RC24489 (W0802-029) February 6, 2008
Computer Science

IBM Research Report

High-Performance Sorting Algorithms on AIX

C. Eric Wu, Gokul Kandiraju, Pratap Pattnaik
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

High-Performance Sorting Algorithms on AIX

C. Eric Wu, Gokul Kandiraju, and Pratap Pattnaik

IBM Research

Abstract

Sorting is a topic which has been studied and researched extensively. Given a number of records and one or more fields as the key, the task is to arrange the records into ascending or descending order. Sorting has been one of the most critical applications on mainframe machines. In this paper we describe our approach of offloading sorting operations from mainframe machines and executing the operations on AIX systems. A library was developed to read records from mainframe DASD disks on an AIX machine and thus make offloading other mainframe operations feasible. We analyze the performance of quick sort and radix sort implementations on AIX using mainframe datasets and find that the sorting operations can be improved up to 70%.

Introduction

Sorting is a topic which has been studied and researched extensively [1]. Given a number of records and one or more fields as the key, the task is to arrange the records into ascending or descending order.

Floyd pioneered the notion of analyzing the number of transfers between primary and secondary storage in 1972 for matrix transposition [2], setting the stage for categorizing sorting techniques into *internal sorting* and *external sorting*. Internal sorting assumes that there is enough memory to store the records for the processor to work on them. When the system memory is relatively small, programmers aware of the limits will try to conserve memory usage. In the case of sorting a user may choose techniques such as multi-way merging or tournament sort to builds a loser tree, in which each element in the tree is the leading element in its own list. The memory usage is small because the number of lists is typically much smaller than the number of records, and lists can be stored in the next level of memory hierarchy or even in external storage such as tapes.

Related Work and Initial Observation

A recent direction in the design of cache-efficient and disk-efficient algorithms and data structures is the notion of *cache obliviousness*, introduced by Frigo et. al. in 1999 [3, 4]. Cache-oblivious algorithms perform well on a multilevel memory hierarchy without knowing parameter details of the hierarchy, only knowing the existence of a hierarchy. Vinther concludes that the quicksort algorithm [5] is also a cache-oblivious sorting algorithm [6]. As modern computers with larger memory capacities and more advanced prefetch features become available every year, there is no need to limit ourselves with external sorting approaches for yesterday's workloads.

Getting Specific: Quick Sort

It has been widely accepted that quicksort is the fastest comparison based sorting algorithm on typical datasets. The time complexity for quicksort is O(N Log N) on average, where N is the number of records. We would like to see how various implementations may diverge from this well-known fact and how we can keep the constant as small as possible. The only additional requirement is to make the sorting results "stable". The stability of a sorting algorithm is the ability to preserve the order of records with identical keys, regardless of sorting for ascending or descending order. An un-stable sorting algorithm will not preserve the original order of the records, while a stable sorting algorithm does as long as the records have identical keys.

The quicksort algorithm starts by picking a pivot that is used to compare with all other records in the input. Records are partitioned into two groups: those with smaller keys and others with larger keys. The algorithm goes on iteratively for the two groups until the sorting is done. In order to make quicksort stable, an additional key is required for the compare operation. We implemented the quicksort algorithm in two different ways, i.e. sorting pointers (SP) and sorting data (SD). Because the size of the key is typically much less than the size of the record, the idea is to use a pointer array in which each element is a pointer pointing to its corresponding record in the SP implementation. The SP quicksort approach swaps pointers instead of records, thus potentially saves time compared with swapping records in the SD quicksort approach. In the 64-bit environment the pointer size is 8 bytes.

Ensuring Stability

For the SP quicksort we use the pointer value as the second key for stability, since input records are typically lined up in virtual memory space in ascending order. For the SD quicksort we added a record number at the end of each record as the second key for stability, thus effectively making each record a little bigger. Input records are randomly generated and the key is a 10-byte ASCII number string, also randomly generated at the beginning of each record.

Experimental Platform

Both the SP and SD quicksort implementations were executed in an IBM POWER5 system with 1.9-GHz processors and 8-Gbyte main memory. The memory hierarchy in the system includes a two-way 64-KByte L1 instruction cache, a four-way 32-KByte L1 data cache, a shared 10-way 1920-KByte L2 cache, and a 12-way 36-MByte private off-chip victim L3 cache.

