Architecture Independent Performance Characterization and Benchmarking for Scientific Applications

Erich Strohmaier, Hongzhang Shan
Future Technology Group
Lawrence Berkeley National Laboratory
One Cyclotron Road, CA 94720
{estrohmaier, hshan@lbl.gov}

Abstract

A simple, tunable, synthetic benchmark with a performance directly related to applications would be of great benefit to the scientific computing community. In this paper, we present a novel approach to develop such a benchmark. The initial focus of this project is on data access performance of scientific applications. First a hardware independent characterization of code performance in terms of address streams is developed. The parameters chosen to characterize a single address stream are related to regularity, size, spatial, and temporal locality. These parameters are then used to implement a synthetic benchmark program that mimics the performance of a corresponding code. To test the validity of our approach we performed experiments using five test kernels on six different platforms. The performance of most of our test kernels can be approximated by a single synthetic address stream. However in some cases overlapping two address streams is necessary to achieve a good approximation.

1. Introduction

When benchmarking computer systems ultimately the performance of our scientific application codes is most important to us. However using them for performance studies requires an enormous amount of time and effort and studies based on them tend to have limited architectural scope. This makes fair comparisons across architectural domains cumbersome, as they require a substantial effort for code adaptation and optimization. The amount of available results therefore tends to be limited. Application codes also tend to have a very limited meaningful life-time as benchmarks, as the actual applications tend to evolve and change at a rapid pace making special benchmark versions obsolete. Recent examples for this type of benchmarking efforts include SPEC HPC [20] and The Matrix [4].

Synthetic benchmarks are much easier to develop, maintain, and use. Synthetic benchmarks such as Linpack [1], NAS Parallel Benchmarks [2], PARKBENCH [3], or SPLASH2 [9] have been developed based on specific codes, which were selected ad-hoc. However, they only reflect the performance of a very narrow set of applications at best and cannot serve as general benchmark against which the progress in real application performance could be judged. It is therefore of great interest to develop a tunable synthetic benchmark, the performance of which we can relate to the performance of our application codes. This requires that we develop a parametric characterization of the performance behavior of scientific application codes first, which will then lead us directly to the design of such a synthetic benchmark.

The initial assumption of this study is that the performance behavior of codes can be characterized by a small set of performance factors, which are specific to the code and independent of the computer architecture. The performance of a class of codes with similar characteristic performance factors should then be closely related to each other on different architectures. A synthetic benchmark implemented such that its execution profile could be tuned by corresponding input parameters to match the chosen characteristic performance factors, could then be used as a proxy for the performance behavior of codes with similar characteristic parameters. Therefore such a benchmark can be used as a more realistic indicator of achievable performance on existing or new platforms. Since the design of the benchmark is only guided by the application characterization methodology and is independent of any specific architecture, the benchmark can be used for a long time and across all platforms. As a synthetic benchmark it greatly reduces maintenance and provides increased portability. Due its compact code size it also is substantially easier to use with simulators.

During the last decades memory access became the dominant performance factor for many codes. Therefore data access is the initial focus of our application performance characterization project (Apex). We developed a performance characterization for memory access and a corresponding benchmark – the memory access probe (Apex-MAP). Our performance characterization is based on assumptions about the relative importance of different aspects of data access patterns. In order to show the validity of these assumptions, we selected five scientific kernels and investigated their performance and the performance of Apex-MAP on six different platforms. Currently these include the IBM Power3 (375MHz and 200MHz), Power4, Intel Xeon, AMD Opteron, and the Cray X1. The kernels used are CG from the NAS benchmark suite, FFT, Nbody, and Radix Sorting from SPLASH2, and dense Matrix Multiplication.

We also need to choose an appropriate measure for how well our synthetic benchmark Apex-MAP approximates real application codes or how much of the intrinsic properties of a real code have been captured by Apex-MAP. Several different metrics have been adopted in different studies in the past. Metrics used include function-level execution profiles [16], linkage distance obtained through cluster analysis [10], cache-miss or hit ratio [6,7,12,17], the squared Euclidean distance [18], or statistical method [11]. For Apex-MAP we prefer to choose a metric closely related to performance. Our metric is the ratio of the average access time per memory access between our synthetic program and the real code. In the ideal case this ratio would always be one on all platforms for any problem size.

The remainder of this paper is organized as follows. Section 2 describes our application performance characterization process. Section 3 discusses the implementation of Apex-MAP. We describe the six platforms and the five kernels we selected in Section 4. The performance results of APEX-Map and the results of our validation study are discussed in section 5. Finally we summarize our findings and describe related ongoing work.

