MOB: A Memory Organization Benchmark

Josep M. Blanquer and Robert C. Chalmers
Department of Computer Science
University of California
Santa Barbara, CA 93106-5110
{blanquer,robertc}@cs.ucsb.edu

October 2000

Abstract

It is often difficult to elicit many details of a computer's memory architecture from published specifications, but these parameters can be quite valuable when evaluating software performance over a number of architectures. *MOB*, the Memory Organization Benchmark, provides a platform-independent benchmark suite capable of characterizing the properties of a memory sub-system. Currently, *MOB* automitically detects parameters such as the number of caches, the size of each cache, block size, associativity, replacement policy, write policy, as well as similar properties for TLBs. The benchmarks have been tested over a wide range of machines and operating systems.

1 Introduction

Computers are already an essential part of our everyday routine. New demands force technology to evolve, but different apsects of computer architectures evolve at varying rates. This has led to modern systems formed from a collection of specialized subsystems that interchanged and matched according to a designers functional and cost requirements. It is often difficult to elicit many details of a computer's architecture from published specifications, but these parameters can be quite valuable when evaluating software performance over a number of architectures.

MOB, the Memory Organization Benchmark, provides a platform-independent benchmark suite capable of characterizing the properties of a memory sub-system. Currently, *MOB* automitically detects parameters such as the number of caches, the size of each cache, block size, associativity, replacement policy, write policy, as well as similar properties for TLBs. *MOB* was designed witha number of goals in mind:

- Indepence from CPU architecture and OS.
- Use as few assumptions about the underlying memory sub-system as possible.
- Minimize the inter-dependency between benchmarks.
- Provide inline analysis of results and useful conclusions about cache and TLB parameters.

The benchmarks have been tested over a wide range of machines including Intel, Sparc and Alpha processors and a number of operating systems, such as FreeBSD, Linux, Solaris and DEC Unix. Throughout the design and experimentation, we found that the most interesting results and the hardest challenges came from unexpected idosynchrousies in the architectures and peculiar optimizations at the OS level. Testing across platforms has resulted in a more complete, stable and effective tool.

The rest of the paper is organized as follows: Section 2 introduces the basic techniques used to probe the memory system and describe the general approach behind our inline anaylsis. We discuss *MOB* as a tool in Section 3 and elaborate on the specific analysis for each benchmark including the limitations of our heuristics. In Section **??**, we explore many anomalies which we encountered through experimentation. Related work is covered in Section 5 and we conclude the paper in Section 6.

2 Probing the Memory System

Memory sub-systems are designed hierarchically such that memory nearest the processor is accessible within a single or at most a very few clock cycles [?]. However, as memory speeds increase, so do the costs and the constraints on design. Faster caches are in turn smaller caches. Moving up through the heirarchy, caches and eventually main memory grow larger and suffer longer access delays. These two characteristics of the memory sub-system, speed and size differentiation, make it possible to deduce properties of each cache.

Although not striclty part of the cache hierachy, Translation Lookaside Buffers (TLBs) are an important part of the memory sub-system. Memory is typically partitioned into a set of contiguous blocks or pages. Applications, however, view their entire address space as contiguous using virtual addresses. The operating system must convert each virtual address into a physical address representing the actual location of a page in memory. Most modern processors maintain a small cache of virtual to physical translations, the TLB [?]. TLB properties can be deduced in much the same way as normal cache properties.

The remainder of this section elaborates on the techniques used to deduce cache and TLB properties. We cover many basic techniques that have been developed in previous research and present how this paper extends on those techniques with an emphasis on inline analysis.

2.1 Basic Techniques

When an attempt is made to access a portion of memory that is not resident in cache, a miss penalty is incurred due to the increased delay. A simple technique to measure this miss penalty was employed as early as 1977 [1]. The cache was filled by reading an array of values. The miss penalty was then measured as the difference between a cache hit, re-reading the same data, and reading a separate set of values thus causing a cache miss on each access. The researchers, however, already knew the properties of the caches being investigated.

Saavedra and Smith extended this technique in 1995 [2] in order to detect, not just miss penalties, but the underlying characteristics of the caches themselves, such as size and associativity. In their experiments, the authors developed a micro-benchmark that varied the size of memory accessed and distance (stride) between adjacent accesses. They categorized the accesses into four distinct *regimes*. For each regime, they computed the number of expected cache misses and derived a theoretical miss penalty. Thomborson and Yu later extended this classification to a total of six regimes [3].

