Crossroads
The ACM Student Magazine

Optimizing Application Performance: A Case Study Using LMBench

by ***M Tyler Maxwell*** and ***Kirk W. Cameron***

Introduction

In the world of computing, speed is everything. In the consumer and business world, no one wants to use a piece of software that takes minutes to complete its task when competing software can do roughly the same task in seconds. A spreadsheet application that takes even seconds to sum an entire column exhibits unacceptable performance. In the scientific world, time is even more critical; a meteorology program that takes forty-eight hours to predict the weather for twenty-four hours in the future is useless.

We could try running the code on faster computers. However, the execution time of a piece of software is not only dependent on the machine that it runs, but also on the design of the software. By exerting some effort while writing our code, we can possibly gain performance.

Typically, when optimizing software performance we use an optimizing compiler that is proficient at general optimizations. Optimizing compilers have many techniques to speed up execution at the level of the processor, such as encouraging processor pipelining via loop unrolling [8]. Native compilers are designed for specific platforms; they have the ability to perform architecture-dependent optimizations. However, compiler optimizations are not sufficient if we want to optimize to a system's full potential. The performance of a computer is not only dependent on the processor; the overall performance of a system is the result of a synergy between multiple subsystems. Some subsystems, such as the memory hierarchy, experience slower rates of technological growth than the central processor [9]. To ignore these lagging subsystems would be a terrible mistake, because these subsystems might account for a great deal of our performance degradation. Even though compilers can perform specific optimizations for a given instruction set architecture (ISA), it is likely that these other subsystems comprising the computer will be different across various computers of the same ISA. Because of this, compilers often do not optimize for subsystems. We conclude that a given compiler (particularly a GNU compiler) will probably not have enough information at its disposal to achieve the same improvements in performance compared to a well-informed programmer optimizing an algorithm by hand before optimized compilation.

Programmers must design system-specific algorithms to take advantage of subsystems (particularly the memory hierarchy). We need tools to collect empirical data to apply to these algorithms. LMBench is a microbenchmarking suite that we will use to determine the characteristics of a machine's memory hierarchy. After we have made conclusions regarding the memory hierarchy, we will optimize a matrix multiplication algorithm using the information that we obtain. **Figure 1** shows the application timing results from such an approach. A naive matrix multiply algorithm is compiled with all compiler optimizations (1). (All code in this article is compiled with the IA-64 GNU gcc compiler using the −O3 optimization flag.) Curve (2) is the same algorithm compiled with optimizations on, but also using general hand optimization techniques. Finally, curve (3) shows targeted optimization of the memory hierarchy. This implementation uses the general hand optimizations, an optimizing compiler, and hand optimization using information gained from microbenchmarking. Notice that the speed improvement also improves as the data set gets larger. It is exciting that with moderate effort we can speed a code a great deal more than just relying on the compiler to do it for us.

We choose matrix multiplication as an algorithm to target because the working set (data that can be processed independent of other data elements) is easy to identify. In other algorithms, the identity of the working set will be different, and perhaps will be harder to determine. The specific techniques to implement working set optimizations will also be different across various problems. Because of this, hand optimizations across algorithms using the same principle might yield different degrees of speedup.

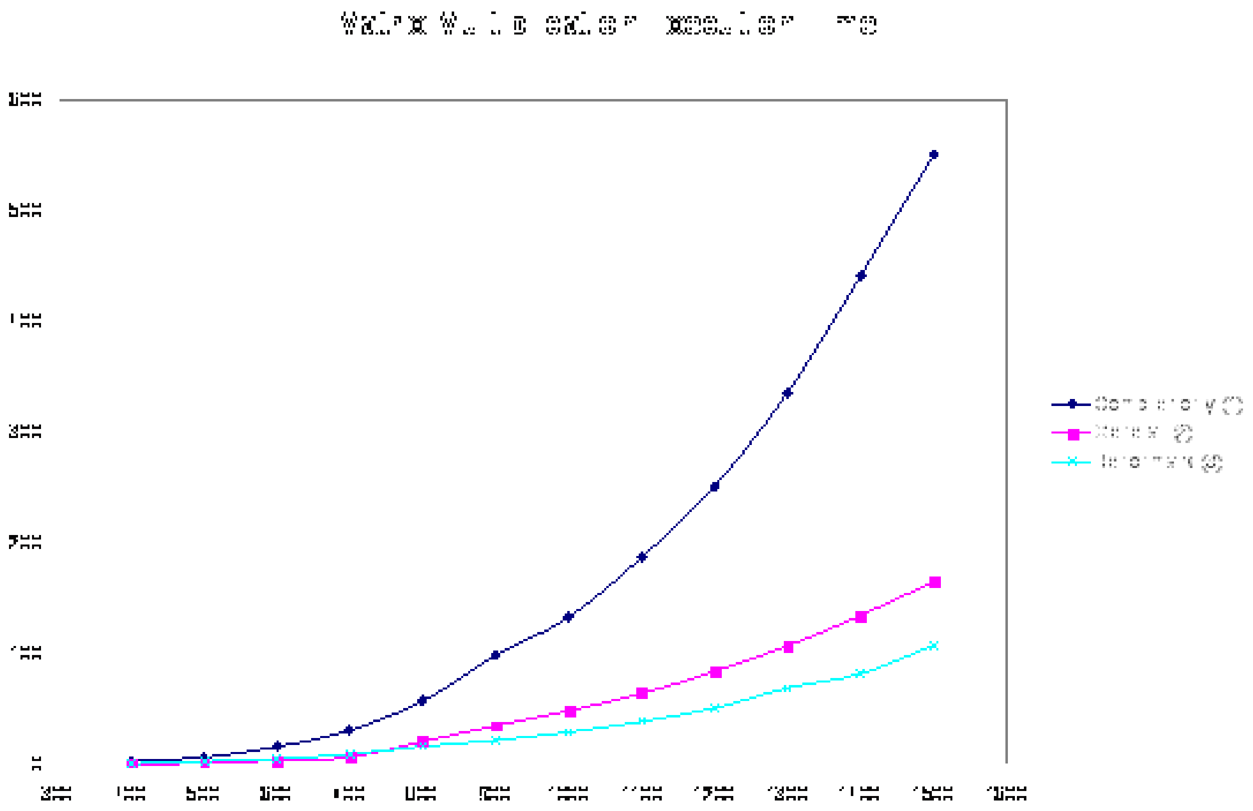


Figure 1: Matrix multiply performance with various levels of optimization.