2. Application Performance Characterization Methodology

Starting point of this project is to develop a characterization of application performance behavior focusing initially on data access. Several methods for characterizing the data access behavior of codes have been proposed and studied previously [5,6,7,8,10,12,13,17,19]. D. Thiebaut, J.L. Wolf, and H.S. Stone [6] describe a characterization of memory traces using hyperbolic probability distributions with two parameters which correspond to the working set size and the locality of the reference. They measure the accuracy of this characterization in terms of cache-miss ratios and lifetime functions (the ratios of unique memory addresses to total memory accesses). Conte and Hwu [13,19] introduced the *inter-reference temporal density function* and *the inter-reference spatial density function* to measure temporal and spatial locality. The *inter-reference temporal density function f(x)* is defined as the probability of having *x* unique references between successive references to the same item. The *inter-reference spatial density function f(x)* is defined as the probability that between references to the same item, a reference to an item *x* units away occurs. Recently, M. Wang [17] introduced a statistical PQRS model to capture the temporal and spatial behavior of the applications as well as the correlations between them. These characterization methods are based on execution traces, which are affected by the specific compilers and architectures used, and thus not architecture independent. Most efforts to characterize codes are similarly based on low-level observations on the system level and reflect machine properties, rather than pure workload or code properties. In contrast our project develops a hardware independent characterization for code performance.

2.1 Main Factors of Performance

We assume that the data access of any code can be described as several concurrent streams of addresses which in turn can be characterized by a single, unique set of performance related factors. As performance factors for our characterization we chose:

- The regularity of the access pattern.
- The size of the data set accessed.
- The length of contiguous data access, which is also referred to as Spatial Locality.
- The above average re-use of data items, referred to as Temporal Locality.

The definition of the first three seems straightforward, while finding a truly hardware independent definition of the temporal locality of a code proves to be challenging. Several different methods to quantify temporal locality have been proposed [5,6,7,8], but most of them are hardware dependent or have other properties, which make them unsuited for our purpose.

Regularity We distinguish only two extreme categories of regularity: *random access* and *regular access*. Random access characterizes streams of addresses without regularity in the addresses. Examples would be address streams generated by pointer chasing or the address stream of an indirectly accessed data array. Regular access characterizes streams of addresses easily captured by regular expressions. Typical example would be stride access patterns or the stride one access to the index array of an indirect access pattern. Most of our current work is focused on code with irregular data access and we therefore ran most of our experiments using random access.

Data Set Size The amount of memory accessed *M* is evidently an important factor influencing the performance of an address stream. Its impact on performance is becoming more important with the increasing complexity of memory hierarchies in modern system architectures.

Spatial Locality We characterize spatial locality by the number of contiguous memory locations *L* accessed in succession (e.g. vector-length). Many modern systems benefit greatly from contiguous memory access patterns.

Temporal Locality and Re-Use To define the temporal locality of a code independent of hardware concepts such as cache sizes is difficult at best. We start by looking at the cumulative temporal distribution function of a memory accesses stream. To arrive at a problem size independent definition of temporal re-use we then approximate these temporal distribution functions by a scale-invariant power distribution function. The shape parameter (α) of the power distribution function is used to quantify data reuse ($0 <= \alpha <=1$). It turns out, that α is closely related to a properly defined, scale-invariant re-use number *k*, and thus allows for an easy interpretation. First we define a re-use number in such a way, that it is zero if the address stream in consideration accesses the complete data set of size *M* with maximum distance between two accesses to *any* memory location. One example for such an address stream can be generated by a single stride one access to a data array of size *M*. All equivalent address patterns can be generated from this pattern by permutations of the addresses. Any other access pattern would have access to at least one memory location within less than M cycles. For each address *X* in the address stream we now look at a window of the next *M*-1 addresses in the stream (Fig 1). Then we count how often *X* is accessed within this window. We now define our re-use number *k* as the average of all these counts over the whole address stream.
Next we define the cumulative temporal distribution function. Suppose *N* is the number of total memory accesses. Let *X* be the memory location of *n-th* memory access ($n <= N$). If *X* has been accessed earlier and the last access is the *m-th* memory access, then we say the temporal distance for *n-th* memory access is *n-m*. If *X* has not been accessed earlier, then we assume an infinite repetition of the whole address stream. The temporal distance is then given by $n + N – k$ and *k* is the location of last appearance of memory location *X* in the address stream. If *X* is only accessed one time, the temporal distance is *N*. The cumulative temporal distribution function is defined using this temporal distance and show for each temporal distance *t* the probability *p*, of accessing a memory location again within the next *t* memory accesses. Temporal distances distribution functions are very similar to the stack distances distribution functions [15] and the inter-reference temporal distributions [19,13] except that we count all the references instead of unique references when computing the distance.