The regimes provide us with boundary conditions where we expect to see an increase in the access times. Using these increases or transitions, Saavedra and Smith [2] and Thomborson et. al [4, 3] were able to accurately detect several cache and TLB properties which we describe here in the context of our implementation. We, then, continue the discussion in the Section 2.2 highlighting a number of novel benchamarks we have implented beyond this common base.

Cache Size. To detect the number of caches in the hierarchy and the size of each cache, we fix the stride and increment the total size of memory accessed. For each size, we measure the average time per access. We expect to see the access times increase substantially once the cache containment boundary is broken, causing the cache to fetch a line from the next level. For example, in a system with a 4KB level one (L1) cache, we should witness equivalent access times when limiting the total memory to 1, 2, and 4KB chunks since the accesses do not cause capacity misses. Beyond 4KB, however, the access times should markedly increase. The same effect will occur for the next cache level until we finally force each access to fetch lines directly from main memory. The clarity of each transition depends largely upon the replacement policy of each cache.

Miss Penalty. Having performed the previous benchmark, the miss penalty incurred by each level in the cache hierachy can be calculated directly from the access times.

Block Size. Caches are divided into a number of separate entries or blocks. Each block represents a small portion of contiguous memory. For this benchmark, memory is allocated in a multiple of the cache size but smaller than the next level cache. The stride is then incremented. Strides smaller than the block size will benefit due to multiple hits per cache block. The fraction of misses increases as the stride approaches the block size. Once the stride bypasses the block size, each access should cause a capacity miss and an equivalent access time.

Number of Blocks. The total size of the cache and the size of each individual block necessarily determines the number of blocks that can be maintained simultaneously. Thus, the total number of blocks in a cache can be calculated as the cache size divided by the block size.

Associativity. Caches are usually indexed by the low-order bits of the physical or virtual[1] memory address. Each index may refer to a single cache block (direct-mapped) or a number of blocks each with an identifying tag. The number of blocks per index is referred to as the cache's associativity[2]. To detect the associativity, we must attempt to access memory at the same index by adjusting the stride appropriately. In this benchmark, we allocate memory twice the size of the cache and set the stride to equal the block size multipled by the number of indices (or the cache size divided by the candidate associativity), accessing twice as many entries as should fit in the cache. For a cache size of 16K with a candidate associativity of 4, the benchmark would cover a 32K chunck of memory in 4K strides. This would cause 8 entries to be read at a single index if the associativity were truly 4, thus causing misses on every access.

Write Policy. Saavedra and Smith's benchmark mixed reads and writes. Enyou and Thomborson [4] separated reads from writes in an attempt to determine the write policy of the cache. When a write is performed the value can be written back to the cache (write-back) or through to the next level (write-thorough). If a value will be changed repeatedly, writing through to memory could be extraordinarily expensive. However, for multi-processor systems with separate caches, synchronization becomes an issue and a write-through policy may be more effective than a complicated cache coherency algorithm. Another option exists. If a block that is being written to is not already resident in the cache, should the value be read before it is written (write-allocate)? If the value is within a block with other values that will not be read soon, then it is advantageous to write without first reading the entire block (write-noallocate). These two decisions lead to four possible configurations. In most systems, write-back and write-allocate are implemented together while write-through is usually combined with write-noallocate.

TLB Properties. Detecting properties of the TLB, such as page size, number of entries and associativity, is very similar to detecting cache parameters. The page size is analagous to the block size of the cache and can be determined by changing the stride until each access causes as TLB miss and a subsequent page fault. Many CPUs support multiple page sizes, but most OSs only use one to simplify management. To find the number of entries in the TLB, we use the page size as a stride and increment the total memory accessed until we exhaust the TLB capacity. Correctly detecting TLB parameters is hard due to cache interference. For virtually addressed caches, the TLB and cache are referenced in parallel and TLB misses are concealed by cahce hits. On the other hand, cache misses through mutliple levels may completely over-shadow the TLB misses. Saavedra and Smith realized this fact but felt that it was relatively straightforward to isolate the individual effects. We found that in performing an inline analysis this was not always the case.

