On The Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications

Richard C. Murphy†, *Member, IEEE,* Peter M. Kogge, *Fellow, IEEE*

***Abstract*— This paper compares the SPEC Integer and Floating Point suites to a set of real-world applications for high performance computing at Sandia National Laboratories. These applications focus on the high-end scientific and engineering domains, however the techniques presented in this paper are applicable to any application domain. The applications are compared in terms of three memory properties: first,** *temporal locality* **(or reuse over time); second,** *spatial locality* **(or the use of data "near" data that has already been accessed); and third,** *data intensiveness* **(or the number of unique bytes the application accesses). The results show that real world applications exhibit significantly less spatial locality, often exhibit less temporal locality, and have much larger data sets than the SPEC benchmark suite. They further quantitatively demonstrates the memory properties of real supercomputing applications.**

***Index Terms*— B.8.2 Performance Analysis and Design Aids, C.1.0 General, C.4.c Measurement techniques, C.4.g Measurement, evaluation, modeling, simulation of multiple-processor systems**

I. Introduction

The selection of benchmarks relevant to the supercomputing community is challenging at best. In particular, there is a discrepancy between the workloads that are most extensively studied by the computer architecture community, and the codes relevant to high performance computing. This paper examines these differences *quantitatively* in terms of the memory characteristics of a set of a real applications from the high-end science and engineering domain as they compare to the SPEC CPU2000 benchmark suite, and more general High Performance Computing (HPC) benchmarks. The purpose of this paper is two-fold: first to demonstrate what general memory characteristics the computer architecture community should look for when identifying benchmarks relevant to HPC (and how they differ from SPEC); and second, to quantitatively explain application's memory characteristics to the HPC community, which often relies on intuition when discussing memory locality. Finally, although most studies discuss temporal and spatial locality when referring to memory performance, this work introduces a new measure *data intensiveness* that serves as the biggest differentiator in application properties between real applications and benchmarks. These techniques can be applied to any architecture-independent comparison of any benchmark or application suite, and this study could be repeated for other markets of interest. The three key characteristics of the application are:

1) Temporal Locality: the reuse over time of a data item from memory;
2) Spatial Locality: the use of data items in memory near other items that have already been used; and,
3) Data Intensiveness: the amount of unique data the application accesses.

Quantifying each of these three measurements is extremely difficult (see Section IV and Section II). Beyond quantitatively defining the measurements, there are two fundamental problems: first, choosing the applications to measure; and second, performing the measurement in an architecture-independent fashion that allows general conclusions to be drawn about the application rather than specific observations of the applications performance on one particular architecture. This paper addresses the former problem by using a suite of real codes that consume significant compute time at Sandia National Laboratories; and it addresses the latter by defining the metrics to be orthogonal to each other, and measuring them in an architecture independent fashion. Consequently, these results (and the techniques used to generate them) are applicable for comparing any set of benchmarks or applications memory properties without regard to how those properties perform on any particular architectural implementation.

The remainder of this paper is organized as follows: Section II examines the extensive related work in measuring spatial and temporal locality, as well as application's working sets; Section III describes the Sandia integer and floating point applications, as well as the SPEC suite; Section IV quantitatively defines the measures of temporal locality, spatial locality, and data intensiveness; Section V compares the application's properties; Section VI presents the results; and Section VII ends with the conclusions.

II. Related Work

Beyond the somewhat intuitive definitions of spatial and temporal locality provided in computer architecture text books [14], [27], there have been numerous attempts to quantitatively define spatial and temporal locality [37]. Early research in computer architecture [7] examined *working sets*, or the data actively being used by a program, in the context of paging. That work focused on efficiently capturing the working set in limited core memory, and has been an active area of research [9], [31], [35].

†Richard Murphy is at Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

More recent work is oriented towards addressing the memory wall. It examines the spatial and temporal locality properties of cache accesses, and represent modern hierarchical memory structures [6], [10], [32]. Compiler writers have also extensively examined the locality properties of applications in the context of code generation and optimization [11], [12].

In addition to definitions that are continuously refined, the methodology for modeling and measuring the working set has evolved continuously. Early analytical models were validated by small experiments [7], [9], [31], [35], while modern techniques have focused on the use of trace-based analysis and full system simulation [18], [21], [32].

Because of its preeminence in computer architecture benchmarks, the SPEC suite has been extensively analyzed [12], [18], [19], [21], [33], as have other relevant workloads such as database and OLTP applications [3], [6], [20].

Finally, the construction of specialized benchmarks such as the HPC Challenge RandomAccess benchmark [8] or the STREAM benchmark [22] is specifically to address memory performance. Real world applications on a number of platforms have been studied extensively [26], [36].

III. Applications and Benchmarks

This section describes a set of floating point and integer applications from Sandia National Laboratories, as well as the SPEC Floating Point and Integer benchmark suites to which they will be compared. The HPC Challenge RandomAccess benchmark, which measures random memory accesses in GUPS (Giga-Updates Per Second), and the STREAM benchmark, which measures effective bandwidth, are used as comparison points to show very basic memory characteristics. Finally, LINPACK, the standard supercomputing benchmark used to generate the Top500 list is included for comparison. In the case of MPI codes, the user portion of MPI calls is included in the trace.

A. Floating Point Benchmarks

Real scientific applications tend to be significantly different from common processor benchmarks, such as the SPEC suite. Their datasets are larger, the applications themselves are more complex, and they are designed to run on large-scale machines. The following benchmarks were selected to represent critical problems in supercomputing seen by the largest scale deployments in the United States. The input sets were all chosen to be representative of real problems, or, when they are benchmark problems, are the typical performance evaluation benchmarks used during new system deployment. Two of the codes are benchmarks, sPPM (see Section III-A.5), which is part of the ASCI 7x benchmark suite (that set requirements for the ASCI Purple supercomputer), and Cube3 which is used as a simple driver for the Trilinos linear algebra package. The sPPM code is a slightly simplified version of a real-world problem, and, in the case of Trilinos, linear algebra is so fundamental to many areas of scientific computing that studying core kernels is significantly important.

