Static Performance Evaluation for Memory-Bound Computing: the MBRAM Model

Gene Cooperman[1], Xiaoqin Ma[1] and Viet Ha Nguyen[1]
College of Computer Science, 161 CN
Northeastern University
Boston, MA 02115
{gene,xqma,vietha}@ccs.neu.edu
Phone: +0 617-373-8686 (FAX: +0 617-373-5121)
Contact: Gene Cooperman

***Abstract*— We present the MBRAM model for static evaluation of the performance of memory-bound programs. The MBRAM model predicts the actual running time of a memory-bound program directly from pseudo-code. This means that the final running time can be predicted even before the program has been developed and benchmarked.**

In contrast to the "Big Oh" complexity model, which measures the time solely by counting the number of instructions executed, the MBRAM model predicts running times based on memory accesses, cache parameters, RAM bandwidth, and other important architectural parameters. As a result, the MBRAM model correctly ranks orders the actual running times for implementations of seven different $O(n \log n)$ sorting algorithms. In our suite of benchmarks, the MBRAM model consistently underestimates the actual running times, with errors ranging from 10% to 44%.

I. Introduction

We describe a model for static evaluation of the performance of memory-bound programs. Such a model implicitly assumes that the CPU is operating in parallel with the RAM. Furthermore, one has limited data parallelism, in that all of the bytes of a cache line are read or written in parallel. A *memory-bound* program is a program in which most of the time is spent waiting for memory operations to complete. By concentrating on memory-bound programs, we are able to demonstrate a surprisingly accurate, yet simple model of the running time. The model has the added advantage of statically predicting the running times.

By *static* prediction, we mean evaluation based solely on the code and knowledge of architectural parameters of the target computer. This assumes that the structure of the program is sufficiently transparent, so that for a given data input, the number of times that an instruction is executed can be directly estimated without recourse to running the program. This is true for many common subroutines. This is demonstrated for permutation multiplication, matrix multiplication and sorting. Note also that in this situation, the running time can be estimated directly from pseudo-code, prior to implementing an algorithm.

[1]This work was partially supported by the National Science Foundation under Grant CCR-0204113, and by the Institute for Complex Scientific Software (ICSS, `http://www.icss.neu.edu/`).

An evaluation model that is both simple and static is important in:

- compiler optimization through program transformation; and
- algorithm tuning by implementors.

The model is also important for diagnosing performance bugs, by noting when the time for an implementation differs too much from the predicted time.

The new MBRAM model (*Memory Bound Realistic Analytical Model*) determines a lower bound on the running time of implementations of memory-bound programs. The MBRAM model can be understood in a rough way as counting the total number of memory cycles over the life of the process. This is in contrast to the traditional "big Oh" or "RAM model of computation" (not to be confused with RAM memory) as taught to undergraduates, in which one counts the number of dynamic instructions executed. More precisely, the primary architectural parameters of the MBRAM model are

- the sequential bandwidth of RAM (β_1);
- the random access bandwidth of RAM (β_2: the bandwidth, assuming that a cache block is loaded from RAM with each successive cache block coming from a random location in RAM);
- the cache size (C) and cache block size (B) for the largest level of cache; and
- the branch misprediction pipeline.

For the suite of memory-bound applications on which the model was tested, the experimental running times were always within 44% of predictions and usually much closer. Since the prediction is always a lower bound, this error range is equivalent to $\pm$22%. For some cases, such as permutation multiplication on newer Pentiums, the model was accurate to within 5%. The test suite covers a broad variety of data access patterns. They include permutation multiplication (for random permutations), matrix multiplication, and seven variations of sorting. This suite covers random word access, stride access, a heap data structure, merging of multiple streams of sorted data, and other access patterns.

The issue can be seen starkly in the comparison in Figure 1, below. A novel, faster, two-pass algorithm for permutation multiplication is compared with the traditional algorithm. An analysis using the RAM model (unit time per operation) would predict the running time of the two-pass model to be twice as long as the standard one-pass implementation. While this is roughly true on the Pentium III, the *opposite* is true on the Pentium 4. The RAM model fails to account for the high bandwidth and longer latency of DDR-RAM.

The rest of this paper is organized to: describe the MBRAM model in Section II; apply the MBRAM model to permutation multiplication in Section III; apply the model to matrix multiplication in Section IV; and apply the MBRAM model to sorting of integers V. Each of these three applications show excellent agreement between experimental running times and the MBRAM model. This indicates that each of them are memory-bound.

A. Related Work

There are several examples of newer complexity models more accurately taking into account architectural features. These include the LogP model of Culler et al. [1], the BSP model of Valiant [2], and the disk latency model for parallel disks by Vitter [3]. Like the MBRAM model, These models are used as a guide toward algorithm design in their domains, but they are not usually used to predict actual running times.