2.2 Novel Benchmarks

With our benchmark suite, we have extended the previous work already discussed. All of the previous benchmarks assumed separate instruction and data caches or simply ignored the effects of a shared cache. None attempted to detect the parameters of the instruction caches or TLB. Our benchmark suite includes a complete set of tests that probes the instruction caches and TLB similarly to their data counterparts. We also attempt to detect the replacement policy employed by each cache and distinguish be-

[1] The effects of phyically and virtually addressed caches are discussed in more detail in Section ?.

[2] A cache with only a single index is fully associative.

```
JMP_BLOCK __asm( "add %o0,%o1,%o0 \n\t jmp %o0 \n\t nop" )
NOP_BLOCK __asm( "nop" )
RET_BLOCK __asm( "jmp %o2 \n\t nop" )
```

Figure 1: Instruction probe primitives for the Sparc architecture (C defines).

tween shared and independent instruction and data caches.

Instruction Caches and TLBs. To probe an instruction cache, it is necessary to construct blocks of code of *no-ops* interspersed with *jump* instructions at the correct striding distance. To achieve this as portably as possible, we have defined a small number of primitives that must be supplied in assembly for a given architecture. Figure 1 shows an example set of primitives defined for the Sparc architecture.

Sharing In addition to detecting instruction cache and TLB properties, our benchmark attempts to detect when separately detected data and instruction caches are actually a single shared cache. This is accomplished by comparing the separate access times with a third test that interleaves data reads and instruction executions to detect whether conflicts indicate a shared cache.

Replacement Policies. When a new cache line is allocated, space must be made by evicting an existing cache block from the same associative set. The decision of which block to evict depends on the replacement policy. Three standard policies in use are Least Recently Used (LRU), First In First Out (FIFO) and random replacement. In the case of LRU, the block that was most recently accessed is most likely to remain in the cache. For FIFO, the first block loaded will be freed no matter how often it has been accessed. The previous benchmarks made assumptions about the replacement policy of the caches. Often, LRU is assumed [2, 3]. Enyou and Thomborson included FIFO [4] as well as LRU, but none of the benchmarks attempted to distinguish which replacement policy was in use. In reality, many of the caches the we investigated use a pseudo-LRU or random replacement policy. Our benchmark attempts to detect the implemented policy by performing three separate tests which remain within the associativity or break the associativity to varying degrees.

2.3 Portability and Coverage

An important goal for all the benchmarks discussed so far has been portability. Designing a benchmark that is specific to a particular architecture by writing in assembly or taking advantage of profiling timers may be easier and possibly give better results, but it necessarily limits the usefulness of the benchmark. Our benchmarks are written in C, but as discussed previously, we do define a few primitives that are written in assembler in order to perform instruction cache tests. However, we have supplied primitives for each of the architectures that we have tested and an automated configure script includes the appropriate set for the target machine.

Although Saavedra and Smith ran their benchmark across a number of architectures and operating systems, their sample systems are somewhat out-dated. Thomborson et al. report results from only the Intel pentium class processors running Windows NT [4] and Linux [3]. Table 1 lists the range of architectures and operating systems that we have tested. We found that each new architecture posed new problems and that much of what we learned came from overcoming these sometimes small yet dramatic differences. Different operating systems on the same architecture posed other problems, such as an optimization in Linux that caused all accesses to hit in the L1 cache no matter how large a piece of memory we covered. These interesting issues and lessons learned are discussed in further detail in Section **??**.

2.4 Inline Analysis

The key contribution of this paper is the inline analysis. All previous work present results as an intuitive analysis of graphs produced by the benchmarks. The primary goal driving the design of our benchmark suite was to provide an automated tool which probes a target machine and returns the parameters of interest. This turned out to be much more difficult than it first seemed. The two primary obstacles were overcoming noisy data and converting our in-

Arch	Processor	OS
Intel	Pentium 133	FreeBSD 3.3
Intel	Pentium Pro 200	FreeBSD 3.4
Intel	Pentium III 500	FreeBSD 3.4
Intel	Pentium II 400	Linux 2.2.17
Intel	Pentium II 350	Solaris x86 5.7
Sparc	Turbo 170	Solaris 5.7
Sparc	Ultra	Solaris 5.7
Sparc	Ultra IIi 333	Solaris 5.7
Sparc	Ultra IIi 333	Linux 2.2.12
Alpha	21064	DEC Unix

Table 1: Architectures and operating systems tested.

tuition into a strict analysis. In the current implementation, much of the analysis consists of heuristics used to detect expected transistions or patterns in the access times. We found that through careful coding and hand-tuning of probe loops that we could return fairly consistent results that compared well with built-in profiling timers available on some architectures.