All of the codes are written for MPPs using the MPI programming model, but, for the purposes of this study, were traced as a single node run of the application. Even without the use of MPI the codes are structured to be MPI-scalable. Other benchmarking (both performance register and trace-based) has shown that the local memory access patterns for a single node of the application and serial runs are substantially the same.

1) LAMMPS: LAMMPS represents a classic molecular dynamics simulation designed to represent systems at the atomic or molecular level [28], [29]. The program is used to simulate proteins in solution, liquid crystals, polymers, zeolites, and simple Lenard-Jones systems. The version under study is written in C++, and two significant inputs were chosen for analysis:

- *Lenard Jones Mixture*: This input simulated a 2048 atom system consisting of three different types;
- *Chain*: simulates 32000 atoms and 31680 bonds.

LAMMPS consists of approximately 30,000 lines of code.

2) CTH: CTH is a multi-material, large deformation, strong shock wave, solid mechanics code developed over the last three decades at Sandia National Laboratories [16]. CTH has models for multi-phase, elastic viscoplastic, porous and explosive materials. CTH supports multiple types of meshes:

- Three-dimensional rectangular meshes;
- two-dimensional rectangular and cylindrical meshes; and
- one-dimensional rectilinear, cylindrical, and spherical meshes.

It uses second-order accurate numerical methods to reduce dispersion and dissipation and produce accurate, efficient results. CTH is used extensively within the Department of Energy laboratory complexes for studying armor/anti-armor interactions, warhead design, high explosive initiation physics and weapons safety issues. It consists of approximately 500,000 lines of Fortran and C.

CTH has two modes of operation: with or without adaptive mesh refinement (AMR)[1]. Adaptive mesh refinement changes the application properties significantly and is useful for only certain types of input problems. One AMR problem and two non-AMR problems were chosen for analysis.

Three input sets were examined:

- **2-Gas:** The input set uses an $80 \times 80 \times 80$ mesh to simulate two gases intersecting on a 45 degree plane. This is the most "benchmark-like" (e.g., simple) input set, and is included to better understand how representative it is of real problems.
- **Explosively Formed Projectile (EFP):** The simulation represents a simple Explosively Formed Projectile (EFP) that was designed by Sandia National Laboratories staff. The original design was a combined experimental and modeling activity where design changes were evaluated computationally before hardware was fabricated for testing. The design features a concave copper liner that is formed into an effective fragment by the focusing of shock waves from the detonation of the high explosive.

[1] AMR typically uses graph partitioning as part of the refinement, two algorithms for which are part of the integer benchmarks under study. One of the most interesting results of including a code like CTH in a "benchmark suite" is its complexity.

The measured fragment size, shape, and velocity is accurately (within 5%) modeled by CTH.

- **CuSt AMR:** This input problem simulates a 4.52 km/s impact of a 4 mm copper ball on a steel plate at a 90 degree angle. Adaptive mesh refinement is used in this problem.

3) Cube3: Cube3 is meant to be a generic linear solver and drives the Trilinos [15] frameworks for parallel linear and eigensolvers. Cube3 mimics a finite element analysis problem by creating a beam of hexagonal elements, then assembling and solving a linear system. The problem can be varied by width, depth, and degrees of freedom (e.g., temperature, pressure, velocity, or whatever physical modeling the problem is meant to represent). The physical problem is three dimensional. The number of equations in the linear system is equal to the number of nodes in the mesh multiplied by the degrees of freedom at each node. There are two variants based on how the sparse matrices are stored:

- **CRS:** a 55x55 sparse compressed row system; and
- **VBR:** a 32x16 variable block row system.

These systems were chosen to represent a large system of equations.

4) MPSalsa: MPSalsa performs high resolution 3D simulations of reacting flow problems [34]. These problems require both fluid flow and chemical kinetics modeling.

5) sPPM: The sPPM [5] benchmark is part of the ASCI Purple benchmark suite as well as the 7× application list for ASCI Red Storm. It solves a 3D gas dynamics problem on a uniform Cartesian mesh using a simplified version of the PPM (Piecewise Parabolic Method) code. The hydrodynamics algorithm requires three separate sweeps through the mesh per time step. Each sweep requires approximately 680 flops to update the state variables for each cell. The sPPM code contains over 4000 lines of mixed Fortran 77 and C routines. The problem solved by sPPM involves a simple, but strong (about Mach 5) shock propagating through a gas with a density discontinuity.

B. Integer Benchmarks

While floating point applications represent the classic supercomputing workload, problems in discrete mathematics, particularly graph theory, are becoming increasingly prevalent. Perhaps most significant of these are fundamental graph theory algorithms. These routines are important in the fields of proteomics, genomics, data mining, pattern matching and computational geometry (particularly as applied to medicine). Furthermore, their performance emphasizes the critical need to address the von Neumann bottleneck in a novel way. The data structures in question are very large, sparse, and referenced *indirectly* (e.g., through pointers) rather than as regular arrays. Despite their vital importance, these applications are significantly underrepresented in computer architecture research, and there is currently little joint work between architects and graph algorithms developers.

In general, the integer codes are more "benchmark" problems (in the sense that they use non-production input sets), heavily weighted towards graph theory codes, than are the floating point benchmarks.