Background

Memory Hierarchy

The memory hierarchy is a major subsystem that contributes greatly to overall system performance. Since the early 1980's, the rate of speed increase per year for processors has outpaced the speed increase per year of memory. A CPU in a modern computer is small, fast, and expensive, while main memory (RAM) is large, cheap, and comparatively slower. Computer architects have remedied this situation by creating a hierarchy of memories: faster memory near the CPU, and slower memory near the main memory. **Figure 2** shows a typical memory hierarchy. The faster memory near the CPU (called cache) is smaller in capacity and more expensive, while the main memory is slightly slower, less expensive, and therefore has a higher capacity. It is desirable to keep data that is likely to be needed next by the CPU where it can be accessed more quickly than if it had been in the main memory.

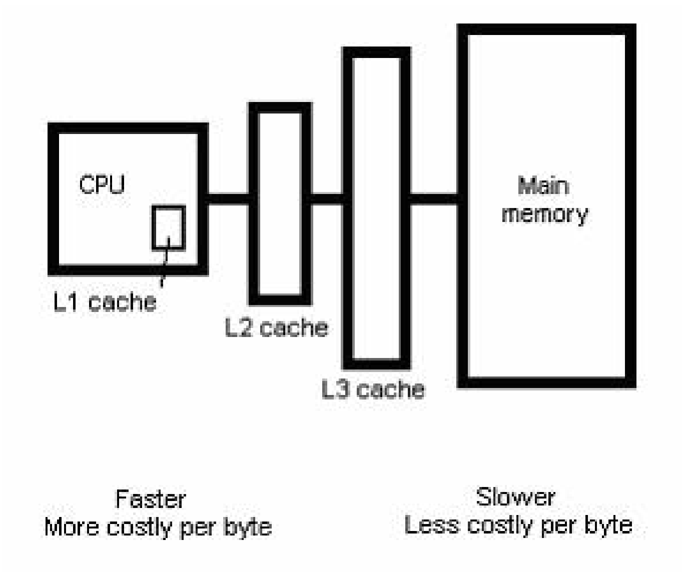


Figure 2: A typical memory hierarchy.

Cache can speed up our software because of a concept inherent in program execution known as **locality**. Locality assumes that if a piece of data has been accessed it is highly likely that it will be accessed again soon (temporal locality) and that its neighbors in main memory will be also accessed soon (spatial locality). If codes exhibit locality, then resulting applications will run faster because the data will be in nearby fast cache [5].

Commodity cache behavior is deterministic; it does not change its behavior for an access pattern inherent in a program. If we write a program that accesses its data in a way that unknowingly defeats the purpose of the cache, we forego a great deal of performance improvement. As programmers it is therefore our responsibility to design our code with the memory hierarchy in mind. For various methods of optimization we require detailed information about the sizes of the levels of cache and the cache's blocking characteristics. Our optimizations will only focus on the sizes of the various levels of the cache, because considering associativity (and hence replacement policy) requires increased understanding of the underlying architecture with minimal impact on performance. Most commodity higher level caches have set associativity (4-way or 8-way) that performs almost just as well as fully associative caches, so we ignore associativity [1].

Microbenchmarking is a good, empirical way to determine cache characteristics before advanced optimization can take place. In this article we will use LMBench to determine the cache characteristics of a quad processor HP Itanium SMP computer. We then illustrate LMBench's usefulness in implementing nearly optimal hand optimization via a technique known as blocking.

Matrix Multiplication

Matrix multiplication is a common operation that can benefit greatly from the efficient use of cache. We can attempt to reduce the asymptotic time complexity of matrix multiplication by using Strassen's algorithm [6] instead of the naive $O(N^3)$ algorithm, but implementing this algorithm is very complex and has many drawbacks. **Figure 3** shows naive code that we would normally create to multiply two matrices. This code is not cache-friendly.

```
for(j = 1; j <= size; j++) {
      for (l = 1; l <= size; l++) {
            for (i = 1; i <= size; i++) {
                  C[i][j] = C[i][j] + B[l][j] * A[i][l];
            }
      }
}
```

Figure 3: Naive matrix multiplication.

At a size of 1000 by 1000 elements the completion time on the IA-64 machine used in this article is 2 minutes and 25 seconds (with aggressive optimizations). Using general cache optimization techniques and the cache-size-specific optimization technique blocking, we reduce the execution time for 1000 by 1000 elements to 30 seconds.

General optimizations applied to naive matrix multiplication include loop unrolling, the use of temporary variables, matrix transposition, and loop reordering. Loop unrolling targets efficiency in processor pipelining. Since a loop is a set of instructions repeated some number of times, at times we may replace a loop by a series of interleaved executions of the instruction block. Optimizing compilers do an excellent job of unrolling loops by determining dependencies and deducing branch evaluations automatically. Because we are interested in optimizing memory system performance, we assume O3 optimization provides gains due to such techniques.

Storing the value of the inner loop result in a temporary variable is also optimizing because the compiler allocates the inner loop value to a constant location in memory (which the cache is likely to store).

```
for(i = 1; i <= size; i++) {
      for (j = 1; j <= size; j++) {
            temp = C[i][j];
            for (l = 1; l <= size; l++) {
                  temp = temp + B[l][j] * A[i][l];
            }
            C[i][j] = temp;
      }
}
```

Figure 4: Naive matrix multiplication with loop reordering.

Loop reordering involves changing the order of the `for` loops so that the array access pattern is optimal. **Figure 4** shows naive code modified using a temporary variable and loop reordering. The algorithm for multiplying two matrices together requires multiplying elements in a given row by elements in a given column. That is, if one matrix is accessed by row element, the other must be accessed by column element. This can be very inefficient because of the way the matrices are stored in memory. Programming languages such as C (row-major) or Fortran (column-major) provide a default mapping of array elements to the linear storage found in memory. A matrix A[i][j] in a row-major language like C may be stored in memory as an array A[i] of pointers to arrays of k length. Therefore, A[i][j] is contiguous to A[i][j+1] and not to A[i+1][j]. **Figure 5a** shows a row-major matrix accessed in an efficient manner. Because the next element needed is contiguous to the current element, the next element needed is more likely to reside in cache. The matrix in **Figure 5b** is not being accessed efficiently because the inner `for` loop goes from B[l][j] to B[l+1][j], meaning successive elements are less likely to reside in cache.