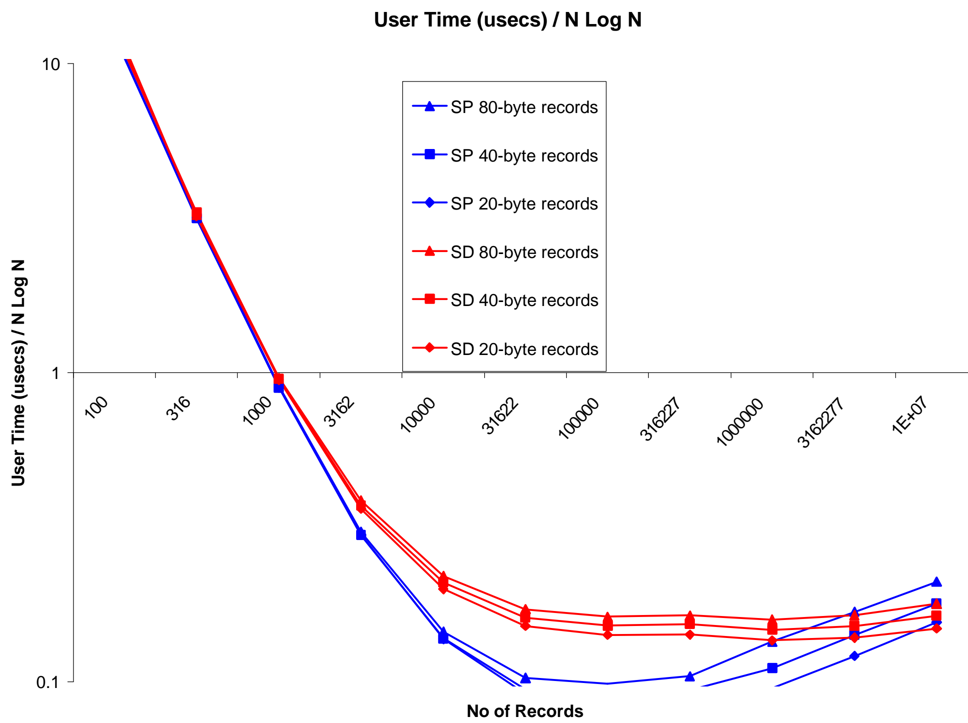

Figure 1(a). User time divided by N Log N, for record sizes 20, 40, and 80 bytes

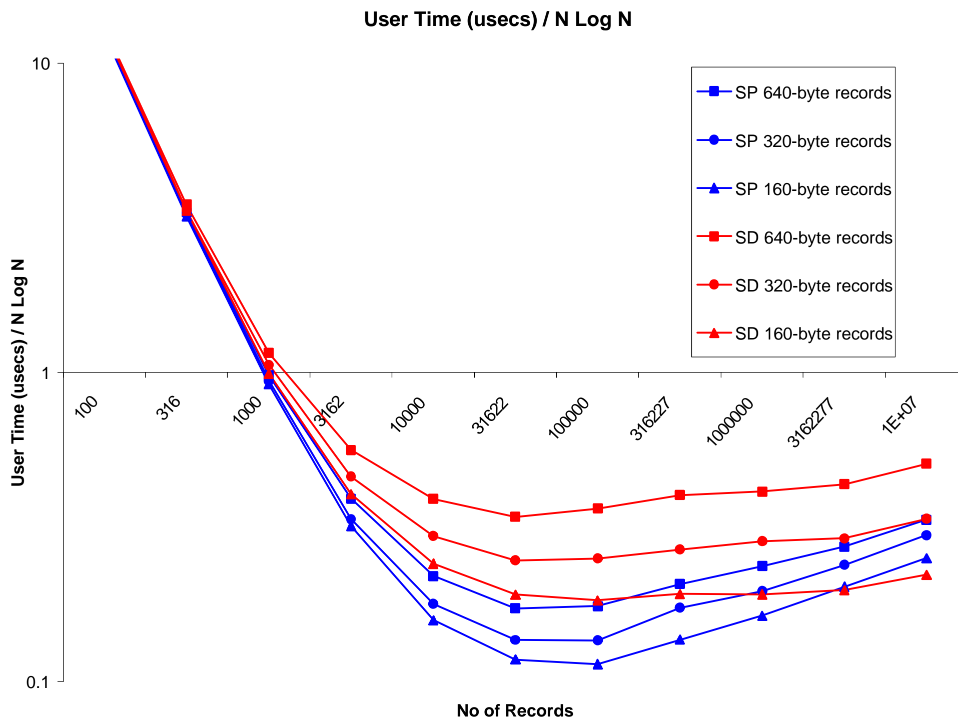

Figure 1(b) User time divided by N Log N, for record sizes 160, 320, and 640 bytes

Initial Observations

Figure 1 show the user time in micro-seconds divided by N Log N, which would be flat if user time was truly proportional to N Log N. We ran both the SP and SD quicksort implementation for record sizes ranging from 20 bytes to up to 640 bytes. These are the original record sizes, i.e. they do not include the added second key for stability in the case of SD quicksort. It can be seen in Figure 1(a) that the SP quicksort has an edge over the SD quicksort when the number of records is relatively small. However, a cross-over point occurs somewhere < 3162277 when the record size is, say 80 bytes. That is, the SD implementation actually runs faster than the SP implementation for N >= 3162277 and record size = 80 bytes. The same can be seen for records with smaller sizes, as the cross-over points for record size = 20 and record size = 40 occur when the numbers of records are both between 10000000 and 3162277.

We increased the record size to 160, 320, and 640 bytes and the results were shown in Figure 1(b). It can be seen that for record size = 160 bytes a cross-over point occurs again at somewhere < N = 3162277. Although there is no cross-over point found for the cases of record size = 320 and 640 bytes in Figure 1(b), we could not rule them out either. This is because that while the number of comparisons is independent of the presorted-ness of the input records, the number of swaps is highly dependent on it. Once the pivot is selected among the records, records with smaller keys are swapped to positions at one side of the pivot. With randomly generated keys in the records the number of swaps is roughly half of the number of comparisons in the first iteration. Since record order is not changed within the sub-partitions in subsequent iterations, randomly selecting a new pivot in a sub-partition would generate half as many swaps as comparisons. On the other hand, if selected pivots lean to one side, the number of swaps could be as many as comparisons. In such cases quicksort complexity goes to $O(n^2)$. It may be true that we can find a big enough record size so that such cross-over point will never occur for our average or random input record sets, however, we want to have a better understanding and find a good, solid sorting approach.

At this point we assume that the cross-over points are mainly caused by delays in the memory hierarchy for the SP sorting approach. Performance analysis with cycle-per-instruction (CPI) stacks will be used to show where the cycles are spent in both implementations.

Sorting Implementations and Analysis

One area we are interested in is to get data sets from System z volumes and work on them in other systems, such as the IBM POWER5 system. A data set is a collection of logically related data and can be a source program, a library of macros, or a file of data records used by a processing program. In this study we developed library routines to read the most commonly used sequential data sets from Direct Access Storage Device (DASD) volumes. The recording surface of a volume is divided into multiple concentric cylinders, each of which contains many tracks. The number of tracks and their capacity vary with the device. Information is recorded on all DASD volumes in a standard format called Count-Key-Data (CKD) format [7].

The Volume Table of Contents (VTOC) on a DASD is used to manage the storage and placement of data sets [8]. A VTOC is a data set that describes the contents of the direct access volume on which it resides. It is composed of 140-byte Data Set Control Blocks (DSCBs) that correspond either to a data set or to contiguous, unassigned tracks on the volume. A data set is defined by one or more DSCBs in the VTOC of each volume on which it resides. Contiguous tracks called extents are specified in the DSCBs to indicate where the records are stored in the volume.

The amount of space required for a data set is specified when a data set is allocated. Records stored in the tracks could be fixed-length or variable-length, and could have a blocked or unblocked format. Logical records in a data set can be bundled together as a block to save space, creating the so-called blocked format. The logical record length and block size of a data set are specified in its corresponding DSCB as data set parameters. A variable-length record is preceded by a record length field in the track indicating the length of the record. While we developed library routines to access fixed-length or variable-length, blocked or unblocked sequential data sets from DASD volumes, we use blocked fixed-length data sets randomly generated from System z as input for the rest of the report.