LaMarca and Ladner [4], [5] moved a step closer to an analytical model by quantifying the number of cache misses for a variety of sorting algorithms, and comparing those numbers to experimentally acquired data. They also present experimental data for the number of instructions and for CPU time for each algorithm. Although they can predict the number of cache misses and correlate it to running times, they do not predict the running time itself.

The concept of a "memory wall" was introduced by Wulf and McKee [6]. The algorithm for faster permutation multiplication of Section III is described in a research note by Cooperman and Ma [7]. The work of Cooperman and Robinson [8] uses the idea of fast disk-based permutation multiplication to develop a disk-based membership algorithm for mathematical groups. The work of this paper is motivated by the prior results of Cooperman and Grinberg [9], who found the parallel shared memory performance of coset enumeration to be strongly dependent on memory speed rather than CPU speed.

II. Definition of MBRAM Model

We group the parameters of the MBRAM model according to memory access, cache, and branch misprediction. We then present the *Model Rules* for calculating the running times.

a) Memory access.

First we define a *sequential stream* as a sequential pattern of reads from or writes to main memory (either in the forward or backward direction). The sequence must be of length at least twice the size of cache block.

β_1 (bytes/second): memory bandwidth for one of at most μ_1 concurrent streams in a region of code, with no other concurrent memory accesses.

β_2 (bytes/second): memory bandwidth for all other memory accesses.

μ_1 (typically $\mu_1 = 2$): number of concurrent sequential streams allowed.

On current memory architectures, β_2 is experimentally found to be the same for both read and write accesses. Further developments in memory chip design may create a situation requiring separate values for $\beta_2^{(read)}$ and $\beta_2^{(write)}$.

The motivation for μ_1 is that most RAM chips have multiple independent memory banks. Current PC-133 and DDR-266 RAM have four memory banks. Hence, an extended burst of consecutive reads or writes will access a single page of RAM. Access to a new page of RAM not currently associated with some memory bank incurs a delay. Since the memory bank addresses are mapped in advance, it is difficult to predict how the application data will be mapped to distinct memory banks. Hence, we conservatively suggest $\mu_1 = 2$ as an average value.

b) Cache.

Next, we introduce three cache-related parameters. The three parameters are important for the restrictions they add to the model.

C: cache size

B: cache line (or cache block) size

$\mu_2 = C/B$: the maximum number of cache buffers

In applying branch misprediction to the model, we will assume the best possible branch prediction based on the history all previous branch sites in the code. For example, in a comparison-based sort, one would expect a branch misprediction after a comparison exactly half of the time. In that case, if there are $n \log n$ comparisons, then the model would charge $m(n \log n)/2$ due to branch misprediction.

c) Branch misprediction.

Finally, we add one more parameter.

m (seconds): penalty charged for a branch misprediction.

Strictly speaking, branch misprediction is not a major component of the time of memory-bound programs, since it does not involve waiting on access to main memory. We take the branch misprediction as a parameter but ignore the CPU clock rates and the number of CPU instructions for two reasons: (1) in some comparison-intensive applications such as comparison-based sorts, the branch misprediction penalty accounts for a large part of the running time of the CPU; and (2) the number of branches can be easily estimated from pseudo-code.

d) Model Rules.

1) A read from memory to cache or a write from cache to memory (whether associated with β_1 or β_2) is charged for reading or writing data in multiples of the cache block size B. If less data is accessed, one rounds up to the next multiple. In particular, a read of a single random

CPU/RAM	New, Two-Pass Algorithm	Traditional Algorithm
2.66 GHz Pentium 4 / DDR-266 RAM	0.042 s	0.159 s
0.6 GHz Pentium III / PC-100 RAM	0.131 s	0.097 s

Fig. 1. Two-Pass Permutation Multiplication versus Traditional Algorithm: `for (i=0; i<1048576; i++) Z[i]=Y[X[i]];`

word in main memory is charged B/β_2. An access to c cache blocks is charged cB/β_1 or cB/β_2.

