Portable Techniques to Find Effective Memory Hierarchy Parameters

Keith Cooper
Rice University
Houston, Texas

Jeffrey Sandoval
Cray, Incorporated
St. Paul, Minnesota

***Abstract*—Application performance on modern microprocessors depends heavily on performance related characteristics of the underlying architecture. To achieve the best performance, an application must be tuned to both the target-processor family and, in many cases, to the specific model, as memory-hierarchy parameters vary in important ways between models. Manual tuning is too inefficient to be practical; we need compilers that perform model-specific tuning automatically.**

To make such tuning practical, we need techniques that can automatically discern the critical performance parameters of a new computer system. While some of these parameters can be found in manuals, many of them cannot. To further complicate matters, compiler-based optimization should target the system's behavior rather than its hardware limits. Effective cache capacities, in particular, can be smaller than the hardware limits for a number of reasons, such as sharing between cores or between instruction and data caches. Physical address mapping can also reduce the effective cache capacity.

To address these challenges, we have developed a suite of portable tools that derive many of the effective parameters of the memory hierarchy. Our work builds on a long line of prior art that uses micro-benchmarks to analyze the memory system. We separate the design of a reference string that elicits a specific behavior from the analysis that interprets that behavior. We present a novel set of reference strings and a new robust approach to analyzing the results. We present experimental validation on a collection of 20 processors.

I. Introduction

Application performance on today's multi-core processors is often limited by the performance of the system's memory hierarchy. To achieve good performance, the code must be carefully tailored to the detailed memory structure of the target processor. That structure varies widely across different architectures, even for models of the same ISA. Thus, performance is often limited by the compiler's ability to understand model-specific differences in the memory hierarchy and to tailor the program's behavior accordingly.

This paper presents a set of techniques to discover, empirically, the capacities and other parameters of the various levels in the data memory hierarchy, both cache and TLB. Our toolset computes *effective* values for the various memory-hierarchy parameters that it measures, rather than finding the full hardware capacity. We define *effective capacity* to mean the amount of memory at each level that an application can use before the access latency begins to rise. The effective value for a parameter can be considered an upper bound on the usable fraction of the physical resource.

In the best case, effective capacity is equal to physical capacity. For example, on most microprocessors, the effective L1 data cache capacity is identical to the physical capacity, because the L1 data cache is not shared with other cores, it is separate from the L1 instruction cache, and it is virtually mapped. In contrast, an L2 cache for the same architecture might be shared among cores. It might contain the images of all those cores' L1 instruction caches. It might hold page tables, loaded into L2 by hardware that walks the page table. Each of these effects might reduce the effective L2 cache capacity; modern commodity processors exhibit all three.

A compiler that blocks loops to improve memory access times should achieve better results using these effective cache sizes than it would using the physical hardware limits because the effective number captures the point on the curve of access cost versus capacity where access costs begin to rise. The compiler's goal should be to tile the computation into that fraction of cache that does not cause access time to rise. Several authors have advocated the use of effective capacities rather than physical capacities [1]–[3].

This paper describes techniques that measure effective capacities for a single-threaded application, running on a quiescent system—that is, no other tasks are making significant demands on the memory system. While this scenario is the best case for effective capacities, it presents significant challenges. To obtain clean data, the techniques must carefully isolate specific behaviors, separating, for example cache misses from TLB misses. They must also reduce the impact of transient behavior, such as interference from autonomous processes such as operating system daemons. To produce consistent results, the data requires interpretation. That analysis must be automatic and robust if the tools are to be portable. The tools have been tested on across a broad collection of systems; Section VI shows results from twenty systems.

This paper does not address the problem of finding the effective parameters seen by a single thread in a multithreaded computation, whether on one core or many cores. Rather, these techniques lay the groundwork for a careful investigation of that phenomenon: carefully validated techniques for measurement and analysis of the simpler single-thread behavior. Neither does the paper address the problem of measuring instruction cache capacity, unless that level of cache is shared between the instruction and data cache hierarchies.

This paper builds on a long line of prior work, described in Section II. It extends that work in several important ways. Our focus is on robust micro-benchmarks and automated analysis to interpret the results. The microbenchmarks, described in Section IV, use carefully designed reference strings to isolate

Rice University Computer Science Technical Report 11-06, December 2011

and measure specific memory hierarchy behaviors. They adopt a disciplined approach to time measurement that provides clean, reproducible data. The automated analysis, described in Section V, incorporates a sophisticated multi-step technique to filter, smooth, and interpret the data. It produces a consistent interpretation of the micro-benchmark results across many systems. Finally, Section VI describes our experience using the tools to characterize more than twenty distinct processors and processor models.

II. Why Not Use Existing Tools?

This problem is not new. Prior work has described several systems that attempt to characterize the memory hierarchy [4]–[9]. Our goal has been to build a set of tools that derive the effective memory-system parameters in an automated way. From our perspective, previous systems suffer from several specific flaws.

1) We found no single set of tools that measured the full set of cache and TLB parameters that a compiler needs. With some of the systems, the papers lay out techniques for measuring higher levels of cache or TLB that the distributed software does not implement. Several of the systems rely on a human to interpret the results.
2) The prior tools are not easily portable to current machines. Some rely on system-specific features such as superpages or hardware performance counters to simplify the problems. Others were tested on older systems with shallow hierarchies; they and produce odd and inaccurate results on modern processors. Our tools limit themselves to portable C code with POSIX calls.
3) Sharing in the memory hierarchy complicates the problems of both measurement and analysis enough so that the tools need to model it and account for it explicitly. The techniques in this paper address one part of that problem—understanding behavior.[1]
4) Our own experience has shown us that robust analysis of the data is difficult. For example, several prior systems rely on threshold values to detect transitions in the data—for example, an increase in access time of $>20\%$ indicates a new level in the hierarchy. As systems evolve and models proliferate, such threshold-based techniques invariably fail.
5) Finally, previous tools try to solve for multiple parameters at once. When the code works, this approach is fine. However, if the code finds a wrong answer for one parameter, it inevitably is wrong for the others. For example, colleagues showed us an example where X-RAY [9] computed an L1 associativity of 9 rather than 8, with the result that it reported an L1 capacity that was $\frac{9}{8}$ of the physical value [10].