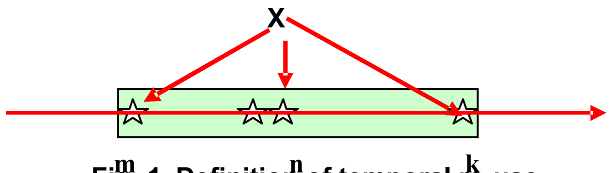

Fig. 1. Definition of temporal re-use

Fig. 2 displays the cumulative temporal distribution for FFT for three sizes. The data set of 16KB has a total of 784215 memory accesses. The reuse number for distance 1 is 49957, which gives a probability for immediate reuse of 49957/784215 = 6.3%. To describe these cumulative temporal distribution functions with a single number, we now approximate their shape by a generic function with only a single parameter other than *N*. By using a power function with a single shape parameter we can achieve such an approximation in a scale-invariant way. In practice this leaves us with the question how to construct a generic address stream with such a temporal distribution function. It turns out that an address stream generated with a power distributed random number generator approximates such a distribution very closely.
We therefore decided to use a non-uniform power function random address stream, to approximate the performance behavior of the address stream of codes. It turns out that the shape parameter α ($0<=\alpha<=1$) of the power function is related to the previously defined re-use number *k*. This can be seem by the existence of the fix-point of $p(M)=k/(k+1)$ for all possible address traces. The parameter α therefore defines temporal re-use as well as *k* and can be directly used in its place. The smaller the value of α is, the higher the re-use value *k*, and the higher the temporal locality. A value of α=0 means the program will access a single address repeatedly, while α=1 indicates a uniform random memory access.

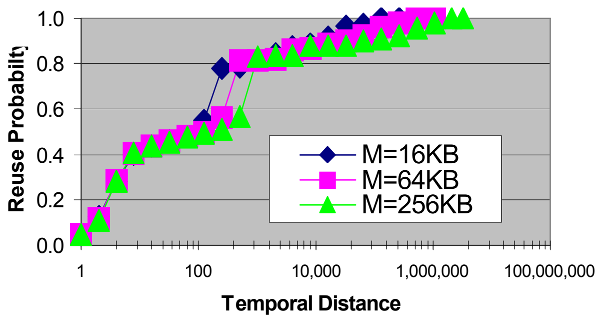

Fig. 2. Cumulative temporal distribution function of FFT for different sizes.

2.2 Determining Parameter Values for Code Characterization

Ideally we should be able to derive the parameters characterizing a code by analyzing its address trace. In practice we end up with cumulative temporal distribution functions and values of L and α, which depend on the effects of compilers and architectures used. In order to reduce such effects and to test the general validity of our ideas, we decided to initially use a statistical back-fitting procedure to find the characteristic parameters (α, L). On all platforms, we run our test applications using various data set sizes. Then, we compute the average time per memory access using the total runtimes divided by the total number of memory accesses. The total number of memory accesses is obtained from a reference platform (IBM Power3 using HPM toolkit [14]). Next, we run Apex-MAP using a variety of parameters (M = memory size used by codes; L = 1, 2, 4, …, 16384; α=0.001, 0.0025, 0.005, 0.01, …, 1.0). By this, we explore a very large part our parameter space.

We now determine a single parameter set (L, α) for each application, which fits the results on all systems simultaneously best. Let $y=(y_1,y_2, \ldots, y_n)$ represent the list of application's average access times for n different data set sizes, $x_{(L,\alpha)}=(x_1,x_2, \ldots, x_n)$ represent the corresponding Apex-MAP outputs for parameter set (L, α). We expect that a linear relationship between y and $x_{(L,\alpha)}$, i.e. $y » c\ x$ exists. The ideal case would be c=1, which however is usually not the case. Most codes include multiple access streams and the effects of these streams mix together. Imagine that one code has two access streams: one stream accesses a small amount of data in the cache while the other one accesses a large amount of data in the main memory. The performance of the code will be totally dominated by the second stream. In this situation, it will be good approximation to use the second stream to represent the code's memory behavior. The average time per memory access x however will be two times longer than y.

Table 1

The global R^2 for Radix (Columns: α; Rows: L).