SP and SD Quicksort Implementation

A number of fixed-length (record size 80 bytes), blocked format data sets were created on DASD volumes with number of records ranging from 100 to 10000000. Figure 2(a) shows user time in micro seconds divided by N log N, where N is the number of records. Records were created randomly from System z and stored at DASD volume before getting sorted by the POWER5 system. As seen in Figure 1, there is a cross-over point in Figure 2(a) when the number of records is between 1000000 and 3162277. Using high performance counters available in the POWER5 architecture we obtained the number of cycles per instruction (CPI) in Figure 2(b). CPI for the SD quicksort implementation is relatively flat, ranging from 1.21 when the number of records is small to 1.13 when the number of records is 10000000. On the other hand, CPI for the SP quicksort implementation rises quickly as the number of records increases.

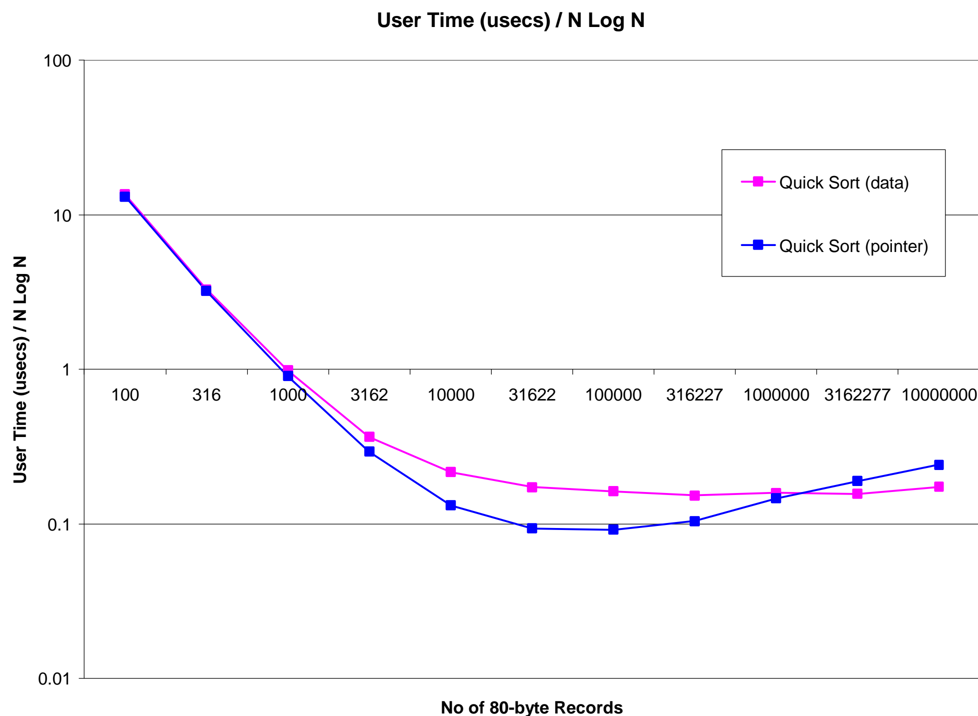

Figure 2(a). User time divided by N Log N, 80-byte DASD data sets

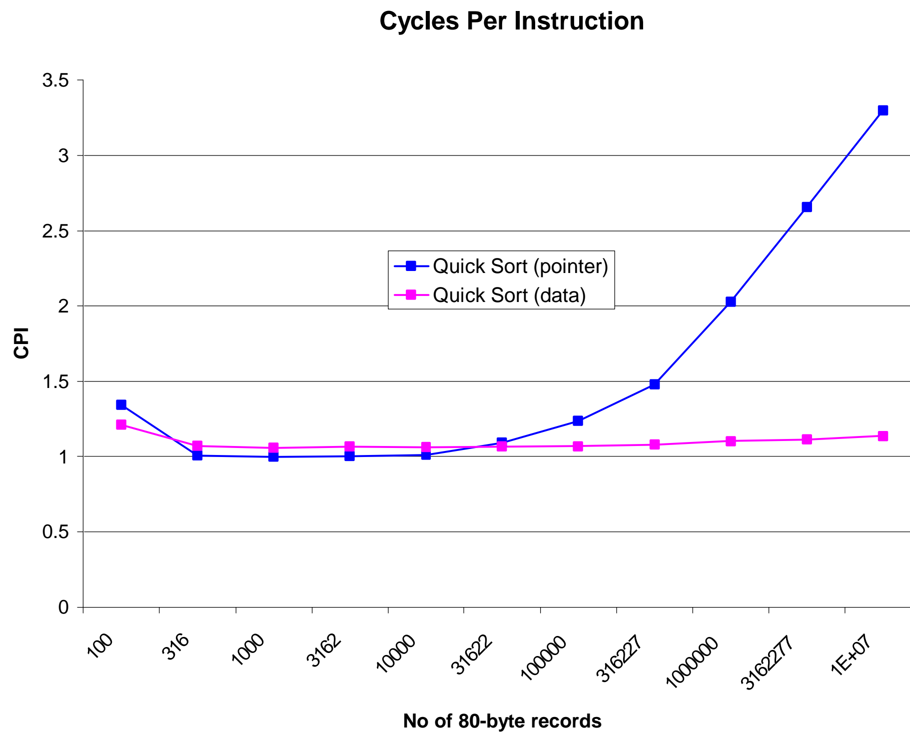

Figure 2(b). Cycles per instruction, fixed-length (80-byte) DASD data sets

The quicksort algorithm is shown in pseudo-C code for its simplicity in List 1. The initial lower and upper values are the lowest and the highest indexes in the array. The array is partitioned into sub-partitions repeatedly based on a chosen pivot. Pivots could be chosen randomly, however, a good choice should partition the records evenly into two sub-partitions. Multiple (say 3 or 5) indexes could be generated and the one in the middle could be used as the pivot.

```
void quicksort(array, lower, upper) {
  if (upper > lower) {
    select a pivot;
    swap(array, pivot, upper);
    m = lower;
    for (i = lower; i < upper - 1; i++) {
      if (array[i] <= array[upper]) swap(array, m++, i);
    }
    if (array[upper] <= array[m])
      swap(array, m, upper);
    else
      m = upper;
    quicksort(array, lower, m-1);
    quicksort(array, m+1, upper);
  }
}
```

List 1. Pseudo-C code for quicksort. The SD and SP implementations use the record array and the pointer array respectively.

