Automatic Memory Hierarchy Characterization

Clark L. Coleman and Jack W. Davidson
Department of Computer Science, University of Virginia
E-mail: {clc5q, jwd}@cs.virginia.edu

Abstract

As the gap between memory speed and processor speed grows, program transformations to improve the performance of the memory system have become increasingly important. To understand and optimize memory performance, researchers and practitioners in performance analysis and compiler design require a detailed understanding of the memory hierarchy of the target computer system. Unfortunately, accurate information about the memory hierarchy is not easy to obtain. Vendor microprocessor documentation is often incomplete, vague, or worse, erroneous in its description of important on-chip memory parameters. Furthermore, today's computer systems contain complex, multi-level memory systems where the processor is but one component of the memory system. The accuracy of the documentation on the complete memory system is also lacking. This paper describes the implementation of a portable program that automatically determines all of a computer system's important memory hierarchy parameters. Automatic determination of memory hierarchy parameters is shown to be superior to reliance on vendor data. The robustness and portability of the approach is demonstrated by determining and validating the memory hierarchy parameters for a number of different computer systems, using several of the emerging performance counter application programming interfaces.

1. Introduction

Because the gap between processor performance and memory performance continues to grow, performance analysis and compiler optimizations are increasingly focused on the memory hierarchy of the target computers. To optimize memory system performance we need to know, for any level of cache or TLB (translation lookaside buffer), the size, line size, associativity, write allocation and replacement policies, and whether each cache or TLB is split or unified.

One approach to determining these parameters is to search vendor documentation for the relevant numbers. There are numerous deficiencies in this method, based on the authors' experiences:

1. Vendor documents can be incorrect. For example, the Intel P6 Family Instruction Set Architecture Manual [5], in its description of the CPUID instruction, describes the 16KB L1 (Level One) data cache associativity as being 2-way when it is in fact 4-way. Later corrections to this value and several TLB values in this manual confirm the presence of errors over time.

2. Vendor documents can be vague, such as when a single manual attempts to describe several related processors within a family. This often leads to describing the memory hierarchy with a range of possibilities, or only with the common denominator parameters.

3. Vendor manuals often describe memory hierarchy components that are used by the operating system but are not accessible to user-level code under that operating system [7], such as the large-page TLBs in the Intel P6 family processors [5]. Automatic characterization reveals the memory hierarchy as actually seen by the application programs on the system.

4. Some parameters are dependent on the system, not the CPU, and thus are not documented fully in CPU manuals. For example, off-chip secondary cache sizes typically vary in powers of two among systems using a certain CPU. The size is best determined dynamically.

5. Gathering information through searches of processor, system, and OS documents, followed by email correspondence to resolve ambiguities and errors, is very time-consuming in comparison to a dynamic characterization that runs in only a few minutes.

For these reasons, documentation is an insufficient source for needed memory hierarchy information. A superior approach would be to design a program that can reliably perform dynamic memory hierarchy char-

acterization directly on the machines to be characterized. We have successfully designed and tested such a program, which we describe in this paper.

1.1. Dynamic Measurement using Timing Tools

Prior tools have measured latency, bandwidth, and cache sizes using the timing precision available in the standard Unix/C environment [6, 1]. Through repeated memory accesses (in which lengthy straight-line code repeatedly dereferences and then increments a pointer) and timings, these programs can detect the slowdown that occurs when a data array exceeds the size of a level of cache, and hence can determine the size of that level of data (or unified) cache, and its latency and bandwidth. These tools are valid and useful for their intended application.

There are limitations in this approach, however. Existing tools do not measure instruction caches, nor instruction or data TLBs. Thus, they do not give a complete, precise measurement of the parameters required for optimal use of the memory hierarchy.

To address the problem of gathering accurate, reliable data about the characteristics of all aspects of the memory hierarchy, we have designed a portable program, called AMP (for Automatic Memory Hierarchy Parameterization), that automatically determines all key memory hierarchy characteristics including those of the instruction caches and TLBs. Most modern processors incorporate hardware performance counters that can count certain events, such as cache or TLB misses. However, there are several challenges to using such counters.