One or more of these issues arose with each prior system that we tested. These shortcomings motivated our current set of tools. We have built a set of tools that measure a broad variety of cache and TLB parameters, that are portable across a variety of systems, that provide accurate results for shared levels in the memory hierarchy, that include a robust automatic analysis without arbitrary threshold values, and that solve for each parameter independently to avoid compounding errors.

[1]We have also developed tools that derive a graph of sharing relationships between cores and between the instruction cache and data cache hierarchies. Space limitations prevent us from describing those tools in this paper.

III. Literature Review

All memory characterization work appears to derive from Saavedra and Smith [8]. They use a Fortran benchmark to observe memory behavior. It measures the time needed to stride through an array of length N with a stride of s. They generate plots with varied values of N and s and manually interpret the results to determine the cache and TLB capacity, linesize (pagesize), and associativity. They use a single benchmark to determine all characteristics, which requires careful disambiguation between various effects. In contrast, our work uses distinct access patterns for each effect and relies on a robust automated analysis that interprets results for both physically and virtually mapped caches.

In `lmbench`, McVoy and Staelin replaced array access with a linked list traversal to allow indirect and randomized access patterns [7]. This advance was necessitated by improvements in hardware prefetching. Our tools leverage this approach.

In X-RAY, Yotov *et al.* addressed both algorithmic and implementation issues in prior work [9], [11]. X-RAY uses a single test to detect cache capacity and associativity, probing the cache to determine its *shape*. While that works reasonably well on an unshared cache, such as an L1 data cache, features such as sharing or a victim cache can create unexpected results. X-RAY also requires superpages to characterize physically mapped levels in the cache—a serious problem for portability. In contrast, our tools measure each parameter separately; the same tools handle physically and virtually mapped caches.

Both Servet [6] and P-RAY [5] extend prior work to characterize sharing and communication aspects of multi-core clusters. These approaches do not address the issues that we tackle in this paper. Our work on improving cache characterization methodology from the perspective of a single thread is orthogonal to the work on characterizing shared resources. This paper provides a principled foundation for automatic resource characterization, which is necessary for future extensions to multi-core architectures.

IV. Portable Micro-benchmarks

This section describes the micro-benchmarks that we developed to measure specific behaviors. The micro-benchmarks include a general test for effective cache capacity at all levels, a similar test for effective TLB capacity at all levels,and a test for cache linesize at all levels. We rely on the operating-system pagesize reported by the POSIX `sysconf` interface.

Because L1 data cache linesize is useful to reduce spatial locality in the general tests, our tools use a specialized test to find L1 cache parameters. The parameters are later confirmed by the more general tests. This specialized test, the *gap test*, is based on ideas found in X-RAY [9] running in an infrastructure

```
baseline ← time for the G(2,LB,0) reference string
for n ← 2 to MaxAssociativity
  for k ← LB to UB
    t ← time for the G(n,k,0) reference string
    if (t > baseline)
      L1Assoc ← n - 1
      L1Size ← L1Assoc × k
      break out of both loops

for offset ← 1 to pagesize
  t ← time for the G(n,k,offset) reference string
  if t = baseline
    L1LineSize = offset - 1
    break out of loop
```

Fig. 1. Pseudocode for the Gap Test

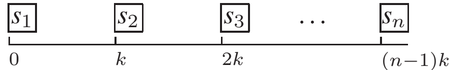
that we developed to obtaining accurate measurements. It measures capacity, associativity, and linesize. Either sharing or physical address mapping can defeat the *gap test*, which makes it unsuitable for caches beyond the L1 cache.

All of the tests rely on a standard C compiler. We use standard POSIX interfaces to build an accurate timer and an allocator that returns page-aligned arrays. (All of our tests use page-aligned arrays to eliminate one source of variation between runs.)

A. Gap Test

We describe the gap test first because it exposes many of the complications that arise in building micro-benchmarks to expose memory hierarchy behavior. The gap test is simple and intuitive. It relies directly on hardware effects caused by the combination of capacity and associativity.

The gap test accesses a set of n locations spaced a uniform k bytes apart. We call this set a *reference string*. We describe the reference strings for the gap test with a tuple *G(n,k,o)* where n is the number of locations to access, k is the number of bytes between those locations, and o is an offset added to the start of the last location in the set. The reference string *G(n,k,0)* generates the following locations:



*G(n,k,**4**)* would move the nth location out another four bytes.

As its first step, the gap test finds cache capacity and associativity. It uses the reference strings to conduct a series of parameter sweeps over n, k, and o, organized as shown in Figure 1. It measures the time taken to run each reference string. It conducts a simple analysis on those results. We describe how to run and time a reference string in subsections below. The test takes four inputs: a lower bound on cache size, *LB*; an upper bound on cache size, *UB*; an upper bound on associativity, *MaxAssoc*, and the OS pagesize from `sysconf`.

The intuition behind this parameter sweep is simple. Consider a direct-mapped cache. The algorithm first tries the set of reference strings from *G(2,**LB**,0)* to *G(2,**UB**,0)*. When k reaches the L1 cache size, the two locations in the reference string will map to the same cache location and each reference will miss in the L1 cache. That effect raises t above *baseline*. The code records cache capacity and associativity, and terminates the loop.

With a set associative cache, the sweep will continue until n is one greater than the associativity and $(n-1)\cdot k$ equals the cache capacity. At that point, the locations in the reference string all map to the same set and, because there are more references than ways in the set, the references will begin to miss. For smaller values of n, all the references will hit in cache and the time will match the baseline time.

The second part of the algorithm uses the same effect to find linesize. It already has values for n and k that match capacity and associativity. It runs a parameter sweep on o in the reference string *G(n,k,**o**)*, When o, the offset in the last block, reaches the linesize, the last access in the string maps into a different set in cache, all n references hit in cache, and the measured time returns to *baseline*.

Of course, both steps assume that we can accurately measure the running time of the reference string and that compulsory misses at the start of that run do not matter.