1) Graph Partitioning: There are two large-scale graph partitioning heuristics included here: Chaco [13] and Metis [17]. Graph partitioning is used extensively in automation for VLSI circuit design, static and dynamic load balancing on parallel machines, and numerous other applications. The input set in this work consists of a 143,437 vertex and 409,593 edge graph to be partitioned into 1,024 balanced parts (with minimum edge cut between partitions).

2) Depth First Search (DFS): DFS implements a Depth First Search on a graph with 2,097,152 vertices and 25,690,112 edges. DFS is used extensively in higher-level algorithms, including identifying connected components, tree and cycle detection, solving the two-coloring problem, finding Articulation Vertices (e.g., the vertex in a connected graph that, when deleted, will cause the graph to become a disconnected graph), and topological sorting.

3) Shortest Path: Shortest Path computes the shortest path on a graph of 1,048,576 vertices and 7,864,320 edges, and incorporates a breadth first search. Extensive applications exist in real world path planning and networking and communications.

4) Isomorphism: The graph isomorphism problem determines whether or not two graphs have the same shape or structure. Two graphs are isomorphic if there exists a one-to-one mapping between vertices and edges in the graph (independent of how those vertices and edges are labeled). The problem under study confirms that two graphs of 250,000 vertices and 10 million edges are isomorphic. There are numerous applications in finding similarity (particularly, subgraph isomorphism) and relationships between two differently labeled graphs.

5) BLAST: The Basic Local Alignment Search Tool (BLAST) [1] is the most heavily used method for quickly searching nucleotide and protein databases in biology. The algorithm attempts to find both local and global alignment of DNA nucleotides, as well identifying regions of similarity embedded in two proteins. BLAST is implemented as a dynamic programming algorithm.

The input sequence chosen was obtained by training a hidden Markov model on approximately 15 examples of piggyBac transposons from various organisms. This model was used to search the newly assembled aedes aegypti genome (a mosquito). The best result from this search was the sequence used in the blast search. The target sequence obtained was blasted against the entire aedes aegypti sequence to identify other genes that could be piggyBac transposons, and to double check that the subsequence is actually a transposon.

6) zChaff: The zChaff program implements the Chaff heuristic [23] for finding solutions to the Boolean satisfiability problem. A formula in propositional logic is *satisfiable* if there exists an assignment of truth values to each of its variables that will make the formula true. Satisfiability is critical in circuit validation, software validation, theorem proving, model analysis and verification, and path planning. The zChaff input comes from circuit verification and consists of 1,534 Boolean variables, 132,295 clauses with five instances, that are all satisfiable.

TABLE I
SPEC CPU2000 INTEGER SUITE

Benchmark	Lang.	Description
164.gzip	C	Data Compression
175.vpr	C	FPGA Placement and Routing
176.gcc	C	GNU C Compiler
181.mcf	C	Combinatorial Optimization
186.crafty	C	Chess
197.parser	C	Word Processing
252.eon	C++	Visualization
253.perlbmk	C	PERL
254.gap	C	Group Theory
255.vortex	C	Object Oriented Database
256.bzip2	C	Data compression
300.twolf	C	VLSI Placement and Routing

TABLE II
SPEC FLOATING POINT SUITE

Benchmark	Lang	Description
168.wupwise	F77	Quantum Chromodynamics
171.swim	F77	Shallow Water modeling
172.mgrid	F77	Multi-grid Solver
173.applu	F77	Parabolic PDEs
177.mesa	C	3d Graphics
178.galgel	F90	Comp. Fluid Dynamics
179.art	C	Adaptive Resonance Theory
183.equake	C	Seismic Wave Propagation
187.facerec	F90	Face Recognition
188.ammp	C	Computational Chemistry
189.lucas	F90	Primary Number Testing
191.fma3d	F90	Finite Element Crash Simulation
200.sixtrack	F77	High Energy Physics Accelerator
301.apsi	F77	Pollutant Distribution

C. SPEC

The SPEC CPU2000 suite is by far the most currently studied benchmark suite for processor performance [4]. This work uses both the SPEC-Integer and SPEC-FP components of the suite, as summarized in Tables I and II respectively, as its baseline comparison for benchmark evaluation.

1) SPEC Integer Benchmarks: The SPEC Integer Suite, summarized in Table I, is by far the most studied half of the SPEC suite. It is meant to generally represent workstation class problems. Compiling (176.gcc), compression (164.gzip and 256.bzip2), and systems administration tasks (253.perlbmk) have many input sets in the suite. These tasks tend to be somewhat streaming on average (the perl benchmarks, in particular, perform a lot of line-by-line processing of data files). The more scientific and engineering oriented benchmarks (175.vpr, 181.mcf, 252.eon, 254.gap, 255.vortex, and 300.twolf) are somewhat more comparable to the Sandia integer benchmark suite. However selectively choosing benchmarks from SPEC produces generally less accurate comparisons than using the entire suite (although it would lessen the computational requirements for analysis significantly).

It should be noted that the SPEC suite is specifically designed to emphasize computational rather then memory performance. Indeed, other benchmark suites, such as the STREAM benchmark or RandomAccess focus much more extensively on memory performance. However, given the nature of the memory wall, what is important is a mix of the two. SPEC, in this work, represents the baseline only because it is, architecturally, the most studied benchmark suite. Indeed, a benchmark such as RandomAccess would undoubtedly overemphasize the memory performance at the expense of computation, as compared to the real-world codes in the Sandia suite.

2) SPEC Floating Point Benchmarks: The SPEC Floating Point suite is summarized in Table II, and primarily represents scientific applications. At first glance, these applications would appear very similar to the Sandia Floating Point suite; however the scale of the applications (in terms of execution time, code complexity, and input set size) differs significantly.

