Efficient Characterization of Hidden Processor Memory Hierarchies

Keith Cooper[0000−0003−4288−4847] and Xiaoran Xu[0000−0003−4450−8757]

Rice University, Houston TX 77005, USA
{keith, xiaoran.xu}@rice.edu

Abstract. A processor's memory hierarchy has a major impact on the performance of running code. However, computing platforms, where the actual hardware characteristics are hidden from both the end user and the tools that mediate execution, such as a compiler, a JIT and a runtime system, are used more and more, for example, performing large scale computation in cloud and cluster. Even worse, in such environments, a single computation may use a collection of processors with dissimilar characteristics. Ignorance of the performance-critical parameters of the underlying system makes it difficult to improve performance by optimizing the code or adjusting runtime-system behaviors; it also makes application performance harder to understand.
To address this problem, we have developed a suite of portable tools that can efficiently derive many of the parameters of processor memory hierarchies, such as levels, *effective capacity* and latency of caches and TLBs, in a matter of seconds. The tools use a series of carefully considered experiments to produce and analyze cache response curves automatically. The tools are inexpensive enough to be used in a variety of contexts that may include install time, compile time or runtime adaption, or performance understanding tools.

Keywords: Efficient Characterization · Hidden Memory Hierarchies · Code Performance · Portable Tool

1 Introduction and Motivation

Application performance on modern multi-core processors depends heavily on the performance of the system's underlying memory hierarchy. The academic community has a history of developing techniques to measure various parameters on memory hierarchies [1–14]. However, as deep, complex, and shared structures have been introduced into memory hierarchies, it has become more difficult to find accurate and detailed information of performance-related characteristics of those hierarchies. What's more, to an increasing degree, large-scale computations are performed on platforms where the actual characteristics of the underlying hardware are hidden. Knowledge of the performance-critical parameters of the underlying hardware can be useful for improving compiled code, adjusting runtime system behaviors, or understanding performance issues. Specifically, to achieve the best performance on such platforms, the code must be tailored to the detailed memory structure of the target processor. That structure varies widely across different architectures, even for models of the same instruction set architecture (ISA). Thus, performance can be limited by the compiler's ability to understand model-specific differences in the memory hierarchy, to tailor the program's behavior, and to adjust runtime behaviors accordingly. At present, few compilers attempt aggressive model-specific optimization. One impediment to development of such compilers is the difficulty of understanding the relevant performance parameters of the target processor. While manufacturers provide some methods to discover such information, such as Intel's CPUID, those mechanisms are not standardized across manufacturers or across architectures. A standard assumption is that such data can be found in manuals; in truth, details such as the latency of an L1 TLB miss on an Intel Core i7 processor are rarely listed. What is worse, even the listed information may differ from what a compiler really needs for code optimization, such as full hardware capacity vs. *effective capacity.*

Effective capacity is defined as 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. Several authors have advocated the use of effective capacities rather than physical capacities [15–17]. In the best case, *effective capacity* is equal to *physical capacity.* For example, on most microprocessors, L1 data cache's effective and physical capacity are identical, because it is not shared with other cores or instruction cache, and virtually mapped. In contrast, a higher level cache for the same architecture might be shared among cores; contain the images of all those cores' L1 instruction caches or hold page tables, locked into L2 or L3 by hardware that walks the page table. Each of these effects might reduce the *effective cache capacity* and modern commodity processors exhibit all three.

Contribution This paper presents a set of tools that measure, efficiently and empirically, the *effective capacity* and other parameters of the various levels in the data memory hierarchy, both cache and TLB; that are portable across a variety of systems; that include a robust automatic analysis; and that derive a full set of characteristics in a few seconds. The resulting tools are inexpensive enough to use in a variety of contexts that may include install time, compile time or runtime adaption, or performance understanding tools. Section 4 shows that our techniques produce results with the same accuracy as earlier work, while using a factor of **10x** to **250x** less time, which makes the tools inexpensive enough to be used in various contexts, especially lightweight runtime adaption.

2 Related Work

Many authors describe systems that attempt to characterize the memory hierarchy [1–5, 8], but from our perspective, previous systems suffer from several specific flaws: (1) The prior tools are not easily portable to current machines [1–3]. 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; multi-level caches and physical-address tags create complications that they were not designed to handle. On contrast, our tools characterize various modern processors using only portable C code and Posix calls. (2) Some previous tools solve multiple parameters at once [4, 5], which is not robust since if the code generates one wrong answer for one parameter, it inevitably causes the failure of all the other parameters. Noisy measurements and new hardware features, such as sharing or victim caches, can also cause these tests to produce inaccurate results. (3) Finally, the time cost of measuring a full set of characteristics by previous tools is very large, e.g. 2-8 minutes by Sandoval's tool [8] (See Section 4). At that cost, the set of reasonable applications for these measurements is limited. For these techniques to find practical use, the cost of measurement and analysis must be much lower.

