OXFORD UNIVERSITY PRESS The Computer Journal

BlackjackBench: Portable Hardware Characterization with Automated Results Analysis

Journal:	*The Computer Journal*
Manuscript ID:	COMPJ-2012-03-0208
Manuscript Type:	Original Article
Date Submitted by the Author:	31-Mar-2012
Complete List of Authors:	Danalis, Anthony; University of Tennessee, Luszczek, Piotr; University of Tennessee, Marin, Gabriel; Oak Ridge National Laboratory, Vetter, Jeffrey; Oak Ridge National Laboratory, Dongarra, Jack; University of Tennessee,
Key Words:	Micro-Benchmarks, Hardware Characterization, Statistical Analysis

BlackjackBench: Portable Hardware Characterization with Automated Results Analysis

ANTHONY DANALIS[1], PIOTR LUSZCZEK[1], GABRIEL MARIN[2], JEFFREY S. VETTER[2] AND JACK DONGARRA[1]

[1]*University of Tennessee, Knoxville, TN, USA*
[2]*Oak Ridge National Laboratory, Oak Ridge, TN, USA*
Email: {adanalis,luszczek,dongarra}@eecs.utk.edu, {maring,vetter}@ornl.gov

DARPA's *AACE* project aimed to develop Architecture Aware Compiler Environments. Such a compiler automatically characterizes the targetted hardware and optimizes the application codes accordingly. We present the *BlackjackBench* suite, a collection of portable micro-benchmarks that automate system characterization, plus statistical analysis techniques for interpreting the results. The BlackjackBench benchmarks discover 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 executables generated by existing compilers from 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 these features of modern multicores can be discovered programmatically. We also show how the features could potentially interfere with each other resulting in incorrect interpretation of the results, and how established classification and statistical analysis techniques can reduce experimental noise and aid automatic interpretation of results. We show how effective hardware metrics from our probes allow *guided tuning* of computational kernels that outperform an autotuning library further tuned by the hardware vendor.

Keywords: Micro-Benchmarks, Hardware Characterization, Statistical Analysis

Received 00 January 2009; revised 00 Month 2009

1. INTRODUCTION

Compilers, autotuners, numerical libraries, and other performance sensitive software need information about the underlying hardware. If portable performance is a goal, automatic detection of hardware characteristics is necessary given the dramatic changes undergone by computer hardware. Several system benchmarks exist in the literature [1, 2, 3, 4, 5, 6, 7, 8]. However, as hardware becomes more complex, new features need to be characterized, and assumptions about hardware behavior need to be revised, or completely redesigned.

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

1. A collection of portable micro-benchmarks that can probe the hardware and record its behavior while control variables, such as buffer size, are varied.
2. A statistical analysis methodology, implemented as a collection of scripts for result parsing, examines the output of the micro-benchmarks and produces the desired system characterization information, e.g. effective speeds and sizes.

BlackjackBench was specifically motivated by the effort to develop architecture aware compiler environments [9] that automatically adapt to hardware, which is unknown to the compiler writer, and optimize application codes based on the discovery of the runtime environment.

Often, important performance related decisions take into account *effective* values of hardware features, rather than their peak values. In this context, we consider an effective value to be the value of a hardware feature that would be experienced by a user level application written in C (or any other portable, high level, standards-compliant language) 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 – complex – hardware characteristics such as modern sophisticated cache prefetchers, and the interaction between the cache and TLB hierarchies, etc. Furthermore, 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. As a consequence, BlackjackBench does not merely output large sets of data that require human intervention and comprehension; it shows information about the hardware characteristics of the tested platform. Moreover, BlackjackBench does not rely on assembler code, specialized kernel modules and libraries, nor non-portable system calls. Therefore, it is a portable system characterization tool.

2. RELATED WORK

Several low level system benchmarks exist, but most have different target audiences, functionality, or assumptions. Some benchmarks, such as those described by Molka et. al [10], aim to analyze the micro-architecture of a specific platform in great detail and thus sacrifice portability and generality. Others [11] sacrifice portability and generality by depending upon specialized software such as PAPI [12]. Autotuning libraries such as ATLAS [13] 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 [14], most of which we need to subsume in order to target a much broader feature spectrum.

Other benchmarks, such as *CacheBench* [8], or `lmbench` [6, 7] 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 "answers" about the values that characterize the hardware platform.

X-Ray [1, 2] is a micro- and nano-benchmark suite that is close to our work in terms of the scope of the system characteristics. There are, however, a number of features that we chose to discover with our tests that are not addressed by X-Ray. There are also differences in methodology which we mention, where appropriate, throughout this document. One important distinguishing feature is X-Ray's emphasis on code generation as part of the benchmarking activity, while we give more emphasis on analyzing the resulting data.

P-Ray [3] is a micro-benchmark suite whose primary aim is to complement 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. In particular, the authors express interest in testing IBM Power and Intel Itanium architectures, which we did in our work.

Servet [5] is yet another suite of benchmarks that attempts to subsume both X-Ray and P-Ray by measuring a similar set of hardware parameters. It adds measurements of interconnect parameters for communication that occurs in a cluster of multicore processors with distributed memory. The methodology and measurement techniques in Servet complement, rather than imitate, those of X-Ray and P-Ray. And they remain in sharp contrast with our work. Unlike Servet, we do not focus on the actual hardware parameters of the tested system. Rather, we seek the observable parameters that often enough are below vendors' advertised specifications. Servet aims for maximum portability of its constituent tests, as does our work, but we were unable to compare this aspect of our efforts as the authors only presented results from Intel Xeon and Itanium clusters.

In summary, our work differs from existing benchmarks in the methodology used in several micro-benchmarks, the breadth of hardware features it characterizes, the automatic statistical analysis of the results, and the emphasis on effective values and the ability to address modern, sophisticated architectures.

We consider the use of BlackjackBench as a tool for model-based tuning and performance engineering to be related to autotuning based existing exhaustive search approaches [15, 16], analytical search methodologies [17], and techniques based on machine learning [18].