D. RandomAccess

The RandomAccess benchmarks is part of the HPC Challenge suite [8] and measures the performance of the memory system by updating random entries in a very large table that is unlikely to be cached. This benchmark is specifically designed to exhibit very low spatial and temporal locality, and a very large data set (as the table update involves very little computation). It represents the most extreme of memory intensive codes, and is used as a comparison point to the benchmarks and real applications in this work.

E. STREAM

The STREAM benchmark [22] is used to measure sustainable bandwidth on a platform and does so via four simple operations performed non-contiguously on three large arrays:

- **Copy:** $a(i) = b(i)$
- **Scale:** $a(i) = q * b(i)$
- **Sum:** $a(i) = b(i) + c(i)$
- **Triad:** $a(i) = b(i) + q * c(i)$

To measure the performance of main memory, the STREAM rule is that the data size is scaled to four times the size of the platform's L2 cache. Because this work is focused on architecture independent numbers, the each array size was scaled to 32MB, which is reasonably large for a workstation.

IV. METHODOLOGY AND METRICS

This work evaluates the temporal and spatial locality characteristics of applications separately. This section describes the methodology used in this work and formally defines the temporal locality, spatial locality, and data intensiveness measures.

A. Methodology

The applications in this were each traced using the Amber instruction trace generator [2] for the PowerPC. Trace files containing 4 billion sequential instructions were generated by identifying and capturing each instruction executed in critical sections of the program. The starting point for each trace was chosen using a combination of performance register profiling

of the memory system, code reading, and, in the case of SPEC, accumulated knowledge of good sampling points. The advice of application experts was also used for the Sandia codes. The traces typically represent multiple executions of the main loop (multiple time steps for the Sandia floating point benchmarks). These traces have been used extensively in other work, and are well understood [25].

B. Temporal Locality

The application's *temporal working set* describes its *temporal locality*. As in prior work [25], a temporal working set of size N is modeled as an N byte, fully associative, true least recently used cache with native machine word sized blocks. The hit rate of that cache is used to describe the effectiveness of the fixed-size temporal working set at capturing the application's data set. The same work found that the greatest differentiation between conventional and supercomputer applications occurred in the 32KB-64KB level one cache sized region of the temporal working set. The temporal locality in this work is given by a temporal working set of size 64 KB. The temporal working set is measured over a long-studied 4 billion instruction trace from the core of each application. The number of instructions is held constant for each application. This puts the much shorter running SPEC benchmark suite on comparable footing to the longer running supercomputing applications.

It should be noted that there is significant latitude in the choice of temporal working set size. The choice of a level one cache sized working set is given for two reasons: first, it has been demonstrated to offer the greatest differentiation of applications between the floating point benchmarks in this suite and SPEC FP; and second, while there is no direct map to a conventionally constructed L1 cache, the L1 hit rate strongly impacts performance. There are two other compelling choices for temporal working set size:

1) **Level 2 Cache Sized:** in the 1-8 MB region. Arguably, the hit rate of the cache closest to memory most impacts performance (given very long memory latencies).
2) **Infinite:** describes the temporal hit rate required to capture the application's total data set size.

Given N memory accesses, H of which hit the cache described above, the temporal locality is given by: $\frac{H}{N}$.

C. Spatial Locality

Measuring the spatial locality of an application may be the most challenging aspect of this work. Significant prior work has examined it as the application's stride of memory access. The critical measurement is how quickly the application consumes all the data presented to it in a cache block. Thus, given a cache block size, and a fixed interval of instructions, the spatial locality can be described as the ratio of data the application actually uses (through a load or store) to the cache line size. This work uses an instruction interval of $1,000$ instructions, and a cache block size of 64-bytes. For this work, every $1,000$ instruction window in the application's 4 billion instruction trace is examined for unique loads and stores. Those loads and stores are then clustered into 64-byte blocks, and the ratio of used to unused data in the block is computed. The block size is chosen as a typical conventional cache system's block size. There is much more latitude in the instruction window size. It must be large enough to allow for meaningful computation, while being small enough to report differentiation in the application's spatial locality. For example, a window size of the number of instructions in the application should report that virtually all cache lines are 100% used. The $1,000$ instruction window was chosen based on prior experience with the applications [24].

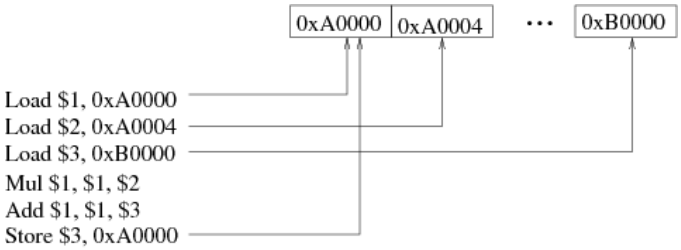

Fig. 1. An example of temporal locality, spatial locality, and data intensiveness.

Given U_{1000} unique bytes accessed in an average interval of 1,000 instructions that are clustered into L 64-byte cache lines, the spatial locality is given by $\frac{U}{64L}$.

D. Data Intensiveness

One critical yet often overlooked metric of an application's memory performance is its *data intensiveness*, or the total amount of unique data that the application accesses (regardless of ordering) over a fixed interval of instructions. Over the course of the entire application, this would be the application's memory footprint. This is not fully captured by the measurements given above, and it is nearly impossible to determine from a cache miss rate. This differs from the application's *memory footprint* because it only includes program data that is accessed via a load or store (where the memory footprint would also include program text). Because a cache represents a single instantiation used to capture an application's working set, a high miss rate could be more indicative of the application accessing a relatively small amount of memory in a temporal order that is poorly suited to the cache's parameters, or that the application exhibits very low spatial locality. It is not necessarily indicative of the application accessing a large data set, which is critical to supercomputing application performance. This work presents the data intensiveness as the total number of unique bytes that the application's trace accessed over its 4 billion instruction interval.