The other related works have different focuses with ours. P-Ray [6] and Servet [7] characterized sharing and communication aspects of multi-core clusters. Taylor *et al.* [9] extended memory characterization techniques to AMD GPUs. Sussman *et al.* [10] arbitrated between different results produced from different benchmarks. Abel [11] measured physical capacities of caches. Servet 3.0 [12] improved Servet [7] by characterizing the network performance degradation. Casas *et al.* [13, 14] quantified applications' utilization of the memory hierarchy.

3 The Algorithms

This section describes three tests we developed that measure the levels, capacity and latency for cache and TLB, along with associativity and linesize for L1 cache. All of the tests rely on a standard C compiler and the Posix libraries for portability, and timings are taken based on `gettimeofday()`. All the three algorithms rely on a data structure *reference string*, implemented as a circular chain of pointers, to create a specific pattern of memory references. The reference strings are different for each test to expose different aspects of the memory hierarchy, but their construction, running and timing are shared as presented below.

3.1 Reference Strings

A reference string is simply a series of memory references—in this paper, they are all load-operations—that the test uses to elicit a desired response from the memory system. A reference string has a *footprint*, the amount of contiguous virtual address space that it covers. In general, the tests use a specific reference string, in a variety of footprints, to measure memory system response. By constructing, running different footprints and recording the times spent, the test builds up a response curve. Fig. 1 shows a typical response curve running on an Intel T9600.

Running a Reference String: The microbenchmarks depend on the fact that we can produce an accurate measurement of the time required to run a refer-

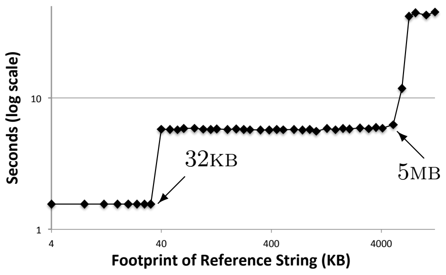


Fig. 1: Example Response Curve

```
$loads \leftarrow$ number of accesses
$start \leftarrow$ timer()
while $loads - - > 0$ do
    $p \leftarrow *p$
end while
$finish \leftarrow$ timer()
$elapsed \leftarrow finish - start$
```

Fig. 2: Running a Reference String

ence string, and that we can amortize out compulsory start-up cache misses. 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 instantiate the reference string as an array of pointers whose size equals the footprint of the string. Inside this array, the tools build a circular linked list of the locations. (In C, we use an array of *void* $**$.) The code to run the reference string is simple as shown in Fig. 2. The actual implementation runs the loop enough times to obtain stable results, where "enough" is scaled by the processor's measured speed. We chose the iteration count by experimentation where the error rate is 1% or less.

Timing a Reference String: To time a reference string, we insert calls to the the Posix *gettimeofday* routine around the loop as a set of calipers. We run multiple trials of each reference string and record the minimum measured execution time because outside interference will only manifest itself in longer execution times. Finally, we convert the measured times into "cycles", where a cycle is defined as the time of an integer add operation. This conversion eliminates the fractional cycles introduced by amortized compulsory misses and loop overhead.

The basic design and timing of reference strings are borrowed from Sandoval's work [8], since he already showed that these techniques produce accurate timing results across a broad variety of processor architectures and models. Our contribution lies in finding significantly more efficient ways to manipulate the reference strings to detect cache parameters. The L1 cache test provides a good example of how changing the use of the string can produce equivalent results with an order of magnitude less cost (see Section 4 for cost and accuracy results). Also, we significantly reduce the measurement time of data while keeping the same accuracy. We repeat each test until we have not seen the minimum time change in the last 25 runs. (The value of 25 was selected via a parameter sweep from 1 to 100. At 25, the error rate for the multi-cache test is 1% or less.)