2.4.1 Timing

Although the approch to timing memory accesses seems straight-forward, there are some subtle problems that must be addressed to ensure that the measured times truly reflect the underlying phenomemon being explored. In this section, we discuss three common timing issues we faced while designing our benchmarks: timer granularity, effects of loop design and control overhead.

Granularity. Events being measured are on the scale of nanoseconds, but UNIX standard time functions like **gettimeofday()** operate with a resolution of microseconds. To work at course granularities, it is necessary to average the result over a large number tests. This requires the use of loop constructs that repeat for some minimum time that is measurable. Loop counts that are too low result in skewed times due to the overhead of the timing calls themselves. On the other hand, counts that are two high cause the benchmarks to become unnecessarily sluggish. Appropriate loop counts are a function of the number of accesses and the speed of each access. In their design of the mhz benchmark to determine a processor's clock speed, Staelin and McVoy [5] introduce a novel method to automatically scale the loop count to ensure an acceptable error range. Currently, we use constant counts determined from experiment. However, to gain confidence in our precsion, we have compared our resultant times with those from high precision profiling registers in the Intel architecture.

Loops. Other benchmarks implement a single loop with a single high level operation which repeats once for each access. To overcome compiler optimizations, Thomborson et al. actually add the accessed value to a local variable. Their assumption is that the compiler will not optimize out the instruction, as it might do with just a repeated read, but will implement the local variable as a register, thus negating the cost of a write on each access. This, however resulted in higher than expected times in their resulting access times[3]. Instead, we chose to compile our benchmarks without compiler optimizations (-W0). Furthermore, we coded a series of 32 accesses per loop. We found that in some benchmarks, tight loops caused odd pecularities in timings possibly due to failed speculative execution and branch prediction.

Overhead. With compiler optimizations disabled, the overhead due to the loops themselves becomes apparent. For each loop iteration, the loop control variable must be incremented. If array accesses are used, e.g. **a[i]**, then incrementing the loop count is all that is necessary. However, this fixes the stride based on the size of the element type, one word for an **int[]**, and could actually cause increased overhead if irregular strides are required since an array access implicitly requires and offset calculation based on the index. We chose a more flexible design using pointer arithmetic which allows for variable strides and has no "hidden" costs. It does, however have an explicit cost of incrementing the pointer by the stride for each access as well as the loop count to ensure enough accesses are performed, and without the compiler optimizations this became noticable. The solution was to declare all control variables with the

[3] Results for a Pentium III 500 showed cache hit times near 60 ns. For a comparable system, our benchmarks report 3-4 ns.

register qualifier in decreasing order of sensitivity. The success of this optimization is dependent on the architecture and the compiler. On RISC architectures this results in most, if not all, of the variables used between time calls residing in registers. For systems with fewer general pupose registers, e.g. i386, the primary variables such as the memory pointer and stride are held in registers, but some less referenced values are not.

Control Times. As an absolute measure of control overhead, we also implement a version the access loop that differs only in the fact that it does not access the memory in the array. This provides a means to calculate the actual time required to access the cache or TLB as the measured access time less the control time. The control times also serve as a measure of the system noise since they should remain constant as the stride changes for a given test. This noise level becomes an important tool in determining which changes are significant when detecting transitions in the access times.

2.4.2 Transitions

Inspecting the range of resulting timing graphs, we have classified the expected transitions into three distinct classes: single, stair-step and plateau. For each class, we have developed a number of heuristics enabling us to detect the appropriate transition point or points. Here, we introduce each class of transition and their associated heuristics. Concrete examples and limitations are explored further in the discussion of the implementation in Section 3.

The heuristics were developed through experimentation over a range of architectures and operating systems, and it always boiled down to a problem of deciding which changes were "big enough" to be considered a transition. To aid in weeding out system noise, we compute a lower *bound* equal to twice the standard deviation of the control times. We also employ smoothing and spike detection algorithms to exclude extreme outliers from the analysis.

Single Transitions. For properties such as associativity, we expect a graph with a single transition. Access times begin to increase as the stride shrinks causing more blocks to be accessed at a single index than the associativity allows. At some point, enough blocks will cause a capacity miss on every access and the access times will stabalize at a maximum. The steepness of this transition depends heavily on the replacement policy of the cache or TLB, so we choose the first substantial increase between two adjacent points. When detecting TLB page sizes, we actually search for the largest single transition throughout the entire graph, but we exclude transitions that are part of a spike where the point following the transition drops again by half.