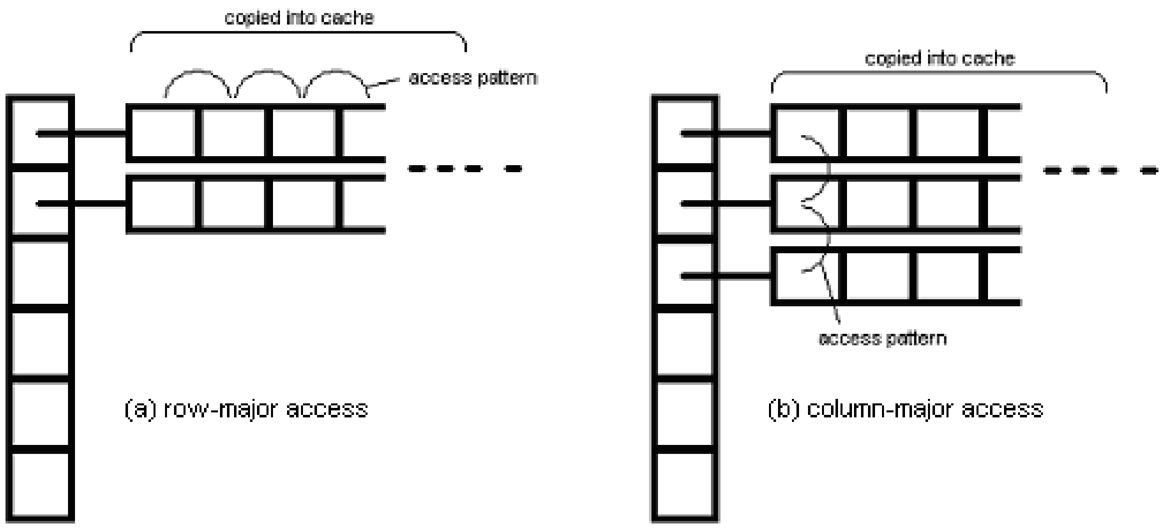


Figure 5: Access patterns in row-major storage.

We can use loop reordering to make one matrix follow a more optimal access pattern. Due to the nature of the matrix multiply algorithm, the access pattern of one matrix will be more optimal than the access pattern of the other. It is useful to transpose the less than optimal matrix so that both matrices follow the optimal access pattern. If y is the variable of the inner loop, then given an indexing scheme for A of A[x][y], we optimize a matrix B by changing the indexing of a matrix B[y][x] to an index B[x][y]. If this optimized matrix is initially input in a transposed manner, then we will maintain mathematical correctness. B[l][j] is replaced by B[j][l] in the code, and the matrix B is input as B^T. (Since the algorithm multiplies a transposed matrix, $(B^T)^T = B$. [3]) The "for" loop will then cause B[j][l+1] to be accessed on the next cycle, resolving the inefficient memory access. We used this technique in the optimized code to further increase speedup.

The algorithm using blocking and the general techniques is shown in **Figure 6**. This code was inspired by code from Hennessey and Patterson [1].

```
for (jj=1; jj<=size; jj+=N) {
    for (kk=1; kk<=size; kk+=N) {
       for (i=1; i<=size; i++) {
              for (j=jj; j<=findMin(jj+N-1,size); jj++) {
                     temp = 0;
                     for (k=kk; k<=findMin(kk+N-1,size); k++) {
                           temp += B[i][k]*A[j][k];
                     }
                     C[j][i]+=temp;
              }
       }
    }
}
```

Figure 6: Matrix multiply with blocking and general cache optimizations.

Blocking involves dividing the matrix size into smaller sub-matrices that are executed iteratively. If we choose optimal block sizes, the sub-matrices (or working set) will fit into cache thereby reducing execution time. The value of N in an N x N sub-matrix should be such that each block, or sub-matrix, is small enough so that three sub-matrices are able to fit into a level of cache during its execution. Knowing the value of N is precisely why the compiler can not typically apply blocking to a code. This information is system dependent, since systems with the same ISA may have various amounts of available cache. Microbenchmarks, small specialized codes designed to measure system characteristics, can determine how large the caches are. Once we collect this empirical data, we can use that information to determine the value of N.

LMBench Overview

LMBench is a suite of microbenchmarks for UNIX that one can download from ftp://ftp.bitmover.com/

lmbench/old/lmbench-2.0.tgz. After downloading the archive and unzipping it to a directory (the root directory is acceptable; it makes a subdirectory "LMbench"), the binaries are produced by entering that subdirectory and typing

```
make
```

This command will compile the source for a particular machine. If one chooses to make modifications to the source code (which is kept in the "src" directory), invoking this command again will cause the modified code to be recompiled. The executables will appear in a subdirectory of "bin," the name of the subdirectory depending on the architecture. Since we are using an IA-64 machine using Linux, a directory called "ia64-linux" is created in the "bin" directory.

The microbenchmark in the LMBench suite that we will focus heavily on, Lat_mem_rd, can be used to determine the sizes of the various levels of cache. It has several other uses, but for now we will simply estimate the cache sizes so that we can estimate an appropriate N. Lat_mem_rd takes two or more arguments from the command line.

```
lat_mem_rd <arraysize> <stridesize1> <stridesize2> ...
```

The argument <arraysize> is given in megabytes, and the values for <stridesize> are given in bytes. An array size that is known to be much larger than the cache (hundreds of megabytes), and various stride sizes that are powers of two, starting at 2^4 (16) and ending at 2^{10} (1024) are a set of values that typically give adequate results. We can adjust these values for larger cache sizes. To run the microbenchmark for multiple strides, we may enter more than one stride. It is also easiest to run this command while redirecting its output to a file. An example execution is shown below, using an array size of 250 megabytes with a stride size of 16 and 32 bytes, and having output redirected to a file `lmr.data`. Because the output is to standard error, we must use the ">&" operator.

```
lat_mem_rd 250 16 32 >& lmr.data
```

When the command finishes executing, the contents of the file will be a list of ordered pairs. At the top of file and between each stride subset will be a comment indicating the stride used for the data set. The first number in any ordered pair is the array size used (in megabytes), and the second number is the load time (in nanoseconds per load). This data should be graphed with the array size on the x-axis and the load time on the y-axis.