3.2 L1 Cache

Because the L1 data-cache linesize can be used to reduce spatial locality in the multi-level cache test and the TLB test, we use an efficient, specialized test to discover the L1 cache parameters. Two properties of L1 caches make

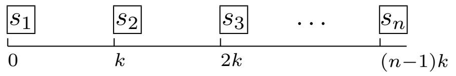
```
baseline ← time for G(2,LB/2,0)
1. for k ← LB/MaxAssoc to UB/MaxAssoc
     t ← time for G(MaxAssoc+1,k,0)
     if t > baseline
         L1Size = k * MaxAssoc
         break

2. for n ← MaxAssoc; n ≥ 1; n ← n/2
     t ← time for G(n+1,L1Size/n,0)
     if t ≤ baseline
         L1Assoc = n * 2
         break

3. for offset ← 1 to pagesize
     t ← time for G(L1Assoc+1,L1Size/L1Assoc,offset)
     if t ≤ baseline
         L1LineSize = offset
         break
```

Fig. 3: Pseudo Code for the L1 Cache Test

the test easy to derive and analyze: the lack of sharing at L1 and the use of virtual-address tags. The L1 test relies directly on hardware effects caused by the combination of capacity and associativity. We denote the L1 reference string as a tuple G(n,k,o), where n is the number of locations to access, k is the number of bytes between those locations (the "gap"), 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:



And *G(n,k,4)* would move the n^{th} location out another four bytes.

Both X-Ray [4] and Sandoval's [8] "gap" test use a similar reference string. However, they require to iterate k across the full range of cache sizes, and n from 1 to the actual associativity plus one. Our algorithm orders the tests in a different way that radically reduces the number of combinations of n and k that it must try. It starts with the maximum value of associativity, **MaxAssoc**, say 16, and sweeps over the gap size to find the first value that causes significant misses, as shown in the first loop in Fig. 3.

It sweeps the value of k from LB / MaxAssoc to UB / MaxAssoc, where **LB** and **UB** are the lower and upper bounds of the testing cache. For L1 cache, we choose 1KB and 4MB respectively. With $n = \text{MaxAssoc} + 1$ and $o = 0$, the last and the first location will always map to the same cache set location. None of them will miss until $k * MaxAssoc$ reaches the L1 cache size. Fig. 4 shows how the string G(33, 1KB, 0) maps into a 32KB, 8-way, set-associative cache. With these cache parameters, each way holds 4KB and all references in G(33, 1KB, 0) map into sets 0, 16, 32, and 48. The string completely fills those four cache sets, and overflows set 0. Thus, the 33^{rd} reference will always miss in the L1 cache.

The first loop will record uniform (cycle-rounded) times for all iterations, which is called the **baseline** time since all the references hit in L1 cache so far. Until it reaches G(33, 1KB, 0), at which time it will record a larger time due to the miss on the 33^{rd} element, the larger time causes it to record the cache size and exit the first loop.

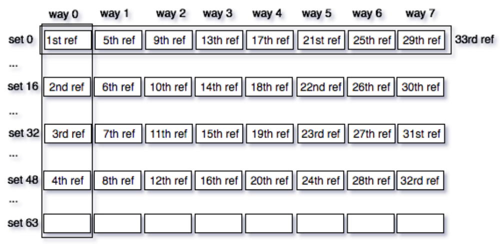


Fig. 4: Running G(33,1KB,0) on a 32KB, 8-Way, Set-Associative Cache

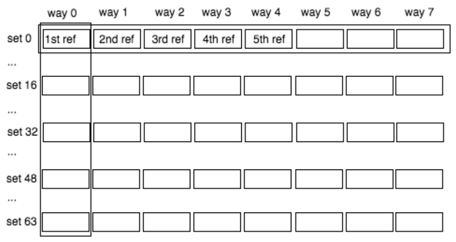


Fig. 5: Running G(5,8KB,0) on a 32KB, 8-Way, Set-Associative Cache

The second loop in Fig. 3 finds the associativity by looking at larger gaps (k) and smaller associativities (n). It sweeps over associativity from $\text{MaxAssoc} + 1$ to 2, decreasing n by a factor of 2 at each iteration. In a set-associative cache, the last location in all the reference strings will continue to miss in L1 cache until n is less than the actual associativity. In the example, a 32KB, 8 way L1 cache, that occurs when n is $\text{L1Assoc} / 2$, namely 4, as shown in Fig. 5. At this point, the locations in the reference string all map to the same set and, because there are more ways in the set than references now, the last reference will hit in cache and the time will again match the baseline time.

At this point, the algorithm knows the L1 size and associativity. So the third loop runs a parameter sweep on the value of o, from 1 to pagesize (in words). When o reaches the L1 linesize, the final reference in the string maps to the next set and the runtime for the reference string drops back to the original baseline—the value when all references hit in cache.