	0.001	0.0025	0.005	0.010	0.025	0.05	0.1	0.25	0.5	1.0
1	0.7	0.9	0.8	0.5	-0.7	-2.0	-3.3	-4.7	-5.6	-5.6
2	0.6	0.8	0.9	0.7	0.0	-1.0	-2.1	-3.5	-4.6	-4.6
4	0.5	0.7	0.8	0.8	0.3	-0.4	-1.5	-2.8	-3.5	-3.8
8	0.3	0.6	0.8	0.8	0.6	0.0	-0.7	-1.8	-2.6	-2.7
16	0.2	0.5	0.7	0.8	0.7	0.4	-0.2	-1.2	-1.7	-2.0
32	0.2	0.4	0.5	0.7	0.8	0.6	0.3	-0.3	-0.7	-0.8
64	0.1	0.3	0.4	0.6	0.8	0.7	0.6	0.3	0.1	0.0
256	0.0	0.1	0.2	0.3	0.5	0.7	0.8	0.7	0.6	0.6
1024	0.0	0.1	0.1	0.2	0.4	0.6	0.8	0.8	0.7	0.6
4096	0.0	0.1	0.1	0.3	0.5	0.7	0.8	0.7	0.6	0.6
16384	0.0	0.1	0.2	0.3	0.6	0.8	0.7	0.1	-0.4	-0.6

We then use linear regression to examine which $x_{(L,\alpha)}$ has the best relationship with y on all platforms. This is done by computing the coefficient of determination R^2 across all platforms and selecting the highest value. The global R^2 values for radix are shown in Table 1. Different columns show different values of temporal reuse α and rows different vector length L. Since we assume a restricted linear relationship $y=a\ x_{(L,\alpha)}$ (and not a general linear model with intersection term $y=a\ x_{(L,\alpha)} + b$), the values of R^2 can be negative. However, no value can be greater than 1. The closer the value to 1, the greater is the degree of linear association between y and $x_{(L,\alpha)}$. We now chose the corresponding values (L, α) of the largest R^2 value as characteristic parameters for the code. The results for our test kernels are shown in Table 2.

Regular access is described by only two parameters (M, S), where S is the stride. In this case characteristic parameters are determined mainly by code inspection. We find that only the kernel MM-stride (described later) needs to be characterized with a regular access pattern and that S is equal to the matrix width of B in AB=C.

Table 2
The characterization parameters for the test applications.

	Radix	FFT	Nbody	MM	MM-Stride	CG
L	1	4096	4096	8	1	1
α	0.0025	0.1	0.025	0.005	0.25	1

3. Implementation of Apex-MAP

The synthetic benchmark, Apex-MAP, has six parameters. Four of them are from the application characterization. They are the amount of memory accessed (M), the temporal reuse of data (α), the vector length of data access (L), and the stride width (S). We distinguish only two extreme categories of regularity: random access and regular access. If codes are characterized as random access, Apex-MAP needs three parameters: M, L, and α. In the regular case, it needs only two parameters M and S. The other two parameters of Apex-MAP are the number of times to repeat the experiment (N) and the index buffer size (I) (for random access only). These parameters are necessary for the implementation of Apex-MAP itself.

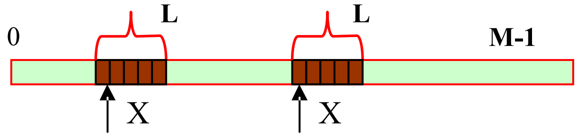

Fig. 3. The random memory access pattern of Apex-MAP.

Apex-Map uses indexed access to simulate random access streams as illustrated in Fig. 3. The random starting addresses (X) are aligned by length L and generated by a power distribution function controlled by the parameter α. Once an address is accessed, the following L continuous addresses will also be accessed. The starting addresses cannot be dynamically generated since the time needed to generate an address is too long compared with the memory access time. Therefore they have to be computed in advance and stored in an index buffer. Here is the core part of the Apex-Map code:

```
for ( i = 0; i < N; i++) {
   initIndexArray(I);
   CLOCK(time1);
   for (j = 0; j < I/4; j++) {
        pos  = ind[j*4];
        pos1 = ind[j*4+1];
        pos2 = ind[j*4+2];
        pos3 = ind[j*4+3];
        for (k = 0; k < L; k++) {
           res0 += data[pos  + k];
           res1 += data[pos1 + k];
           res2 += data[pos2 + k];
           res3 += data[pos3 + k];
        }
   }
   CLOCK(time2);
}
```

The *initIndexArray* function will compute the random starting addresses and fill them in an index buffer *ind* whose length is set by parameter I. The default value for I is 1024. The existence of this index array will influence the cache and occupy load/store units. Thus it has to be relatively small to reduce its impact on the average time per memory access. However, it also has to be large enough so that the measured time differences between *time1* and *time2* are accurate. The elapsed time between *time1* and *time2* is also corrected for the overhead of the timer call *CLOCK*. The index loop is unrolled four times as some compilers cannot automatically optimize the loop well on some platforms. We have to hand-optimize the loops by unrolling them to reduce compiler effects.