Stair-Steps. When detecting cache sizes, it is possible to encounter multiple transtions in a single graph, one per cache level. Thomborson et. al actually disabled the L2 cache and focused on only a single cache. Saavedra and Smith only encountered one system with two cache levels. All of the systems that we experimented with contained two caches. We detect multiple cache levels by identifying any number of increasingly larger transitions. The control bound is used to distinguish the first transition from normal system perterbance. Then, subsequent transitions must be greater than one third the previous transition to be considered.

Plateaus. Benchmarks to determine cache block size and the number of TLB entries result in graphs with a characteristic plateau. As the stride is increased, a larger percentage of accesses cause misses. When the correct stride is reached, each access should cause a misss and all strides larger should continue this trend, thus creating a plateau at the end of the graph. Our heuristic is composed of three key steps. (1) The data is smoothed using a typical function, $\gamma_{t+1} = \gamma_t(1-\alpha) + x_t(\alpha)$ with $\alpha = 0.15$. A lower bound is chosen as halfway between the minimum (min) and maximum (max) of the smoothed data, and only data points above this lower bound are considered. (2) A variance (v) is caculated using a decremented error margin (e) as $v = (max - min) * e$. The error margin begins at 0.16 and decrements by 0.02. (3) The average difference (d) between the candidate point and the next three points is computed. The candidate point is awarded a vote if the change on the left is at least twice the variance, v, and the future difference, d is less than the variance. Finally, the point that receives the majority of votes over the range of error margins is chosen as the start of the plateau.

3 The Tool

Rather than a run-once experiment, *MOB* was designed as a convenient, portable tool similar to *lmbench* [6], a popular benchmark suite which tests data transfer rates. New benchmarks can easily be added to the *MOB* suite to probe other properties of caches and TLBs, such as write buffer depth or multiple page sizes, as well as possibly other components of the memory sub-system. The main executable, **inspector**, is run with flags indicating which benchmarks to perform, the number of trials per test, the length of each trial as well as the verbosity level.

At the end of each run, a human readable/editable output file is generated with a separate section for each component detected. Properties of each component are listed in key/value pairs. This output file can then be used as input for further benchmarks. Strict benchmark dependencies are defined, and parameters needed by one benchmark can be supplied through the input file. If necessary parameters are not available then the appropriate benchmarks will automatically be run to determine those parameters. A sepcial verbosity level is provided which produces separate output that can be parsed by a supplied script to produce graphs through *gnuplot*.

The remainder of this section evaluates *MOBs* performance, expanding on the previous discussion of benchmark techniques. We explain the in-line analysis in more detail and discuss some of the limitations found in our methodology and heuristics.

3.1 Cache Benchmarks

In the next paragraphs, we present our the implementation of our tests, based on the already explained techniques in section 2. We detail our choice of parameters such as stride and memory sizes, define the limitations that the tests may have and explain how the heuristics work for each of the tests' data.

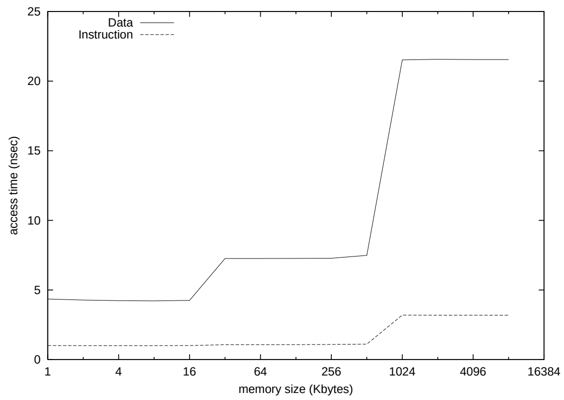

Figure 2: Pentium-III 450MHz with 16K/512K L1/L2 caches

3.1.1 Levels and Sizes

In this test, the stride is fixed to be the size of an integer and the size of a *no-op* for data and instruction caches respectively. With this stride then we access several memory sizes starting at 1Kbyte and up to 8 Mbytes in increments of powers of 2. The analysis of this test uses the stair-step function. Each of the base point of the steps indicate the each of the cache sizes. As seen in figure 2, until 16Kbytes the access times are low until pas the 16Kbytes mask, which indicates that we have hits at the same cache level. Once we try to access 32Kbytes, the times increase meaning that we start having some misses which indicates that the cache cointainment boundary is broken. The same exact thing happens after 512Kbytes where level 2 cache needs to start fetching lines from main memory.