Although the quick sort algorithm is well known for its outstanding average performance, its worst complexity could be $O(n^2)$ if the choices of pivots leaned to either side. We use random pivots in both the SD and SP implementations, so the poor performance of the SP implementation when N increases in Figure 2 must come from somewhere else. The SD implementation uses an array of records, and the SP implementation uses a pointer array with pointers pointing to individual records in the record array. Thus, each comparison in the SD implementation with the pivot involves sequential access to the records in the record array, while each comparison in the SP implementation causes the indirect access of a record, which, in turn, may cause delay in the memory hierarchy. We will examine where the cycles were spent later in this section.

Radix Sort

To provide comparison and increase our choices of sorting techniques we implemented the radix sort, also known as the distribution sort [9]. Radix sort is a sorting algorithm that sorts numbers by processing individual digits. Because numbers can represent strings of characters and specially formatted floating point numbers, radix sort is not limited to integers.

A least significant digit (LSD) radix sort has time complexity in $O(N \cdot K)$, where N is the number of records and K is the average key length. It serves as an alternative to other high-performance comparison-based sorting algorithms that require $O(N \log N)$ execution time. In radix sort each record is placed into one level of buckets corresponding to the value of the rightmost k bits of each key, where $k < K$. Each bucket preserves the original order of the records, thus making it a stable sorting algorithm. We implemented a non-recursive radix sort with double-linked lists for buckets, and re-arranged the records in place after iteration. The process repeats itself with the next neighboring k bits until there are no more bits to process.

```
void radixsort(array, k) {
  for each k bits in key {
    for (i = 0; i < array.size(); i++) {
      determine bucket number b using the k bits;
      buckets[b].add(array[i]);
    }
    i = 0;
    for (b = 0; b < 2^k; b++) {
      for (j = 0; j < buckets[b].size(); j++)
        swap(array, i++, buckets[b][j]);
      buckets[b].clear();
    }
  }
}
```

List 2. Radix sort algorithm.

A simplified radix sort algorithm is shown in List 2, where the partial key masked by the k bits in each record determines the bucket number for each record. We use a record pointer array and doubled-linked lists to implement buckets. At the end of each scan we re-arrange the pointer array in place, based on the order of records in the buckets. Similar to the SP quicksort implementation keys are indirectly accessed through pointers in the radix sort implementation, thus it may cause delay in the memory hierarchy.

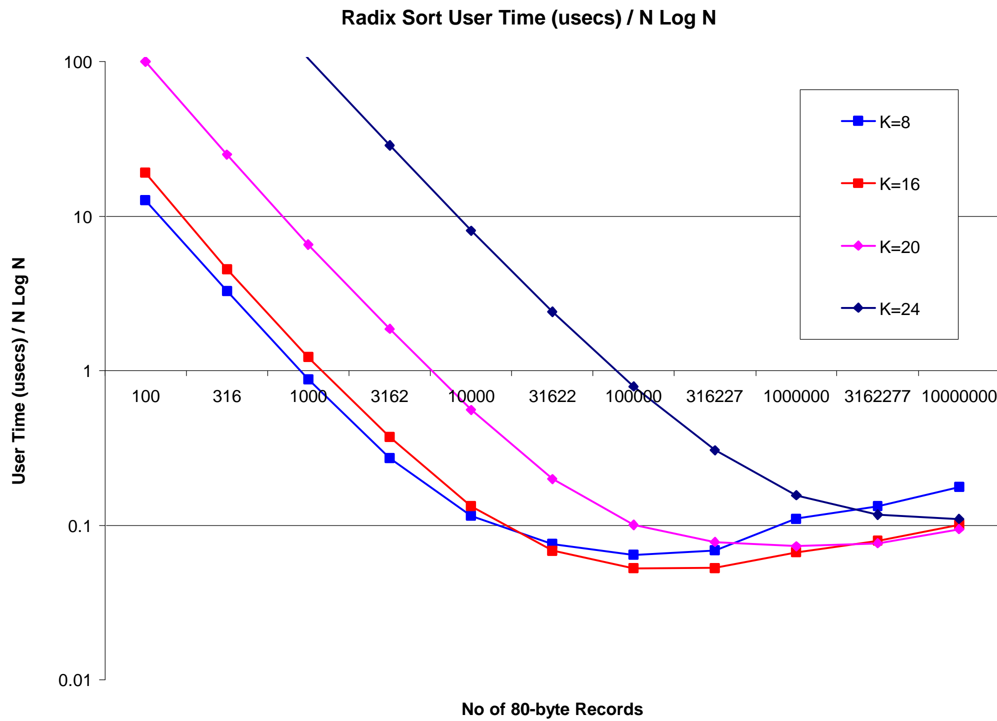

Figure 3. User time (micro seconds) with various k's in radix sort divided by N Log N

Figure 3 shows the user time in micro seconds divided by N Log N in radix sort for the values of k, where 2^k is the number of buckets used in the algorithm. Note that the key is a 10-byte EBCDIC number string. It can be seen that using a large k does not always help, as the number of buckets increases and many buckets could be empty. For example, there are 16 million buckets in the case of k = 24, however, we have only 1000 possible partial keys for a 3-byte EBCDIC number string. The number of empty buckets is determined by the key values, which could vary dramatically from one data set to another. In general, to reduce the number of empty buckets one could choose the largest k so that $2^k = N$, which may also decrease the user time.

CPI Stack: Where the Cycles Were Spent

Profiling is a common approach to collect timing and resource utilization for a workload. The POWER5 processor provides on-chip performance monitor units (PMUs) to record performance events through six performance monitor counters (PMCs). As a result, with an appropriate set of performance monitor application programming interfaces (PMAPIs) designed to provide access to those PMCs, we can profile many performance-sensitive events related to the core or the memory subsystem.

We use a CPI breakdown model similar to the one used in [10, 11] that breaks the CPI into a base component when the processor is completing work (group completed), and a stall component when the processor is not completing instructions (total cycles – group completed). The stall component is divided into cycles when the pipeline was empty (GCT empty) and cycles when the pipeline was not empty but completion is stalled (stall – GCT empty). The GCT empty cycles can be further partitioned into I-cache miss penalty, branch redirection penalty, and GCT others such as store stall and flush penalty. Completion stall cycles could be caused by a fixed-point unit (FXU), a floating-point unit (FPU), a load-store unit (LSU), or other units such as the branch

unit (BRU) or conditional register unit (CRU). LSU stall cycles could be further divided into cycles stalled due to D-cache miss penalty, LSU reject due to address translation, and LSU other reasons [10, 11].