3. BENCHMARKS

In this section we describe the operation of our micro-benchmarks and discuss the assumptions about compiler and hardware behavior that make our benchmarks possible. We also present experimental results, from diverse hardware environments, as supporting evidence for the validity of our assumptions.

A key thesis of this work is that *only hardware characteristics with a significant impact on application performance are important*. Therefore, our benchmarks vary controlled variables, such as buffer size, access pattern, number of threads, variable count, etc., in order to observe variations in performance. Our benchmarks rely on assumptions about the behavior of the hardware under different circumstances and try to trigger different behaviors by varying the circumstances.

Our benchmarks regulate control variables, such as buffer size, access pattern, number of threads, etc., in order to observe variations in performance. 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 to optimization tools such as compilers, auto-tuners, etc. Our benchmarks rely on assumptions about the behavior of the hardware under different circumstances, and try to trigger different behaviors by varying the circumstances.

The memory hierarchy in modern architectures is rather complex, with sophisticated hardware prefetchers, victim caches, etc. As a result, several details must be addressed to attain clean results from the benchmarks. Unlike benchmarks [5, 4] that use constant strides when accessing their buffers, we use a technique known as pointer chasing (or pointer chaining) [3, 1, 6]. 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.

1. The initialization of the buffer is not part of the timed code. Therefore, it does not cause noise in the performance measurement loop, unlike calculations of offsets or random addresses done on the critical path. Furthermore, since we can afford for the initialization to be slow, we can use sophisticated pseudo-random number generators, when random access patterns are desirable.
2. 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).
3. It eliminates the possibility that the hardware prefetcher(s) can guess the address of the next element, when the access pattern is random, regardless of the sophistication level of the prefetcher.

To minimize the loop overhead, the pointer chasing loop is unrolled over 100 times. Finally, to minimize random noise in our measurements, we repeat each measurement several times (≈ 30).

3.1. Cache Hierarchy

Improved cache utilization is one of the most performance critical optimizations in modern computer hardware. Processor speed has been increasing faster than memory speed for several decades, making it increasingly harder for main memory to feed all processing elements with data quickly enough. To bridge the gap, fast, albeit small, cache memory has become necessary for fast program execution. In recent years, the pressure on main memory has increased further, as the number of processing elements per socket has been going up [19]. As a result, most modern processor designs incorporate complex, multi-level cache hierarchies that include both shared and non-shared cache levels between the processing elements. In this section we describe the operation of the micro-benchmarks that detect the cache hierarchy characteristics, as well as any eventual asymmetries in the memory hierarchy.

3.1.1. *Cache Line Size*

The assumption that enables this benchmark is that *upon a cache miss the cache fetches a whole line*[3] (A1). As a result, 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 the size of a cache line can result in, at most, two cache misses.

To utilize this observation, our benchmark allocates a buffer large enough to be much larger than the cache, and performs memory accesses in pairs. The buffer is aligned to 512 bytes, which is a size assumed to be safely 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, in a sense, touches the first and the last elements of a memory segment with extent D of a random buffer location. To achieve this, 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 size, each even access will be in the same line as the previous odd access, and this will result in a cache hit. If D is larger than the cache line size, the two addresses will map to different cache lines and both accesses will result in cache misses.

Clearly, 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 further away caches, or the RAM, 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, on a Core 2 Duo processor, can be seen in Figure 1.

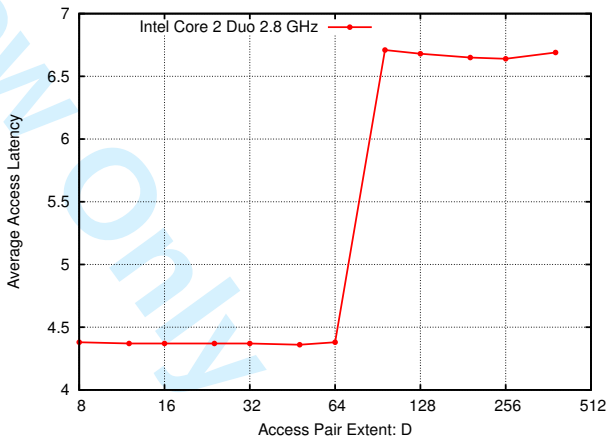

FIGURE 1. Cache Line Size Characterization on Core 2 Duo

3.1.2. *Cache Size and Latency*

Using the cache line size information we can detect the number of cache levels, as well as their sizes and access latencies. The enabling assumption is that *performing multiple accesses to a buffer that resides in the cache is faster than performing the same number of accesses to a buffer that does not reside in the cache (or only partially resides in the cache)* (A2).

The benchmark starts by allocating a buffer that is expected to fit in the smallest cache of the system; for example, a buffer only a few cache lines large. Then we access the whole buffer with stride equal to the cache

[3] On some architectures the cache fetches two consecutive lines upon a miss. Since we are interested in *effective* sizes, for such architectures we report the cache line to be twice the size of the vendor documented value.

line size. By making the access pattern random to avoid prefetching effects, and by continuously accessing the buffer until every element has been accessed multiple times to amortize start-up and cold misses overhead, we can estimate the average latency per access.

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 buffers that reside in different levels of cache. Therefore, 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 equal the number of caches plus one, the extra step corresponds to the main memory, and the Y value at the apex of each step will correspond to the access latency of each cache.

In order to eliminate false steps, the cache benchmarks need to minimize the effects of the TLB. To achieve that goal, the accesses to the buffer are not uniformly random. Instead, the buffer is logically split into segments equal to a TLB 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. In a typical modern architecture the value of S_P/S_L is around 64. While this approach does not completely eliminate the cost of a TLB miss, it significantly amortizes it. Indeed, the data sets we have gathered from real hardware are easy to correlate with the characteristics of the cache hierarchy and exhibit little interference due to the TLB. Figure 2 shows the output of the benchmark on an Atom processor.