2) One may apply β_1 to a region of code for with at most μ_1 sequential streams and no other memory accesses. (This rule is subject to revision, as the architectures evolve both for hardware prefetch of streams by the CPU, and for the ability to maintain separate memory banks with fast sequential access by the DRAM.)
3) For each branch instruction, one predicts whether the branch is taken, based on the static program, and also based on the dynamic history for all branch instructions of taking a branch. (This is the same decision made by the CPU's branch predictor.) From this, one determines the probability π that the prediction will be wrong. Each execution of the branch is then charged πm. (This prediction is often very simple. For example, if a loop contains many iterations, one predicts that the branch that tests continuation of the loop is always taken. Then the probability of error is $\pi \approx 0$. So, one is charged $0m = 0$. In a second example, suppose a branch is taken 1/10 of the time. If the program provides no clues, then the branch predictor will predict that the branch is never taken, for lack of other knowledge. Then $\pi = 1/10$ and one is charged $\pi m = m/10$. Note that one is never charged more than $m/2$.)
4) The cache initially holds no data.
5) Each read causes the associated data to be brought into cache.
6) Data in the cache may not grow beyond C. If data does grow beyond C, some data is ejected from the cache on an LRU (Least Recently Used) basis. (In fact, cache associativity could modify this rule, but for the sake of simplicity of the model, we ignore such issues.)
7) Any write to main memory requires the corresponding address to be previously loaded to cache. (This rule reflects current hardware implementations, but future implementations may relax this rule.)
8) One may not have more than μ_2 cache buffers. If the program is accessing memory in a pattern with more than μ_2 concurrent streams, then each access to memory is considered a random access. Each access to memory is then charged B/β_2. For example, if the number of concurrent streams is smaller or equal to μ_2, the cost for accessing N 4-byte integers will be $4N/\beta_1$ or $4N/\beta_2$ (according to whether the stream is supported by hardware-prefetch or not). But if the number of concurrent streams exceeds μ_2, then the cost for accessing N 4-byte integers will BN/β_2 .

III. Permutation Multiplication

The traditional permutation multiplication is expressed by the simple formula below, with the X and Y arrays as input, and the Z array as output.

```
int X[N], Y[N], Z[N];
for (i = 0; i < N; i++) Z[i] =
Y[X[i]];
```

A. Determination of MBRAM Parameters

For the Pentium 4, we use the MBRAM model parameters of Figure 2. We were able to derive all numbers based purely on the cache block size, the length of the branch misprediction pipeline [**?**, page 4 in URL], and the memory specifications [**?**], [**?**]. The PC-66, PC-100 and PC-133 RAM all use SDRAM with timing parameters of 2-2-2. This corresponds to a latency of four memory cycles. For example, PC-133 RAM has a latency of 4/(133 MHz)=30 ns.

DDR-266 RAM (also known as PC-2100) uses RAM with 2.5-3-3 timing parameters. For DDR-266, a latency of 10 memory cycles (75 ns) on the 133 MHz bus was assumed. This is needed to handle the worst case latency, occuring for a WRITE memory page miss occuring after a WRITE burst of length four.

In each case, the latency λ was found as the number of clock cycles divided by the memory bus speed. The time t to access a cache block is the time for $B/8$ memory bus cycles, since the Pentiums use an 8-byte bus. Taking f as the frequency yields $t = B/(8f)$, this yields:

$$\beta_1 = B/t = 8f$$
$$\beta_2 = B/(\lambda + t) = B/(\lambda + B/\beta_1)$$

The parameters β_1 and β_2 based on specifications agree to within 5% with benchmarks on the Pentiums. The benchmarks access each cache block once, either in sequence (for β_1) or at random (for β_2). For some other processors, the benchmarks of β_1 and β_2 were up to 50% slower, indicating a less optimized CPU. In those other processors, benchmarks on the individual CPUs are needed to estimate β_1 and β_2.

Over time, we expect all CPUs to be sufficiently optimized to directly use the memory specifications for β_1 and β_2. Some of these optimizations include a non-blocking cache, hardware prefetch, dynamic instruction re-ordering, high instruction throughput (ILP and high CPU clock rate), and other optimizations to allow CPU execution to to be overlapped with main memory access. Offloading video traffic from the memory bus, as is done with the Pentium AGP bus, is also important.

The branch misprediction penalties are also estimated based on vendor specifications. In the case of the Pentium, Intel

CPU/RAM	bandwidth (β_1, GB/s)	bandwidth (β_2, GB/s)	branch misprediction (m, ns)	L2 cache block (B bytes) / cache (C KB)
2.66 GHz Pentium 4 / DDR-266 RAM	2.12	0.95	7.52 (20 cycles)	128 B / 512 KB
1.7 GHz Pentium 4 / PC-133 RAM	1.06	0.85	11.76 (20 cycles)	128 B / 256 KB
0.6 GHz Pentium III / PC-100 RAM	0.80	0.46	16.67 (10 cycles)	32 B / 512 KB
0.35 GHz Pentium II / PC-66 RAM	0.53	0.26	28.57 (10 cycles)	32 B / 512 KB

Fig. 2. MBRAM Parameters for Different Generations of Pentiums

reports that the P6 core (Pentium II and III) have a 10 stage branch misprediction pipeline and the Pentium 4 has a 20 stage branch misprediction pipeline [**?**, page 4 in URL].