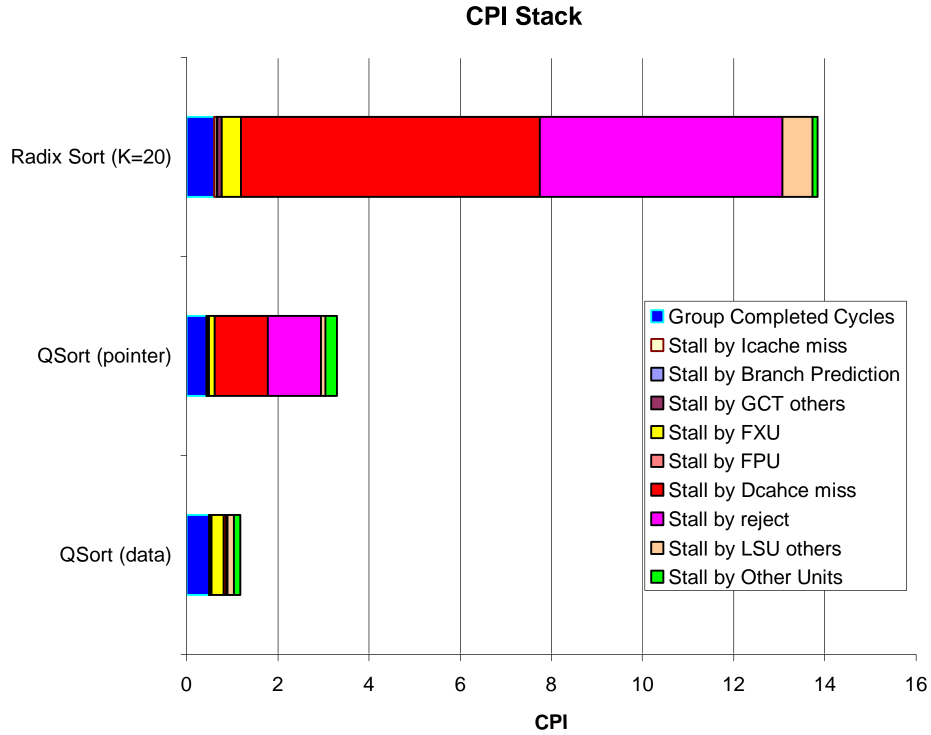

Figure 4(a). Cycles per instruction (CPI) stack

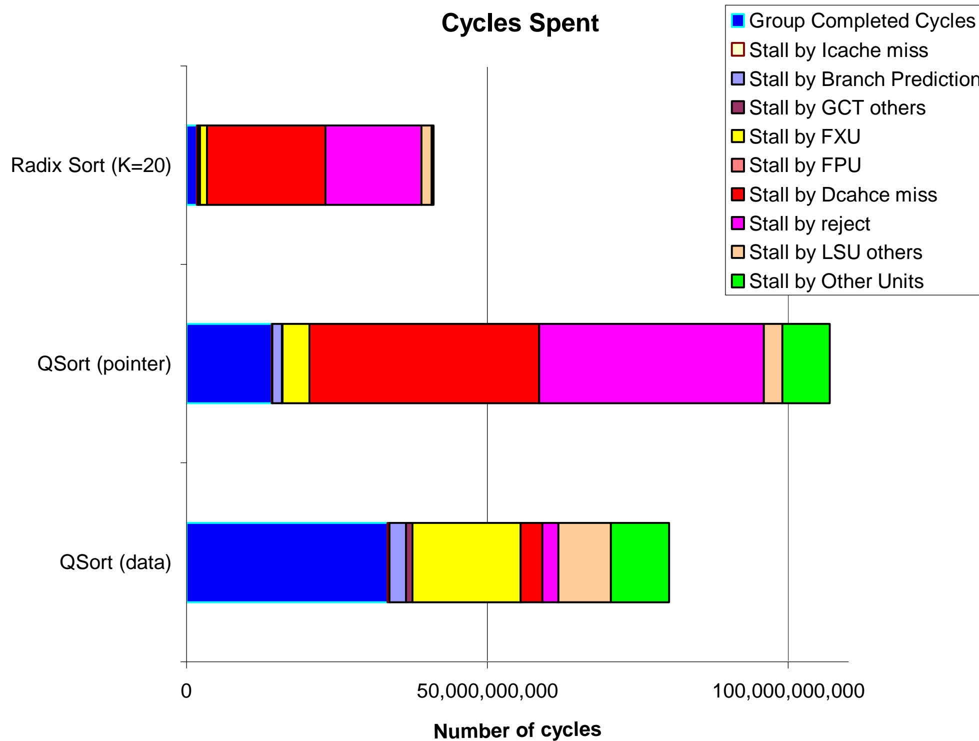

Figure 4(b). Cycles spent in SP/SD quicksort and radix sort

Figure 4(a) shows the CPIs for the three implementations sorting 10 million 80-byte records, with the CPI for the radix sort highest among the three. On the other hand, the number of completed instructions varies drastically. While the radix sort executed roughly 2.65 billion instructions, the SP and SD quicksort implementations completed more than 30 and 65 billion instructions, respectively. Note that the same quicksort algorithm is used for both the SP and SD implementations. The main difference is that the SD implementation uses the record array and swaps records, while the SP implementation uses the indirect pointer array and swaps pointers. Figure 4(b) shows where the cycles were spent or stalled by various units.

It can be seen from Figure 4(b) that the radix sort is the fastest, although its completion stall cycles made up more than 94% of the time. The stall cycles are mainly caused by LSU due to D-cache miss or LSU reject such as ERAT miss. The SP quicksort implementation suffers from more than 85% completion stall cycles, in which the majority was caused by LSU due to D-cache miss or LSU reject. Compared with the radix sort and the SP quicksort, the SD quicksort implementation has a much smaller stall by LSU D-cache miss (4.47%) and stall by LSU reject (3.3%). As the result, although the SD quicksort has to swap 88-byte records compared with swapping 8-byte pointers in the SP implementation, it still runs faster than the SP quicksort in this case.