First, there is no common set of counters available across all processors. The counters available and what events are recorded vary widely even among processors from the same vendor. Second, accessing the counters that are available is machine and operating system dependent. To overcome these challenges, we have designed a middle layer module that acts as an interface between AMP and the application program interface (API) for accessing the counters. There are currently several research groups providing APIs for access to these performance counters. The APIs germane to this project are described in Section 2.1.

A third, and perhaps most difficult challenge, is that it is not feasible or reasonable to run measurement programs such as AMP on a standalone machine. For example, many systems require connections to the network for file system access. Configuring a machine to run in a standalone fashion is often not an option (e.g., removing a compute server from the pool of available servers), and when it is an option, taking a machine and rebooting it in standalone mode is time-consuming. To address this problem, we designed AMP so it can be run on a machine in a standard networked environment. By carefully controlling the experiments that are run and using statistical techniques, AMP accurately determines the memory hierarchy parameters on a machine being run in a standard computing environment.

The remainder of the paper has the following organization. In the next section, we address issues regarding portability and the robustness of our measurements on multi-tasking systems. Section 2 describes the algorithms developed to compute the various memory system parameters. Section 4 discusses the measurements collected for a variety of systems, Sections 5 and 6 describe the implementation and availability of the software, and Sections 7 and 8 summarize the contributions of the work and future enhancements.

2. Robustness and Portability

In this section, we address issues confronted when running AMP on multiple targets that have different performance counters available, with different access methods. We also show how we solved the problems introduced by competition on multi-tasking systems.

2.1. Performance Counter APIs and Portability

AMP currently runs primarily on top of the PCL performance counter API [2], available on several hardware platforms. A similar API effort is PerfAPI [8]. The Rabbit API is available for x86/Linux measurements only [4]. AMP has been ported to these APIs. Development work was undertaken in the DOS/DJGPP environment [3], which permitted direct counter access in privileged mode under DOS, without multitasking overhead.

We have created a middle layer module that acts as an interface between the code implementing the algorithms described above and the performance counter API. Queries (available in all three APIs) are used to determine what events are available on a system (and thus, how many levels of cache have associated counters). Porting to a new performance counter API only requires changes to this middle layer module.

A machine-dependent module can use machine-specific instructions to flush caches, TLBs, etc., but can be made to just return `false` for all functions to minimize porting effort. When a return value of `false` is seen,

the cache flushing proceeds as described in Section 3.1., Step 1, in which code or data accesses are used to evict from the cache the code or data of interest.

2.2. Robustness and Reliability

We observed during testing that AMP suffered a loss of reliability whenever the APIs (or competing system processes) caused cache misses unrelated to AMP's algorithms. A voting scheme eliminated this problem. The main program actually gathers measurements in a voting loop. An outline of the measurement and voting approach is as follows.

```
1. while (vote limit not reached) do
2.   Perform experiment measuring cache
     misses 8 times
3.   Take minimum of these 8 numbers
4.   Use the minimum to compute the
     result (e.g. line size)
5.   Store result as the current vote
6. endwhile
7. The final value = most common vote
```

Using the minimum misses from multiple repetitions discards high numbers of misses caused by context switching or other anomalies. Our hypothesis is that no event in the system will reduce measured cache misses, but many events could increase them, so the minimum is the most accurate measurement. (Note that the APIs used count misses on a per-process basis; the problem is not the accidental counting of misses suffered by another process, rather the increase in AMP process cache misses caused by cache evictions that occur during execution of another process, causing certain data items to not be found in the cache as expected.) Thus, computing the L1 D-cache line size involves taking eight measurements, recording the minimum number of misses among those eight, using that minimum to compute a line size, and making that line size the first vote, etc. An important point here is that all measurements are designed to produce hundreds of cache misses, or at least dozens of TLB misses; AMP does not depend on a precise number of misses that is small, as this would be unreliable even after implementing this voting scheme. In all, 64 measurements will produce 8 reliable votes. Prior to implementing the voting scheme, AMP produced inconsistent results on different runs on very lightly loaded Unix systems. With voting, AMP returns the same parameters on more than 90% of all runs.

Another boost to our measurement reliability is the computation of the performance counter API overhead. The API might make small cache perturbations, which need to be removed from our measurements. At startup, AMP loops through an array that has been read into the cache and should be small enough to remain there for a brief duration. AMP counts the cache misses (which would be zero, if there were no API overhead) and takes the minimum of several counts as the overhead for that measurement. This is done for all caches and TLBs. The overhead is subtracted from each measurement described in this paper before any cache miss number is used in determining any cache parameter.