B. Memory-Aware Permutation Multiplication

The model of Section II leads naturally to the following new, two-pass algorithm, described in [7], [8]. The constraint is that each block of the D array fits in half the cache. Furthermore, the number of blocks of the D array must be less than the number of cache blocks (less than C/B). A later remark of this section describes how to extend the algorithm by using more passes when these constraints cannot be met.

The key to the algorithm is that if each block of the D array contains S entries, then the first block of the D array will ultimately contain the first S entries of the Y array, but permuted according to the order in which they will be written into Z. The second block of D will similarly contain the next S entries of Y, and so on. Phase I determines how to permute each block. Phase II locally permutes the data from each segment of Y to the corresponding block of D (and the segment of Y and block of D both fit in cache). Phase III copies the permuted data from the different blocks of D to Z, merging the blocks according to the desired permutation.

```
#define BLOCK_LENGTH \
  (CACHE_SIZE/2/sizeof(int))
#define NUMBER_OF_BLOCKS \
  (ARRAY_LENGTH / BLOCK_LENGTH)
int X[ARRAY_LENGTH], Y[ARRAY_LENGTH],
   Z[ARRAY_LENGTH];
int D[ARRAY_LENGTH];
int *D_ptr[NUMBER_OF_BLOCKS];

// Phase I: distribute value, X[a],
//   into block given by D_ptr[block_num]
int block_num, i, j;
for (block_num= 0; block_num <
       NUMBER_OF_BLOCKS; block_num++)
  D_ptr[block_num] =
     & D[block_num * BLOCK_LENGTH];
for (i = 0; i < ARRAY_LENGTH; i++){
  block_num = X[i] / BLOCK_LENGTH;
  *(D_ptr[block_num]) = X[i];
  D_ptr[block_num]++;
}

// Phase II: for D[i] == X[a],
//   replace the value X[a] by Y[X[a]]
for (i = 0; i < ARRAY_LENGTH; i++)
  D[i] = Y[ D[i] ];

// Phase III: copy value Y[X[a]]
//   from D_ptr[block_num] to Z[a]
for (block_num = 0; block_num <
      NUMBER_OF_BLOCKS; block_num++)
  D_ptr[block_num] =
    & D[block_num * BLOCK_LENGTH];
for (i = 0; i < ARRAY_LENGTH; i++) {
  block_num = X[i] / BLOCK_LENGTH;
  Z[i] = *(D_ptr[block_num]);
  D_ptr[block_num]++;
}
```

Figure 3 shows the experimental results across Pentium generations. Note the close agreement of experiment and prediction across CPU generations.

With each new generation of the Pentium and memory, the new algorithm has become progressively faster relative to straightforward permutation multiplication. This is due to the growing CPU/memory latency gap of newer generations of the Pentium CPU/memory subsystem, The accuracy of the MBRAM model rises to within 5% on later generations, corresponding to the rise in the CPU-memory gap.

The data of Figure 3 for the new algorithm represents an optimized version of the straightforward code. Memory layout of the D blocks was altered to avoid cache-aliasing, and zero-mapped pages of the operating system were first instantiated. The unoptimized version was about 20% longer.

C. Analysis and Experimental Results

In Figure 3, we have seen the experimental and predicted results. The implementation uses optimizations to avoid cache aliasing and to instantiate zero-mapped pages in advance. Without those optimizations, the experimental times are about 20% higher.

We analyze the traditional permutation multiplication. We assume 4 byte integers and that X array is a random permutation. Since X is read consecutively, it costs $4\frac{N}{\beta_1}$ to read X. By Model Rule 1 of Section II, the random accesses to the Y array cost $\frac{B}{\beta_2}n$. The Z array must be read into cache before being written (Model Rule 7), costing $2\cdot 4\frac{N}{\beta_1}$. So, the predicted total cost is

$$\left(\frac{12}{\beta_1}+\frac{B}{\beta_2}\right)N.$$

For the new permutation algorithm, Phase I is charged $3 \cdot 4\frac{N}{\beta_2}$. (X is a read stream, while each stream corresponding to `D[Cursor[block_num]]` is a write stream requiring a read-modify-write operation.) Note that we require

CPU/RAM	Time (new Fast alg., s)		Time (traditional alg., s)	
	Experiment	Predicted	Experiment	Predicted
2.66 GHz Pentium 4 / DDR-266 RAM	0.042	0.042	0.159	0.147
1.7 GHz Pentium 4 / PC-133 RAM	0.060	0.047	0.176	0.158
0.6 GHz Pentium III / PC-100 RAM	0.131	0.087	0.097	0.083
0.35 GHz Pentium II / PC-66 RAM	0.222	0.151	0.148	0.143