SP Pointer Array with Primary Keys

One simple way to improve the performance of the SP quicksort implementation is to include the primary key in the pointer array. Comparisons between keys can then be done sequentially inside the pointer array without any indirect access, unless secondary keys, if any, are needed for comparison.

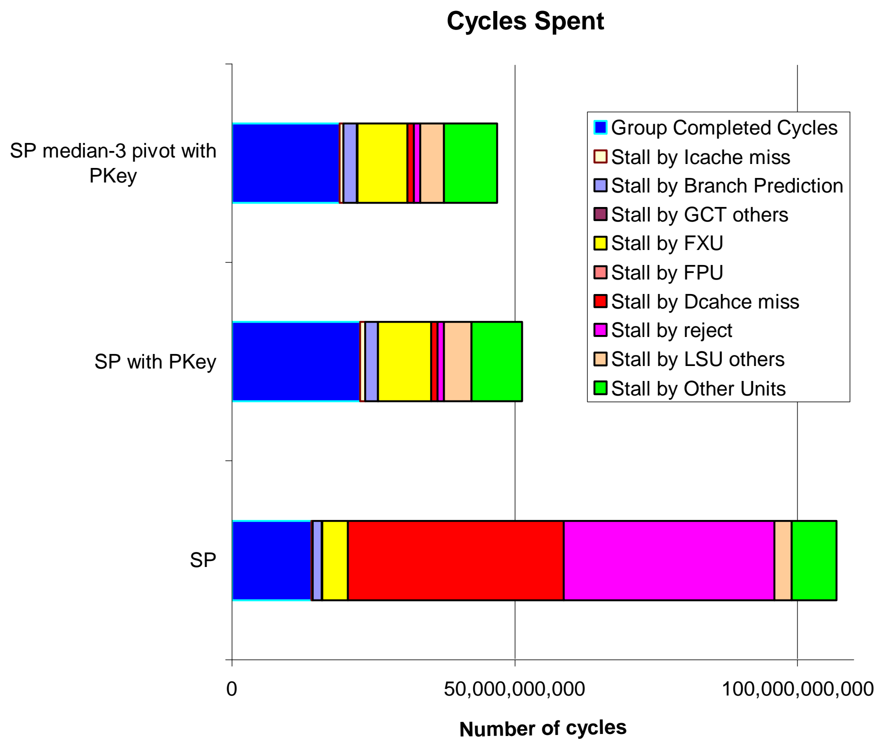

Figure 5. Cycles spent in (i) SP with PKey and median-3 pivots, (ii) SP with PKey, and (iii) original SP quicksort implementation without the primary key

In Figure 5 we use the data set with 10 million 80-byte records again to compare the three implementations. With the primary keys included in the pointer array, the first two implementations, one with median-3 pivots and the other with random pivots, have slightly more than 4% stall cycles due to D-cache miss or LSU reject. Note that more than 70% of the cycles were stalled due to the same D-cache miss or LSU reject for the original SP implementation without the primary key. Thus, the SP implementations with primary key in the pointer array indeed eliminate stalls in the LSU and improve performance.

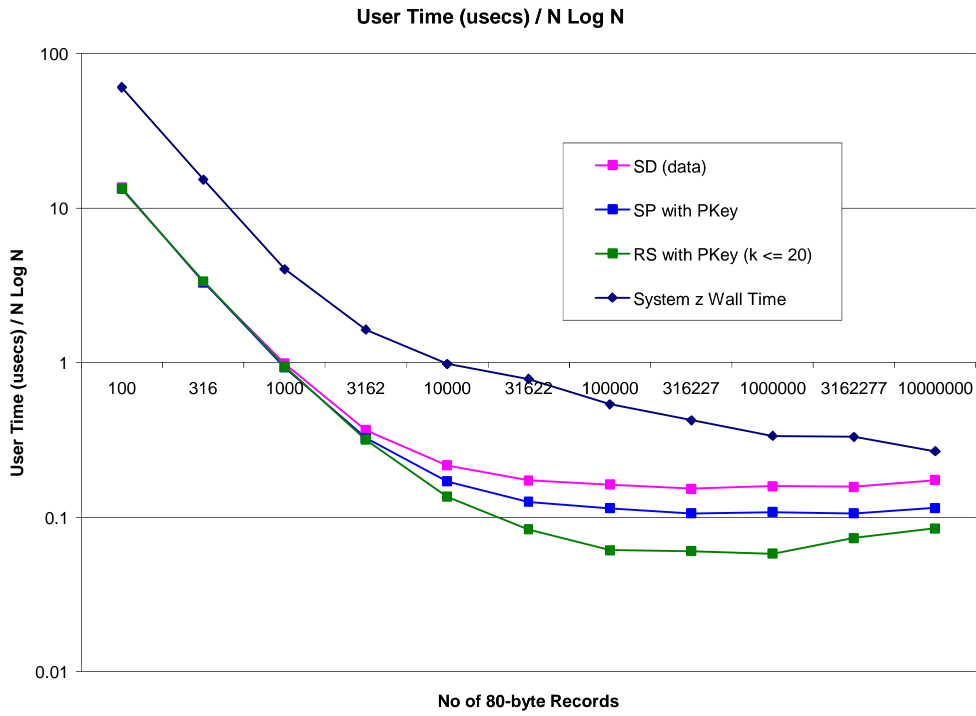

Figure 6(a). User time (micro seconds) divided by N Log N. The SP quicksort with primary key performs consistently better than the SD quicksort