3. Algorithms and Measurements

The subsections below describe our measurement algorithms in terms of performance counters while making reference to particular machine dependencies only where necessary. Keep in mind when reading these measurement algorithms that each measurement is actually a sequence of repeated measurements, followed by a voting scheme, as discussed in Section 2.2.

In the data cache discussions, AMP uses a very large integer array declared as a C-language `static` global array within the main code module. The main module contains all the measurement functions for the data cache elements of the memory hierarchy. The array is configurable in size using named constants and currently consists of 256 rows, each being 128KB in size, for a total size of 32MB. This is substantially larger than any currently known data cache. Hereafter, we refer to this data structure simply as "the array."

The strategy of AMP is to first determine the line size (or TLB page size) for each level of the memory hierarchy in turn. Once the line size has been determined, AMP knows how often misses will occur in further experiments that access uncached memory regions at that level. For example, if the L1 D-cache line size is 32 bytes, then accessing a 1024 byte region of memory that is *not* present in the L1 D-cache should produce 1024/32 = 32 L1 D-cache misses. The approach for other parameters, such as total cache size and associativity, is to perform certain accesses that attempt to thrash that cache. If thrashing occurs, AMP will get the maximum possible miss rate for that access sequence, which is once per cache line. Details are in the sections below.

3.1. Data Cache Line Size

The line size of a data cache is determined using the following algorithm.

```
1. Flush first few rows of the array from
   the cache
```

```
2. Read current value of performance
   counter for cache misses
3. Read the first row of the array
4. Read value of the performance counter
   again
5. Compute the miss rate
6. Set line size to the power of two that
   is nearest to the reciprocal of the
   miss rate
```

The L1 data cache will be our initial example.

Step 1: First, the data cache must have the first few rows of the array flushed from it. If a machine-dependent module is able to execute a cache flush instruction, it does so. Otherwise, AMP reads the second half of the array (i.e. the final 16MB of the array, at its current size) several times (not just once, in case a non-LRU replacement scheme is used.) If the first few rows of the array were ever present in the data cache, they should now be evicted by the later rows. This will be true as long as 16MB exceeds the size of any level of cache.

Steps 2–5: Turn on the *L1 data cache miss* performance counter, read its initial value, then read the first row of the array. Read the performance counter again, subtract its initial value from the new value to compute the number of L1 data cache misses that occurred. The number of misses is divided into the number of integers in one row of the array to give the miss rate, expressed as the proportion of integers that caused misses.

Step 6: The final step is to determine which power of 2 is nearest to the reciprocal of the miss rate. This is the cache line size. (AMP assumes that all cache line sizes are a number of bytes that is a power of 2, which is true for all data caches with which we are familiar.) The formula to compute the line size from the miss rate is:

$$LineSize = sizeof(int) \times 2^{\left\lfloor \frac{\log \frac{1}{MissRate}}{\log 2} + 0.5 \right\rfloor}$$

where *log* refers to the natural logarithm, which is divided by *log 2* to produce the logarithm base 2 of the reciprocal of the miss rate. This value is then rounded by adding 0.5 and applying the floor function (truncation). Raising 2 to this power gives us the number of integers in the cache line. A final multiplication by the number of bytes per integer converts the units to bytes.

For other levels of cache, the appropriate performance counter is used, and the same algorithm will produce the L2 line size, Data TLB (DTLB) page size, etc. Reliability of line size and page size computations is discussed in Section 2.2.

One unfortunate hurdle encountered in the validation of this algorithm was the presence of hardware errata producing invalid numbers from the L1 D-cache read miss counter on Sun UltraSparc-I and UltraSparc-II systems [9]. This counter produces more read misses than there actually were data reads performed. We detected these anomalous numbers, and investigation turned up the errata sheets for the CPUs. AMP works around this problem by using a change in the code, controlled by compilation directives for UltraSparc targets, that uses *writes* instead of reads in the accesses to the first array row. The write miss counter is verified to work properly on these systems, and line size can be determined as accurately as on other CPUs.