Running a Reference String To measure the running time for a reference string, the tool must instantiate the string and walk its references enough times to obtain an accurate timing. Our tools build the reference string into an array of pointers that contains a circular linked list of the locations. (In C, we use an array of `void **`.) The code to run the string is simple:

```
loads ← number of accesses
start ← timer()
while (loads - - > 0)
  p ← *p;
finish ← timer()
elapsed ← finish - start
```

The implementation unrolls the loop by a factor of ten to make loop overhead small relative to the memory access costs. The tool selects a number of accesses that is large enough so that the fastest test, *G(2,LB,0)*, runs for at least 1,000 timer ticks.

Timing a Reference String The loop that runs the reference string computes elapsed time using a set of calipers, the calls to *timer*, placed immediately outside the minimal timing loop. In practice, obtaining good times is difficult. Our task is made more difficult by the desire to run on arbitrary POSIX systems in multiuser mode (e.g., not in single-user mode). To obtain sufficiently accurate timings in this environment, we use a simple but rigorous discipline.

First, we use an accurate timer. It calls the POSIX `gettimeofday` routine and combines the resulting `tv_sec` and `tv_usec` values to produce a double-precision floating-point value. We scale the number of accesses to the apparent resolution of this timer, determined experimentally.

Second, we run many trials of each reference string and keep the minimum measured execution time. We want the shortest time for a given reference string; outside interference manifests

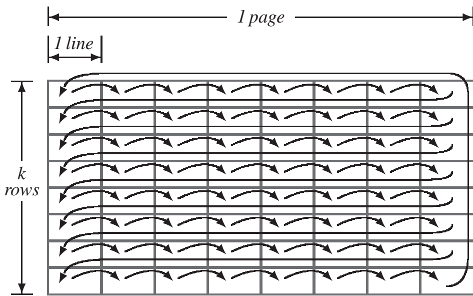


Fig. 2. Cache-Only Reference String

itself in longer times. To find the shortest time, we run the test repeatedly until we have not seen the minimum time change in the last *Trials* runs. A typical value for *Trials* is 100.

Finally, we convert the measured times into cycles. We carefully measure the time taken by an integer add and convert the measured time into integer-add equivalent units. Specifically, we multiply to obtain nanoseconds, divide by the number of accesses, and round the result to an integral number of integer-add equivalents. This conversion eliminates the fractional cycles introduced by amortized compulsory misses and loop overhead.

Experimental validation on a broad variety of machines shows that these techniques produce accurate results for the L1 cache characteristics of a broad variety of architectures (See Section VI). Our other tests use the same basic techniques with different reference strings.

Reducing the Running Time Figure 1 suggests that the parameter sweeps sample the space at a fine and uniform grain. We can radically reduce running time by sampling fewer points. On most systems, for example, the size of the gap, k, will be an integral multiple of 1 KB. Associativity is unlikely to be odd. Linesize is likely to be a power of two. The current implementation uses LB = 1 KB, UB = 16 MB, and an initial 1 KB increment that increases in steps as k grows.[2] It tests n for the values 2 and odd numbers from 3 to 33. It varies o over powers of two from `sizeof(void*)` to pagesize.

Limitations The gap test only works if it can detect the actual hardware boundary of the cache. We do not apply the gap test beyond L1 for several reasons. Higher levels of cache tend to be shared, either between I-cache and D-cache, or between cores, or both. Operating systems lock page table entries into higher-level caches. Higher levels of cache often use physical rather than virtual addresses. Each of these factors can cause the gap test to fail. It works on L1 precisely because L1 data caches are core-private and virtually mapped, and page tables are locked into L2 or L3 cache.

B. Cache-Only Test

The cache-only test avoids the weaknesses that the gap test exhibits for upper level caches by solving for cache capacity in isolation from associativity. It also isolates cache effects

[2]When the test samples the interval from 2^n to 2^{n-1}, it uses an increment of *max(1024,*2^{n-2}*)*. Thus, for $n \geq 12$, it tests 2^n, 2^{n+1}, and three points between, spaced 2^{n-2} bytes apart. For smaller n, it tests at 1 KB intervals.

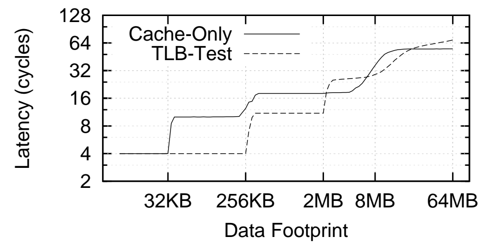


Fig. 3. Intel E5530 response, log-log plot

from TLB effects. It reuses the infrastructure from the gap test to run and time the cache-only reference string.

The cache-only reference string, $C(k)$, minimizes the impact of TLB misses. The parameter k specifies the reference string's memory footprint. The generator also uses the OS pagesize and an estimate of L1 linesize. In practice, L1 linesize is used to accentuate the system response by decreasing spatial locality, so any value greater than `sizeof(void*)` works.

Given k, the L1 linesize, and the OS pagesize, the generator builds an array of pointers that spans k bytes of memory. The generator constructs an index set, the column set, that covers one page and accesses one pointer in each line on the page. It constructs another index set, the row set, that contains the starting address of each page in the array. It shuffles both the column and row sets into random order.

To build the linked list, it iterates over the pages in the row set. Within a page, it links together the lines in the order specified by the column set. It links the last access in one page to the first access in the next page. If pagesize does not divide k, it generates a partial last row in random order. The last access then links back to the first, to create the circular list. Figure 2 shows the cache-only reference string without randomization; in practice, we randomize the order within each row and we randomize the order of the the rows.

To measure cache capacity, the test uses this reference string in a simple parameter sweep:

$$\textit{for } k \leftarrow LB \textit{ to } UB$$
$$t_k \leftarrow \textit{time for } C(k)$$

The implementation, of course, is more complex, as described in Section IV-A. The sweep produces a series of values, t_k, that form a piecewise linear function describing the processor's cache response.

The *cache only* line in Figure 3 shows the results of the cache-only test on an Intel E5530 Nehalem. Note the sharp transition for the L1 cache at 32 KB and the softer transitions for L2 and L3 caches. Our analysis reports an effective L2 capacity of 224 KB from this dataset. (See Table I.)

As long as pagesize is large relative to linesize, $C(k)$ produces clean results that isolate the cache behavior. To draw consistent conclusions from the data, however, requires the analytical techniques explained in Section V.