LMBench contains more than just Lat_mem_rd in its suite of microbenchmarks. **Figure 7** lists all of the microbenchmarks in the LMBench suite, along with a brief description of what they measure [4].

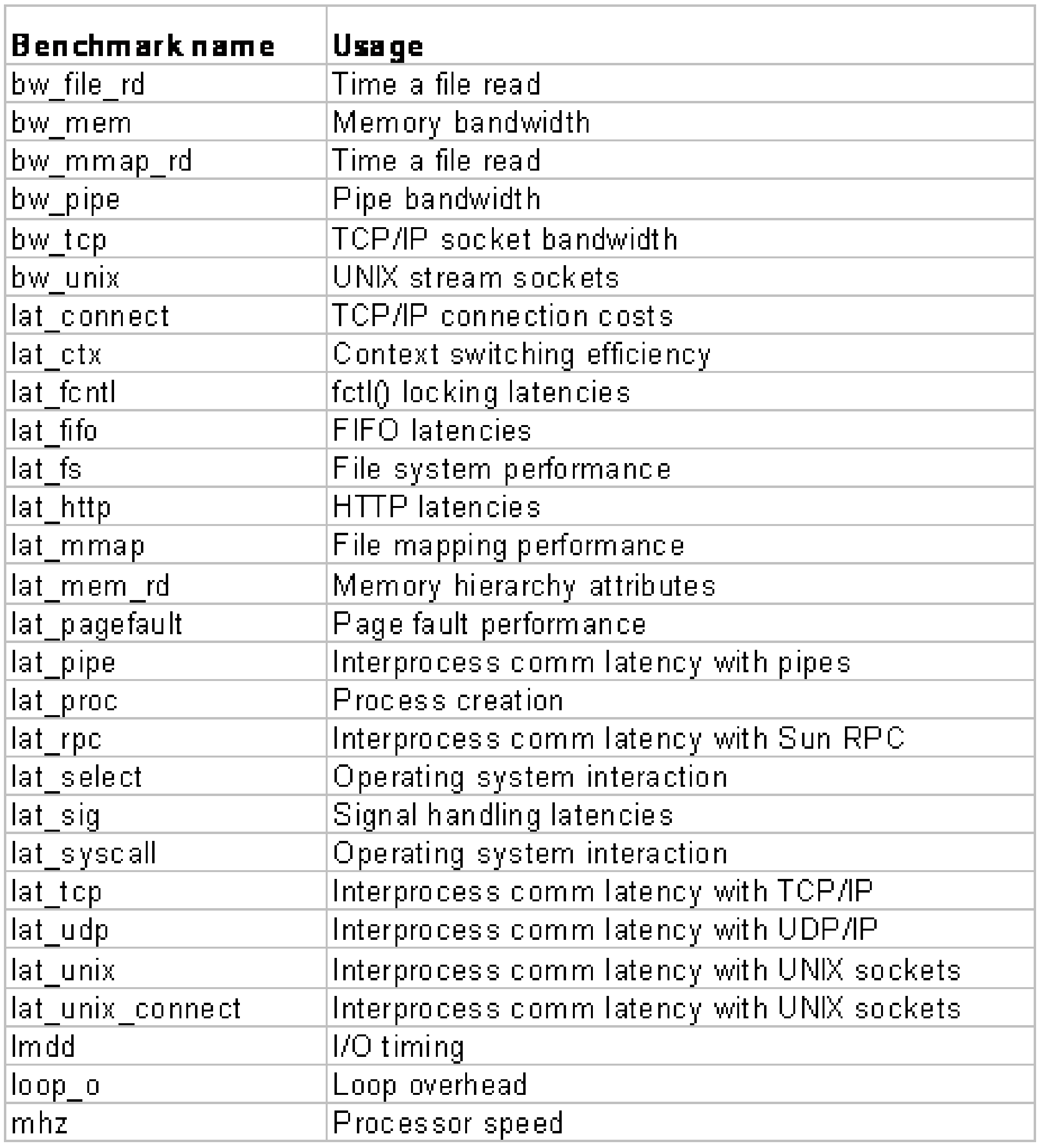
The Bw_mem microbenchmark is another useful microbenchmark which measures memory bandwidth for various memory operations. To specify which operation to run, the arguments in Figure 8 are used at the command line. The simplified command-line syntax of Bw_mem is

```
bw_mem <size> <opcode>
```

The argument <size> is the amount of data to operate on, and the argument <opcode> is the operation code found in **Figure 8**. When giving the <size> argument, ending the argument in "k" or "m" will indicate to the microbenchmark that the size is given in either kilobytes or megabytes, respectively. There is another optional argument affecting alignment that we will not discuss.

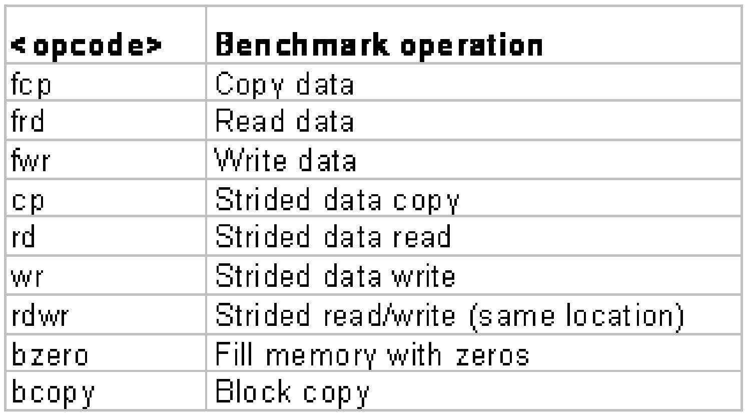
Before running the microbenchmark, every occurrence of the bcopy() command in the Bw_mem source should be changed to memcpy(). The bcopy() command is obsolete on most UNIX systems; memcpy() should be used in its place. The function memcpy() is optimized for the particular architecture on which it has been compiled, so it gives optimal results. Its performance indicates how fast the memory system can copy data.

Bw_mem outputs data in an ordered pair, the first number being the size considered and the second number being the bandwidth in megabytes per second. To convert megabytes per second into the units we used (cycles per byte), one may use the formula in **Figure 9**. The period of the processor is equal to the inverse of the processor speed (in Hertz).