This is directly measured by counting the total number of unique bytes accessed over the given interval of 4 billion instructions. This is the same as the unique bytes measure given above, except it is measured over a larger interval (U_{4B}).

E. An Example

Figure 1 shows an example instruction sequence. Assuming that this is the entire sequence under analysis, each of the metrics given above is computed as follows:

Temporal Locality: is the hit rate of a fully associative cache. The first 3 loads in the sequence (of $0xA0000$, $0xA0004$, and $0xB0000$) miss the cache. The final store (to $0xA0000$) hits the item in the cache that was loaded 3 memory references prior. Thus, the temporal locality is:

$$\frac{1\ hit}{4\ memory\ references} = 0.25$$

Spatial Locality: is the ratio of used to unused bytes in a 64-byte cache line. Assuming that each load requests is 32-bits, there are two unique lines requested, $0xA0000$ (to $0xA0040$), and $0xB0000$ (to $0xB0040$). Two 32-bit words are consumed from $0xA0000$, and 1 32-bit word from $0xB0000$. The spatial locality is calculated as:

$$\frac{12\ consumed\ bytes}{128\ requested\ bytes} = 0.09375$$

Data Intensiveness: is the total number of unique bytes consumed by the stream. In this case, 3 unique 32-bit words are requested, for a total of 12 bytes.

V. Initial Observations of Program Characteristics

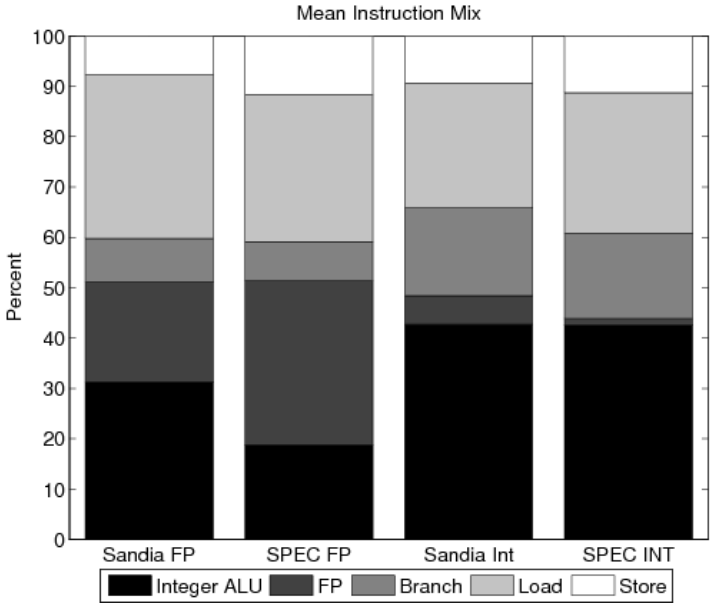

Fig. 2. Benchmark Suite Mean Instruction Mix

Figure 2 shows the instruction mix breakdown for the benchmark suites. Of particular importance is that the Sandia Floating Point applications perform significantly more **integer** operations than their SPEC Floating Point counterparts, in excess of 1.66 times the number of integer operations, in fact. This is largely due to the complexity of the Sandia applications (with many configuration operations requiring integer tests, table look ups requiring integer index calculations, etc.) as well as their typically more complicated memory addressing patterns [30]. This is largely due to the complexity of the algorithm, and the fact that significantly more indirection is used in memory address calculations. Additionally, in the case of the floating point applications, although the Sandia applications perform only about 1.5% more total memory references than their SPEC-FP counterparts, the Sandia codes perform 11% more loads, and only about $\frac{2}{3}$ the number of stores, indicating that the results produced require more memory inputs to produce fewer memory outputs. The configuration complexity can also be seen in that the Sandia codes perform about 11% more branches than their SPEC counterparts.

In terms of the integer applications, the Sandia codes perform about 12.8% fewer memory references over the same number of instructions, however those references are significantly harder to capture in a cache. The biggest difference is that the Sandia Integer codes perform 4.23 times the number of floating point operations as their SPEC Integer counterparts. This is explained by the fact that three of the Sandia Integer benchmarks perform somewhat significant floating point computations.

TABLE III
Sandia Integer Applications with Significant Floating Point Computation

Application	Percent Floating Point Instructions
Chaco	15.84%
DFS	14.74%
Isomorphism	13.41%

Table III summarizes the three Sandia Integer Suite applications with significant floating point work: Chaco, DFS, and Isomorphism. Their floating point ratios are quite below the median for SPEC FP (28.69%), but above the Sandia Floating Point median (10.67%). They are in the integer category because their primary computation is an integer graph manipulation, whereas CTH is in the floating point category even though runs have a lower floating point percentage (a mean over its three input runs of 6.83%), but the floating point work is the primary computation. For example, Chaco is a multilevel partitioner and uses spectral partitioning in its base case, which requires the computation of an eigenvector (a floating point operation). However, graph partitioning is fundamentally a combinatorial algorithm, and consequently in the integer category. In the case of CTH, which is a floating point application with a large number of integer operations, it is a shock physics code. The flops fundamentally represent the "real work", and the integer operations can be accounted for by the complexity of the algorithms, and the large number of table look-ups employed by CTH to find configuration parameters. In either case, the SPEC FP suite is significantly more floating point intensive.

VI. Results

The experimental results given by the metrics from Section IV are presented below. Each graph depicts the temporal locality on the X-axis, and the spatial locality on the Y-axis. The area of each circle on the graph depicts each application's relative data intensiveness (or the total amount of unique data consumed over the instruction stream).

Figure 3 provides the summary results for each suite of applications, and the RandomAccess memory benchmark.