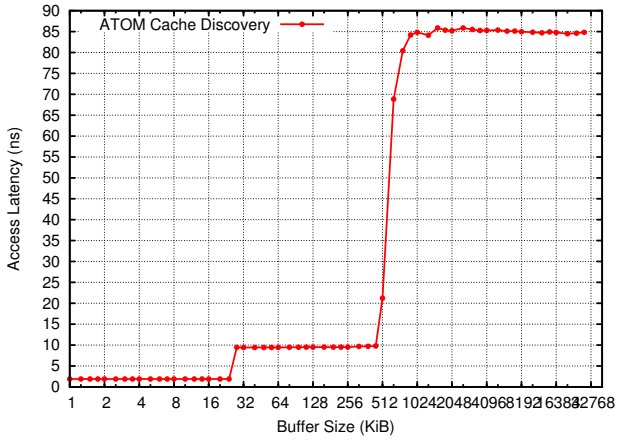

FIGURE 2. Cache Count, Size and Latency Characterization on Atom

3.1.3. *Cache Associativity*

This benchmark is based on the assumption that *in a system with a cache of size S_c, two memory addresses A_1 and A_2, where $A_1 \% S_c == A_2 \% S_c$, will map to the same set of an N-way set associative cache* (A3). The benchmark assumes knowledge of the cache size, potentially extracted from the execution of the benchmark mentioned above.

Assuming that the cache size is S_c, we allocate a buffer many times larger than the cache size, $M * S_c$. Next, the benchmark starts accessing a part of the buffer that is $K * S_c$ 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 3.

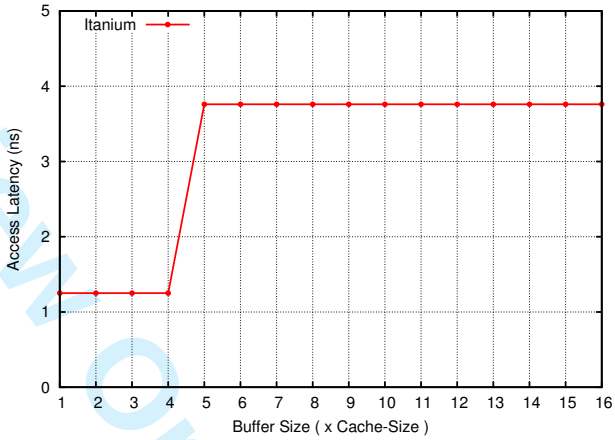

FIGURE 3. Cache Associativity Characterization on Itanium II

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

3.1.4. *Asymmetries in the Memory Hierarchy*

With the move to multi-core processors, we witnessed the quasi-general introduction of shared cache levels to the memory hierarchy. Multicore processors introduced sharing of caches at some levels to the memory hierarchy. A shared cache design provides larger cache capacity by eliminating data replication for multi-threaded applications. The entire cache may be used by the single active core for single-threaded workloads. More importantly, a shared cache design eliminates on-chip cache coherence at that cache level. In addition, it resolves coherence of the private lower level caches internally within the chip and thus reduces external coherence traffic. One downside of shared caches is a larger hit latency which may cause increased cache

contention and unpredictable conflicts. Shared caches are not an entirely new design feature. Before level two caches were integrated onto the chip, some SMP architectures were using external shared L2 caches to increase capacity for single threaded workloads, and to reduce communication costs between processors.

Over the last decade, we also witnessed the integration of the memory controller into the processor chip even for desktop grade processors. On the positive side, the on-chip memory controller increases memory performance by reducing the number of chip boundary crossings for a memory access, and by removing the memory bus as a shared resource. Memory bandwidth can be increased by adding additional memory controllers. Multi-socket systems now feature multiple memory controllers. It is expected that as the number of cores packed onto a single chip continues to increase, multiple on-chip memory controllers will be needed to keep up with the memory bandwidth demand. This higher memory bandwidth availability comes at the expense of increased hardware complexity in the form of Non-Uniform Memory Access (NUMA) costs. Cores experience lower latency when accessing memory attached to their own memory controller.

Both of these hardware features can potentially create asymmetries in the memory system. They may cause subsets of cores to have an affinity to each other – cores communicate faster with other cores from the same subset than with cores that are not part of the subset. For example, a cache level may be shared by only a subset of cores on a chip as is the case with the L2 cache on the Intel Yorkfield family of quad core processors. Thus, cores that share some level of cache may exchange data faster among themselves than with cores that do not share that same level of cache. For NUMA architectures, data allocated on one NUMA node is accessed faster by cores located on the same NUMA node than by cores from different NUMA nodes.

To understand such memory asymmetries, we use a multi-threaded micro-benchmark based on a producer-consumer pattern. Two threads access a shared memory block. First, one thread allocates and then initializes the memory block with zeros. The benchmarks assume that a NUMA system implements the first-touch page allocation policy. By allocating a new memory block and then touching it, the memory block will be allocated in the local memory of the first thread. Page allocation policy can be also set using OS specific controls, e.g., the `numactl` command on NUMA-aware Linux kernels. 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 takes its turn. 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 with other data. This approach causes the two threads to act both as producers and as consumers, switching roles after each complete update of the memory block. We repeat this process many times to minimize secondary effects and we compute the achieved bandwidth for different memory block sizes. We use a range of block sizes, from a block size smaller than the L1 cache size to a block size larger than the size of the last level of cache.

We measure the communication bandwidth between all pairs of cores. To be OS independent, the benchmark must assume that a thread is executed on the same core for its entire life. For best results, we control the placement of the threads using an OS specific API for pinning threads to cores. The hwloc [20] library provides a portable API for setting thread affinity, and we plan to integrate it into the benchmark suite. Next, the communication profiles are analyzed in decreasing order of the memory block size, to detect any potential cliques of cores that have an affinity to each other. The algorithm starts with a large clique that includes all the cores in the system. 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 with each other at this smaller memory block level. In the end, the algorithm will produce a locality tree that captures all detectable asymmetries in the memory hierarchy.