Benchmark name	**Usage**
bw_file_rd	Time a file read
bw_mem	Memory bandwidth
bw_mmap_rd	Time a file read
bw_pipe	Pipe bandwidth
bw_tcp	TCP/IP socket bandwidth
bw_unix	UNIX stream sockets
lat_connect	TCP/IP connection costs
lat_ctx	Context switching efficiency
lat_fcntl	fctl() locking latencies
lat_fifo	FIFO latencies
lat_fs	File system performance
lat_http	HTTP latencies
lat_mmap	File mapping performance
lat_mem_rd	Memory hierarchy attributes
lat_pagefault	Page fault performance
lat_pipe	Interprocess comm latency with pipes
lat_proc	Process creation
lat_rpc	Interprocess comm latency with Sun RPC
lat_select	Operating system interaction
lat_sig	Signal handling latencies
lat_syscall	Operating system interaction
lat_tcp	Interprocess comm latency with TCP/IP
lat_udp	Interprocess comm latency with UDP/IP
lat_unix	Interprocess comm latency with UNIX sockets
lat_unix_connect	Interprocess comm latency with UNIX sockets
lmdd	I/O timing
loop_o	Loop overhead
mhz	Processor speed

Figure 7: LMBench suite.



<opcode>	Benchmark operation
fcp	Copy data
frd	Read data
fwr	Write data
cp	Strided data copy
rd	Strided data read
wr	Strided data write
rdwr	Strided read/write (same location)
bzero	Fill memory with zeros
bcopy	Block copy

Figure 8: Opcode arguments for Bw_mem.

Figure 9: Megabytes per second conversion into cycles per byte.

Experimental Results

HP Itanium Architecture

The machine used in this article is a quad-processor IA-64 Itanium 800Mhz with 8.4GB of shared RAM, running on Linux (SMP #56 version 2.4.18). Each chip possesses 3 levels of cache. The first two levels of cache are on the die and the third level is in the processor cartridge. The L1 cache is split, 16KB for integer data and 16KB for instruction data. The dedicated L2 data cache is 96KB in size, and the L3 cache is unified at a size of 4MB. Most current architectures include memory hierarchies of similar configuration, save for the large L3 cache.

Often, users do not know the exact configuration of the memory hierarchy and thus need to verify or discover it empirically. We verify these cache characteristics through microbenchmarking.

Lat_mem_rd

Before we can interpret and make any use of the data the microbenchmark returns, it is useful to understand how Lat_mem_rd generates these numbers. The microbenchmark allocates an integer array in memory that is <arraysize> in size. Let us suppose that we choose an array size of 250 megabytes. An array of that size means that the entire array will not fit into the cache. The microbenchmark starts at the end of the array, and reads backwards through the array. A pointer chasing routine is used that causes the array accesses to be dependent on one another. The location for the next read is stored in the current read data, creating a dependency that the processor cannot resolve until the read completes fully. Pointer chasing defeats all latency hiding techniques in the cache, such as pre-fetching and block transfers. This assures us that the average memory access time per load is equal to the maximum latency to the level of the cache where the data resides. Because the array size is varied, plateaus will appear in the graph at references to higher levels of cache. By looking at these latency transitions in a graph of the data, we can identify the sizes of the various levels of cache, and the associated latencies.

The distance traversed in the integer array is specified by the stride size. (For a stride size of 1, we would visit every element of the array.) For every stride specified at the command line, a set of data is produced using the method described above. Varying the stride has no benefit in determining the size of the cache levels; rather, it is used to determine the cache block size. This type of analysis is beyond the scope of this article, but we can use knowledge of cache block sizes to enhance cache performance.

We ran Lat_mem_rd on the IA-64 machine, with the following command line arguments.

```
lat_mem_rd 250 16 32 64 128 256 512 1024
```

Figure 10 shows the data obtained from the microbenchmark. We convert the output data from megabytes into bytes, for readability. Notice that there are several plateaus in the graph. The first plateau lasts until about 16KB, the next plateau stops at 96KB, and the third plateau seems to stop at around 1MB, but it slowly continues upward to reach a plateau that lasts beyond the graph. The points at which these plateaus end are the cache sizes for the various levels in the memory hierarchy. The very last plateau is main memory. If we extended the graph to beyond the size of main memory, then we would have another plateau that would represent the paging disk. The behavior becomes more erratic with disk accesses because the behavior of the I/O subsystem is dependent on other system processes. The graphs for the varying strides all have the same size plateaus, because the results of the stride size are only useful for determining the cache block size.

The last cache plateau (next to the last level) increases slowly starting at 1MB and ends at around 4MB. This indicates that the L3 cache is 4MB, but that it is unified instruction and data. That means that the test array the microbenchmark created had to share space with program code, effectively reducing the size of the array that could be copied into that level.