The main feature or our regular access pattern is stride access. The following code illustrates this:

```
for  ( i = 0; i < N; i++) {
   CLOCK (time1);
   for  (k = 0; k < m ; k++) {
      for  (j = k; j < M; j += S) {
         res0 += data[j];
      }
   }
   CLOCK (time2);
}
```

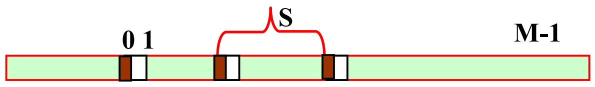

Fig. 4. The regular memory access pattern of Apex-MAP.

The first iteration starts from offset 0 and accesses the data array with stride S. The next iteration starts from offset 1 and so on as shown in Fig. 4. The reason to start from different offsets in successive iterations is to ensure that the access pattern uses the whole data array. The output of Apex-MAP is the average access time per memory access both in cycles and nanoseconds.

4. Test Platforms and Applications

4.1 Platforms

We selected six platforms using different processors, including IBM Power3 running at 200MHz, and at 375MHz, Cray X1 at 800MHz (vector processor), IBM Power4 at 1.3GHz, AMD Opteron at 1.6GHz, and Intel Xeon at 2.8GHz. The design philosophy and implementations vary widely among these processors and thus provide a good test-bed for our project. Further details can be found at the respective manufactures websites.

IBM Power3: Up to eight instructions can be executed per cycle. It can prefetch up to four streams of data from memory or L2 cache into L1 data-cache. A 256-entry two-way set associative TLB is used to access 4KB memory pages. The latency from L1 cache to register is 1 cycle with the width 2*8 bytes/cycle, from L2 to L1 is 6 to 7 cycles with the width 32 bytes/cycle, from memory to L2 or L1 is ~35 cycles with 16 bytes/2 cycles.

IBM Power4: The Power4 chip contains two 64-bit microprocessors, a interface controller, a 1.41MB L2 cache, a L3 cache directory, and other controllers. Eight independent units execute instructions in parallel. A processor is capable of tracking over 200 instructions in-flight. It is capable of managing up to eight data cache line requests to the L2 cache (and beyond). All data stored to cache lines that exist in the store-through L1 data cache are also sent to the L2 cache. It supports up to eight prefetch streams. Latency between registers and L1 is ~4 cycles; between registers and L2 ~14 cycles.

Intel Xeon: The Intel Xeon processor is based on the Intel NetBurst micro-architecture. It has a deep twenty-stage pipeline, allowing up to 126 instructions in flight, 48 loads and 24 stores in pipeline. The out-of-order execution core can dispatch up to six ops per cycle. The latency of the L1 data cache is 2 cycles. The L2 cache is a full-speed, unified on-die Advance Transfer Cache. It supports a quad-pumped, scalable bus clock for 4X effective speed, delivering up to 3.2GB/s of bandwidth.

AMD Opteron: The Opteron is a three-way superscalar processor. It's control unit can fetch, decode, and issue up to three instructions per cycle. The integer and floating-point scheduler can simultaneously issue up to nine micro-ops. The Opteron supports up to three coherent HyperTransport links, which provide up to 19.2GB/s peak bandwidth per processor, and has a 128-bit high-performance DDR SDRAM memory controller.

Cray X1: The single-streaming processor (SSP) of the X1, contains two vector pipes running at 800MHz. Each SSP contains 32 vector registers holding 64 double-precision words and can have up to 512 addresses simultaneously in flight. It contains a two-way out-of-order superscalar processor running at 400MHz. The multi-streaming processor (MSP) combines four SSPs into one logical computational unit. The four SSPs share a 2-way set associative 2MB data Ecache, a unique feature for vector architecture that allows extremely high bandwidth (25-51 GB/S). Our results are generated on SSP.

4.2 Test Kernels

We select five scientific kernels for this study:

Radix: Sorts integer keys using the radix algorithm. The radix size determines the number of iterations. In each iteration it computes first the histograms and then moves the keys according to the histograms. Two main arrays are alternating used as source and destination. The source data are read sequentially while the destination data are written scattered. Access to the histogram is random. The computational intensity (CI) is very low. Data movement dominates the code and increasing the data set size does not improve the CI.

FFT: A double-precision complex 1-dim FFT obtained from SPLASH2 suite. The data set is an array of n double-precision complex data points to be transformed, and an array of double-precision complex data to be used as roots of unity. There are total six steps: i) transpose matrix, ii) perform 1-D FFTs individually on rows of size $\sqrt{n}$, iii) multiplying the elements of the resulting complex matrix by the corresponding roots of unity, iv) transpose matrix, v) perform 1-D FFTs on individual rows, vi) transpose matrix. Without the transpose, the computational intensity is ~1.25 for n = 4K and slowly increasing with larger data sets.