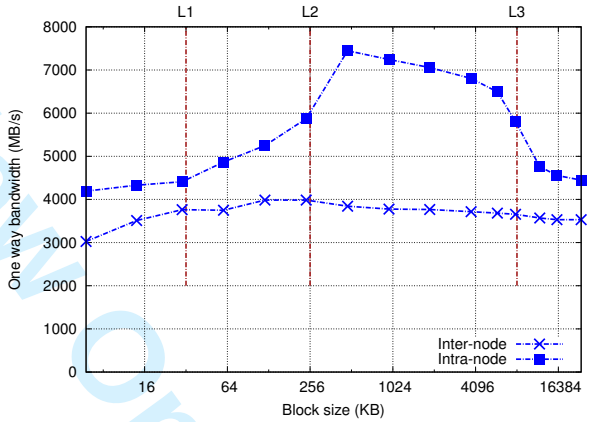

FIGURE 4. One-way, inter-core communication bandwidth for different memory block sizes and core placements on a dual-socket Intel Gainestown system.

Figure 4 shows aggregated results for an Intel Gainestown system with two sockets and Hyper-Threading disabled. The *x* axis represents the memory block size, and the *y* axis represents the bandwidth observed by one of the threads. The bandwidth is computed as *number_updated_lines* * *cache_line_size* / *time*, where *number_updated_lines* is the number of cache lines updated by the first thread. 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. when the two cores reside on different NUMA nodes, curve labeled *inter-node*;
2. when the two cores are on the same NUMA node, curve labeled *intra-node*.

For this system, 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 cores, 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 size four cores at the next level corresponding to the two NUMA nodes in the system.

3.2. TLB Hierarchy

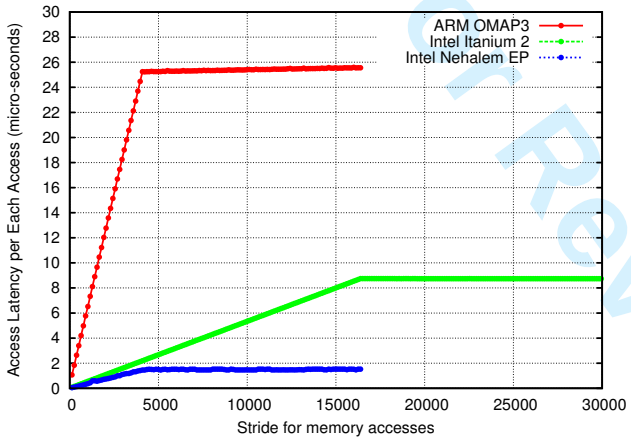

FIGURE 5. Graph of timing results that reveal the TLB page size.

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 methodology. Accordingly, we will focus on the description of our TLB benchmarking techniques rather than present differences and similarities with the cache benchmarks.

The crucial feature that any TLB benchmark should posses is the ability to alleviate cache effects on the measurements. Both conflict and capacity misses coming from data caches should either be avoided at runtime or filtered out during the analysis of the results. We chose the latter, as it has the added benefit of 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 eliminated by the high cost of a TLB miss in the last level of TLB. Handling these misses requires the traversal of the OS page table stored in the main memory – the combined latency exceeds the cost of a miss for any level of cache. Typical timing curves for this benchmark are shown in Figure 5. 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. For our example, the page size is 4KB on both ARM and Nehalem, and 16KB on Itanium.

There are programmatic means of obtaining the TLB page size, such as the legacy POSIX call `getpagesize()` or the more modern `sysconf(_SC_PAGE_SIZE)`. One obvious caveat of using any of these functions is portability. A more important caveat is the fact that these functions return the system page size rather than the hardware TLB page size. For most practical purposes they are the same, but modern processors support multiple page sizes and large page sizes are often used to increase performance. Under such circumstances, our test delivers the observable size of a TLB page, which is the preferred value from a performance stand point.

A similar argument can be made about counting the actual page faults rather than measuring the execution time and inferring the number of faults from the timing variations. There are system interfaces such as `getrusage()` on Unix. The pitfall here is that these interfaces assume the faults will either be the result of the first touch of the page or that the fault occurred because the page was swapped out to disk. Again, our technique sidesteps these issues all together.

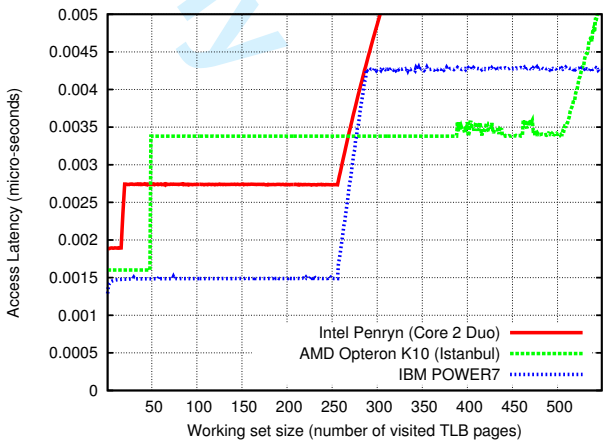

FIGURE 6. Graph of results that reveal the number of levels of TLB and the number of entries in each level.

Once the actual TLB page is known, it is possible to proceed with discovering the sizes of the levels of the TLB

hierarchy. Probably the most important task is to minimize the impact of the data cache. 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 [4]. 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 6 shows the timing graphs on a variety of platforms. Both Level 1 and Level 2 TLBs are identified accurately. As we mentioned before, we aim at the observable parameters: even if the TLB size is 512 entries, some of these entries will not contain user data. In a running process, 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.

3.3. Arithmetic Operations

Information about the processing units, such as the latency and throughput of various arithmetic operations, are important to compilers and performance analysis tools. Such information is needed to produce efficient execution schedules that attempt to balance the type of operations in loops with the number and type of resources available on the target architecture.

3.3.1. 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 cost per operation is computed by dividing the wall clock time of the loop by the total number of operations executed. The loop is executed twice, using the same number of iterations but with different unroll factors, and the difference of the two execution times is divided by the difference in operation counts between the two loops. This approach eliminates the effects of the loop overhead, without increasing the unroll factor to a very large value, which would significantly increase the time spent in compiling the micro-benchmarks and could create instruction cache issues. To account for run-time variability, each benchmark is executed six times, and the second lowest computed value is selected.