Fig. 3. Fast multiplication of two random permutations on 1,048,576 points (4 MB per permutation)

```
for (i=0; i<1048576; i++) Z[i]=Y[X[i]];
```

`number_of_blocks`$+1 \leq \mu_2$ by Model Rule 8 of Section II. Phase II is charged $3 \cdot 4\frac{N}{\beta_2}$ for reading `D` and `Y` and writing to `D`. Phase III is charged $4 \cdot 4\frac{N}{\beta_2}$ for reading `X` and `D` and executing a read-modify-write on `Z`. Thus, the total cost is

$$40\frac{N}{\beta_2}.$$

IV. Matrix Multiplication

Matrix multiplication is demonstrated as an example with stride access, and with the more CPU-intensive operation of matrix multiplication. Such issues are common in many numerical analysis programs. The straightforward row major integer matrix multiplication algorithm is implemented with the results shown in Figure 4. (Blocked matrix multiplication routines are more efficient, but they also tend further toward the CPU-bound domain, and hence further away from applicability for the MBRAM model.)

The Pentium II and Pentium III cases were omitted because our Pentium III did not have enough memory for the large matrix multiplication, and the shorter cache block size of Pentium II and III imply that the formula (2) would not be valid for those cases.

After doubling the number of scalar multiplications in the inner loop on the 1.7 GHz Pentium 4, the CPU time was found to increase by 20% for $n = 500$. Hence, the program is only partially CPU-bound. The remaining inaccuracy is accounted for by the frequent accesses to L2 cache and by the TLB misses. The MBRAM model charges nothing for either case.

We next derive the MBRAM formulas for the time to execute PQ for matrices P and Q stored in row major order, of dimension n, and with each entry of size w bytes. We analyze only the n^3 terms. Consider first the case when a matrix row is larger than a cache block ($wn > B$), and one matrix row and two matrix columns fit inside the cache ($wn + 2Bn < C$), but the entire matrix does not fit inside the cache ($wn^2 > C$). (Note that for row major order, each entry of a matrix column lies in a separate cache block, and so a column requires Bn space.) After multiplying by the first column, the second column will be contained in the same cache blocks as the first column, and so on. Each entry of Q, and therefore each cache block corresponding to Q, is loaded into cache n times, and there are n^2 entries or wn^2/B cache blocks. So, the MBRAM formula is $wn^3/\beta_2 + O(n^2)$.

Next, consider the case when a matrix column is larger than the entire cache ($Bn \geq C$). In this case, each inner product of a row and column requires one to load both the row and the column. This costs $wn/\beta_1 + Bn/\beta_2$ seconds. A matrix multiplication computes n^2 inner products. So, the MBRAM formula is $wn^3/\beta_1 + Bn^3/\beta_2 + O(n^2)$.

V. Sorting Algorithms

In Figure 5, we compare several well-known sorting algorithms to show agreement of theory and experiment. The MBRAM predictions tend to be about 2/3 of the measured running times, and always serves as a lower bound. Furthermore, the ordering of the times of the MBRAM formulas reflects the ordering of the measured running times. Thus, the MBRAM model serves as an excellent guide for improved sorting implementations.

The MBRAM formulas are based on estimates of the number of accesses to main memory and the number of branch mispredictions. While there is not room for the derivation in this note, the full derivation can be found in Appendix A. For mergesort and quicksort, the times are dominated by the branch mispredictions, while the data access dominates the time for heapsort. The non-comparison-based sorts eliminate the cost of branch misprediction entirely.

As always, the MBRAM prediction is a lower bound. The predicted times are approximately 2/3 of the experimental times. We hypothesize that the additional experimental time is due to intensive use of the L2 cache at the lower levels of the recursion (for which the MBRAM model charges nothing), along with the fact that no attempt was made to optimize the code over the straightforward implementations. This points up the opportunity for further optimizations by overlapping the CPU-intensive lower recursion levels with the memory-intensive upper recursion levels.

The results are shown in Figure 5. The derivations of the MBRAM formulas are contained in Appendix A. The non-comparison-based sorts (bucket sort and radix sort) diverge further from the predictions because a portion is CPU-bound, and it is not overlapped by branch mispredictions or by memory accesses. Specifically, they use the L2 cache intensively, and tend to have many TLB misses. (There are only 64 TLB entries on the Pentium 4.)