Matrix-Multiplication (MM) and MM-stride: Matrix multiplication ($A * B = C$). The computational intensity of this regular kernel is around 1 and has no relation with data set size. There are two standard ways to program the code: one uses the sequential access (MM) and the other uses stride access (MM-stride). This is implemented by exchange the inner and outer loops. The following code is used for MM-stride. By exchange the loop j and loop k, we get MM. The performance of these two are substantially different from each other and characterized using different parameters.

```
For (i = 0; i < M; i++)
   For (j = 0; j < N; j++)
      For (k = 0; k < K; k++)
          C[i,j] += A[i,k] * B[k, j];
```

Nbody: Simulates the interaction of a system of bodies in 3 dim over a number of time steps, using a hierarchical N-body method. Browsing the oct-tree and calculating the gravitational force consume most of the time. Browsing the tree causes a lot of pointer tracing. However, the working set size is relatively small and the code shows very good caching behavior. The computational intensity is ~1.1 for 1k bodies and decrease gradually to 0.954 for 4M bodies.

CG: Solver for sparse linear systems using conjugate gradient method obtained from NAS benchmark suite. The main memory operations are random access to a small data set and sequential access to a large data set at the same time. The computational intensity is ~0.67.

5. Performance Results of Apex-MAP

We now analyze the ratios of the average access time between Apex-MAP and our test kernels to determine to what extent the performance of Apex-MAP can represent their performance. In the ideal case the ratio for a given kernel would not change with problem size and be =1. In this case Apex-MAP would completely explain the scaling behavior of the test codes. Considering the generality of our methodology we consider it satisfactory achieving a ±25% range. First we investigate, if a single random access streams can represent the kernels. Fig. 5. – 10. present the performance ratios for Radix, FFT, Nbody, MM, MM-stride, and CG individually. The Y-axis is the ratio of average access times. On the X-axis are the different memory sizes used by different problem sizes. Each line represents all the matching problem sizes on one specific platform. In the ideal case all lines are horizontal and overlap, i.e., the application's average memory access time can be accurately predicted by APEX-Map. We notice that for Radix, Nbody, and FFT, all the six lines are pretty close to Y=1 with almost all the points falling into the range 0.8 ~ 1.2. For MM, the range on the AMD Opteron becomes a little larger to 0.57 ~ 1.47. However, the results for CG on smaller data set and MM-Stride are not satisfactory. The ratios vary for MM-Stride between 0.35 ~ 1.56 and for CG between 0.89 ~ 1.85.

The main access pattern of the stridden matrix multiplication is stride access for matrix B and no reuse before the whole matrix has been accessed. For such codes, it will be difficult to use random access stream to accurately simulate the stride access and we use regular access instead. Since different matrix sizes will lead to different stride size, the characterization parameters are directly related with data set sizes. S equals the width of matrix B. Fig. 12. is the new ratio after using the regular access to characterize the stridden Matrix Multiplication.

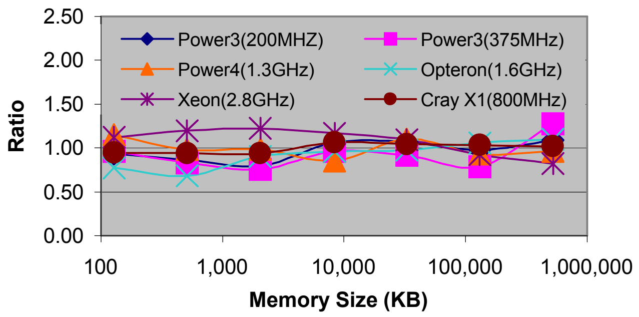

Fig. 5. The performance ratio for Radix using one random access stream

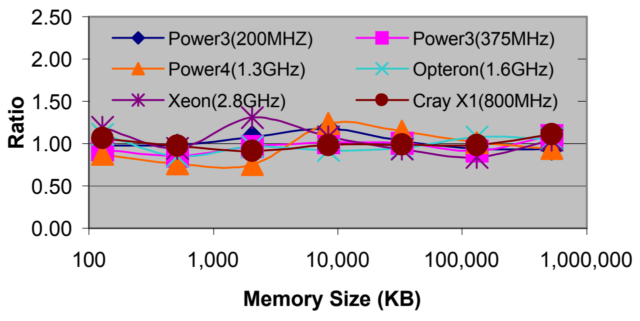

Fig. 6. The performance ratio for FFT using one random access stream

There are three main streams in CG: the stride-1 access to the large sparse matrix (column index and matrix-data) and the random access to the smaller vector. By only simulating the random access using Apex-MAP, we see (Fig. 10.) that the ratios are relatively high on Intel Xeon and AMD Opteron platforms for smaller data sets, but they fit well on the other platforms. This is mainly due to the inefficiency of these platforms for large sequential access patterns. Their average access time for large sequential access is higher than the average access time for the random access to a smaller data set. The effect of the sequential access cannot be ignored on these two processors. On the other platforms the random access dominates the average access times and sequential access times are negligible compared with random access times. Overall we find that the performance approximation by Apex-MAP can be improved by averaging the results of two different streams, one random access as above and one stride-1 access. The results for an approximation with two access streams are shown Fig. 11.

