Using Microbenchmarks to Evaluate System Performance

Brian N. Bershad
Richard P. Draves
Alessandro Forin

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

1 Introduction

It has become nearly impossible to write a paper about anything in operating systems without including some discussion of performance. Usually, the performance section concentrates on a few "microbenchmarks" which demonstrate that whatever it is which is being described in the paper can, has been, or might be, efficiently implemented. For example, the time to execute a null remote procedure call, a system trap, or to access a page of a mapped file have all been used at one time or another to show that the system implementing the function is either efficient (if it was built by the authors) or inefficient (if it was built by anybody else).

[0]This research was sponsored in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035 and in part by the Open Software Foundation (OSF). Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award. Draves was supported by a fellowship from the Fannie and John Hertz Foundation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF, the Fannie and John Hertz Foundation, the NSF, or the U.S. government.

Two implicit assumptions underlie the use of microbenchmarks. First, it is assumed that the time required for the microbenchmark to exercise the code path in question is the same as it is when the code path is used by real programs. Second, there is the assumption that a microbenchmark is actually representative of something which is either important in its own right, or which has a measurable impact on overall system performance. In this paper we point out the vulnerability of the first assumption by showing the significant variation that can occur with even simple microbenchmarks. Identifying weaknesses in the second assumption is something best done on a case by case basis.

2 The Root of the Problem

As Shakespeare once probably said, "the fault, dear Brutus, lies not in our stars, but in our memory systems." High performance RISC processors live or die as a result of their memory system. Therefore, so do many microbenchmarks. In particular, (at least) two components of memory system design, the cache and the write buffer, must be considered when using microbenchmarks.

2.1 Cache Effects

The behavior of cache memory can distort the performance of a microbenchmark. Caches permit rapid access to memory locations which have been recently referenced. Cache collisions occur when a sequence of references resolve to the same cache line. The RISCier the processor, the simpler the cache structure, and the more vulnerable it is to collisions. The simplest caches, which are small (less than 64K) and direct mapped, present the greatest problem for microbenchmarks since they have little protection against collisions.

Cache collisions can occur between memory in the same address space, or between memory in different address spaces. In the first case, the performance of a particular instance (that is, a linked binary) of the microbenchmark can be stable, but the performance of different versions of the same program can actually be quite variable. In the second case, when collisions occur between memories in separate address spaces, such as between the kernel and an application, then performance may be variable even from run to run, depending on how the system has allocated memory.

When microbenchmark performance is used to tune pieces of the system by changing code and rerunning the microbenchmark to evaluate the change, the variability caused by cache effects can cause the wrong conclusions to be reached. It becomes difficult to say whether performance changed because of the tuning, or, say, because the linker assigned different addresses to object files.

2.2 Write Buffers

Many RISC processors have write buffers which are several entries deep. The write buffer allows the processor to proceed while previous stores complete asynchronously. Once the write buffer fills, however, subsequent stores slow the processor to memory speed. Often, microbenchmarks are structured to eliminate side effects which would normally occur during real applications of the code being tested. For example, a test to evaluate the performance of spinlocks might only acquire and release a lock in succession through a loop, rather than doing any real work (such as modifying a shared variable) inside the loop. Depending on the depth of the write buffer, a write stall could be avoided because nothing was computed inside the loop. In this case, the benchmark doesn't measure the true cost of the stores in the spinlock primitives. Consequently, the microbenchmark overestimates the actual performance of the primitives, since they would never be used in the way they were being tested.

3 Some Real Examples

A potential problem is different than one which occurs in real life. Our experiences with measuring operating system performance at CMU have shown that, in real life, microbenchmarks can be misleading. In this section we briefly describe two situations where microbenchmark results were, or could have been, affected by interactions with the memory system.

3.1 The Null RPC – Cache Interference Within the Kernel

Mach is a "small kernel" operating system which provides support for fast cross-address space communication [Draves et al. 91]. On the DecStation 3100, which is a MIPS R2000-based system running at 16.67 MHz, null RPCs between two address spaces on the same machine typically take about 95 μsecs, or about 1600 cycles.