Quicksort is of special interest. One can split the total time of 1.51 = 0.38 + 1.13, where the first term accounts for data movement, and the second term accounts for branch mispredictions (comparisons). The given formula assumes that the input array is partitioned in half at each level. As shown in the appendix, the time $(N/2)m\log_2 N$ is a lower bound for

CPU/RAM	Time (Mat. Dim. $n = 500$)		Time (Mat. Dim. $n = 4000$))	
	Experiment	Predicted	Experiment	Predicted
2.66 GHz Pentium 4 / DDR-266 RAM	1.67*	0.53 (1)	11,241.23	8,743.91 (2)
1.7 GHz Pentium 4 / PC-133 RAM	1.47*	0.59 (1)	11,306.00	9,879.16 (2)

Fig. 4. Matrix Multiplication over Integers (word size $w = 4$); Predictions according to MBRAM formula (1) wn^3/β_2 (when $wn > B,\ Bn < C$)); or (2) $wn^3/\beta_1 + Bn^3/\beta_2$ (when $Bn \geq C$)
*Computation is CPU-bound

Sorting Algorithm	Exper. (s)	Pred. (s)	MBRAM formula
Quicksort	2.40	1.51	$\frac{2wN}{\beta_1}(\log_2 N - \log_2 C + 1) + \frac{N}{2} m \log_2 N$
Mergesort	3.14	1.75	$(\frac{wN}{\beta_2} + \frac{2wN}{\beta_1})(\log_2 N - \log_2 C + 1) + \frac{N}{2} m \log_2 N$
Heapsort	24.88	16.83	$\frac{B}{\beta_2} N(\log_2 N - \log_2 (C/w) + \log_2 \log_2 N) + mN \log_2 N$
[Input data uniformly distributed between 0 and $N = 8$ Meg]			
Simple Bucket Sort ($b = 64$)	0.62	0.47	$\frac{3wN}{\beta_2}(\lceil \log_b N \rceil - \lfloor \log_b (C/2) \rfloor + 1)$
Distribution-count Bucket Sort ($b = 64$)	0.92	0.57	$\frac{3wN}{\beta_2}(\lceil \log_b N \rceil - \lfloor \log_b (C/2) \rfloor + 1) + \frac{wN}{\beta_1}(\lceil \log_b N \rceil - \lfloor \log_b C \rfloor + 1)$
Simple Radix Sort ($b = 64$)	0.77	0.47	$\frac{3wN}{\beta_2} \lceil \log_b N \rceil$
Distribution-count Radix Sort ($b = 64$)	1.01	0.60	$(\frac{wN}{\beta_1} + \frac{3wN}{\beta_2}) \lceil \log_b N \rceil$

Fig. 5. Prediction and Experiment for Various Sorting Algorithms; 1.7 MHz Pentium 4 with PC-133 RAM; input data is uniformly distributed array size N = 8 Meg (Nw = 32 MB); cache size C = 256 KB; integer word size $w = 4$ bytes

the time due to branch mispredictions, and $(N/2)m(\log_2 N + \log_2(4/5))$ is an upper bound. For the N in the table, the upper bound is only 1.4% larger than the lower bound.

Hence, the MBRAM formula predicts that the cost of branch mispredictions (or comparisons) is highly insensitive to good or bad pivots. This leaves only the small time for data movement (0.38) to be improved by a better choice of pivots.

The above analysis implies that the well-known *median-of-three* optimization for choosing pivots will have very little effect for implementations on the 1.7 MHz Pentium 4 with PC-133 RAM. (The pivot is chosen as the median of the array values at the middle of the array and at the two extremes.) This is borne out by our tests. The median-of-three optimization resulted in a time of 2.38 s, or only 1% improvement.

VI. Acknowledgements

We gratefully acknowledge helpful discussions with Sergey Bratus, Tom Cormen, David Kaeli and Rajmohan Rajaraman.

References

[1] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP : Towards a realistic model of parallel computation," in *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993, pp. 1–12.
[2] L. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, pp. 103–111, 1990.
[3] J. Vitter, "External memory algorithms and data structures: Dealing with massive data," *ACM Computing Surveys (CSUR)*, vol. 33, pp. 209–271, 2001.
[4] A. LaMarca and R. Ladner, "The influence of caches on the performance of sorting," in *Proc. of the 8th annual ACM-SIAM Symp. on Discrete Algorithms*, 1997, pp. 370–379.
[5] ——, "The influence of caches on the performance of sorting," *J. Algorithms*, vol. 31, pp. 66–104, 1999.
[6] W. Wulf and S. McKee, "Hitting the memory wall: Implications of the obvious," *ACM Computer Architecture News*, vol. 23, pp. 20–24, 1995.
[7] G. Cooperman and X. Ma, "Overcoming the memory wall in symbolic algebra: A faster permutation algorithm (research note)," *SIGSAM Bulletin*, pp. 1–4, Dec. 2002.
[8] G. Cooperman and E. Robinson, "Memory-based and disk-based algorithms for very high degree permutation groups," in *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '03)*. ACM Press, 2003, pp. 66–73.
[9] G. Cooperman and V. Grinberg, "Scalable parallel coset enumeration: Bulk definition and the memory wall," *J. Symbolic Computation*, vol. 33, pp. 563–585, 2002.
[10] A. LaMarca and R. Ladner, "The influence of caches on the performance of heaps," *Journal of Experimental Algorithmics*, vol. 1, 1996.