3.2. Data Cache Size

Determination of the size of a data cache uses steps similar to determining the line size:

```
1.  SizeHypothesis = MIN_CACHE_SIZE
2.  while (SizeHypothesis is <=
    MAX_CACHE_SIZE) do
3.    Read SizeHypothesis bytes at
      beginning of the array
4.    Read performance counter for cache
      misses
5.    Reread SizeHypothesis bytes from
      beginning of array
6.    Reread performance counter
7.    if one miss per cache line then
8.      Exit the loop
9.    else
10.     Double the SizeHypothesis
11. endwhile
```

The algorithm iterates over cache size hypotheses starting with a defined minimum cache size. This is a named constant in the code that is currently 1024 bytes. (For modern processors that have performance counters, no cache will be this small, and the processors of several CPU generations ago that had cache sizes as small as 1KB did not have performance counters and will not be subject to our measurements.)

Steps 3–6: The first 1KB of the array is read to place it into the cache (if it will fit.) The appropriate performance counter is started, the hypothesized cache size is read a second time, and the new value of the counter is read. A miss rate for this second pass through the hypothesized cache size is computed from the difference in the counter values read.

AMP defines the *expected failure miss rate* as the miss rate that it will see if the hypothesized cache size exceeds the actual cache size and the first read pass

wrapped around the cache and evicted itself, causing the second read pass to miss once for each cache line. Thus, the expected failure miss rate is the reciprocal of the already-computed cache line size.

Steps 7–10: If the miss rate does not exceed a threshold fraction of the expected failure miss rate (a named constant, currently 0.80, empirically derived from experiments on several systems), then AMP concludes that the test data fit into the cache. In this case, the hypothesized size is doubled, and AMP iterates again.

When AMP finds a miss rate that indicates failure, it could assume that the *previous* hypothesis (the largest size that did *not* fail) is the cache size. However, not all cache sizes are powers of two; for example, the Alpha 21164 CPU has an on-chip unified secondary cache that is 96KB in size [10]. To accurately measure the size of such a cache, AMP iterates between the last non-failure test size and the first failed test size, in increments of 25% of the difference between them.

3.3. Data Cache Associativity

Given the cache size, the associativity can be found by experiments that try to thrash the cache.

```
1.  AssocHypothesis = 1
2.  while (AssocHypothesis < NbrOfCache-
    Lines) do
3.    Access the array (2 * AssocHypothe-
      sis) times at spacings CacheSize /
      AssocHypothesis) bytes apart
4.    Read value of performance counter
      for cache misses
5.    Re-access the array (2 * AssocHy-
      pothesis) times at spacings
      (CacheSize / AssocHypothesis) bytes
      apart
6.    Re-read performance counter
7.    Compute miss rate
8.    if miss rate less than thrashing
      threshold then
9.      Increase AssocHypothesis to next
        feasible value
10. endwhile
11. if Thrashed then
12.   return AssocHypothesis
13. else
14.   return NbrOfCacheLines (i.e.
      fully associative)
```

Step 1: Start with direct-mapped as our hypothesis.

Step 2: The limit is a fully associative cache.

Steps 3-10: Choose an access pattern that would thrash if the true associativity matched our hypothesis. For example, if a primary data cache is known to be 16KB in size, then repeatedly and alternately accessing two array elements that are 16KB apart would create a miss rate close to 100% if the cache is direct-mapped (1-way associative), as the elements would map to the same cache line and evict each other from the cache. In this case, the function would terminate and return 1 as the associativity.

Steps 11-12: If thrashing is not observed, the hypothesis is advanced to the next integer that is a factor of the number of lines in the cache. For example, with a 16KB cache with 32 byte lines, there are 512 lines in the cache. The associativity should be a factor of 512. In this example, the factors are all powers of two, so the next associativity hypothesis is double the previous hypothesis, but this cannot be assumed. AMP will work correctly on machines with 3-way, 5-way, etc. caches.

In general, for an associativity hypothesis of k, AMP repeatedly and sequentially accesses $2k$ array elements spaced N/k bytes apart, where N is the array size. For the 2-way associative hypothesis in the 16KB array, AMP accesses 4 elements at relative addresses within the array of 0, 8KB, 16KB, and 24KB. This will thrash a 2-way (but not a 4-way) associative cache. Testing proceeds until AMP sees cache thrashing.