3.3 Multilevel Caches

The L1 cache test relies on the fact that it can precisely detect the actual hardware boundary of the cache. We cannot apply the same test to higher level caches for several reasons: higher level caches tend to be shared, either between instruction and data cache, or between cores, or both; higher level caches tend to use physical-address tags rather than virtual ones; operating systems tend to lock page table entries into one of the higher level caches. Each of these factors, individually, can cause the L1 cache test to fail. It works on L1 precisely because L1 data caches are core-private and virtually mapped, without outside interference or page tables.

Our multi-level cache test avoids the weaknesses that the L1 cache test exhibits for upper level caches by detecting cache capacity in isolation from associativity. It uses a reference string designed to expose changes in cache response while isolating those effects from any TLB response. It reuses the infrastructure developed for the L1 cache test to run and time the cache reference string.

The multi-level cache test reference string $C(k)$ is constructed from a footprint k, the OS pagesize obtained from the *Posix* `sysconf()`, and the L1 cache linesize.[1] Given these values, the string 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 L1 cache line on the page. It constructs another index set, the row set, that contains the starting address of each page in the array. Fig. 6 shows the cache reference string without randomization; in practice, we randomize the order within each row and the order of the the rows to eliminate the impact of hardware prefetching schemes. Sweeping through the page in its randomized order before moving to another page, minimizes the impact of TLB misses on large footprint reference strings.

To measure cache capacity, the test could use this reference string in a simple parameter sweep, for the range from **LB** to **UB**. As we mentioned, **LB** and **UB** are the lower and upper bounds of the testing cache and we choose 1KB and 32MB respectively for the whole multi-level cache.

```
for $k \leftarrow$ Range(LB,UB)
    $t_k \leftarrow$ time for $C(k)$ reference string
```

The sweep produces a series of values, t_k, that form a piecewise linear function describing the processor's cache response. Recall that Fig. 1 shows the curve from the multi-level cache test on an Intel T9600. The T9600 featured a 32KB L1 cache and a 6MB L2 cache, but notice the sharp rise at 32KB and the softer rise that begins at 5MB. It's the effect of *effective capacity*.

The implementation of the algorithm, of course, is more complex. Section 3.1 described how to run and time a single reference string. Besides that, the pseudo code given inline above abstracts the choice of sample points into the notation Range(LB,UB). Instead of sampling the whole space uniformly, in both the multi-level cache and the TLB test, we actually only space the points uniformly below 4KB (we test 1, 2, 3, and 4KB). Above 4KB, we test each power of two, along with three points uniformly-spaced in between since sampling fewer points has a direct effect on running time. The pseudo code also abstracts the fact that the test actually makes repeated sweeps over the range from LB to UB. At each size, it constructs, runs, and times a reference string, updating the minimal time for that size, if necessary, and tracking the number of trials since the time was last lowered. Sweeping in this way distributes outside interference, say from an OS daemon or another process, across sizes, rather than concentrating it in a small number of sizes. Recreating the reference string at each size and trial allows the algorithm to sample different virtual-to-physical page mappings.

Knocking Out and Reviving Neighbors: Most of the time spent in the multi-level cache test is incurred by running reference strings. The discipline of running each size until its minimum time is "stable"—defined as not changing in the last 25 runs, means that the test runs enough reference strings. (As we mentioned, the value of 25 was selected via a parameter sweep from 1 to 100 and at 25, the error rate for the multi-cache test is 1% or less.)

[1] In practice, the L1 linesize is used to accentuate the system response by decreasing spatial locality. Any value greater than $sizeof(void*)$ works, but a value greater than or equal to linesize works better.