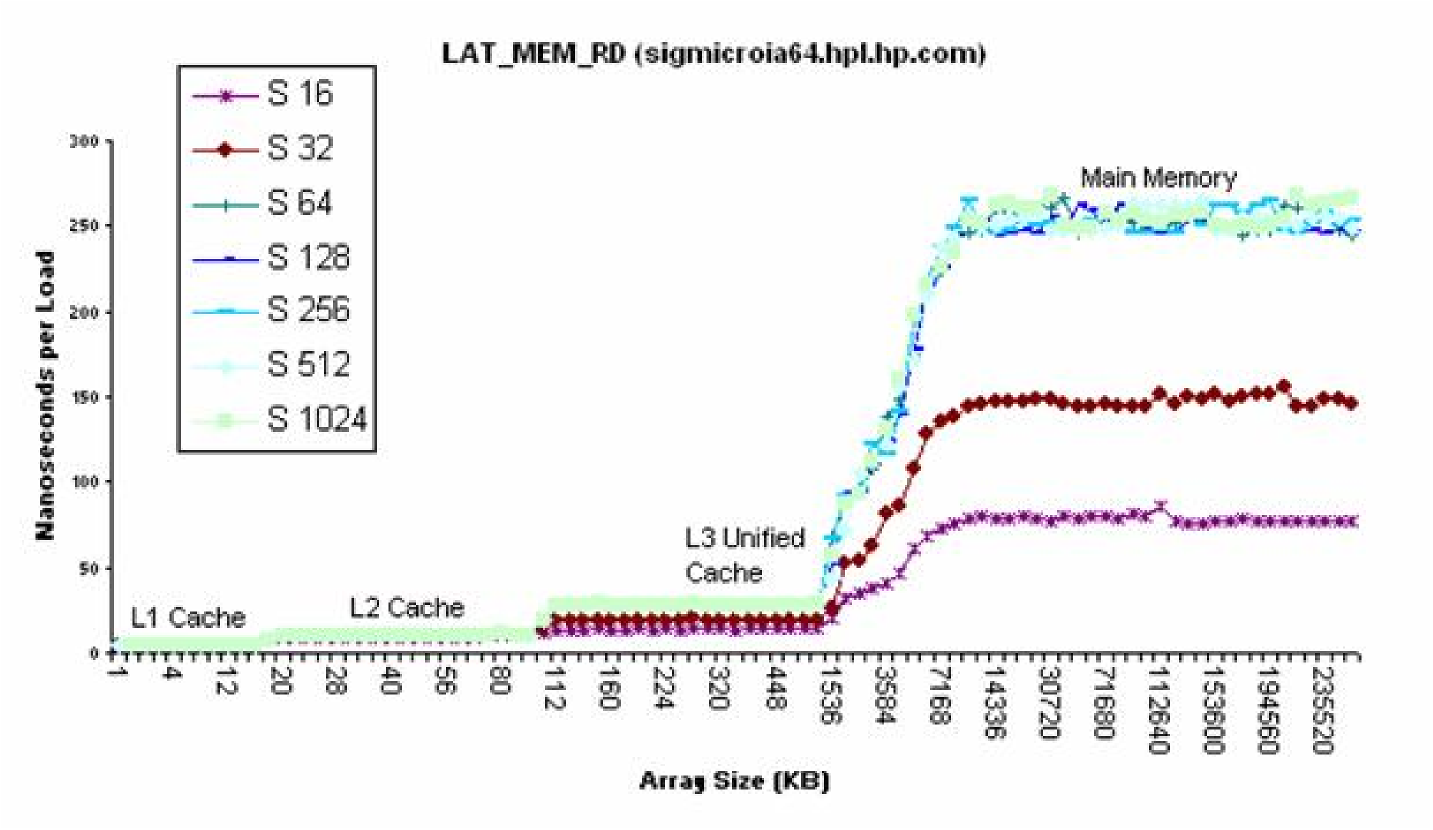


Figure 10: Lat_mem_rd plot.

Determining N

We have just determined the size of each level of the cache. Because the blocking algorithm works best when the sub-matrices are sized to fit in the cache during a sub-matrix execution, we divide the size of the target cache by three, because there are three sub-matrices (A, B, and the resultant C). Which cache should we target? In this case, the L2 cache looks like a prime target because the performance differential between the L1 cache and the L2 cache looks to be much less than the differential between the L2 and L3 cache.

Targeting L2 cache, we are allowed a working set of 96KB. A (N x N) sub-matrix block should be no larger than the cache size. Because we have three matrices that we would like to store in L2, we must divide the cache size by three to find the byte size of a single sub-matrix. Dividing 96KB (96 * 1024 bytes) by three, we obtain 32KB (32,768 bytes). On the IA-64, each element of the matrix occupies eight bytes (the size of a double pointer to a double type). If we divide the byte size of the one sub-matrix by the byte size of one element of the sub matrix, we will obtain the number of elements in one sub-matrix. We divide 32KB by eight and we get 4096 bytes. **Figure 11** shows this general formula for matrix multiplication.

Figure 11: Formula used to compute block size for matrix multiply.

The value that we need for the blocking optimization is N, which is equal to the length of a single sub-matrix. The matrices are square, so we calculate N by taking the square root of the block size (the number of elements in a single sub-matrix). By taking the square root of 4096, we get a value for N of 64.

To verify that 64 elements is indeed an optimal value for N, the algorithm was run for various values of N. This was done for varying sizes of the matrix, to show that matrix size (at large sizes) is independent of the optimal block size. The graph below shows the execution times for the varying runs.

For every size matrix from 800x800 to 1500x1500, the lowest execution time occurs when the value of N is greater than 15 and less than 100. The actual optimal value of N found is 80, but the value that we determined using the Lat_mem_rd microbenchmark is only slightly less optimal. This is because cache size alone is not an absolute predictor of performance; there are other issues that potentially affect the operation of the cache that we did not cover in this article.

If the use of microbenchmarks to find an optimal value of N does not always return the actual optimal result, then why do we not run the matrix multiplication for every possible value of N? To find the actual optimal value might take a long time, since one would need a small granularity and therefore the test would take days for large matrices.