3.4. Write Allocation Policy

AMP can determine whether a cache allocates a cache line upon a write miss. After flushing the cache, AMP writes to a region of the array that is smaller than the size of the cache being tested. Subsequent reads to the same region will be hits if the write misses caused cache line allocations, and will miss at the rate of once per line if the cache employs a no-write-allocate policy. Because AMP assumes that each downstream (larger) cache includes the entire contents of each upstream (smaller) cache, once AMP reaches a level in the cache hierarchy with a write-allocate policy, all downstream caches are assumed to be write-allocate caches, also.

3.5. Replacement Policy

The three common approaches to determining which set to replace after a cache miss are true LRU, pseudo-LRU, and random. The two LRU schemes will be similar for all repeatable access sequences: a certain set will always be chosen for replacement for a given sequence of hits and misses. Depending on the pseudo-LRU implementation and the access sequence, true LRU and pseudo-LRU might choose different sets. Random

replacement will choose a set to replace based upon some random value supplied by the system, and will not demonstrate repeatable behavior for all repetitions of an access sequence. AMP is thus able to distinguish between LRU and random. Ongoing work will differentiate pseudo and true LRU.

For 4-way and 8-way associative caches and TLBs, AMP accesses *N* array elements that map to the same set, where *N* is the associativity. Then it accesses the first *N-1* elements again, followed by an *(N+1)*st element that maps to the same set. This last element will cause eviction of one of the first *N* elements. Turning on the appropriate cache miss counter and accessing element *i* will determine if element *i* was replaced. Repeating the entire experiment and iterating *i* from 0 to N-1 will give AMP a statistical picture of the replacement policy. If only a single set among the first *N* sets is ever replaced when the *(N+1)*st element is accessed, then some form of LRU replacement is being used. If the misses are scattered throughout all *N* sets, the replacement was random.

3.6. Data Cache: Split or Unified

It is essential to know whether a given level of cache is data-only or unified when analyzing performance or performing certain compiler optimizations. AMP can detect a unified cache by simply reading a portion of the array that equals the cache size, then executing a synthesized chunk of object code that is at least that size (but which does not perform data operations), and then re-read the portion of the array. If the second pass through the data array misses once per cache line, then the code execution must have evicted the data from the cache in question, and AMP concludes that it is unified.

An important result from the determination of unified or split status is that AMP can obtain some useful information in the absence of a complete set of performance counters. If a CPU has an L2 data cache miss counter, but not an L2 instruction cache miss counter, AMP can characterize the line size, total size, associativity, and write policies of the L2 cache using the data counters only. After AMP determines that the cache is unified, the absence of an L2 instruction miss counter is not a problem. The same applies to unified TLBs.

3.7. Instruction Cache Line Size

Instruction cache line size is computed in the same way as the data counterpart, except that AMP executes a large block of straight-line code instead of accessing a region of a data array. The code block is generated from macros that repeatedly perform additions and subtractions on a pair of variables, leaving them with their initial values so that no overflow will occur when the macro is repeated thousands of times. The function containing the large block of code is compiled without optimization to ensure that the code does not disappear during compilation. The `gcc` compiler is used in order to take advantage of a non-standard extension to the C language that it provides, viz. the ability to take the object code address of a label in the code using the "`&&`" operator. This operator is used to compute the size of a block of code, which is then used (along with the cache miss counter values) to compute the miss rate. AMP then computes the line or page size directly from the miss rate using the equation from Section 3.1., dropping the `sizeof(int)` multiplication.

3.8. Instruction Cache Size

AMP computes the I-cache sizes using a function that contains a sequence of `switch` statements with 32 `cases` each, each `case` containing a code macro that generates 1KB of object code for the target machine. This macro is obtained from a header file that is generated automatically. A preliminary step in the software building process for AMP uses the `gcc '&&'` operator to compute object code sizes for the code macro, along with other pieces of code such as the surrounding control flow and a `NOP` instruction on the target machine. Using these sizes, the preliminary program generates a header file with macros that will expand to 1KB and 4KB of object code on the target machine, within a few bytes.