NEED to say about probing in between...

The main limitation of this test comes from the fact that only sizes that are powers of two are probed. For cache sizes that are not powers of two the suite would be incorrectly rounding down the size to the nearest power of two. For the same reason, systems with two consecutive caches levels whith sizes that are consecutive powers of two, would be detected as one cache since it would appear as a single but higher step transition.

3.1.2 Sharing

For the sharing test we use the same stride value as in the levels and sizes test and we also fix the memory size to the cache size we're probing. Three different values are used for the analysis: *a)*the average time to read that size, b) the average time to execute as many *no-ops* that fit within that size and *c)* the time resulting from interleaving *a* and *b*. The analisys compares whether *c* is larger than *a* and *b* combined.

In order to compare data acces and instruction execution times, we also scale the execution time values. The scaling is based on the size difference between the data stride (integer) and the *no-op* sizes. For example in the Intel architecture the integer and *no-op* sizes are 32 and 8 bits respectively, so the test will scale the time to execute 1 *no-op* (1 byte) by 4 so the time reflects how much time does it take to execute the same amount of memory as a single data access.

To account for noise, a cushion is used to determine if the difference is relevant. A heuristic bound of one third of the largest individual test is used. A cache will be considered shared when *c* is larger than *a* and *b* plus the cushion.

3.1.3 Block Size

This benchmark uses fixed memory size corresponding to the cache size we're probing. That memory is then run in different strides, rangeing from one wordup to 128 bytes. In the case of the instruction caches, the minimum stride size we can use is constrained by the size of an individual jump block.

Figure 3 shows the test output for an UltraSparc CPU with 32 and 64 bytes block size for level 1 and 2 caches respectively. As can be seen for level 2 cache, as the stride gets closer to the appropriate block size, the times increase since there are less hits per fetched block. Once it reaches the real block size, every access hits a new block thus causing a miss every time. The analysis uses the plateau detection function to get the data point at the beggining of the plateau and select it as the appropriate block size for the cache.

In caches that implement sub-blocks, it is possible that the test detects the sub-block size instead of the block size since it is the unit of replacement. In Figure 3 the effect of 16 bytes sub-blocking strategy of the level 1 UltraSparc data cache can be observed.

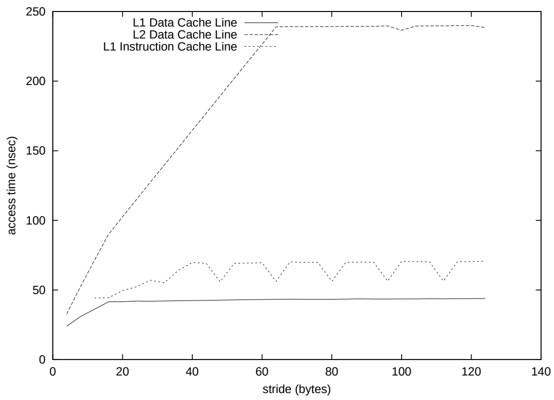

Figure 3: UltraSparc-II with 32/64 L1/L2 block sizes.

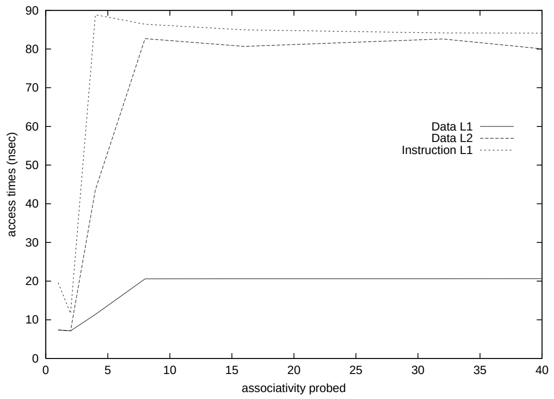

Figure 4: Pentium-III 450MHz with 4-way assoc. caches.