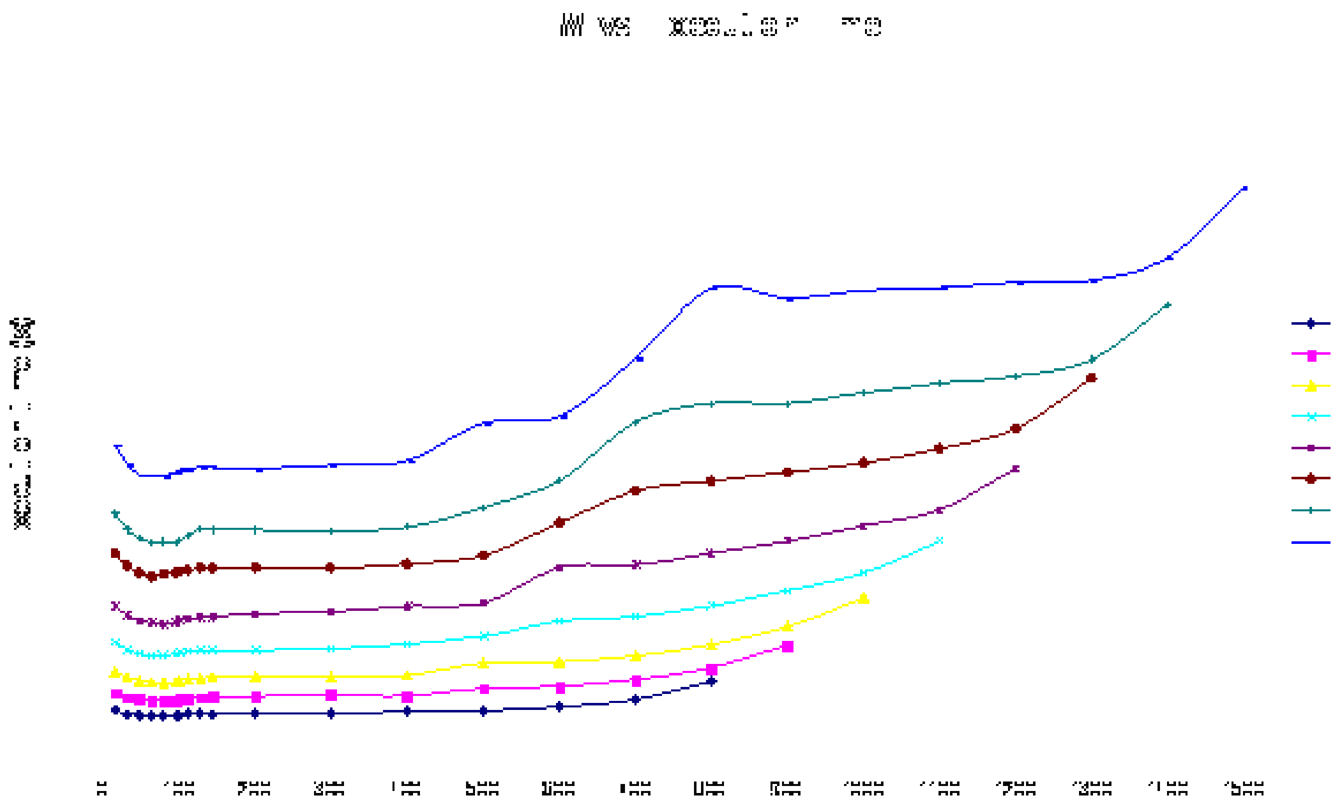


Figure 12: N versus execution time for various matrix sizes.

Simply picking a small value for N is also not feasible, since there is a spike in execution time due to additional overhead when the N reaches a certain value near zero. Microbenchmarking should always provide an adequate guess, and is a useful way to find the optimizing information we need.

How much did we speed up the matrix multiplication over the original? If we graph speedup curves for both the algorithm using row-column ordering only and the algorithm using row-column reordering and blocking, we will be able to see how well we did in our optimizations. **Figure 13** shows speedup as a function of the size of the matrix [7].

Figure 13: Speedup equation as a function of size.

Both curves in **Figure 14** represent the factor in speed increase over the naive algorithm compiled with O3 optimization. (A value greater than one indicates speedup.) The dashed curve is the speedup factor for our general optimizations against the compiler optimized naive algorithm, and the solid curve is the speedup for the blocking optimization. For matrices 800x800 and above, we achieved a speedup of over 2 by using general cache optimizations, which require no specific information about the cache.

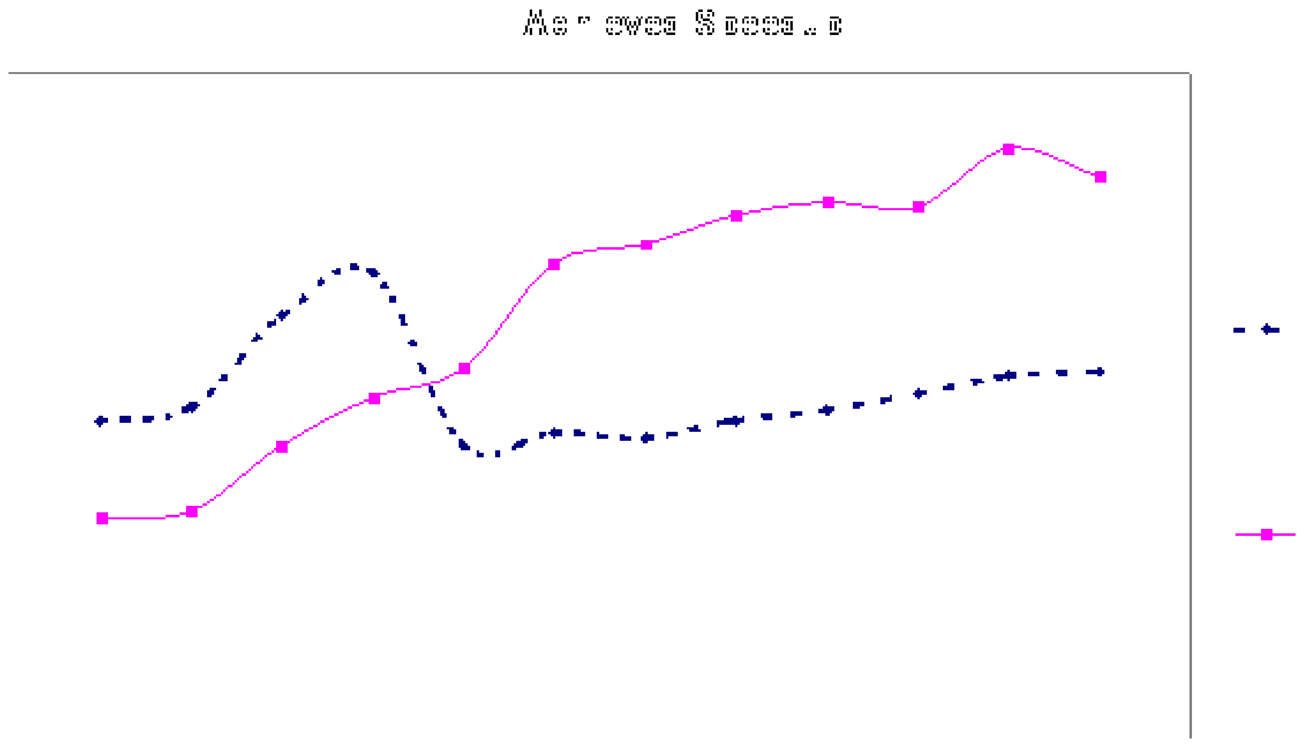


Figure 14: Achieved speedup for both optimizations.

However, by using a simple microbenchmark such as Lat_mem_rd in LMBench, we were able to obtain a speedup of 4 for 900x900 and above 5 for 1500x1500. Blocking is not optimal for smaller matrix sizes because small matrices already fit into levels of the cache; the additional overhead skews performance. Speed becomes more critical as the size of the matrices increase, requiring further demand on the memory subsystem. The speedup over the original seems to get better as the matrix size increases, meaning that a great deal of the execution time growth can be done away with by using blocking in concert with general optimizations.