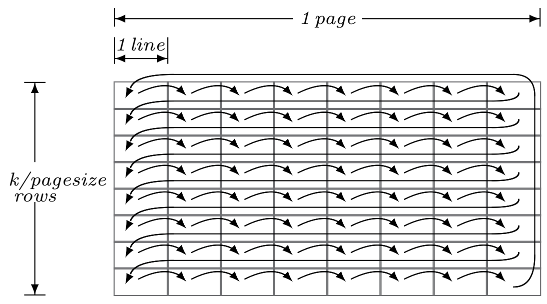


Fig. 6: Cache Test Reference String

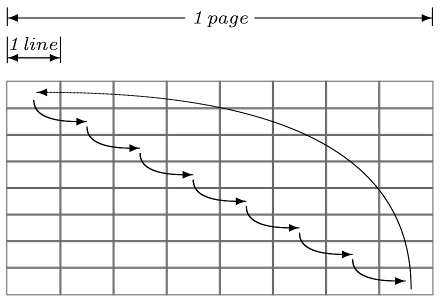


Fig. 7: TLB Test Reference String

In Fig. 1, points that fall in the middle of a level of the memory hierarchy have, as might be expected, similar heights in the graph, indicating similar average latencies. Examining the sweep-by-sweep results, we realized that values in those plateaus quickly reach their minimum values. This is another kind of "stability" of data. To capitalize on this effect, we added a mechanism to knock values out of the testing range when they agree with the values at neighboring sample sizes. As shown in Fig. 8, after every sweep of the reference strings, the knockout phase examines every sample size k in the Range(LB, UB). If t_k, the time of running $C(k)$ string, equals both t_{k-1} and t_{k+1}, then it cautiously asserts k is a redundant sample size and can be knocked out. (It sets the counter for that size that tracks iterations since a new minimum to zero.) In the next sweep of Range(LB, UB), these sample sizes will be omitted.

The knock-out mechanism can eliminate a sample size too soon, e.g. a sample size k and its neighbors have the same inaccurate value. When this occurs, the knockout test may eliminate out k prematurely. To cope with this situation, we added a revival phase. When any point reaches a new minimum, if it has a neighbor that was previously knocked out, it revives that neighbor so that it is again measured in the next sweep of Range(LB, UB).

The knockout-revival optimization significantly reduced the cost of the multi-level cache test, from minutes in earlier work to seconds, as details shown in Section 4.

3.4 TLB Test

The TLB test closely resembles the multi-level cache test, except that it uses a reference string that isolates TLB behavior from cache misses. It utilizes the same infrastructure to run reference strings and incorporates the same discipline for sweeping a range of TLB sizes to produce a response curve. It benefits significantly from the knockout-revival mechanism.

The TLB reference string, $T(n, k)$, accesses n pointers in each page of an array with a footprint of k bytes. To construct $T(1, k)$, the generator builds a column index set and a row index set as in the multi-level cache 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

```
while not all $t_k$ are stable do
    for $k \leftarrow$ Range(LB,UB)
        $t_k \leftarrow$ time for $C(k)$ reference string
        if $t_k$ is a new minimum && $k$'s neighbors
        have been knocked out
            revive $k$'s neighbors to Range(LB,UB)
    for $k \leftarrow$ Range(LB,UB)
        if $t_k == t_{k's\ neighbors}$
            knock out $k$ from Range(LB, UB)
```

Fig. 8: Pseudo Code for Multi-Level Cache Test

```
for $k \leftarrow$ LB to UB
    $t_{1,k} \leftarrow$ time for $T(1,k)$
    if there's a jump from $t_{1,k-1}$ to $t_{1,k}$
        for $n \leftarrow$ 2, 3, 4
            $t_{n,k-1} \leftarrow$ time for T(n,k-1)
            $t_{n,k} \leftarrow$ time for T(n,k)
        if there's a jump from $t_{n,k-1}$ to $t_{n,k}$
        when n=2,3,4
            report a TLB size
```

Fig. 9: Pseudo Code for the TLB Test

string that accesses one line per page, and spreads the lines over the associative sets in the lower level caches. The effect is to maximize the page footprint, while minimizing the cache footprint. Fig. 7 shows $T(1,k)$ 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, to avoid the effects of a prefetcher and to successive accesses to the same page.

The multi-level cache 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 misses because any action that amortizes cache misses also partially amortizes TLB misses. When the TLB line crosses a cache boundary, the rise in measured time is indistinguishable from the response to a TLB boundary. However, we could rule out false positives by running $T(2,k)$ reference string and following the rule that if $T(1,k)$ shows a TLB response at x pages, then $T(2,k)$ should show a TLB response at x pages too if x pages is indeed a boundary of TLB. Because $T(2,k)$ uses twice as many lines at x pages as $T(1,k)$, if it's a false positive response caused by the cache boundary in $T(1,k)$, it will appear at a smaller size in $T(2,k)$.

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 $2*m$ cache lines at x pages, and the processor has cache boundaries at m and $2*m$ lines, both reference strings will discover a suspect point at x pages and the current analysis will report a TLB boundary at x pages even if it's not. Using more tests, e.g., $T(3,k)$ and $T(4,k)$, could eliminate these false positive points.