C. TLB Test

The TLB test uses a reference string that isolates TLB behavior from cache misses and runs it in the same infrastructure

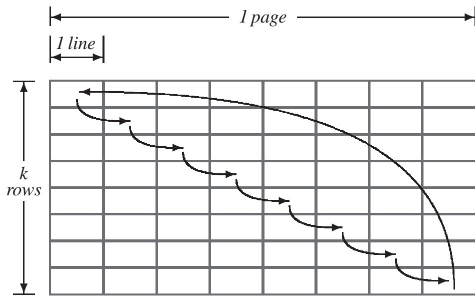


Fig. 4. TLB-Test Reference String

from the earlier tests. It produces a piecewise linear function that describes the processor's TLB response. Again, the data must be subjected to further analysis.

The TLB reference string, *T(n,k)*, accesses *n* pointers in each page of an array of *k* bytes. To construct *T(1,k)*, the generator builds a column index set and a row index set as in the cache only test. It shuffles both sets. To generate the permutation, it iterates over the row set choosing pages. It chooses a single line within the page by using successive lines from the column set, wrapping around in a modular fashion if necessary. The result is a string that accesses one line per page, and spreads the lines over the associative sets in the lower level caches. Figure 4 shows $T(1,k \cdot pagesize)$, without randomization.

For $n > 1$, the generator uses *n* lines per page, with a variable offset within the page to distribute the accesses across different sets in the caches and minimize associativity conflicts. The generator randomizes the full set of references, both to avoid the effects of a prefetcher and to avoid successive accesses to the same page.

The *TLB-Test* line in Figure 3 shows TLB test results for an Intel Nehalem E5530 processor. For the TLB data, the x-axis represents total footprint covered by the TLB, or $pages \times pagesize$. Notice the sharp transitions at 256 KB and 2 MB.

Eliminating False Positives The cache-only test hides the impact of TLB misses by amortizing those misses over many accesses. Unfortunately, the TLB test cannot completely hide the impact of cache because any action that amortizes cache misses also partially amortizes TLB misses. To see this, consider the log-log plot in Figure 5 which depicts the set of feasible memory-footprints that we can test. The x-axis shows the number of lines in a given footprint, while the y-axis shows the number of pages. Labeled dotted lines show boundaries of cache and TLB levels.

Consider the footprint of the cache-only string, *C(k)*, as *k* runs from one to large. *C(1)* generates the footprint (1,1) in the plot. *C(2)* generates (1,2), and so on. When *k* reaches pagesize ÷ linesize, it jumps from one page to two pages. *C(k)* forms a step function that degenerates to a line due to the log-log form of the plot. In contrast, the TLB string, *T(1,k)*, has a footprint that rises diagonally, at one page per line.

The plot predicts points where performance might change. When the line for a given reference string crosses a cache or TLB boundary in the memory hierarchy, performance may jump. With *C(k)*, we see a jump when it crosses cache

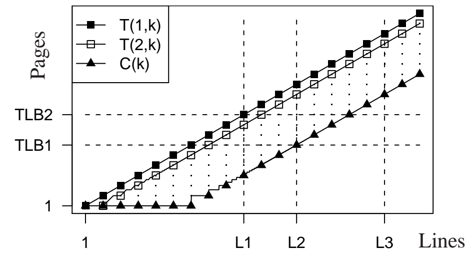


Fig. 5. Memory Hierarchy Search Space

boundaries but not when it crosses TLB boundaries—precisely because the order of access amortizes the TLB misses. Of course, if the hardware responds with a rise in access time before the actual boundary, the test shows that point as the effective boundary.

When the TLB line crosses a cache boundary, the rise in measured time is indistinguishable from the response to a TLB. The plot, however, gives us an insight that allows us to rule out false positive results. The line for *T(2,k)* parallels the line for *T(1,k)*, but is shifted to the right. If *T(1,k)* shows a TLB response at *x* pages, the *T(2,k)* shows a TLB response at *x* pages. Because *T(2,k)* uses twice as many lines at *x* pages as *T(1,k)*, a false positive response caused by the cache in *T(1,k)* will appear at a smaller size in *T(2,k)*.

To detect false positives, the TLB test runs both the *T(1,k)* and *T(2,k)* strings. It analyzes both sets of results, which produces two lists of suspect points in ascending order by *k*. If *T(1,k)* shows a rise at *x* pages, but *T(2,k)* does not, then *x* is a false positive. If both *T(1,k)* and *T(2,k)* show a rise at *x* pages, we report the transition as a TLB size. This technique eliminates most false positive results.

Still, a worst-case choice of cache and TLB sizes can fool this test. If *T(1,k)* maps into *m* cache lines at *x* pages, and *T(2,k)* maps into $2 \cdot m$ cache lines at *x* pages, and the processor has caches with *m* and $2 \cdot m$ lines, both reference strings will discover a suspect point at *x* pages and the current analysis will report a TLB boundary at *x* pages. Using more tests, e.g., *T(3,k)*, *T(4,k)*, and *T(5,k)*, could eliminate these points. In practice, we have not encountered this problem.

D. Linesize

The linesize test operates on a different paradigm than the cache-only test and the TLB test. It cannot rely on effects from associativity, as did the gap test, for two reasons. First, as the response curves from the cache-only test show, the micro-benchmark may not be able to use the full cache; using a smaller footprint will fail to trigger the predictable associativity effects. Second, higher level caches may be physically mapped, which also disrupts the associativity behavior. Thus, the linesize test relies on spatial locality and conflict misses.

The test generates a reference string *L(n,s)*, where *n* is the measured cache capacity and *s* is the stripe, or linesize, to test. For each cache level of size *n* the test performs a parameter sweep over *L(n,s)* for $sizeof(void*) \leq s \leq pagesize \div 2$. To save time we limit *s* to values that are powers of two, but the test works for any *s* within the given bounds.