3.3.2. Instruction throughputs

The throughput $\mathscr{T}(O,T)$ of an operation O, with operands of type T, represents the rate at which one thread of control 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.

Instruction throughputs are measured using an approach similar to the one used for measuring instruction latencies. However, 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 independent operations increases register pressure, potentially causing unnecessary delays due to register spills/unspills. Therefore, for each operation type O and operand type T, multiple micro-benchmarks are generated each with different number of independent streams of operations. 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}.

3.3.3. Operations in flight

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

Operations in flight are measured using an approach similar to the ones used for operation latencies and operation throughputs. For each operation type O and operand type T, multiple micro-benchmarks, with different numbers of independent streams of operations, are generated and benchmarked. However, unlike the operation throughput benchmarks where we are interested in the minimum cost per operation, to understand the number of operations in flight we are looking at the cost per iteration loop. Each independent stream contains the same number of chained operations, and thus, the total number of operations in one loop iteration grows proportionally with the number of streams. When we increase the number of independent streams in the loop, as long as the processor can pipeline all the independent operations, the cost per iteration should remain constant. Thus, the inflection point where the cost per iteration starts to increase yields the number of operations in flight supported by one thread of control.

3.4. Execution Contexts

Modern systems commonly have multiple cores per socket and multiple sockets per node. To avoid confusion due to the overloading of the terms *core*, *CPU*, *node*, etc, by hardware vendors, 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:

1. Floating Point Execution Contexts,
2. Integer Execution Contexts, and
3. 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. As a result, in a system with N floating point contexts (or integer, or memory), we should be able to execute up to N parallel threads that perform intense floating point computations (or integer, or memory) and observe no interference between the threads. 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 $M > N$ threads at least one thread 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 (integral) 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 $M > N$.

While this behavior is true for every machine we have tested, the behavior for $X > N$ depends on the scheduler of the Operating System. Namely, since we do not explicitly bind the threads to cores, the operating system is free to move them around in an effort to distribute the "extra" load caused by the $X - N$ threads (for $X > N$) among all computing resources. If the operating system does so, then the total execution time for $X > N$ will increase almost linearly with the number of threads. If the operating system chooses to leave the threads on the same core for their whole life time, the result will look like a step function with steps at $X = N$, $X = 2 * N$, $X = 3 * N$, etc, and corresponding apexes at $Y = 1$, $Y = 2$, $Y = 3$ and so on. The latter tends to be a rare case, but we observed it in our private Cray XT5 system, Jaguar.

For the case of the Memory Intensive Contexts it is important to distinguish between the ability of the execution contexts to perform memory operations and the ability of the memory subsystem to serve multiple parallel memory requests. 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 on variables initialized to 1 (so the values do not diverge) in a non-decidable way so that the compiler cannot simplify the loop. A sample output of this benchmark on a Power7 processor can be seen in Figure 7.

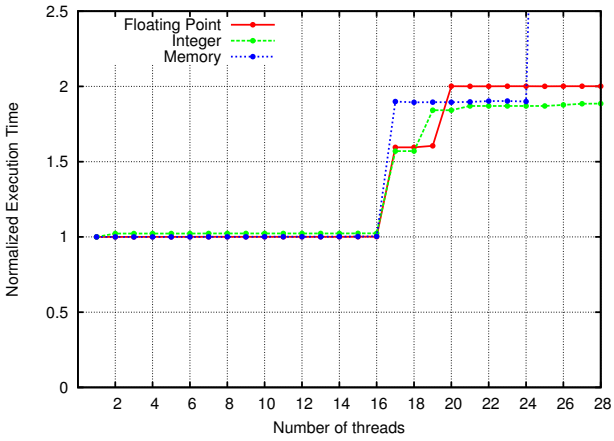

FIGURE 7. Execution Contexts Characterization on Power7

3.5. Live Ranges

An important hardware characteristic for optimization is the maximum number of registers available to an application. Although the number of registers on an architecture is usually known, not all the registers can be used for application variables. For example, either the stack pointer or the frame pointer should be held in a register as at least one of them is essential for retrieving data from the stack (both cannot be stored on the stack). Other constraints or conventions require allocation of hardware registers for special purposes thus reducing the actual number of registers available for applications. We use the term "Live Ranges" to describe the maximum number of concurrently usable registers by a user code.

When the number of concurrently live variables in a program exceeds the number of available registers, the compiler has to introduce a piece of code that "spills" the extraneous variables to memory. 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 that are otherwise used by the original computation performed by the application. 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 R by detecting the step in the resulting data.

The structure of the loop is similar to the one used by X-Ray [1], but is modified in two ways.

1. The `switch` statement, used in the loop by X-Ray, is unnecessary and detrimental to the benchmark. It is unnecessary as the chained dependences of the operations are sufficient to prohibit the compiler from aggressive code optimizion (by simplifying, or

reducing the number of operations in the loop). It is detrimental because some compilers, on some architectures, choose to implement the switch with a jump determined by a register. As a result, the benchmark could report one less register than the number that is available for application variables.

2. 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. We observed this effect when the compiler generated instruction schedules tried to hide the spill/unspill overhead by using the extra resources of the CPU. 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.

In a similar manner to other benchmarks, where sensitive timing is performed, special attention was paid to details that can affect the performance, or the decisions of the compiler. As an example, the K operations that we time have to be performed thousands of times in a loop due to the coarse granularity of widely available timers. However, use of loops introduces latency and poses a dilemma for the compiler as to whether the loop induction variable should be kept in a register. Manually unrolling the body of the loop multiple times amortizes the loop overhead and reduces the importance of the induction variable when register allocation is performed. An example run of this benchmark, on a Power7 processor, can be seen in Figure 8.