The function executes specified (via input parameters) `cases` within the `switch` statement to prime the instruction caches and TLB, then turns on the requested performance counter and executes the same `cases` again. As with the data case, AMP detects the expected failure miss rate when it has exceeded the size of the cache, then performs a finer-grained search between the final pair of powers of two to get the final size.

As secondary and tertiary (L3) caches can be quite large, a clone of this function is provided in which each `case` of the `switch` statement has 4KB of object code from a macro instead of only 1KB. This function is called automatically as the size being tested exceeds the size that can be tested using 1KB macros.

In order to keep the size of these large blocks of synthetic code within very precise bounds, AMP measures,

using the `gcc` && operator, the code size produced by `if` and `switch` statements that surround the code macros. Every few `cases` within each `switch` statement, AMP uses a different macro that produces slightly less code in order to compensate for the overhead of this control code. Because `gcc` cannot compile a `switch` statement with 1024 `cases`, each of which has 1KB of code, AMP uses a sequence of 32 `switch` statements, each of which has only 32 `cases`, to achieve the code size needed. Even so, `gcc` can easily exhaust virtual memory limits on many systems. We also discovered an error in `gcc` code generation for such a large function on Compaq Alpha systems, which has been fixed by the `gcc` maintainers.

3.9. Instruction Cache Associativity

AMP computes associativity using the same function with the large `switch` statements, executing non-contiguous blocks of code located at relative spacings, just as it accessed array elements at certain relative spacings in Section 3.3.

3.10. TLB Measurements

All algorithms have been successfully tested and confirmed to produce valid results for instruction and data TLBs, where the analogous parameters are page size (instead of line size), TLB entries (instead of size in bytes), and associativity.

An interesting anomaly was detected on our MIPS R10000 CPU in an SGI Octane system running the IRIX operating system. Initial efforts to determine the number of TLB entries failed. Inspection of the raw data coming from the TLB miss counters showed that, as our SizeHypothesis neared the size reported in vendor documents (2 MB, in 64 entries that each map pairs of 16 KB sub-pages), thrashing occurred but then sub-sided, reducing the miss rate to nearly zero. After much repetition and validation of these results, vendor employees confirmed that the IRIX OS can be configured to detect TLB thrashing and dynamically resize the pages to use more than 16KB per page. Thus, the repetitions of measurements, designed to increase robustness, gave IRIX time to detect repeated thrashing and eliminate it. AMP was redesigned to detect and report this dynamic resizing of pages, which has only been seen on IRIX systems to date.

Much to our surprise, AMP stopped reporting dynamic page resizing on a certain date. Our system administrators confirmed that a new release of IRIX had been installed, and they had not bothered to enable the page resizing. This confirmed the accuracy of AMP's page resizing detection algorithm.

4. Summary of Measurements

Table 1 summarizes the measurements collected for some popular machines. Associativity is measured in *number of degrees*; all other parameters are measured in *bytes*. N/C indicates that No Counter is available to count misses for that memory hierarchy element.

The systems used, respectively, are a 133MHz Pentium, 200MHz Pentium Pro, 233 MHz Pentium II, 167 MHz UltraSparc-I, and an SGI Octane with 225 MHz R10000 CPU. The Pentium CPUs have entirely different performance counter hardware (and software interfaces) than the Pentium-II/Pentium Pro family CPUs.

For TLB parameters, size is in *number of entries*. All machines had unified L2 caches; the R10000 has a unified TLB. All L1 caches were split. The P5 and UltraSparc L1 D-caches were no-write-allocate. No random replacement schemes were detected. Results were validated using the time consuming sources mentioned previously: vendor documentation, vendor personnel, etc.

System	L1 Code	L1 Data	L2	ITLB	DTLB
	Line/Size/Ass.	Line/Size/Ass.	Line/Size/Ass.	Page/Size/Ass.	Page/Size/Ass.
P5-133	32 / 8K / 2	32 / 8K / 2	N/C	4K / 32 / 2	4K / 64 / 4
PPro-200	32 / 8K / 2	32 / 8K / 2	32 / 256K / 4	4K / 32 / 4	N/C
PII-233	32 / 16K / 4	32 / 16K / 4	32 / 512K / 4	4K / 32 / 4	N/C
USparc-I	32 / 16K/ 2	32 / 16K / 1	64 / 512K / 1	N/C	N/C
R10K-225	64 / 32K / 2	32 / 32K / 2	128 / 1024K / 2	32K / 64 / 64	32K / 64 / 64