3.1.4 Associativity

For this test, the memory size used is fixed at twice the size of the cache. The stride size varies with the associativity tested, and it is equal to the memory size divided by the associativity probed. For example, for a cache size of 16 Kbytes, and a probed associativity of 4, the benchmark would cover a total size of 32 Kbytes of memory in 4 Kbyte strides. Note that this would cause 8 (4*2) blocks to be read at a single index rather than 4 (the probed associativity). This is because the test is designed to break the real associativity of the cache rather than fit within on the probed points. For this reason, data point 2 in Figure 4 shows that trying to fit 4 (2*2) blocks per index didn't cause set containment misses, so the cache is at least 4-way associative.

The analysis of the data uses the point at the first transition to as the right cache associativity, so in the previous example, all caches have the first transition at data point 4 so they all are 4-way associative.

An inherent limitation is that the test wouldn't be able to correctly detect an associativity that's not a power of two. After some experimentation we could see that random and pseudo-random replacement policies produce much less abrupt transitions which was very difficult to detect correctly. That was the reason to use only powers of two.

3.1.5 Replacement Policy

This benchmark tries to decide between LRU and random as the replacement policy of the probed cache. The stride is fixed at cache size divided by its associativity such that uses the same index in all accesses.

Three separate test are performed: *a)* Accesses the cache within its associativity, so no set containment misses occur, *b)* Breaking the associativity by its double and *c* Breaking the associativity by a factor of $1\frac{1}{2}$. In a LRU system cases *b* and *c* would be identical since we're always replacing the oldest accessed set. For a random replacement system, test *c* should cause about half as many misses as test *b*. The analysis considers the cache to be an LRU policy if test *c* is greater than the middle point between *a* and *b*. Otherwise, the system detects it as random.

The main limitation of this test is that not pure LRU but pseudo-LRU policies are widely used in current processors, plus the fact that random policies are really pseudo-random, makes the distinction very difficult to make.

3.1.6 Write Policy

The size of the memory used for this benchmark is set at twice the size of the cache we're probing. Both tests are setup to overrun the L1 write buffers and force writes to go to the next level. The stride size for all tests is fixed at the block size of the probed cache to avoid merging writes in a single write buffer retirement [**?**]. Two different tests are performed such that exploit what we call the best and worst case scenario for any write policy combination.

The first test (best case) is aimed to distinguish between a write-back and a write-through policy. Only the first of the segments is used. The segment is primed with good data first, and then repeatedly written with the same data. For a write-back cache, all writes should be directly modified in the cache, without extra allocation or fetching from the next level, so the average time should be close to a 'hit' time. On the other hand, in a write-through cache every write should pay the price of being written through to the next level, thus getting an average time close to a 'write' time. Since a test in a system would only give us one value, the problem is to classify it as a 'hit' or a 'write' time. We can approach this problem by performing read-access tests and getting an approximation of a hit time as a comparison basis.

The second test (worst case) is designed to see if either a line is fetched on a write miss or an already allocated block is reused. We'll call these two policies fetching and non-fetching, which match the fetch-on-write and not-fetch-on-write policies in [**?**]. In the test, entries of the two segments are repeatedly written causing a write miss on every access. In the case of a fetching policy, in both write-through and write-back caches, every write would cause both a write ('write' time) and a block to be brought from the next level ('miss' time) . In the case of a non-fetching policy, only a write to the next level would be needed. This brings again the problem of distinguishing between a 'miss+write' and a 'write' times. In the case of a write-through cache, we can use the 'write' time from the first test as the gauge for comparison, but for write-back caches other reference times would be needed.

3.2 TLB Benchmarks

The benchmarks that the suite currently has detect: the main page size of the architecture, the number of entries for each detected TLB, their associativty and wether they are shared or separate between instruction and data. Even the TLB benchmarks share most of the already explained ideas, there are some new issues that we don't find in cache benchmarking. Figure 5 show the output data for the page size test in a Pentium Pro system running FreeBSD.