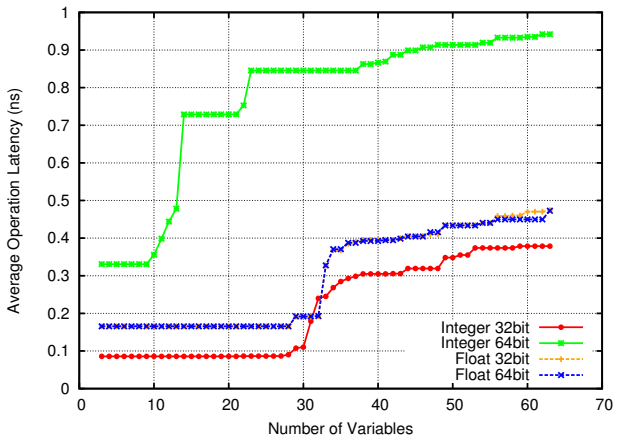

FIGURE 8. Live Ranges Characterization on Power7

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. This is the case since the pipeline is not fully utilized when the data dependencies do not allow several operations to execute in parallel. However, this benchmark is only aiming to identify the latency associated with spilling registers to memory, which results in increased latency. For this reason, 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 4.1.

3.6. OS Scheduler time slot

Multithreaded applications can tune synchronization, scheduling, and work distribution decisions by using information about the typical duration of time a thread will run on the CPU uninterrupted after the OS schedules it for execution. We refer to this time duration as the OS Scheduler time slot. The assumption that enables this benchmark is that *when a program is being 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 (because the program execution was preempted by the OS between these instructions) will be separated by a time interval on the order of the OS Scheduler time slot, T_s, which is several orders of magnitude larger than $1/F$* (A6).

Our benchmark consists of several threads. Each thread executes the following pseudocode:

$te = ts = t0 = timestamp()$
$while(te - t0 < TOTAL\_RUNNING\_TIME)$
  $OP_0 \cdots OP_N$
  $te = timestamp()$
  $if(te - ts > THRESHOLD)\ record(te - ts)$
  $ts = te$

In other words, if T_i was larger than a predefined threshold then a loop that executes a minimal number of operations takes a timestamp and records the length of time T_i since the previous timestamp, The threshold must be chosen such that it is much longer than the time the few operations and the timestamp function take to execute, but much shorter than the expected value of the scheduler time slot, T_s. In a typical modern system, a handful of arithmetic operations and a call to a timestamp function, such as gettimeofday, take less than a microsecond. On the other hand, the scheduler time slot is typically in the order of, or larger than, a millisecond. Therefore, a THRESHOLD of $20\mu sec$ safely satisfies all our criteria. Since we do not record the duration, T_i, of any loop iteration with $T_i < THRESHOLD$, and we have chosen THRESHOLD to be significantly larger than the pure execution time of each iteration, 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 (i.e., cores), 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 other esoteric OS details, the mode of the output data distribution (i.e., the most common

measurement taken) will be the duration of the scheduler time slot, T_s. Indeed, experiments on several hardware platforms and OSes have produced expected results.

4. STATISTICAL ANALYSIS

The output of the micro-benchmarks discussed in section 3 is typically a performance curve, or rather, a large number of points representing the performance of the code for different values of the control variable. Since our motivation for developing BlackjackBench was to inform compilers and auto-tuners about the characteristics of a given hardware platform, we have developed analyses that can process the performance curves and output values that correspond to actual hardware characteristics.

4.1. 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 – second order – 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: $\forall i : X_i = \min_{j \geq i} X_j$

4.2. 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.

4.2.1. *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 7.

To extract the number of Execution Contexts from this data, we are interested in that first jump from the straight line to the noisy data. However, due to noise, there can be small jumps in the part of the data that is expected to be flat. The challenge is to systematically define what constitutes a small jump versus a large jump for the data sets that can be generated by this benchmark. To address this, we first calculate the relative value increase in every step $dY_n^r = \frac{Y_{n+1} - Y_n}{Y_n}$ and then compute the average relative increase $\langle dY^r \rangle$. We argue that the data point that corresponds to the jump we want (and thus the number of execution contexts) 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 (noise) and a few much larger values (actual steps) will be a value higher than the noise, but smaller than the steps. Thus the average relative increase $\langle dY^r \rangle$ gives us a threshold to differentiate between small and large values for every data set of this type.

4.2.2. *Biggest Step*

The Live Ranges benchmark produces curves that start flat (when all variables fit in registers) then potentially grow slightly (if some registers are unavalable for some reason), then exhibit a large step when the first spill to memory occurs, and then continue growing in a non-regular way as can be seen in Figure 8. Due to the increase in latency before the first spill, the previous approach for detecting the first step is not appropriate for this type of data. However, we can utilize the fact that 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 9 shows the data curves for the integer live ranges along with the relative dY^r values.

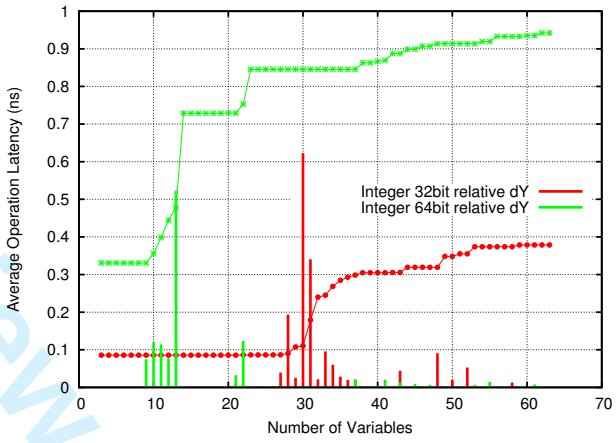

FIGURE 9. Live Ranges and Relative dY

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 $dY^s = dY \times Y$ (instead of the biggest relative step).