Appendix

a) Quicksort.

Consider sorting N integers (w bytes each) using quicksort. We analyze the time as if the input array is always partitioned exactly in half. This is justified in the following paragraphs.

We will recurse $\log_2 N$ times. Each time of recursion will read through all N elements and perform the partitioning process. The cost for reading/writing through N elements in one pass of partition is $2 \cdot \frac{wN}{\beta_1}$. In each pass, there will be N comparisons to be performed on average. The probability that the branch-prediction is correct for those comparisons is a half (since the values to be compared are completely random and have no pattern to predict). So there will be $\frac{N}{2}$ times of branch-misprediction and the total cost for it in one pass is $\frac{N}{2}m$. We have totally $\log_2 N$ passes, $\log_2 C - 1$ (C is the cache size) passes of which are performed in cache. The cost of partitioning in cache does not include the cost for reading/writing data but the branch misprediction cost. So we need $\frac{2wN}{\beta_1} + \frac{N}{2}m$ for one pass in RAM and $\frac{N}{2}m$ for one pass in cache. Therefore, the total cost for sorting N integers according to our model is

$$(\frac{2wN}{\beta_1} + \frac{N}{2}m)(\log_2 N - \log_2 C + 1) + \frac{N}{2}m(\log_2 C - 1).$$

We next analyze the error in this approximation by considering the time when the pivot does not partition the input array in half. The total predicted time splits as 1.51 = 0.38 + 1.13, with 0.38 s for data movement and 1.13 s for branch mispredictions. We will show that the second term, 1.13 s, changes almost not at all under the assumption that the partition does not split the input array in half. Therefore any change in the first term, 0.38 s, will make a small contribution to the overall time.

In analyzing the second term, recall that Model Rule 3 says to count a branch misprediction only to the extent that it differs from the best prediction based on the history. If the partition splits the array into fractions f and $1 - f$ for $0 \le f \le 1/2$, then branch prediction will always predict that data is sent to the larger half. So, an array of length n will have nf branch mispredictions instead of $n/2$.

Let $C(n)$ be the number of branch mispredictions for an array of length n. We show that the MBRAM term for branch mispredictions is bounded by $C(n) \le (n/2)\,(\log_2 n + \log_2(4/5))$, where $\log_2(4/5) \approx -0.32$. To see this, we apply the recurrence for branch mispredictions

$$C(n) = fn + C(fn) + C((1-f)n).$$

We prove that

$$C'(n) = (n/2)\,(\log_2 n + \log_2(4/5))$$

is an upper bound for the previous recurrence. Note that $C'(n) = 0$. We show that $C'(n) \ge fn + C'(fn) + C'((1-f)n)$, to prove that $C'(n)$ is an upper bound for the recurrence. Note that $fn + C'(fn) + C'((1-f)n) = (n/2)\,(\log n + 2f + f\log f + (1-f)\log(1-f) + \log_2(4/5))$. If $f = 0$, then $fn + C'(fn) + C'((1-f)n) = (n/2)\,(\log_2 n + \log_2(4/5))$. We note that $2f + f\log f + (1-f)\log(1-f)$ is minimized when $f = 1/5$, and by plugging in $f = 4/5$, we deduce that $2f + f\log f + (1-f)\log(1-f) \ge \log_2(4/5)$. So, $fn + C'(fn) + C'((1-f)n) \ge (n/2)\,(\log n + 2f + f\log f + (1-f)\log(1-f)) = fn + C'(fn) + C'((1-f)n)$.

For large n, $C'(n) = (n/2)\,(\log_2 n + \log_2(4/5)) \approx (n/2)\,\log_2 n)$. Since $(n/2)\,\log_2 n$ is also a lower bound for $C(n)$, we can take $C(n) = (n/2)\,\log_2 n$, with very small error.

b) Mergesort.

Unlike the case of quicksort, the data array in mergesort is always divided into two equal small arrays. Therefore, mergesort has totally $\log_2 N$ passes, $\log_2 C - 1$ passes of which are performed in cache. The cost for one pass in RAM is $\frac{wN}{\beta_2} + \frac{2wN}{\beta_1} + \frac{N}{2}m$, and for one pass in cache is $\frac{N}{2}m$. Hence, the total cost is

$$\left(\frac{wN}{\beta_2} + \frac{2wN}{\beta_1}\right)(\log_2 N - \log_2 C + 1) + \frac{N}{2}m\log_2 N.$$

c) Heapsort.