Sandoval's test [8] ran the higher line-count TLB strings, $T(2,k)$, $T(3,k)$, and $T(4,k)$ exhaustively. We observe that the values for those higher line-count tests are only of interest at points where the code observes a transition in the response curve. Thus, our TLB test runs a series of $T(1,k)$ strings for k from LB to UB. From this data, it identifies potential transitions in the TLB response graph, called "suspect" points. Then it examines the responses of the higher line-count tests at the suspect point and its immediate neighbors as shown in Fig. 9. If the test detects a rise in the $T(1,k)$ response at x pages, but that response is not confirmed by one or more of $T(2,k)$, $T(3,k)$, or $T(4,k)$, then it reports x as a false positive. If all of $T(1,k)$, $T(2,k)$, $T(3,k)$, $T(4,k)$ show a rise at x pages, it reports that transition as a TLB boundary. This technique eliminates almost

Processor	Capacity(KB)	LineSize (B)	Associativity	Latency Cycle	Cost Secs
Intel Core i3	32	64	8	3	0.49
Intel Core i7	32	64	8	5	0.56
Intel Xeon E5-2640	32	64	8	4	0.49
Intel Xeon E5420	32	64	8	4	0.54
Intel Xeon X3220	32	64	8	3	0.51
Intel Xeon E7330	32	64	8	3	0.52
Intel Core2 T7200	32	64	8	3	0.55
Intel Pentium 4	8	64	4	4	1.71
ARM Cortex A9	32	32	7	4	7.98

Table 1: L1 Cache Results

all false positive results in practice since the situation that all m, $2m$, $3m$, $4m$ cache lines are cache boundaries is extremely unlikely.

Running the higher line-count tests as on-demand confirmatory tests, rather than exhaustively, significantly reduces the number of reference strings run and, thus, the overall time for the TLB test.(See time cost comparison in Section 4.)

4 Experimental Validation

To validate our techniques, we ran them on a collection of systems that range from commodity X86 processors through an ARM Cortex A9. All of these systems run some flavor of Unix and support the POSIX interfaces for our tools.

Table 1 shows the results of the L1 cache test: the capacity, line size, associativity, and latency, along with the total time cost of the measurements. On most machines, the tests only required roughly half a second to detect all the L1 cache parameters, except for the ARM machine and the Pentium 4. The ARM timings are much slower in all three tests because it is a Solid Run Cubox-i4Pro with a 1.2GHZ Freescale iMX6-Quad processor. It runs Debian Unix from a commercial SD card in lieu of a disk. Thus, it has a different OS, different compiler base, and different hardware setup than the other systems, which are all off-the-shelf desktops, servers, or laptops. Thus all of the timings from the ARM system are proportionately slower than the other systems. The Intel Pentium 4 system is relatively higher than the other chips, despite its relatively fast clock speed of 3.2GHZ. Two factors explain this seemingly slow measurement time. The latency of the Pentium 4's last level cache is slow relative to most modern systems. Thus, it runs samples that hit in the cache more slowly than the other modern systems. In addition, its small cache size (384KB) means that a larger number of samples miss in the last level cache (and run slowly in main memory) than the other tested systems.

The multi-level cache and TLB tests produce noisy data that approximates the piecewise linear step functions that describe the processor's response. We developed an automatic, conservative and robust analysis tool which uses a multi-

Processor	Level	Effective Cap.(KB)	Physical Cap.(KB)	Latency Cycle	Cost(Secs)
	1	32	32	3	
	2	256	256	13	
Intel Core i3	3	**2048**	3072	30	2.05
	1	32	32	5	
	2	256	256	13	
Intel Core i7	3	**3072**	4096	36	3.70
	1	32	32	4	
	2	256	256	13	
Intel Xeon E5-2640	3	**14336**	15360	42	4.49
	1	32	32	4	
Intel Xeon E5420	2	**4096**	6144	16	4.68
	1	32	32	3	
Intel Xeon X3220	2	**3072**	4096	15	3.92
	1	32	32	3	
Intel Xeon E7330	2	**1792**	3072	14	6.87
	1	32	32	3	
Intel Core2 T7200	2	4096	4096	15	5.66
	1	8	8	4	
Intel Pentium 4	2	**384**	512	39	8.03
	1	32	32	4	
ARM Cortex A9	2	1024	1024	11	54.57

Table 2: Multilevel Caches Results

step process to derive consistent and accurate capacities from that data. [2] The analysis derive for both cache and TLB, the number of levels and the transition point between each pair of levels (*i.e.*, the effective capacity of each level).