4.2.3. *Quality Threshold Clustering*

Unlike the previous cases, where the analysis aimed to extract a single value from each data set, the benchmark for detecting the cache size, count, and latency has multiple steps that carry useful information. Due to the regular nature of the steps this benchmark tends to generate, 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 [21]. The modification pertains to the cluster diameter threshold used by the algorithm to determine if a candidate point belongs to a cluster or not. In particular, 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. The algorithm we use is represented in pseudocode in Figure 10.

```
1| QT_clustering(G)
2| if |G|≤ 1 then
3|    output G
4| else
5|    for i ∈ G do
6|       flag=true
7|       A_i ← {i} # A_i is the cluster started by i
8|       while flag=trueand A_i ≠G do
9|          find j ∈ (G−A_i) such that diameter(A_i∪{j}) is minimum
10|         if diameter(A_i∪{j}) > 0.25·average(A_i) then
11|            flag=false
12|         else
13|            A_i ← A_i∪{j} # Add j to cluster A_i
14|         end if
15|      end while
16|      identify set C ∈ {A_1,A_2,...,A_|G|} with maximum cardinality
17|      print C
18|      QT_clustering(G−C)
19|   end for
20| end if
```

FIGURE 10. Quality Threshold Clustering Pseudocode

Using QT-Clustering, we can obtain the clusters of points that correspond to each cache level. Thus, we can extract for each cache level the size information from the maximum X value of each cluster, the latency information from the minimum Y value of each cluster and the number of levels of the cache hierarchy from the number of clusters. An example use of this analysis, on the data from a Power7 processor, can be seen in Figure 11. QT-Clustering is also used for the levels of TLB.

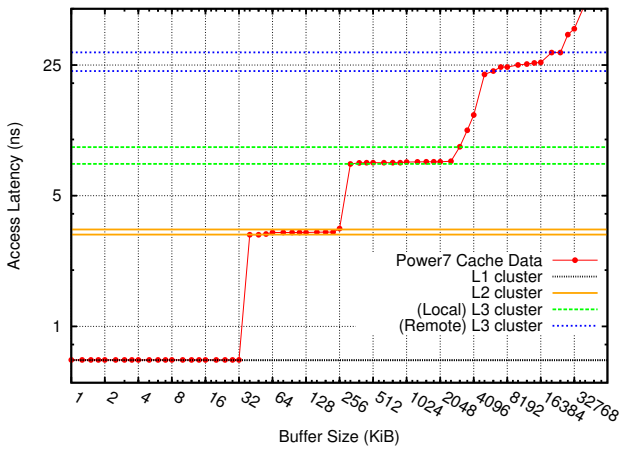

FIGURE 11. QT-Clustering applied to Power7 Cache Data

5. DISCUSSION

Comparing Figures 2 and 11 against vendor documents, one can notice that there are discrepancies in some of the reported values. For example, IBM advertises the L3 cache of the 8 core Power7 processor as 32MiB. Also, according to the vendor, there is no L4 cache. However, our benchmark detects an L3 of size just under 4MiB and a fourth level of cache with a size just under 32 MiB. The reason behind this behavior is that our benchmarks aim at the effective values of the hardware features, as those are viewed using changes in performance as the only criterion. In the case of the Power7, the L3 cache is organized in 4MiB chunks "local" to each core, but shared between all cores. As a result, each core can access the first 4MiB of the L3 faster than it accesses the rest. It then appears as if there were two discreet cache levels. The size being detected is slightly smaller than vendor's claims. This phenomenon occurs in virtually every architecture we have tested, for the largest cache in the hierarchy. The cause differs between architectures, and it is a combination of imperfect cache replacement policies, misses due to associativity, and pollution due to caching of data such as page table entries, or program instructions. As a result, if a user application has a working set equal to the vendor advertised value for the largest cache, it is practically impossible for that application to execute without a performance penalty due to cache misses. Therefore, we consider the size of the largest working set that can be used without a significant performance penalty to be the *effective* cache size, which is the value we are interested in.

A few of our measurements may be affected by a variety of system settings. For example, BIOS on some servers include an option for pair-wise prefetching in Intel Xeon server chips. The Itanium architecture has its peculiarities such as lack of Level 1 caching support for floating-point values and two different TLB page sizes at each level of TLB. In fact, TLB support can be quite peculiar and this includes a separate TLB space for huge page entries on AMD's Opteron chips. Similarly, there are lock-down TLB entries that could potentially be utilized differently by various implementations of the the ARM processor designs. And there is the issue of read-only Level 1 TLB on the Intel Core 2 architecture. Occasionally, the concurrency of in-memory TLB table walkers may limit the performance if multiple cores happen to suffer heavy TLB miss rates and require constant TLB reloading from the main memory. Finally, some OSes (HP-UX and AIX in particular) allow the user's binary executables to include the information on what page size should be used during process execution rather than have a system-wide default value. Such uncommon conditions can be challenging for our benchmarks, in that the produced results could differ from the user's expectations.

6. GUIDED TUNING

Aside from benefiting AACE, we consider BlackjackBench as an indispensable tool for model-based tuning and performance engineering of computational kernels for scientific codes. One such kernel called DSSSM [22], is absolutely essential for good performance of tile linear algebra codes [23]. For the sake of brevity and simplicity of exposition, we will show how BlackjackBench helps us guide a model-based tuning of one of the components of the kernel: a Schur's complement update based on matrix-matrix multiplication [24]. We believe that our approach is a viable alternative to existing exhaustive search approaches [15, 16], analytical search [17], or approaches

based on machine learning [18].

We proceed first by constructing a cache reuse model to maximize the issue rate of floating-point operations. In fact, it is desirable to retire as many such operations in every cycle as is feasible given the number of FPU units. The Schur's complement kernel has a high memory reuse factor that grows linearly with the input matrix sizes. Our model is a constrained optimization problem with discrete parameters:

$$\max_{m+n+k\leq R} \frac{2mnk}{2mnk+(mn+mk+nk)}$$

where m, n, and k are input matrix dimensions and R is the number of registers or the cache size. When the above fraction is maximized, the code performs the most amount of floating point operations per each load operation for the kernel under consideration. BlackjackBench was used to supply the values of R for the register file (Live Ranges) and Level 1 cache size. Optimizations that take into account additional levels of cache are not as important for tile algorithms [23].