TABLE 1. Measured Cache and TLB Parameters

5. Implementation

AMP is implemented in 12 modules of ANSI standard C comprising 29,214 lines of code including comments. It is compiled using `gcc` version 2.95.2. Run times range from 5 to 50 minutes on different systems (absence of certain counters speeds up the run times considerably; TLB measurements are the most time consuming.) The executable file is approximately 9MB. Using a preprocessor symbol, this can be reduced to 800KB to create a DOS bootable diskette with a reduced version of AMP for Intel x86 PCs. The primary code reduction in this version is the removal of the majority of the synthetic code modules discussed in Section 3.8.. This prevents AMP from determining the size of any L2 instruction cache that exceeds 256KB. As x86 PCs have a unified L2 cache, the L2 parameters will have already been determined using data cache measurements, and AMP will be able to determine that the L2 cache is unified if it does not exceed 512KB. The diskette version of AMP can be used to test multiple PCs quickly without installation of any APIs.

6. Software Availability

The file `ftp://ftp.cs.virginia.edu/pub/AMP/README` contains instructions for downloading executables for various target machines. Source code will be available here soon.

7. Ongoing Work

Enhancements nearing completion include miss penalty characterization for caches and TLBs, detection of sub-block and sub-page schemes, and measurement of hardware elements such as branch target buffers. New target systems will be used as they become available, and measurements will be maintained at the FTP site.

8. Summary

Knowing the memory hierarchy parameters of a computer system is vital information for tuning memory performance models and applying optimizations geared to improving memory performance. Unfortunately, obtaining the memory hierarchy parameters of any particular system's memory hierarchy has been difficult. Vendor literature is sometimes erroneous, often hard to find for a particular system, and usually vague. Furthermore, some systems have memory system characteristics that can be set by the operating system. To address these problems, we have developed AMP. AMP accurately determines the memory hierarchy parameters of a computer system by running a set of experiments and using statistical techniques to compensate for any outside interference. Using AMP, a user can determine line size, associativity, capacity, write allocation and miss replacement policies, and organization of the caches and TLBs of the memory hierarchy. Our approach has been verified by running AMP on a variety of architectures and systems. Interestingly, AMP uncovered several important errors and omissions in vendor documents. Many interesting obstacles were overcome, including dynamic TLB page resizing, hardware errata on the counters, process competition on the target systems, and varying overheads of multiple APIs.

9. References

[1] Brown, A., and M. Seltzer, "Operating System Benchmarking in the Wake of *Lmbench*: A Case Study of the Performance of NetBSD on the Intel x86 Architecture," *Proceedings of the1997 ACM SIGMETRICS Conference*, Seattle, WA, June, 1997, pp. 214–224.

[2] Berrendorf, Rudolf, and Heinz Ziegler, *PCL: The Performance Counter Library*, web site `http://www.fz-juelich.de/zam/PCL/`.

[3] Delorie, D.J., developer of DJGPP, web site `http://www.delorie.com/DJGPP/`.

[4] Heller, Don, web site `http://www.scl.ameslab.gov/Projects/Rabbit/`.

[5] Intel Corporation, *Intel Architecture Software Developer's Manual, volume 2: Instruction Set Reference*, 1997. Note page 3-74 for CPUID instruction.

[6] McVoy, L., and C. Staelin, "lmbench: Portable Tools for Performance Analysis," *Proceedings of the 1996 Usenix Technical Conference*, San Diego, CA, January, pp. 279–295.

[7] MIPS Technologies, *MIPS R10000 Microprocessor User's Manual, version 2.0*, October 10, 1996. Compare section 14.6 and section 16.3 re: *wired* TLB entries.

[8] Mucci, Philip J., Shirley Browne, Christine Deane, and George Ho, "PAPI: A Portable Interface to Hardware Performance Counters", *Department of Defense High Performance Computing Modernization Program Users Group Conference*, Monterey, CA, June 7-10, 1999. See `http://icl.cs.utk.edu/projects/papi/`.

[9] Sun Microsystems, *UltraSparc-IIi User's Manual*, 1997.

[10] Compaq Computer Corporation, *Alpha Architecture Handbook*, Version 4, 1998.