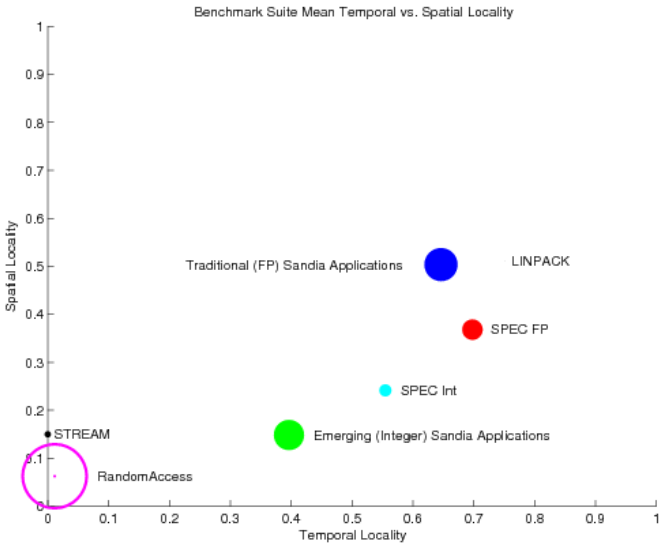

Fig. 3. Mean Temporal vs. Spatial Locality and Data Intensiveness for each benchmark suite.

The Sandia Floating Point suite exhibits approximately 36% greater spatial locality and nearly 7% less temporal locality than its SPEC-FP counterpart. The nearness in temporal locality, and increased spatial locality is somewhat surprising when taken out of context. One would typically expect scientific applications to be less well structured. The critical data intensiveness measure proves the most enlightening. The Sandia FP suite accesses over 2.6 **times** the amount of data as SPEC FP. The data intensiveness is the most important differentiator between the suites. A larger data set size would reflect significantly worse performance in any real cache implementation. Without the additional measure, the applications would appear more comparable. It should be noted that the increased spatial locality seen in the Sandia Floating Point applications is likely because those applications use the MPI programming model, which generally groups data to be operated upon into a buffer for transmission over the network (increasing the spatial locality).

The Sandia integer suite is significantly farther from the SPEC integer suite in all dimensions. It exhibits close to 30% less temporal locality, nearly 40% less spatial locality, and has a unique data set over 5.9 times the size of the SPEC integer suite.

The LINPACK benchmark shows the highest spatial and temporal locality of any benchmark, and by far the smallest data intensiveness (the dot is hardly visible on the graph). It is over 3,000 times smaller than any of the real world Sandia applications. It exhibits 17% less temporal locality and roughly the same spatial locality than the Sandia FP suite. The Sandia Integer suite has half the temporal locality and less than one third the spatial locality.

The STREAM benchmark showed over 100 times less temporal locality than RandomAccess, and 2.4 times the spatial locality. However, critically, the data intensiveness for streams is 1/95th that of RandomAccess. The Sandia Integer Suite is only 1% *less* spatially local than STREAM, indicating that most of the bandwidth used to fetch a cache line is wasted.

While it is expected that RandomAccess exhibits very low spatial and temporal locality, given its truly random memory access pattern, its data set is 3.7× the size of the Sandia FP suite, 4.5× the size of the Sandia Integer suite, and 9.7× and 26.5× the SPEC floating point and integer suites respectively.

Figure 4(a) shows each individual floating point application in the Sandia and SPEC suites. On the basis of spatial and temporal locality measurements alone, the the 177.mesa SPEC FP benchmark would appear to dominate all others in the suite. However, it has the second smallest unique data set size in the entire SPEC suite. In fact, the Sandia FP applications average **over** 9 times the data intensiveness of 177.mesa. There are numerous very small data set applications in SPEC FP, including 177.mesa, 178.galgel, 179.art, 187.facerec, 188.ammp, and 200.sixtrack. In fact, virtually all of the applications from SPEC FP that are "close" to a Sandia application in terms of spatial and temporal locality exhibit a much smaller unique data set. The mpsalsa application from the Sandia suite and 183.equake are good examples. While they are quite near on the graph, mpsalsa has almost 17 times the unique data set of equake. 183.equake is also very near the mean spatial and temporal locality point for the entire Sandia FP suite, except that the Sandia applications average more than 15 times 183.equake's data set size.

Unfortunately, it would be extremely difficult to identify a SPEC FP application that is "representative" of the Sandia codes (either individually, or on average). Often papers choose a subset of a given benchmark suite's applications when presenting the results. Choosing the five applications in SPEC FP with the largest data intensiveness (168.wupwise, 171.swim, 173.applu, 189.lucas, 301.apsi), and 183.equake (because of its closeness to the average and to mpsalsa) yields a suite that averages 90% of the Sandia suite's temporal locality, 86% of its temporal locality, 75% of it's data intensiveness. While somewhat far from "representative", particularly in terms of data intensiveness, this subset is more representative of the real applications than the whole.

Several interesting Sandia applications are shown on the graph. The CTH application exhibits the most temporal locality, but relatively low spatial locality, and a relatively small data set size. The LAMMPS (lmp) molecular dynamics code is known to be compute intensive, but it exhibits a relatively small memory footprint, and shows good memory performance. The temporal and spatial locality measures are quite low. SPPM exhibits very high spatial locality, very low temporal locality, and a moderate data set size.

Figure 4(b) depicts the Sandia and SPEC Integer benchmark suites. These applications are strikingly more different than the floating point suite. All of the applications exhibit relatively low spatial locality, although the majority of Sandia applications exhibit significantly less spatial locality than their SPEC counterparts. The DFS code in the Sandia suite is the most "RandomAccess-like", with 255.vortex in the SPEC suite being the closest counter part in terms of spatial and temporal locality. 255.vortex's temporal and spatial locality are within 25% and 15% of DFS' respectively. However, once again, DFS's data set size is over 19 times that of 255.vortex's.