After one build of the system in which there were no changes to the kernel's RPC path, RPC times increased to about 160 μsecs, or about 2650 cycles. The problem was that the machine-independent IPC code in one object file was colliding in the cache with machine-dependent functions in another object file. The collision occurred because of a change in the size of an unrelated object file. Since cache misses on the DecStation 3100 are resolved in 5 cycles, we concluded that about 215 additional cache misses occurred on the newer system's RPC path. By rearranging the order of the object files for the kernel build, RPC times fell back to 95 μsecs.

If we had used the benchmark to tune the RPC path, to the exclusion of all other measurements, such as number of instructions executed, or number of memory accesses, then the variability could have lead to poor tuning decisions. If the application code itself had interfered with kernel code, then variability would have occurred from run to run.

The best way to deal with the variability problem for tuning is to explicitly flush the cache during each iteration of the trial benchmark, and to then subtract off the cache flush overhead from the measured benchmark time. Although this technique will reveal worst case performance, the performance will at least be reproducible.

Moreover, for tuning, it is important to count the number of loads and stores, rather than just instructions executed or microseconds measured during a run. As processor cycle times decreases, the major determinant of performance is the number of memory accesses which can not be satisfied by the cache. Many of the kinds of benchmarks which arise in operating system performance are stressing components of the system which are executed infrequently enough to make their long term presence in the cache unlikely. Therefore, performance profiles which reflect memory interaction are more likely to be good predictors of actual performance.

3.2 The Simple Spinlock – Failing to Fill the Write Buffer

Mach supports multiple threads of control per address space. Even on a uniprocessor, these threads use simple spinlocks to ensure mutual exclusion in the presence of interleaved execution. Some recent work at CMU focused on improving the performance of test-and-set primitives for architectures which do not have hardware support for atomic read-modify-write sequences [Bershad 91]. A simple microbenchmark evaluating the new locking primitives had the following form:

```
int i;
spinlock lock;

for (i = 0; i < GZILLION; i++)  {
        SPIN_LOCK(lock);
        /* locked code fragment */
        SPIN_UNLOCK(lock);
}
```

On the DecStation 3100, the compiler generated a code sequence in which 8 instructions were executed during each pass of the loop. Within those 8 instructions were two memory stores — one to acquire the lock and one to release it. The DecStation 3100 can complete one store every 6 cycles, but uses a four entry write buffer to allow the CPU to avoid stalling during writes. Multiple writes to the same address are coalesced in the write buffer, so the above loop can proceed at full speed – the effective rate of writes is one per eight cycles.

Unfortunately, the above loop will overestimate the performance of the locking primitives because, in reality, such primitives are always used in conjunction with writes to memory which is shared between multiple threads. Consequently, the above test is optimistic because it includes no writes to memory except the lock itself. If we include a store to shared memory within the loop, we add one instruction to the code path, for a total of nine instructions and two effective stores. Because the DecStation 3100 can retire only one store per six cycles, the augmented microbenchmark takes twelve cycles, not nine.

This example motivates two conclusions. First, one can use microbenchmarks to predict baseline performance only if the measured behavior might really occur. In reality, no multithreaded program would ever acquire and then immediately release a spinlock. There would always be an intervening memory access, and this should be included in the measurements.

Second, understanding the behavior and performance of a microbenchmark may often require understanding subtleties in the architectural implementation (e.g., depth of write buffer, ability to coalesce). It can sometimes be difficult to even discover these details, as they may be considered "confidential and proprietary" by industry.

4 Conclusions

Writing and interpreting microbenchmarks correctly can be difficult. The situation is only going to get worse as the gap between processing speed and memory speed widens. Microbenchmark performance may not be reflective of actual behavior, and even run-to-run results can have high variance. Flushing the cache while running a microbenchmark and counting memory accesses, rather than instructions, are two techniques for reducing the variability of results. Finally, it will become increasingly important to understand low-level details about architectural implementation when interpreting microbenchmarks.

References

[Bershad 91] Bershad, B. N. Mutual Exclusion for Uniprocessors. Technical Report CMU-CS-91-116, School of Computer Science, Carnegie Mellon University, April 1991.

[Draves et al. 91] Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136, October 1991.