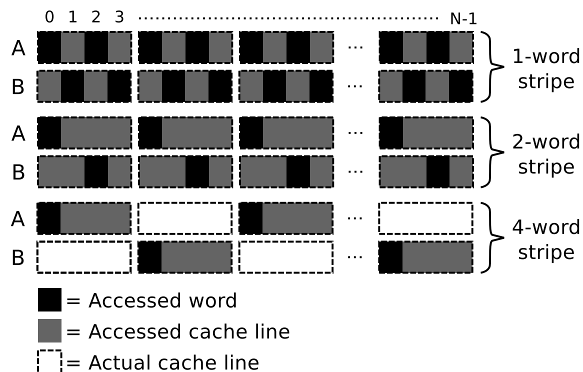


Fig. 6. Linesize micro-benchmark access pattern

L(n,s) generates two complementary striped access patterns, A and B, depicted in Figure 6. Pattern A accesses the first location in each of the even numbered stripes while pattern B accesses the first location in each of the odd numbered stripes. The value of s determines the width of each stripe. Both patterns are constructed to span the entire measured cache capacity, so the combined span is twice the measured cache capacity. But, because each pattern only accesses half of the stripes, the total data footprint is no larger than the cache capacity. The test accesses every location in pattern A followed by every location in B, repeating until sufficient timing granularity has elapsed. The accesses within each pattern are shuffled to defeat a prefetcher.

When patterns A and B both map to the same cache lines, they conflict. For $s < linesize$, each access generates a miss because both A and B access every line. Since the combined patterns span twice the measured cache capacity, the test accesses twice the number of lines in the cache. Once s reaches an integral multiple of the linesize, patterns A and B no longer conflict. Intuitively, each pattern has empty "holes" into which the other pattern fits. The test starts with a small value of s and increases it until A and B do not conflict, at which point the time to run the reference string drops dramatically.

Consider the one-word stripe at the top of Figure 6. Since the linesize in this example is four words, A and B conflict. The test uses the latency measured with the one-word stripe as its baseline. With $s = 2$, A and B still conflict, but spatial locality decreases and run time increases. With $s = 4$, A and B map to different lines, so conflict misses disappear completely and the time to run the reference string drops dramatically.

The analysis portion of this test is straightforward. Measured latency increases relative to the baseline as s increases due to the decrease in spatial locality. As soon as the stripe width is large enough to prevent conflict misses, measured latency drops below the baseline. The effective linesize, then, is equal to the s for which the latency of *L(n,s)* is less than the latency of the baseline, *L(n,sizeof(void*))*. Of course, a system with linesize equal to wordsize would produce the same response for all values of s. We have not encountered such a system.

For the linesize test to function properly both patterns A and B must map to the same cache lines. On a virtually mapped cache we can just create two adjacent arrays for A and B, both of length n. However, physically mapped caches do not guarantee that the arrays map contiguously into the cache. Our key insight is that physically mapped caches provide contiguous mapping *within* each page.

To leverage this observation, the test generate the access patterns at a *pagesize* granularity. It allocates *2*n/pagesize* pages and randomly fills half of them with pattern A and half with pattern B. Because the reference string spans twice as many pages as should fit in cache, on average *2*A* pages will map to each cache set, where A is the cache associativity.

Two competing pages can occupy the cache simultaneously if and only if: (1) one page contains pattern A and other page contains pattern B and (2) the stripe width is an integral multiple of the effective linesize. Otherwise, the two pages conflict with each another. (Note that it suffices to have some, but not all, pages meet condition (1), because avoiding some conflict misses will decrease the time below the baseline time.)

We cannot, in a portable way, control the page mapping. We can, however, draw random samples from a large set of pages and mappings to look for these conditions. The methodology that we developed to run a reference string achieves this effect. If $s < linesize$, then condition (2) never holds and the measured latency remains high. For $s = linesize$ (or an integral multiple of linesize), condition (2) always holds and condition (1) holds in some random samples. If the value of *Trials* is large enough, say 100, the test will find the desired mapping in some of its samples, which will produce the predicted decrease in runtime. In effect, our timing methodology samples over many possible virtual to physical mappings. Because it keeps the minimum time, it finds large enough effects for the analysis to recognize the linesize effect.

E. Associativity

Following X-RAY, our gap test detects associativity in the L1 cache, provided that it is virtually mapped [11]. The X-RAY paper suggests the use of superpages to test associativity in higher cache levels. Because superpage support is not yet portable, we did not follow that path.

With effective sizes smaller than hardware limits and physical address mappings, it is not clear that the compiler can rely on associativity effects in caches at the L2 and higher level. Thus, we do not measure associativity for caches above L1.

We have developed a straightforward test for TLB associativity based on the gap test. It functions well in most cases, but an architect can fool it. The ARM 926EJ-S has a two-part TLB with an 8-page, fully-associative TLB and a 56-page, 2-way set associative TLB. A TLB lookup first consults the small TLB; a miss in the small TLB faults to the larger TLB. The TLB test finds both the 8-page and the 56-page TLB. The associativity test reports that both TLBs are 8-way set associative; we have not been able to devise a reference string that exposes the 2-way associativity in the larger TLB.[3]

[3]The fact that we cannot, in portable C code, discover the associativity suggests that the architects made a good decision. They used a smaller and presumably cheaper associativity precisely in a place where the compiler could neither see nor use the larger associativity.

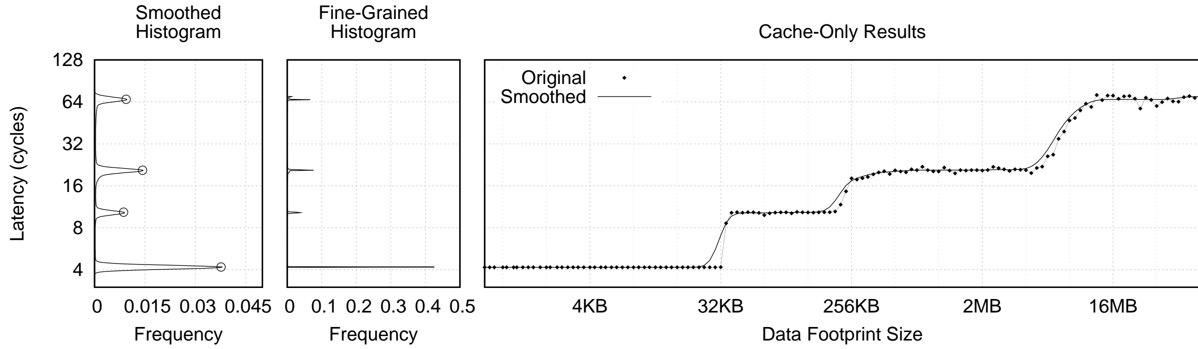


Fig. 7. Histogram Analysis for Intel Xeon E5530 Nehalem, Cache-Only Reference Stream

V. Automatic Analysis

The cache-only and TLB only micro-benchmarks produce piecewise linear functions that describe the processor's response, as shown in Figure 3. The tools use a multi-step analysis to derive consistent and accurate capacities from that data. The analysis derives two key pieces of information from a dataset: (1) the number of levels of cache or TLB and (2) the transition point between each level (*i.e.*, the capacity of each level). The discussion uses data from the cache-only test in its examples. The same analysis is used on the TLB test data.