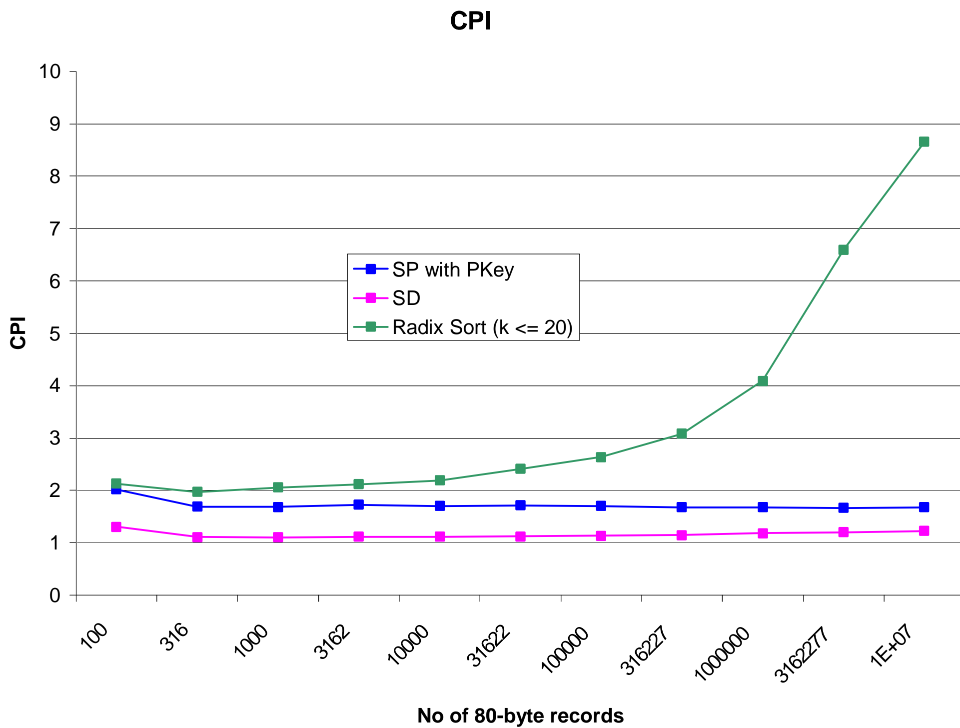

Figure 6(b). Cycles per instruction

Figure 6(a) shows the user time in micro seconds divided by N Log N, for the SD, SP with primary key, and radix sort implementations. The system z wall time in micro seconds divided by N Log N is also shown as a reference. The system z wall time is obtained by running a JCL script calling the SORT program for a given input data set. In general, the savings in sorting time is consistently 50% or more. It can be seen that the SP quicksort with the primary key included in the pointer array performs better than the SD quicksort, and both implementations show a flat CPI in Figure 6(b).

The copy-primary-key approach works well for the SP quicksort implementation, as its stall cycles reduced and CPI improved significantly. As can be seen in the previous section, the user time is literally cut in half, demonstrating that the choice of implementation is critical. Since quicksort performs better with balanced sub-partitions, using median-3 pivots help reduce the number of comparisons at the expense of extra random number generation and selecting the pivot. Using more random numbers to select pivots would have diminishing benefits. In addition, using insertion sort when the number of records in a sub-partition is less than certain threshold may also help reduce the user time.

Radix Sort with Primary Keys

The pointer array for the radix sort uses three pointers for each record: one pointer pointing to the record, one up-link and one down-link pointer are used to build the doubled-link lists for the buckets. Adding the primary key is one way to reduce the number of indirect accesses of the records.

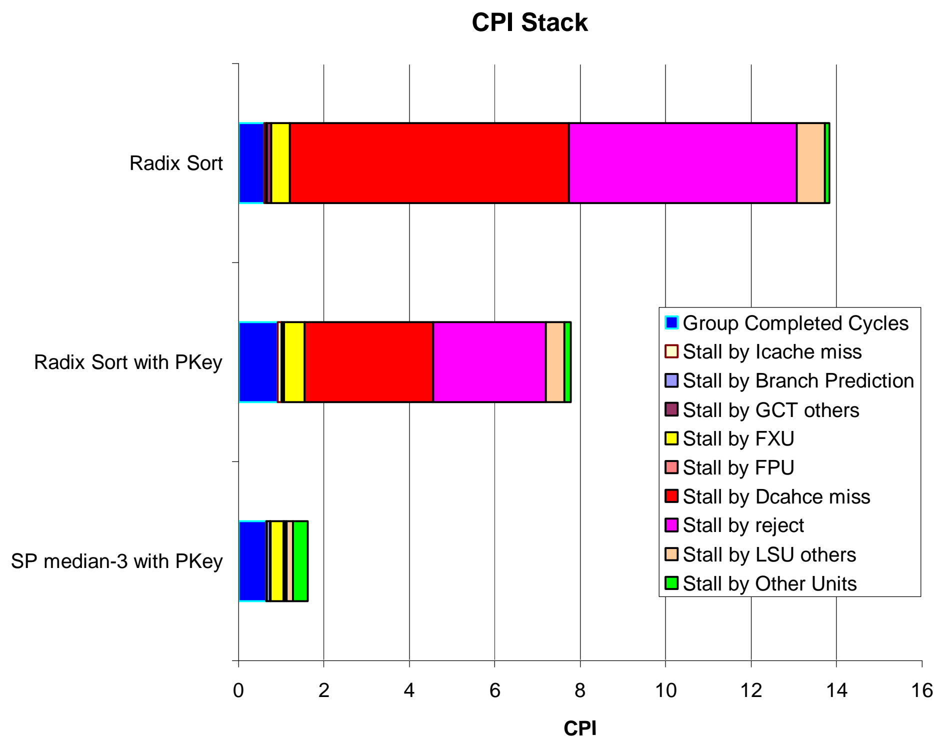

Figure 7(a). Cycles per instruction for (i) radix sort, (ii) radix sort with PKey, and (iii) SP with median-3 pivots and PKey