Virtually indexed caches systems are able to access the cache and the TLB in parallel in such a way that even if there is a TLB miss, it could be masked by the cache hit. Physically indexed caches always need to acces the TLB before the cache indexing can proceed

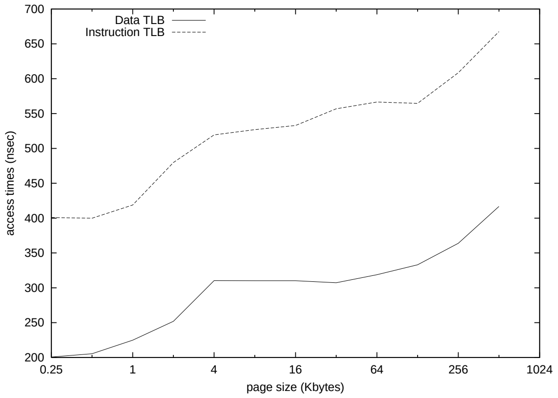

Figure 5: Pentium-Pro 200MHx running FreeBSD-3.4 with 4Kbytes pages

so a TLB miss can never be masked by a cache hit. In order to be independent from the type of cache indexing each access for the TLB tests are designed to cause a cache miss so the masking can never occur.

Due to the restrictions imposed by the underlying cache characteristics the test range may split into different segments. This is done to avoid detecting transitions resulting from effects other than TLB misses, such as cache or set containment breakage.

4 Discussion

5 Related Work

Benchmarking is not a new concept. In fact, there are numerous benchamrks already available to probe properties and evaluate the performance of most of a computer's sub-systems. New benchmark designs leverage existing techniques, improving the efficiency, accuracy and useablility of the software.

A popular benchmark suite, *lmbench* [6], evaluates a system's ability to transfer data between the CPU, memory, disk and network. It improves upon existing benchmarks such as *IOBench* and *IOStone*. Netperf provides a number of benchmarks to measure network throughput and end-to-end latency. Other popular benchmarks, such as SPEC and TPC, focus on application performance, offering a representative workload for a specific class of applications.

Saavedra and Smith [2] introduced the first benchmark which probed a computer's memory hierarchy to detect properties of the caches and TLB such as the size and associativity. Thomborson et al. [4, 3] expanded on this work, separating read accesses from writes and extending the range of address mapping functions to include random accesses. With *MOB*, we have taken this work a step further. Our benchmark suite includes a full set of tests for instruction caches and TLBs, is not limited to a single level of cache hierarchy and provides inline analysis with useful conclusions rather than leaving the user to her own intuition.

6 Conclusions

MOB is a portable and extensible suite of benchmarks which probe the memory system of a computer and automatically detect many properties of the caches and TLBs. The primary contributions of this work include:

- introducing novel benchmarks including a series of tests to identify properties of instruction caches and TLBs where previously only benchmarks for data accesses have been explored
- experimenting over a range of modern architecture and operating system combinations
- implementing inline analysis of benchmark data
- providing a portable, effective tool to the community[4]

Through extensive testing, we have found that the accuracy of individual tests changes across architectures and operating systems. We believe that there is still work ahead of us to better account for many of the complex interactions, like speculative execution and pre-fetching, which affect the memory subsystem in unexpected ways. In the near future, we plan to expand the benchmark suite to test for other properties of the caches and TLBs. For example, we believe it is possible to detect whether each cache is indexed by the virtual or the physical address. We will continue to refine our measurement techniques to reduce anomalous readings and in turn improve the inline analysis. Finally, we would like to investigate more rigorous analysis techniques rather than depending on heuristics based on intuition.

[4]The URL to access the code has been elided to maintain anonimity.

7 Acknowledgments

The authors wish to thank Anurag Acharya for his motivation and insight.

References

[1] B. L. Peuto and L. Shustek, "An instruction timing model of CPU performance," in *Proceedings of IEEE 4th Annual Symposium on Computer Architecture*, (Silver Spring, Maryland, USA), Mar. 1977.

[2] R. H. Saavedra and A. J. Smith, "Measuring cache and TLB performance and their effect on benchmark runtimes," *IEEE Transactions on Computers*, vol. 44, pp. 1223–1235, Oct. 1995.

[3] C. Thomborson and Y. Yuanhua, "Measuring data cache and TLB parameter under Linux," in *Proceedings of Summer Computer Simulation Conference (SCSC'00) (to appear)*, (Vancouver, CANADA), July 2000.

[4] E. Li and C. Thomborson, "Data cache parameter measurements," in *Proceedings of IEEE International Conference on Computer Design (ICCD'98)*, (Austin, Texas, USA), Oct. 1998.

[5] C. Staelin and L. McVoy, "mhz: Anatomy of a micro-benchmark," in *Proceedings of USENIX '96*, (New Orleans, Louisiana, USA), June 1998.

[6] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proceedings of USENIX '98*, (San Diego, California, USA), Jan. 1996.