A more detailed analysis of HeapSort is provided by LaMarca and Ladner [10]. In the spirit of a simple estimate, we make some simplifying approximations. A complete binary tree with N vertices has a depth of at most $\log_2 N$. Assume that procedure *downheap* requires $\log_2 N$ steps. Since the number of nodes of a binary tree per level grows exponentially, this is likely to be a reasonable estimate.

As a further simplifying approximation, assume that the cache holds only the nodes closest to the root. We wish to find the largest level L for which many nodes are held in cache. Specifically, if a node is at level L, then there are 2^L nodes at that level. When a step of *downheap* touches a node at level L, it will also touch $\log_2 N - L$ nodes from lower levels, bringing them into cache. Further, there are $2^{L-1} - 1$ nodes above level L. So, we require that $2^{L-1} - 1 + 2^L + 2^L(\log_2 N - L) \approx C/w$, for w the size of a node in bytes. This yields $L \approx \log_2(C/w) - \log_2(\log_2 N - L + 1.5) \approx \log_2(C/w) - \log_2\log_2 N$. Hence, $\log_2(C/w) - \log_2\log_2 N$ of the steps of *downheap* will access nodes from cache.

Each step will compare the left child, right child and the parent node to determine the largest node, and swap nodes if necessary. On average, each step of *downheap* requires two comparisons with probability 50% of misprediction for each one, and probability 50% for node swapping. The cost of each step is either m (for the first L steps in cache) or $m + \frac{1}{2}\frac{2B}{\beta_2}$ (for the remaining steps).

Procedure *buildheap* builds a heap of size N from the bottom up. Hence, it calls procedure *downheap* $N/2$ times, with later calls to *downheap* containing more steps than earlier calls. The total number of steps will be $N/2 + N/4 + N/8 + \cdots = N$. Of those steps, at most C steps will involve the nodes closest to the root, and hence access to cache. Since $C \ll N$, we assume no accesses to cache.

Procedure *heapsort* calls procedure *downheap* N times. Due to the exponential growth in the number of nodes per level, we again assume $\log N$ steps per call to *downheap*, with $\log C$ steps occuring from cache. We neglect the cost of *buildheap* and copying to the destination array, since both are proportional to N, and are small compared to the $N\log N$ terms. Hence, the total cost of heapsorting N integers is

$$\frac{B}{\beta_2}N(\log_2 N - \log_2(C/w) + \log_2\log_2 N) + mN\log_2 N.$$

d) Bucket sort (uniformly distributed data).

Consider a simple version of non-comparison sorting: bucket sort for uniformly distributed data. We distribute element values in the original arrays into buckets according to their high bits, and then recursively sort them again using bucket sort and the next lower field of bits. Finally, we concatenate the buckets to get the results. Bucket sort is in some sense the opposite of radix sort, which begins with the low bits.

Suppose we use b buckets for each pass, then we need totally $\log_b N$ passes. Of these, $\lfloor \log_b(C/2) \rfloor - 1$ passes are performed in cache and $\lceil \log_b N \rceil - \lfloor \log_b C \rfloor + 1$ passes are in memory. (The $C/2$ in $\lfloor \log_b(C/2) \rfloor - 1$ passes in cache can be seen because the size of an input array must fit in half the cache in order for there to be room in cache for the destination buckets. The "-1" is needed since no matter how small the input array, one pass is needed to copy the array from RAM to cache.)

In each pass, we read sequentially from the source array and distribute values (read-and-modify) into buckets. According to the discussion in Model Rules 8 in Section II, if b is smaller than the maximum number of concurrent streams, μ_2, then the cost for one pass will be $3\frac{wN}{\beta_2}$ with w the integer word size, and the total cost for bucket sorting N integers will be

$$\frac{3wN}{\beta_2}(\lceil \log_b N \rceil - \lfloor \log_b(C/2) \rfloor + 1).$$

We demonstrate on an array of length $N = 8\ Meg$ using $b = 64$ buckets. At the end of the second pass, each bucket is 8 KB. So, we concatenate each 8 KB bucket into the destination array after first sorting it by bucket sort inside cache.

For the distribution count version of this, we add

$$\frac{wN}{\beta_1}(\lceil \log_b N \rceil - \lfloor \log_b C \rfloor + 1).$$

to the previous formula for the counting phase.

e) Radix Sort (uniformly distributed data).

The analysis is similar to that of bucket sort. The primary difference is that none of the passes operate in cache. So, $\lceil \log_b N \rceil - \lfloor \log_b(C/2) \rfloor + 1$ is replace by $\lceil \log_b N \rceil$.