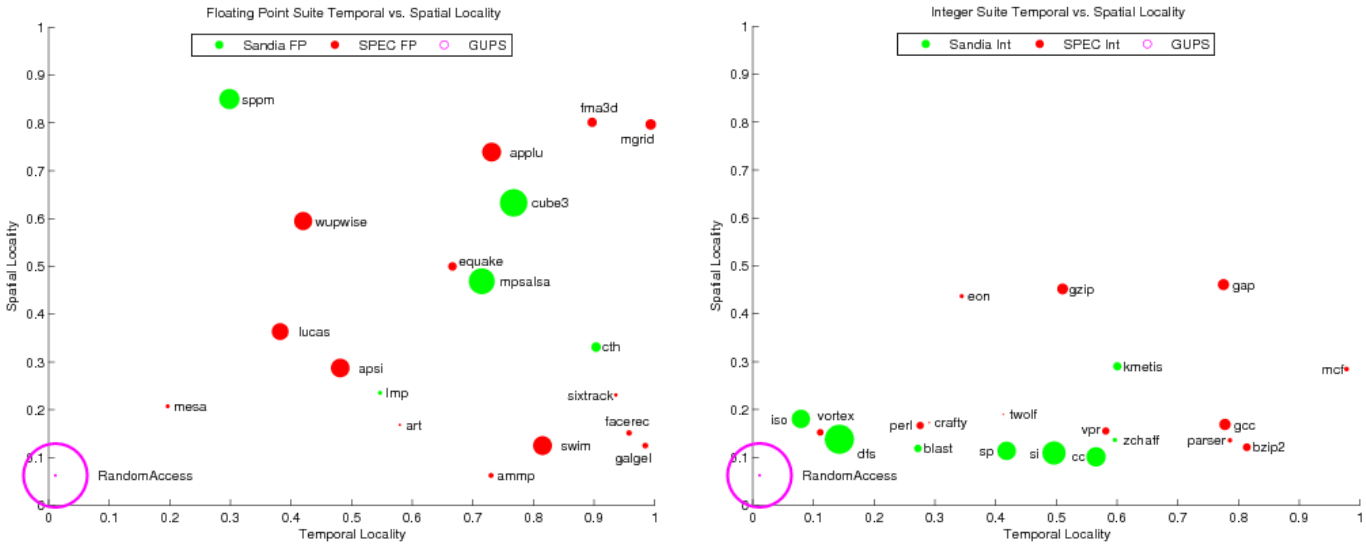

Fig. 4. (a) Integer and (b) Floating Point Applications Temporal vs. Spatial Locality and Data Intensiveness.

300.twolf actually comes closest in terms of spatial and temporal locality to representing the "average" Sandia Integer application, however, the average Sandia code has nearly 140 times the data set size.

VII. Conclusions

This work has measured the temporal and spatial locality, and the relative data intensiveness of a set of real world Sandia applications, and compared them to the SPEC Integer and Floating Point suites, as well as the RandomAccess memory benchmark. While the SPEC floating point suite exhibits greater temporal locality and less spatial locality than the Sandia floating point suite, it averages significantly less data intensiveness. This is crucial because the number of unique items consumed by the application can affect the performance of hierarchical memory systems more than the average efficiency with which those items are stored in the hierarchy.

The integer suites showed even greater divergence in all three dimensions (temporal locality, spatial locality, and data intensiveness). Many of the key integer benchmarks, which represent applications of emerging importance, are close to RandomAccess in their behavior.

This work has further quantitatively demonstrated the difference between a set of real applications (both current and emerging) relevant to the high performance computing community, and the most studied set of benchmarks in computer architecture. The real integer codes are uniformly harder on the memory system than the SPEC integer suite. In the case of floating point codes, the Sandia applications exhibit a significantly larger data intensiveness, and lower temporal locality. Because of the dominance of the memory system in achieving performance, this indicates that architects should focus on codes with significantly larger data set sizes.

The emerging applications characterized by the Sandia Integer suite are the most challenging applications (next to the RandomAccess benchmark). Because of their importance and their demands on the memory system, they represent a core group of applications that require significant attention.

Finally, beyond a specific study of one application domain, this work presents an architecture-independent methodology for quantifying the difference in memory properties between any two applications (or suites of applications). This study can be repeated for other problem domains of interest (the desktop, multimedia, business, etc.).

Acknowledgments

The authors would like to thank Arun Rodrigues and Keith Underwood at Sandia National Laboratories for their valuable comments.

The application analysis and data gathering discussed here used tools that were funded in part by DARPA through Cray, Inc. under the HPCS Phase 1 program.

References

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] Apple Architecture Performance Groups. *Computer Hardware Understanding Development Tools 2.0 Reference Guide for MacOS X*. Apple Computer Inc, July 2002.

[3] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 3–14. IEEE Computer Society, 1998.

[4] Daniel Citron, John Hennessy, David Patterson, and Gurindar Sohi. The use and abuse of SPEC: An ISCA panel. *IEEE Mico*, 23(4):73–77, July-August 2003.

[5] P. Colella and P.R. Woodward. The Piecewise Parabolic Method (PPM) for Gas-Dynamical Simulations. *Journal of Computational Physics*, 54:174–201, 1984.

[6] Z. Cvetanovic and D. Bhandarkar. Characterization of alpha AXP performance using TP and SPEC workloads. In *Proceedings of the 21ST annual international symposium on Computer architecture*, pages 60–70. IEEE Computer Society Press, 1994.

[7] Peter J. Denning. The working set model for program behavior. In *Proceedings of the first ACM symposium on Operating System Principles*, pages 15.1–15.12. ACM Press, 1967.