Table 2 shows the measured parameters from the multi-level cache test: the number of cache levels, effective capacity, latency, and total time required for the measurement. In addition, the table shows the actual physical capacities for comparison against the effective capacities measured by the tests. The tests are precise for lower-level caches, but typically underestimate the last level of cache—that is, their effective capacities are smaller than the physical cache sizes for the last level of cache. As discussed earlier, if the last level of cache is shared by multiple cores, or if the OS locks the page table into that cache, we would expect the effective capacity to be smaller than the physical capacity. The time costs of the multi-level cache test are all few seconds except for the ARM machine because of the same reasons we explained above.

Table 3 shows the results of TLB test: the number of levels, effective capacity for each level ($entries \times pagesize$), and the time cost of the measurement. From Table 3 we see that, on most systems, the TLB test ran in less than 1 second. As with the multi-level cache test, the Pentium 4 is slower than the newer systems. The same factors come into play. It has a small, one-level TLB with a capacity of 256KB. The test runs footprints up to 8MB, so the Pentium 4 generates many more TLB misses than are seen on machines with larger TLB capacities.

[2] The details are omitted due to space limit. Please contact the authors if interested.

Processor	Level	Capacity (KB)	Cost (Secs)
Intel Core i3	1 2	256 2048	0.14
Intel Core i7	1 2	256 4096	0.84
Intel Xeon E5-2640	1 2	256 2048	0.15
Intel Xeon E5420	1 2	64 1024	0.20
Intel Xeon X3220	1 2	64 1024	0.18
Intel Xeon E7330	1 2	64 1024	0.20
Intel Core2 T7200	1 2	64 1024	1.28
Intel Pentium 4	1	256	3.09
ARM Cortex A9	1 2	128 512	8.91

Table 3: TLB Results

The reason why we didn't measure the associativity and linesize for multi-level caches and TLB is that caches above L1 tend to use physical-address tags rather than virtual-address tags, which complicates the measurements. The tools generate reference string that are contiguous in virtual address space; the distance relationships between pages in virtual space are not guaranteed to hold in the physical address space. (Distances within a page hold in both spaces.)

In a sense, the distinct runs of the reference string form distinct samples of the virtual-to-physical address mapping. (Each time a specific footprint is tested, a new reference string is allocated and built.) Thus, any given run at any reasonably large size can show an unexpectedly large time if the virtual-to-physical mapping introduces an associativity problem in the physical address space that does not occur in the virtual address space.

The same effect makes it difficult to determine associativity at the upper level caches. Use of physical addresses makes it impossible to create, reliably, repeatedly, and portably, the relationships required to measure associativity. In addition, associativity measurement requires the ability to detect the **actual**, rather than **effective**, capacity.

The goal of this work was to produce efficient tests. We compare the time cost of our tool and Sandoval's [8] as shown in Table 4. (Other prior tools either cannot run on modern processors or produce wrong answers every now and then.) The savings in measurement time are striking. On the Intel processors, the reformulated L1 cache test is **20** to **70** times faster; the multi-level cache test is **15** to **40** times faster; and the TLB test is **60** to **250** times faster. The ARM Cortex A9 again shows distinctly different timing results: the L1 cache test is 2 times faster, the multi-level cache test is about 8.4 times faster, and the TLB test is about 4.7 times faster.

Processor	Tools	L1 Test Cost	Multilevel Test Cost	TLB Test Cost
Intel Core i3	Our Tool	0.49	2.05	0.14
	Sandoval's tool	27.02	58.16	36.81
Intel Core i7	Our Tool	0.56	3.70	0.84
	Sandoval's tool	34.75	92.35	94.97
Intel Xeon E5-2640	Our Tool	0.49	4.49	0.15
	Sandoval's tool	33.33	65.42	38.79
Intel Xeon E5420	Our Tool	0.54	4.68	0.20
	Sandoval's tool	28.86	150.43	55.56
Intel Xeon X3220	Our Tool	0.51	3.92	0.18
	Sandoval's tool	28.89	121.54	54.17
Intel Xeon E7330	Our Tool	0.52	6.87	0.20
	Sandoval's tool	35.77	228.13	53.24
Intel Core2 T7200	Our Tool	0.55	5.66	1.28
	Sandoval's tool	34.86	200.82	166.19
Intel Pentium 4	Our Tool	1.71	8.03	3.09
	Sandoval's tool	40.45	227.57	194.37
ARM Cortex A9	Our Tool	7.98	54.57	8.91
	Sandoval's tool	16.76	458.55	42.03

Table 4: Our Tool VS. Sandoval's Tool

5 Conclusions

This paper has presented techniques that efficiently measure the ***effective capacities*** and other performance-critical parameters of a processor's cache and TLB hierarchy. The tools are portable; they rely on a C compiler and the POSIX OS interfaces. The tools are efficient; they take at most a few seconds to discover effective cache and TLB sizes. This kind of data has application in code optimization, runtime adaptation, and performance understanding.