The analysis is *automatic*; it needs no human intervention. Manual interpretation of the data is complex and subjective. The analysis uses mathematical optimization to find answers.

The analysis is *conservative*. In the presence of ambiguous results, it favors an underestimate rather than an overestimate, which might cause over-utilization of the cache.

The analysis is *robust*. Each step in the analysis has clear justification. It avoids arbitrary thresholds. Although we cannot prove that it draws perfect conclusions in the presence of noisy data, our thorough testing and analytical justifications increase our confidence that it will at least produce reasonable answers. It holds up experimentally (see § VI).

The following sections describe the three steps of our analysis: (1) filtering noise, (2) determining the number of levels and (3) determining the capacity of each level.

A. Filtering Timing Noise

Timing error is a major obstacle to correctly interpreting the micro-benchmark results. We cannot request single-user or real-time execution in a portable way; thus, the timing results are likely to reflect transient events of the OS or daemon processes. Our tools use a two pronged approach to minimize timing error: we reduce such errors during collection and we filter the data after collection to remove any remaining noise.

Our timing methodology, introduced in Section IV-A, provides the first-line defense against timing error. The tests perform multiple trials for each value in the parameter sweep, but only keep the smallest time. To prevent transient system events from affecting multiple trials of the same parameter value, we sweep across the entire parameter space before repeating for the next trial. Thus, any anomaly is spread across one trial at several parameter values rather than multiple trials at the same value. The test tries each parameter value until it finds *Trials* consecutive attempts with no decrease in the minimum value for that point; typically, $Trials = 100$. This adaptive approach collects more samples when the timing results are unstable and fewer samples when the results are consistent. It always collects at least *Trials* samples per point.

The first step in analysis filters the data to remove noise. Our filtering scheme leverages two observations. First, we assume that cache latency is an integral number of cycles, so we divide the empirical latency by the measured latency of an integer add and round to the nearest integer. For the sizes that fit in a cache, all accesses should be hits and should, therefore, take an integral number of cycles. For sizes that include some misses, the total latency is a mix of hits and misses. Rounding to cycles in these transitional regions produces a slight inaccuracy, but one that has minimal impact. As the data approaches the next cache boundary, all the references are misses in the lower level cache and the latency is, once again, accurate.

Second, we assume that the empirical results approximate an *isotonic*, or non-decreasing latency curve. We don't expect the latency to decrease when data footprint increases. Sometimes, the empirical results contain non-isotonic data points. We correct these anomalies with *isotone regression*, which removes decreasing regions from a curve with a form of weighted averaging. We use the Pool Adjacent Violators Algorithm [12].

B. Determining the Number of Cache Levels

Next, the analysis determines the number of levels in the cache hierarchy. Because this step only determines the rough global structure of the curve, it can use aggressive smoothing techniques, as long as they preserve the curve's important features. The third step, finding transition points, cannot use such aggressive smoothing as it may blur the transitions.

First, the analysis smoothes the curve with a Gaussian filter. The filter eliminates noise while preserving the curve's global shape. It uses a filter window whose width is derived from the minimum distance that we expect between two cache levels. We assume that each cache level should be at least twice as large as the previous level; on a log_2 scale the appropriate window width is $log_2(2) = 1$. With this window, the filter aggressively smoothes out noise between cache levels. It cannot filter out an actual level unless it is less than twice the

size of the previous level. The smoothed curve in the rightmost graph in Figure 7 shows the results of a Gaussian filter applied to the cache-only data points in Figure 3.

Next, the analysis identifies regions in the curve that correspond to levels in the cache. Informally, we expect to find relatively flat regions of the curve that are surrounded by sloped regions. To detect such regions, the analysis computes a one-dimensional density estimate along the y-axis, using a fine-grained histogram. It splits the y-axis into a large number of adjacent bins and computes the number of points that fall in the y-range of each bin. Intuitively, the bins for flat regions have much larger counts than bins for sloped regions. Thus, a cache levels is marked by a region of high density surrounded by regions of low density.

The fine-grained histogram, shown rotated sideways in Figure 7, provides a rough indication of the desired information. Further smoothing with a Gaussian filter clarifies the region structure. The analysis derives the filter window width from the minimum expected magnitude of a transition between regions—that is, the minimum relative cost of a cache miss. We assume that a cache miss incurs at least a 25% performance penalty; this step of the analysis considers anything less to be insignificant. That assumption implies that the window width, on a log_2 scale, should be $log_2(1.25) \approx 0.322$.

With this filter window width, the Gaussian filter consolidates the adjacent bins and produces a smooth curve with clear maxima and minima. The leftmost graph in Figure 7 depicts the smoothed histogram. The final step counts the number of local maxima in the curve by computing the slope of the smoothed histogram. Local maxima correspond to points where the first derivative changes from non-negative to negative. This simple algorithm detects the peaks in the histogram, indicated by the circles on the peaks of the smoothed histogram. Each peak corresponds to a distinct level in the memory hierarchy. If the analysis finds n peaks, that indicates $n-1$ levels of cache, plus main memory. This step concludes by returning the number of levels in the cache.

C. Determining the Size of the Cache Levels

The final analysis step finds the transition points between levels in the curve—the points where latency begins to rise because the cache is effectively full. This section presents an intuitive algorithm to find objectively the optimal points to split the curve, given the number of levels in the cache.