[8] J. Dongarra and P. Luszczek. Introduction to the HPCChallenge Benchmark Suite. Technical Report ICL-UT-05-01, 2005.

[9] Domenico Ferrari. A generative model of working set dynamics. In *Proceedings of the 1981 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 52–57. ACM Press, 1981.

[10] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith. Cache performance of the SPEC benchmark suite. Technical Report CS-TR-1991-1049, 1991.

[11] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *International Conference on Supercomputing*, pages 317–324, 1997.

[12] Swathi Tanjore Gurumani and Aleksandar Milenkovic. Execution characteristics of SPEC CPU2000 benchmarks: Intel C++ vs. Microsoft VC++. In *Proceedings of the 42nd annual Southeast regional conference*, pages 261–266. ACM Press, 2004.

[13] B. Hendrickson and R. Leland. The chaco user's guide — version 2.0. Technical Report SAND94-2692, 1994.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, 2002.

[15] Michael Heroux, Roscoe Bartlett, Vicki Howle, Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, 2003.

[16] E. Hertel, J. Bell, M. Elrick, A. Farnsworth, G. Kerley, J. McGlaun, S. Petney, S. Silling, P. Taylor, and L. Yarrington. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *Proceedings of the 19th International Symposium on Shock Waves*, pages 377–382, July 1993.

[17] George Karypis and Vipin Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[18] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proceedings of the International Sympoisum on Computer Architecture*, pages 15–26, 1998.

[19] Dennis C. Lee, Patrick Crowley, Jean-Loup Baer, Thomas E. Anderson, and Brian N. Bershad. Execution characteristics of desktop applications on windows NT. In *International Symposium on Computer Architecture*, pages 27–38, 1998.

[20] Jack L. Lo, Luiz Andre Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *International Symposium on Computer Architecture*, pages 39–50, 1998.

[21] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the Sixth International Conference on Architectural support for Programming Languages and Operating Systems*, pages 145–156. ACM Press, 1994.

[22] McCalpin, John D. *Stream: Sustainable memory bandwidth in high performance computers*, 1997.

[23] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference*, June 2001.

[24] Richard C. Murphy. *Traveling Threads: A New Multithreaded Execution Model*. Ph.D. Dissertation, University of Notre Dame, May 2006.

[25] Richard C. Murphy, Arun Rodrigues, Peter Kogge, and Keith Underwood. The Implications of Working Set Analysis on Supercomputing Memory Hierarchy Design. In *Proceedings of the 2005 International Conference on Supercomputing*, pages 332–340, June 20-22, 2005.

[26] Leonid Oliker, Andrew Canning, Jonathan Carter, John Shalf, and Stephane Ethier. Scientific Computations on Modern Parallel Vector Systems. In *Proceedings of Supercomputing*, page 10, 2004.

[27] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, 2ed.* Morgan Kaufmann Publishers, 1997.

[28] Steven J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics*, 117:1–19, 1995.

[29] Steven J. Plimpton, R. Pollock, and M. Stevens. Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.

[30] Arun Rodrigues, Richard Murphy, Peter Kogge, and Keith Underwood. Characterizing a New Class of Threads in Scientific Applications for High End Supercomputers. In *Proceedings of the 18th Annual ACM International Conference on Supercomputing*, pages 164–174.

[31] Juan Rodriguez-Rosell. Empirical working set behavior. *Communications of the ACM*, 16(9):556–560, 1973.

[32] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 14–26. ACM Press, 1993.

[33] Rafael Saavedra and Alan Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–84, 1996.

[34] J. Shadid, A. Salinger, R. Schmidt, T. Smith, S. Hutchinson, G. Hennigan, K. Devine, and H. Moffat. MPSalsa: A Finite Element Computer Program for Reacting Flow Problems. Technical Report SAND98-2864, 1998.

[35] D. R. Slutz and I. L. Traiger. A note on the calculation of average working set size. *Communications of the ACM*, 17(10):563–565, 1974.

[36] Jeffrey S. Vetter and Andy Yoo. An Empirical Performance Evaluation of Scalable Scientific Applications. In *Proceedings of Supercomputing*, pages 1–18, 2002.

[37] Jonathan Weinberg, Michael McCracken, Alan Snavely, and Erich Strohmaier. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Supercomputing 2005*, page 50, November, 2005.

Richard C. Murphy is a computer architect in the scalable systems group at Sandia National Laboratories. He received his Ph.D. in computer engineering at the University of Notre Dame. His research interests include computer architecture, with a focus on memory systems and Processing-In-Memory, VLSI, and massively parallel architectures, programming languages, and runtime systems. He spent 2000 to 2002 at Sun Microsystems focusing on hardware resource management and dynamic configuration. He is a member of the IEEE.

Peter M. Kogge was with IBM, Federal Systems Division, from 1968 until 1994, and was appointed an IEEE Fellow in 1990, and an IBM Fellow in 1993. In 1977 he was a Visiting Professor in the ECE Dept. at the University of Massachusetts, Amherst. From 1977 through 1994, he was also an Adjunct Professor in the Computer Science Dept. of the State University of New York at Binghamton. In August, 1994 he joined the University of Notre Dame as first holder of the endowed McCourtney Chair in Computer Science and Engineering (CSE). Starting in the summer of 1997, he has been a Distinguished Visiting Scientist at the Center for Integrated Space Microsystems at JPL. He is also the Research Thrust Leader for Architecture in Notre Dame's Center for Nano-Science and Technology. For the 2000-2001 academic year he was the Interim Schubmehl-Prein Chairman of the CSE Dept. at Notre Dame. Starting in August, 2001 he is the Associate Dean for Research, College of Engineering. Starting in the fall of 2003, he also is a Concurrent Professor of Electrical Engineering. He is a fellow of the IEEE.