This work lays the foundation for two kinds of future work: (1) measurement of more complex parameters, such as discovering the sharing relationships among hardware resources, or measuring the presence and capacities of features such as victim caches and streaming buffers; and (2) techniques for lightweight runtime adaptation, either with compiled code that relies on runtime-provision of hardware parameters or with lightweight mechanisms for runtime selection from pre-compiled alternative code sequences.

References

1. Saavedra, R.H. and Smith, A.J., 1995. Measuring cache and TLB performance and their effect on benchmark runtimes. IEEE Transactions on Computers, 44(10), pp.1223-1235.
2. McVoy, L.W. and Staelin, C., 1996. lmbench: Portable Tools for Performance Analysis. In USENIX annual technical conference (pp. 279-294).

3. Dongarra, J., Moore, S., Mucci, P., Seymour, K. and You, H., 2004. Accurate cache and TLB characterization using hardware counters. Computational Science-ICCS 2004, pp.432-439.
4. Yotov, K., Pingali, K., & Stodghill, P. (2005, September). X-ray: A tool for automatic measurement of hardware parameters. In Quantitative Evaluation of Systems, 2005. Second International Conference on the (pp. 168-177). IEEE.
5. Yotov, K., Pingali, K. and Stodghill, P., 2005. Automatic measurement of memory hierarchy parameters. ACM SIGMETRICS Performance Evaluation Review, 33(1), pp.181-192.
6. Duchateau, A.X., Sidelnik, A., Garzarn, M.J. and Padua, D., 2008, July. P-ray: A software suite for multi-core architecture characterization. In International Workshop on Languages and Compilers for Parallel Computing (pp. 187-201). Springer, Berlin, Heidelberg.
7. Gonzlez-Domnguez, J., Taboada, G.L., Fragela, B.B., Martn, M.J. and Tourino, J., 2010, April. Servet: A benchmark suite for autotuning on multicore clusters. In Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on (pp. 1-9). IEEE.
8. Sandoval, J.A., 2011. Foundations for Automatic, Adaptable Compilation (Doctoral dissertation, Rice University).
9. Taylor, R. and Li, X.. A micro-benchmark suite for AMD GPUs. In Parallel Processing Workshops (ICPPW), 2010 39th International Conference on (387-396). IEEE.
10. Sussman, A., Lo, N. and Anderson, T.. Automatic computer system characterization for a parallelizing compiler. In Cluster Computing (CLUSTER), 2011 IEEE International Conference on (pp. 216-224). IEEE.
11. Abel, A., 2012. Measurement-based inference of the cache hierarchy (Doctoral dissertation, Masters thesis, Saarland University, 2012.
12. Gonzlez-Domnguez, J., Martn, M.J., Taboada, G.L., Expsito, R.R. and Tourino, J., 2013. The Servet 3.0 benchmark suite: Characterization of network performance degradation. Computers & Electrical Engineering, 39(8), pp.2483-2493.
13. Casas, M. and Bronevetsky, G., 2014, May. Active measurement of memory resource consumption. In Parallel and Distributed Processing Symposium, 2014 IEEE 28th International (pp. 995-1004). IEEE.
14. Casas, M. and Bronevetsky, G., 2016. Evaluation of HPC applications memory resource consumption via active measurement. IEEE Transactions on Parallel and Distributed Systems, 27(9), pp.2560-2573.
15. Moyer, S.A., 1991. Performance of the iPSC/860 node architecture. Institute for Parallel Computation, University of Virginia.
16. Qasem, A. and Kennedy, K., 2006, June. Profitable loop fusion and tiling using model-driven empirical search. In Proceedings of the 20th annual international conference on Supercomputing (pp. 249-258). ACM.
17. Luk, C.K. and Mowry, T.C., 2001. Architectural and compiler support for effective instruction prefetching: a cooperative approach. ACM Transactions on Computer Systems, 19(1), pp.71-109.