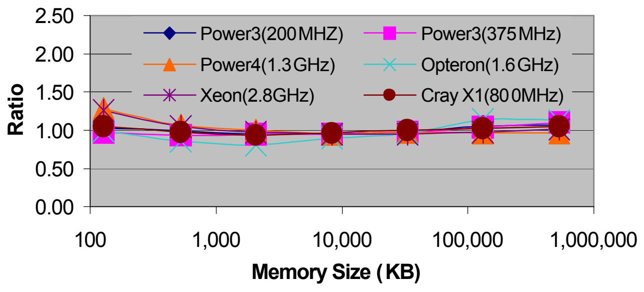

Fig. 7. The performance ratio for Nbody using one random access stream

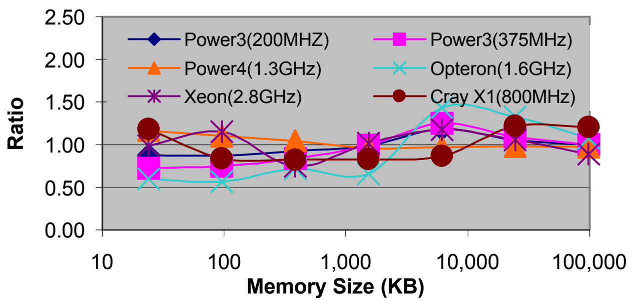

Fig. 8. The performance ratio for MM using one random access stream

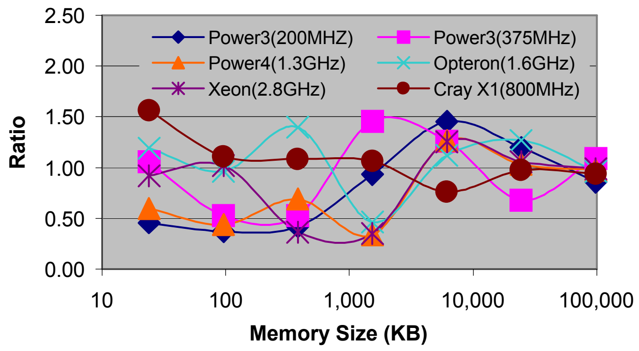

Fig. 9. The performance ratio for MM-Stride using one random access stream

6. Summary and Ongoing Work

In this paper, we describe a method to develop a tunable, synthetic benchmark, which can be used to approximate the performance of application codes. First a hardware independent performance characterization of data access patterns is developed. This characterization is used to implement a synthetic benchmark (Apex-MAP) with a data access behavior tunable by the parameters of the performance characterization. In this framework codes are characterized by a small set of parameters with the goal that the performance of the Apex-MAP benchmark operated with these parameters will reflect their performance. The validity and accuracy of this approach is measured on six platforms using five scientific kernels. In order to capture the performance scaling of the applications with 25%, we need only a single random access stream for Radix, FFT, and Nbody, a single regular access stream for Matrix Multiplication, and two streams (one random access and one regular access) for CG.

Our initial implementation and investigations were done for sequential execution. However, we have developed our characterization and its parameters so that it can be extended directly to include inter-process communication. The parallel version is currently under development.