Bw_mem

Bw_mem was run in a UNIX script [2], with a size starting at 512KB and ending at 1GB. **Figure 15** gives the resulting data series for every operation. Remember that bcopy is actually using memcpy() to perform the block copies, following our modification. Notice how the bandwidth requirements for data sizes near our cache sizes sharply increase. Bw_mem could also be used for determining the cache size, but Lat_mem_rd includes stride options and does not need a script to generate a graph. The graphs that Lat_mem_rd produces are also easier to read for the purposes of determining cache size. Bw_mem provides information useful to applications that perform data streaming, because performance is directly proportional to the optimized platform specific implementation of memcpy ().

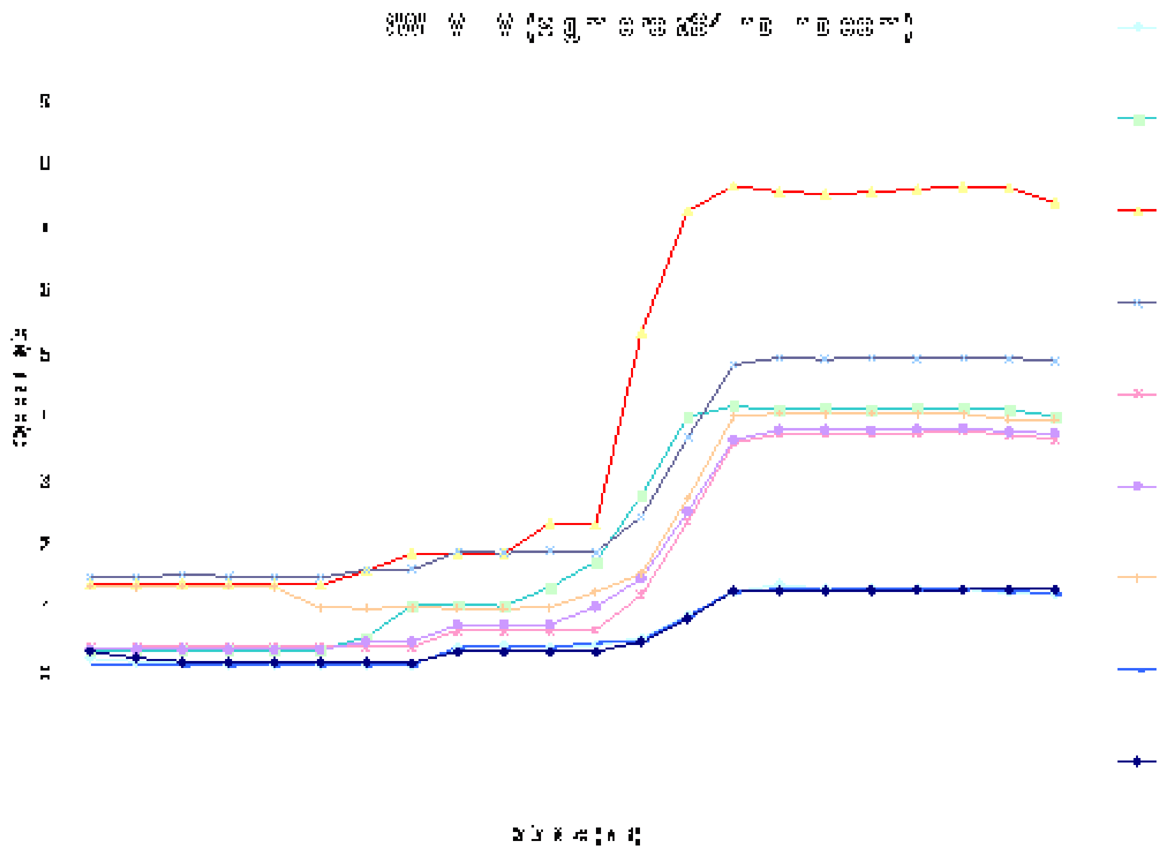


Figure 15: Bw_mem plot.

Conclusions

We have seen how to use a microbenchmark to speed a code significantly. General optimization techniques targeting the given application, such as loop-reordering in our naive matrix multiply, will provide significant performance improvement. More gain can be achieved by adjusting the working set for an application to fit in the cache hierarchy of the target system. We have illustrated the use of LMBench in identifying cache sizes and latencies to improve the performance of matrix multiplication, but our techniques are generally applicable to any application. Our empirical results indicate that memory latency remains a significant contributor to reduced performance on even the most advanced IA-64 based machines.

References

1
Hennessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1996, pp. 373-411.

2
Kochan, S. G. and Wood, P. H. *UNIX Shell Programming*. Hayden Books, 1990.

3
Lay, D. C. *Linear Algebra and Its Applications*. Addison Wesley Longman, 2000, pp. 106-107.

4
McVoy, L. and Staelin, C. "lmbench: Portable Tools for Performance Analysis," *Proceedings of USENIX 1996 Annual Technical Conference*. 1996.

5
D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware / Software Interface*. Morgan Kaufmann Publishers, 1998, pp. 513.

6
Strassen, V. "Gaussian Elimination is Not Optimal." *Numerische Mathematik* **13**, 354-356, 1969.

7
Sun, X.-H. and Ni L. , "Scalable Problems and Memory-Bounded Speedup." *Journal of Parallel and Distributed Computing*, vol. 19, pp. 27-37, 1993.

8
Wall, W. Limits of Instruction Level Parallelism. Paper presented at Conference, 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 1991.

9
Wulf, W. A. and McKee, S. A. "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, 1995.

Biographies

M. Tyler Maxwell (**mtmaxwel@cse.sc.edu**) is a recent graduate from the University of South Carolina, with a B.S. in Computer Engineering. He will be attending the University of Massachusetts Amherst in the fall of 2002, pursuing an advanced degree in Computer Science. His potential research interests include computer architecture, performance evaluation, software engineering, and operating systems.

Kirk W. Cameron (**kcameron@cse.sc.edu**) received his Ph.D. in Computer Science from Louisiana State University in August, 2000. He is currently an assistant professor at the University of South Carolina. His research interests include parallel and distributed systems, scientific computing, performance evaluation, and computer architecture. He has experience working at Intel Corporation in Oregon and on the Parallel Architecture and Performance team at Los Alamos National Laboratory.

Acknowledgements

Portions of this work were supported by ACM SIGMICRO Sponsored Research Award.