Interpreting the cache-latency curve is somewhat subjective, as it entails a judgment call with regard to the capacity/latency tradeoff. The ideal curve would resemble a step function, with long, flat regions connected by short steep transitions. On such a curve, cache capacity is easily determined as the final point before the rise in latency. However, modern processors show soft response curves that rise well before the hardware cache boundary, at least on the higher levels of cache. Some previous approaches try to estimate hardware cache capacity from the shape of the latency curve. In contrast, our analysis finds a number that makes sense for compiler-based blocking of memory accesses. That number, the *effective cache capacity*, corresponds to the point at which access latency starts to rise.

The analysis identifies the largest point in a flat region of the curve. Unfortunately, "flat" is subjective if the transition begins with a gradual slope. Thus, the analysis uses an objective function that selects for points that occur early in the transition. It models a step-function that steps upward at the transition point between two levels. The number of steps should match the number of levels found by the second step in the analysis. Thus, the analysis tries to minimize error between a step-function approximation and the original (unsmoothed) data.

The analysis employs a dynamic programming algorithm, based on extending Perez's polygonal approximation algorithm [13] to a step-function approximation. While the complexity of this algorithm is $\Theta(MN^3)$, where M is the number of levels cache and N is the number of data points, the running time is not a practical problem. The values for M and N are small and the total cost of analysis is insignificant relative to the cost of gathering the data.

Figure 8 shows the result of the step-function approximation on the original data. Smoothing would alter the transition points. The first three steps represent the L1, L2, and L3 caches. The right endpoint of a step indicates that level's capacity. The height of a step indicates its worst-case latency. Although the L2 and L3 transitions are gradual in the data, the approximation conservatively identifies the start of the slope as the effective cache size. A more gentle slope might cause the algorithm to select a larger effective size with a slightly longer latency. The transition points are chosen to minimize the error of the step-function approximation. The rightmost step in the approximation corresponds to main memory and indicates the miss penalty for the L3 cache.

VI. Experimental Validation

To validate our techniques, we run them on a collection of systems that range from commodity X86 processors through an IBM POWER7, an ARM, and the IBM Cell processor in a Sony Playstation 3. All of these systems run some flavor of Unix and support enough of the Posix interface for our tools.

Table I shows the measured cache parameters: linesize, associativity, capacity, and latency for each level of cache that the tools detect. The *Measured* column shows the numbers produced by the tools. Capacities were produced by the cache-only test; the gap test agrees with it on each system we have tested. A blank in the *Measured* column means that the tools

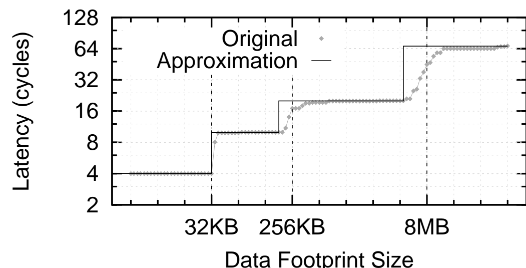


Fig. 8. Step-Function Approximation for Intel Xeon E5530 Nehalem

Processor		Linesize in Bytes		Associativity		Capacity in KB		Latency in Cycles
		Actual	*Measured*	*Actual*	*Measured*	*Actual*	*Measured*	*Measured*
AMD Opteron 2360 SE Barcelona	1	64	64	2	2	64	64	3
	2	64	64	16		512	448	12
	3	64	64	32		2048	1792	46
AMD Opteron 275	1	64	64	2	2	64	64	3
	2	64	64	16		1024	896	17
AMD Opteron 6168 Magny-Cours	1	64	64	2	2	64	64	3
	2	64	64			512	512	13
	3	64	64			12288	5120	32
AMD Phenom 9750 Agena	1	64	64	2	2	64	64	3
	2	64	64	16		512	448	12
	3	64	64	32		2048	2048	31
ARM926EJ-S	1	32	32	4	4	16	16	2
	2	32	32	?		256	224	15
IBM Cell (PS3)	1	128	128	?	4	32	32	2
	2	128	128	?		512	320	20
IBM POWER7	1	128	128	8	8	32	32	1
	2	128	128	8		256	256	6
	3	128	256	?		32768	3072	15
	4		256				20480	51
Intel Core 2 Duo T5600 Merom	1	64	64	8	8	32	32	3
	2	64	128	8		2048	1280	14
Intel Itanium 2 900 McKinley	1	64	64	4	4	16	16	2
	2	128	128			256	256	6
	3	128	128			1536	1024	18
Intel Itanium 2 9040 Montecito	1	64	64	4	4	16	16	2
	2	128	128	8		256	256	6
	3	128	128	12		12288	4096	11
Intel Pentium 4	1	64	64	4	4	8	8	4
	2	64	128			512	256	36
Intel Xeon E5420 Harpertown	1	64	64	8	8	32	32	3
	2	64	128	24		6144	4096	15
Intel Xeon E5440 Harpertown	1	64	64	8	8	32	32	3
	2	64	64	24		6144	4096	15
Intel Xeon E5530 Nehalem	1	64	64	8	8	32	32	4
	2	64	64	8		256	224	10
	3	64	64	16		8192	5120	19
Intel Xeon E7330 Tigerton	1	64	64	8	8	32	32	3
	2	64	128	12		3072	1792	14
Intel Xeon X3220 Kentsfield	1	64	64	8	8	32	32	3
	2	64	64			4096	2560	15
Intel Xeon X5660 Westmere	1	64	64	8	8	32	32	4
	2	64	64	8		256	224	10
	3	64	64	16		12288	8192	22
PowerPC 7455 G4	1	32	32	8	8	32	32	3
	2	64	64	8		256	224	10
	3	128	128	8		2048	1536	32
PowerPC 750 G3	1	32	32	8	8	32	32	2
	2	128	128	2		1024	512	20
Sun UltraSPARC T1	1	16	16	4	4	8	8	4
	2	64	64	12		3072	3072	23

TABLE I
CACHE RESULTS

do not measure that value (e.g., L2 cache associativity). The *Actual* column lists the documented number for that processor, if available. Table II shows the capacity numbers for TLBs on the same systems. We do not show pagesize numbers in the table; they are available from the POSIX `sysconf` call.