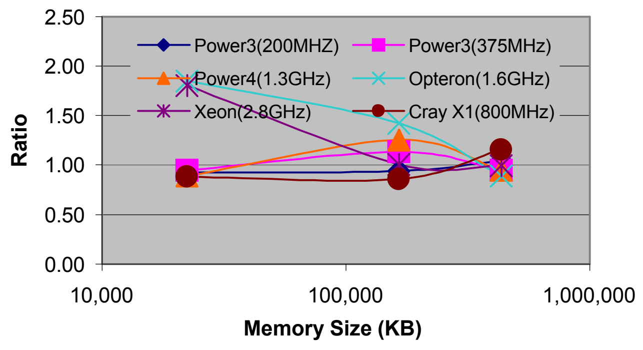

Fig. 10. The performance ratio for CG using one random access stream

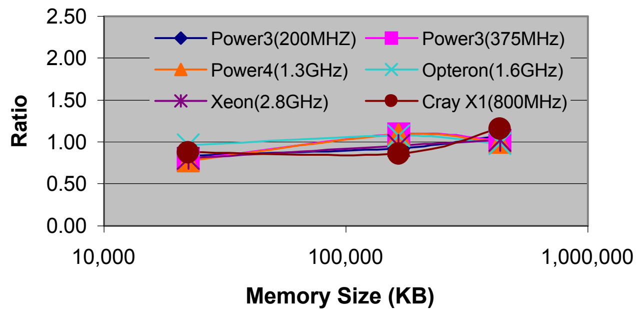

Fig. 11. The performance ratio for CG using two access stream

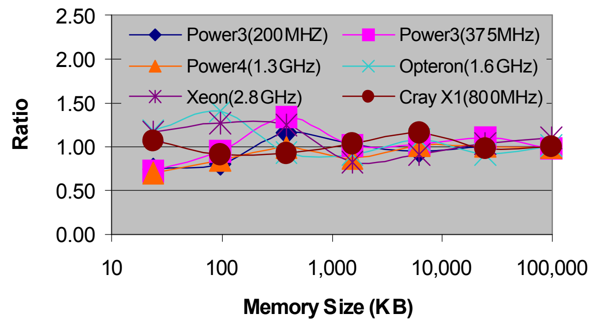

Fig. 12. The performance ratio for MM-Stride using one regular stream

References

1. Linpack, http://www.top500.org/lists/linpack.php.
2. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.
3. ParkBench, http://www.netlib.org/parkbench.
4. Application Performance Matrix, http://www.krellinst.org/matrix/
5. Elizabeth S. Sorenson, "Cache Characterization and Performance studies Using Locality Surfaces". http://citeseer.nj.nec.com/sorensen03cache.html.
6. Dominique Thiebaut, Joel L Wolf, and Harold S. Stone. "Synthetic traces for trace-driven simulation of cache memories". *IEEE Transactions on Computers*, 41(4):388-410, April 1992.
7. Kathryn S. Mckinley and Olivier Temam. "Quantifying loop nest locality using SPEC'95 and the perfect benchmarks". *ACM Transactions on Computer Systems*, 17(4):288-336, November, 1999.
8. F.J. Sanchez and A. Gonzalez. "Data locality analysis of the SPECFP'95". *Digest of Performance Analysis and its Impact on Design (PAID) Workshop*, p 78-84, 1998.
9. Stanford Parallel Applications for Shared Memory, http://www-flash.stanford.edu/apps/SPLASH/.
10. Lieven Eckhout, Hans Vandierendonck, and Koen De Bosschere, "Designing Computer Architecture Research Workloads", *IEEE Computer*, Vol. 36, No. 2, February 2003, pp. 65-71
11. A.K. Agrawala and J.M.Mohr, "A model for workload characterization", *Proceedings of the 1975 symposium on simulation of computer systems*, August 1975.
12. Wing Shing Wong, and Robert J.T. Morris, "Benchmark Synthesis Using the LRU Cache Hit Function", *IEEE Trans. on Computers*, Vol 37, No. 6, June 1988.
13. L. K. John, P. Vasudevan, and J. Sabarinathan, "Workload Charaterization: Motivation, Goals and Methodology", in *Workload Characterization: Methodology and Case Studies*, pp 3-14, IEEE Comp.Soc., 1999.
14. Hardware Performance Monitor (HPM) Toolkit, http://hpcf.nersc.gov/software/ibm/hpmcount/
15. Mark Brehob, Richard Enbody, "An Analytical model of Locality and Caching", *Technical Report*, Michigan State University, MSU-CSE-99-31.
16. A.J. KleinOsowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation – Based Computer Architecture Research", *Computer Architecture Letters*, June 2002, pp 10-13.
17. M. Wang, A. Ailamaki, C. Faloutsos, "Capturing the Spatio-Temporal Behavior of Real Traffic Data", *Performance 2002 (IFIP Intl. Symp. on Comp. Performance Modeling, Measurement, and Evaluation),* Rome, Italy.
18. R. H. Saavedra and A. J. Smith, "Analysis of Benchmark Characteristics and Benchmark Performance Prediction", *ACM Trans. Computer Systems*, Vol. 14, No. 4, pp. 344-384.
19. T. Conte and W-m W. Hwu, "Benchmark Characterization for Experimental System Evaluation", *Proceedings of the 1990 Hawaii International Conference on System Sciences (HICSS),* Vol. 1, Architecture Track, pp. 6-18.
20. Standard Performance Evaluation Corporation, http://www.specbench.org