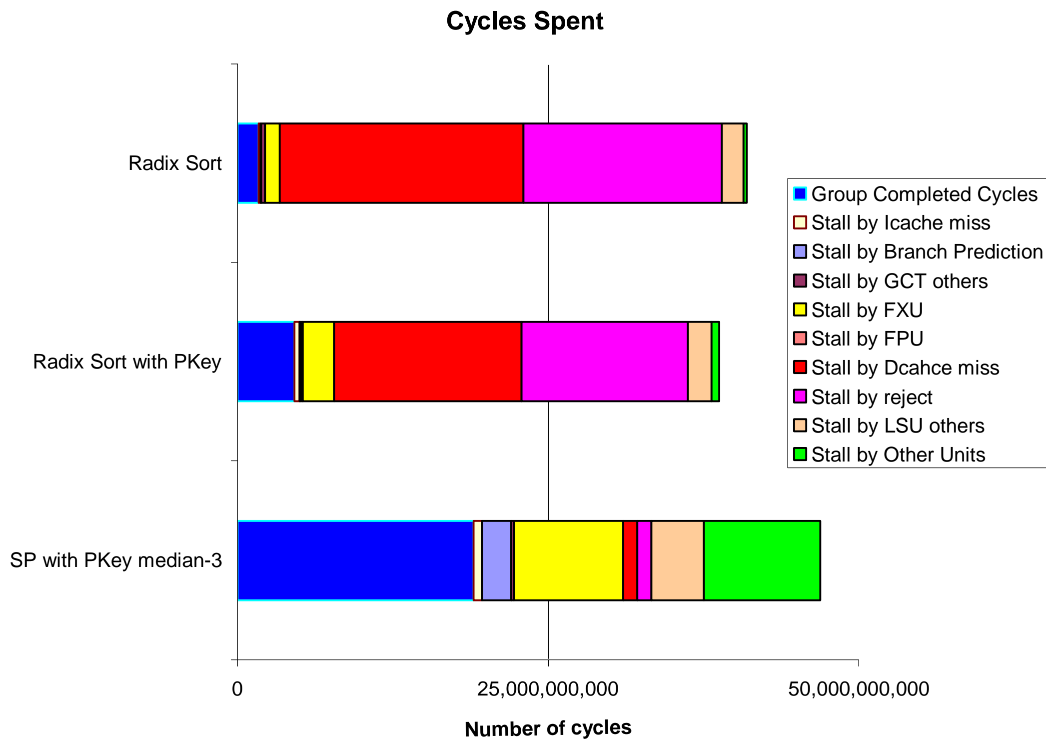

Figure 7(b). Cycles spent in (i) radix sort, (ii) radix sort with primary key, and (iii) SP quicksort with primary key

Figure 7(a) shows the CPI for radix sort, radix sort with primary key, and SP quicksort with median-3 pivots and primary key implementations using the data set with 10 million 80-byte records. It can be seen that with the primary key included in the pointer array the radix sort CPI drops substantially, from 13.8 to less than 8. The number of completed instructions increases from 2.65 billion to 4.4 billion while the total number of cycles drops slightly. The SP quicksort with median-3 pivots and primary key executed more than 27 billion instructions. Note that the CPI value in Figure 6(b) is calculated for the sorting only, which tends to be a little higher than that if the whole program is used.

For radix sort each structure in the pointer array includes an up-link and a down-link for building doubled-linked lists, as well as the record pointer and the primary key. Including the primary key in the pointer array helps reduce the CPI, although the CPI still goes up as the number of records increases, as shown in Figure 6(b). As we pointed out earlier that the key is a 10-byte EBCDIC number string, resulting in many empty buckets for the radix sort. The in-place placement at the end of each k-bit scan does not help either, since doubled-linked lists are made up of indirect pointers, and record pointers along with the primary keys are swapped through the use of these pointers. Our result does show slight advantage in user time for the radix sort with the primary key in the pointer array, as shown in Figure 7(b). As the user time drops below 20 seconds, the saving is close to 70% compared with the system z wall time. On the other hand, more experiments may be needed, with larger data sets and different keys.

Summaries

While we use blocked fixed-length datasets from System z as inputs in this paper, we have also developed library routines to access variable-length, blocked or unblocked sequential data sets

from DASD volumes. Since variable-length records are preceded with record lengths, a pointer to a variable-length record point to the actual starting point of the record content instead of the record-length field. Thus, the same sorting routines work for all four format combinations of a sequential dataset, i.e. fixed-length or variable-length, blocked or non-blocked.

Our experiences show that the radix sort is a fast stable sorting algorithm that may be used when the key length is relatively small. In the examples we have a 10-byte EBCDIC number string as the key, while the record length is 80 bytes. Although the key causes many empty buckets in the radix sort, it still runs faster than SP quicksort with the primary key for the data set with 10 million records. Whether it would stay faster for larger data sets remains to be seen, particularly if the key stays unchanged.

While the radix sort may be faster than the quicksort, the quicksort algorithm is more flexible. The quicksort is a comparison-based approach that can easily accommodate additional keys. Thus, the SP quicksort with the primary key implementation would be a good choice if the key length is relatively large or when there are multiple keys to compare. Our sorting implementation allows multiple keys stored with each record pointer in the pointer array, thus providing a flexible approach for multiple-key sorting operations. Although the result could be highly data dependent, we believe that our implementations provide a good starting point for sorting in AIX systems.

The library routines we developed to access datasets from DASD disks could be used for other applications as well. The potential with such a library is huge. Given such a library users of a UNIX system with links to mainframe disks could have the capability to access mainframe datasets and perform operations on behalf of the mainframe. Thus, the library can be used to offload operations from expensive mainframes to cheaper UNIX systems, saving precious mainframe cycles while possibly speeding up the operations.

References

1. Donald E. Knuth, "The Art of Computer Programming, Vol. 3, Sorting and Searching", Addison-Wesley, 1973.
2. Robert W. Floyd, "Permuting information in idealized two-level storage," Complexity of Computer Calculations, 1972.
3. Matteo Frigo, Charles E. Leiserson, H. Prokop, S. Ramachandran, "Cache-Oblivious Algorithms," ACM Proceedings of the 40th Annual Symposium on Foundations of Computer Science, 1999.
4. Demaine, Erik D., "Cache-Oblivious Algorithms and Data Structures," Lecture Notes in Computer Science, MIT Laboratory for Computer Science, 2002.
5. C. A. R. Hoare, "Quicksort," Comp. J. Vol. 5, pp. 10 – 15, 1962.
6. Kristoffer Vinther, "Engineering Cache-Oblivious Sorting Algorithms," Master's Thesis, University of Aarhus, Denmark, 2003.
7. IBM Manual, "DFSMS: Using Data Sets," SC26-7410-06, Sixth Edition, September 2005.
8. IBM Manual, "DFSMSdfp Advanced Services" SC26-7400-04, Fifth Edition, March 2005.
9. Donald E. Knuth, "The Art of Computer Programming, Vol. 3, Sorting and Searching", Section 5.2.5, Sorting by Distribution, pp. 170 – 178. Addison-Wesley, 1973.
10. Alex Mericas, "POWER5 Performance Monitor Programmer's Guide," IBM Confidential document, version 1.0, July 2004.

11. Duc Vianney, Alex Mericas, Bill Maron, Thomas Chen, Steve Kunkel, and Bret Olszewski, "CPI Analysis on POWER5, Part 2: Introducing the CPI Breakdown Model," IBM Developerworks, http://www-128.ibm.com/developerworks/power/library/pa-cpipower2, April 2006.