The tables are produced by a script that distributes the code, uses `make` to compile and execute it, and retrieves the results. Two of the systems use batch queues; those systems require manual intervention to schedule the job and retrieve the results.

A couple of entries deserve specific attention. The POWER7 has an unusual L3 cache structure. Eight cores share a 32 MB L3 cache; each core has a 4 MB portion of that cache that it can access faster than the remaining 28 MB. The cache-only test discovers two distinct latencies: a 3 MB cache with a 15 cycle latency and a larger 20 MB cache with a 51 cycle latency. Our tests were run on an active system; the effective sizes reflect the actual behavior that a program might see. A compiler that blocks for POWER7 caches would do better to use the tool's

Processor		Capacity in KB	
		Actual	*Measured*
AMD Opteron 2360 SE Barcelona	1	192	192
	2	2048	2048
AMD Opteron 275	1	128	128
	2	2048	2048
AMD Opteron 6168 Magny-Cours	1	192	192
	2	2048	2048
AMD Phenom 9750 Agena	1	192	192
	2	2048	2048
ARM926EJ-S	1	256	32
	2		224
IBM Cell (PS3)	1	?	256
	2	?	4096
IBM POWER7	1	4096	4096
	2	?	32768
Intel Core 2 Duo T5600 Merom	1	64	64
	2	1024	1024
Intel Itanium 2 900 McKinley	1	2048	7680
	2	8192	
Intel Itanium 2 9040 Montecito	1	512	1920
	2	2048	
Intel Pentium 4	1	256	256
Intel Xeon E5420 Harpertown	1	64	64
	2	1024	1024
Intel Xeon E5440 Harpertown	1	64	64
	2	1024	1024
Intel Xeon E5530 Nehalem	1	256	256
	2	2048	2048
Intel Xeon E7330 Tigerton	1	64	64
	2	1024	1024
Intel Xeon X3220 Kentsfield	1	64	64
	2	1024	1024
Intel Xeon X5660 Westmere	1	256	256
	2	2048	2048
PowerPC 7455 G4	1	512	512
	2		1280
PowerPC 750 G3	1	512	512
	2		1280
Sun UltraSPARC T1	1	512	3840

TABLE II
TLB RESULTS

description than to treat it as a unified 32 MB L3 cache.

As discussed in § IV-E, the TLB on the ARM 926EJ-S generates a result that differs from the hardware description. Again, a compiler would do well to use the tools' result rather than the description from the manuals.

Several of our systems have cache designs that use different linesizes for different levels of cache. The Itanium, POWERPC G3, and Sun T1 all use a smaller linesize for L1 and a larger linesize for higher levels of the cache. The POWERPC G4 has a different linesize for each level of cache. The linesize test detects the correct linesize in each case. On the POWER7, Intel T5600, Pentium 4, Intel E540, and the Intel E7330, the tools detect a larger *effective* linesize for the last level of cache. While it is possible that the documentation is incorrect, it seems more likely that the test exposes behavior of the hardware prefetcher or the memory controller. Again, these examples reinforce the need to determine such parameters experimentally rather than rely on documentation.

Effective Cache Sizes The tests measure effective cache size rather than the actual cache size. The discovered effective size is typically smaller than actual size. For L2 cache and beyond, effective size can be as small as 50–75% of actual size. For L1 caches, effective size matched actual size on each system.

VII. CONCLUSION

This paper presents techniques to measure the effective sizes of levels in a processor's cache and TLB hierarchy. The tools are portable; they rely on a C compiler and the POSIX OS interfaces. The tools discover effective cache and TLB sizes that are suitable for use in memory-hierarchy optimizations; in fact, these effective numbers should provide better optimization results than would be obtained using the actual hardware values from the manufacturer's manual. The tools will be available in open source form (before ISPASS).

We are pursuing two extensions of this work. The first will use the micro-benchmarks described in this paper to measure effective capacity when other cores are loaded. The experiments will run a known memory load on all but one core, while measuring cache size on the final core. The second project will explore in more detail the reasons for the discrepancy between effective and physical cache sizes.

REFERENCES

[1] C.-K. Luk and T. C. Mowry, "Architectural and compiler support for effective instruction prefetching: a cooperative approach," *ACM Trans. Comput. Syst.*, vol. 19, no. 1, pp. 71–109, 2001.

[2] S. A. Moyer, "Performance of the IPSC/860 Node Architecture," University of Virginia, Charlottesville, VA, USA, Tech. Rep., 1991.

[3] A. Qasem and K. Kennedy, "Profitable loop fusion and tiling using model-driven empirical search," in *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2006, pp. 249–258.

[4] J. Dongarra, S. Moore, P. Mucci, K. Seymour, and H. You, "Accurate cache and tlb characterization using hardware counters," in *Proceedings of the International Conference on Computational Science (ICCS)*, 2004, pp. 432–439.

[5] A. X. Duchateau, A. Sidelnik, M. J. Garzarán, and D. Padua, "P-ray: A software suite for multi-core architecture characterization," *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, pp. 187–201, 2008.

[6] J. González-Domínguez, G. L. Taboada, B. B. Fraguela, M. J. Martín, and J. Touriño, "Servet: A benchmark suite for autotuning on multicore clusters," in *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, Atlanta, GA, USA, April 2010.

[7] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, California, January 1996.

[8] R. H. Saavedra and A. J. Smith, "Measuring cache and tlb performance and their effect on benchmark runtimes," *IEEE Trans. Comput.*, vol. 44, no. 10, pp. 1223–1235, 1995.

[9] K. Yotov, K. Pingali, and P. Stodghill, "X-ray: A tool for automatic measurement of hardware parameters," in *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, Washington, DC, USA, 2005, p. 168.

[10] "*Omitted for blind review*," *Reference will appear in full paper.*

[11] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 181–192, 2005.

[12] T. Robertson, F. Wright, and R. Dykstra, *Order Restricted Statistical Inference*. John Wiley @ Sons Ltd., 1988.

[13] J.-C. Perez and E. Vidal, "Optimum polygonal approximation of digitized curves," *Pattern Recogn. Lett.*, vol. 15, no. 8, pp. 743–750, 1994.