We put our methodology to the test on a dual-core Intel Core i7 machine with 2.66 GHz clock. This determines the size of register file (16) and Level 1 cache (32 KiB) as supplied by BlackjackBench and confirmed with the hardware specification. Our model points to register blocking with $m = 3$, $n = 3$, and $k = 1$; cache blocking with $m = 63$, $n = 63$, and $k = 1$. These parameters however are not practical because the kernel would most likely always be called with even matrix sizes and so the cleanup code [15] would have to be utilized in the majority of cases which may drastically diminish the performance. Thus, a more feasible register blocking is $m = 4$, $n = 2$, and $k = 1$. Level 1 cache blocking has to be adjusted to match the register blocking. Further adjustment comes from the fact that one of the input matrices in our kernel may have non-unit stride, and loading its entries into the L1 cache brings in one cache line worth of floating-point data, and we need room to accommodate for these extra items. Using BlackjackBench we discover the cache line size and thus, the corresponding change in cache blocking parameters is: $m, n = 56$, and $k = 1$. Since m and n are the same, we refer to them collectively as *matrix size*.

Figure 12 shows performance results for our kernel and the vendor implementation. According to our model and the consideration presented above, optimal performance should occur at matrix size 56 and the figure confirms this finding: the Blackjack line drops for matrix sizes larger than this value. An additional benefit of our guided tuning is the fact that we are able to achieve significantly higher performance for the range of matrix sizes that are the most important in our application.

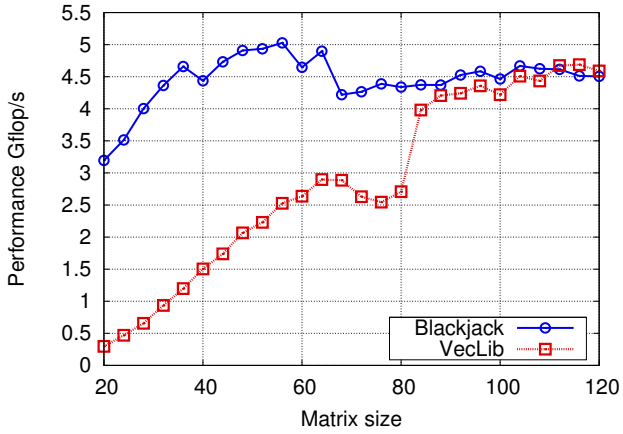

FIGURE 12. Performance of the Schur's complement kernel routines. VecLib is based on ATLAS. Blackjack is a routine optimized for Level 1 cache.

7. CONCLUSION

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

1. Offering micro-benchmarks that can exercise a wider set of hardware features than most existing benchmark suites do.
2. Emphasizing portability by avoiding low level primitives, specialized software tools and libraries, or non-portable OS calls.
3. Providing comprehensive statistical analyses as part of the characterization suite, capable of distilling the results of micro-benchmarks into useful values that describe the hardware.
4. Emphasizing the detection of hardware features through variations in performance. As a result, BlackjackBench detects the *effective* values of hardware characteristics, which is what a user level application experiences when running on the hardware, instead of often unattainable peak values.

We describe how the micro-benchmarks operate, and their fundamental assumptions. We explain the analysis techniques for extracting useful information from the results and demonstrate, through several examples drawn from a variety of hardware platforms and OSes, that our assumptions are valid and our benchmarks are portable.

REFERENCES

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

[2] Yotov, K., Jackson, S., Steele, T., Pingali, K., and Stodghill, P. (2005) Automatic measurement of instruction cache capacity. *Proceedings of the 18th Workshop on Languages and Compilers for Parallel Computing (LCPC)*.

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

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

[5] Gonzalez-Dominguez, J., Taboada, G., Fraguela, B., Martin, M., and Tourio, J. (2010) Servet: A Benchmark Suite for Autotuning on Multicore Clusters. *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Atlanta, GA.

[6] McVoy, L. and Staelin, C. (1996) `lmbench`: portable tools for performance analysis. *ATEC'96: Proceedings of the Annual Technical Conference on USENIX 1996 Annual Technical Conference*, Berkeley, CA, USA, USENIX Association., pp. 23–23.

[7] Staelin, C. and McVoy, L. (1998) mhz: Anatomy of a microbenchmark. *USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, January 15-18, pp. 155–166.

[8] Mucci, P. J. and London, K. (1998) The CacheBench Report. Technical report. Computer Science Department, University of Tennessee Knoxville.

[9] DARPA AACE. `http://www.darpa.mil/ipto/solicit/baa/BAA-08-30_PIP.pdf`.

[10] Molka, D., Hackenberg, D., Schone, R., and Muller, M. S. (2009) Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, pp. 261–270. IEEE Computer Society.

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

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

[13] Whaley, R. C., Petitet, A., and Dongarra, J. J. (2001) Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, **27**, 3–35.

[14] Whaley, R. C. and Castaldo, A. M. (2008) Achieving accurate and context-sensitive timing for code optimization. *Software: Practice and Experience*, **38**, 1621–1642.

[15] Whaley, R. C. and Dongarra, J. (1998) Automatically tuned linear algebra software. *CD-ROM Proceedings of SuperComputing 1998: High Performance Networking and Computing*.

[16] Whaley, R. C., Petitet, A., and Dongarra, J. J. (2001) Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, **27**, 3–35.

[17] Yotov, K., Li, X., Ren, G., Garzaran, M., Padua, D., Pingali, K., and Stodghill, P. (2005) A comparison of empirical and model-driven optimization. *Proceedings of the IEEE*, **93**. special issue on "Program Generation, Optimization, and Adaptation".

[18] Singh, K. and Weaver, V. (2007) Learning models in self-optimizing systems. *COM S*, **612**.

[19] Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Luszczek, P., and Tomov, S. (2006) The impact of multicore on math software. In Kågström, B., Elmroth, E., Dongarra, J., and Wasniewski, J. (eds.), *Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, PARA*, Lecture Notes in Computer Science, **4699**, pp. 1–10. Springer.

[20] Broquedis, F., Clet Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010) hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE (ed.), *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa Italie, 02.

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

[22] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. J. (2008) Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, **20**, 1573–1590. `http://dx.doi.org/10.1002/cpe.1301` DOI: 10.1002/cpe.1301.

[23] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S. (2009) Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, **180**.

[24] Golub, G. and Van Loan, C. (1996) *Matrix Computations*, 3rd edition. Johns Hopkins University Press, Baltimore, MD.