Characteristics of Workloads Used in High Performance and Technical Computing [*]

Razvan Cheveresan[1], Matt Ramsay[2], Chris Feucht[1], and Ilya Sharapov[3]

[1]Sun Microsystems, Architecture Technology Group
4210 Network Circle, Santa Clara, CA 95054

[2]Sun Microsystems, Architecture Technology Group
5300 Riata Park Court, Austin, TX 78727

[3]Apple
1 Infinite Loop, Cupertino, CA 95014

razvan.cheveresan@sun.com, matthew.ramsay@sun.com,
christopher.feucht@sun.com, isharapov@apple.com

ABSTRACT

This paper provides a systematic comparison of various characteristics of computationally-intensive workloads. Our analysis focuses on standard HPC benchmarks and representative applications. For the selected workloads we provide a wide range of characterizations based on instruction tracing and hardware counter measurements.

Each workload is analyzed at the instruction level by comparing the dynamic distribution of executed instructions. We also analyze memory access patterns including various aspects of cache utilization and locality properties of address distributions. Since prefetching plays an important role in the performance of computational workloads, we explore the prefetching potential and for parallel workloads we study the sharing properties of memory accesses. For the purpose of completeness, HPC workloads are compared to two commonly used commercial computing benchmarks.

The results of this work show that the HPC application space is surprisingly diverse, with some codes showing similar data sharing and locality properties with commercial applications. The wide range of studies presented in this paper are instrumental in uncovering the diversity of this application space.

[*]This material is based upon work funded by DARPA under Contract No. NBCH3039002

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Measurement, Performance

Keywords

HPC, workload characterization, instruction decomposition, cache coherency, software prefetch, data locality

1. INTRODUCTION

Workload characterization is an integral part of performance analysis of computer systems. Accurate performance predictions and successful system design rely on characterizations such as memory access patterns or the types of the instructions used in a computation. These properties influence the design of components at all system levels from the microarchitecture and memory system to compilers and operating environments.

A good understanding of workload properties sheds light on resource utilizations in the system and can guide performance optimization both at the software and system configuration level. In addition, workload properties are used to support decisions for purchasing or allocating hardware for different applications.

It is important to analyze a broad portfolio of workloads that fully represent the usage of the modeled architecture at customers' sites. In this paper, our attention is focused on computationally-intensive workloads starting with the industry-standard HPC Challenge and NAS benchmarks. Since these are fairly large benchmark suites, we have selected the most important and computationally demanding programs from both sets.

In addition to the standard HPC benchmarks, several representative full-scale computational applications were also examined. For the purpose of completeness, the properties of these HPC workloads were compared to those of two standard commercial computing benchmarks: TPC-C and SAP SD. For all workloads we used unmodified versions of the programs and applied standard compilation techniques. That ensured that the workloads we analyzed were representative of the programs run by end users.

The main focus of this study is to provide a comprehensive account of a variety of performance characteristics for computationally intensive workloads. Although individual characterizations are not novel, it is our belief that the results of a broad study of performance properties for a wide class of HPC workloads and comparisons against standard commercial or enterprise benchmarks are valuable.

The remainder of the paper is divided such that the next two sections (Sections 2 and 3) give an overview of the characterization approaches suggested for use in high performance computing and brief descriptions of the workloads we used in our studies. Section 4 discusses the methodology and techniques we used for collecting and validating instruction traces. The remaining sections of the paper present the results from our studies. First, Section 5 discusses the characterization of each application at the instruction level. The remaining sections analyze memory access patterns, in particular the data locality properties of each workload (Section 6), the sensitivity to various cache configurations (Section 7), the data sharing properties of parallel versions of each workload (Section 8), and the analysis of prefetch efficiency for each code (Section 9). The results and conclusions of our studies are summarized in the last section of the paper.

2. BACKGROUND

Over the last decade, a variety of workload characterization studies were performed for computationally-intensive programs. These studies are typically based on information collected either by using on-chip hardware counters or by processing instruction traces of applications.

Similar to our trace-based instruction decomposition study, Rupnow et. al. [24] provide an instruction category breakdown of SPEC-FP2000 [1] and an internal benchmark suite from Sandia National Labs. Their work concluded that SPEC-FP undervalues the importance of integer instructions in real scientific codes. We extend their work to draw conclusions between scientific applications and important commercial OLTP benchmarks.

Characterizations often use standard methodologies for the analysis of the memory access properties of a workload [9]. Regularity of memory accesses can be based on the analysis of striding properties [8]. A particular aspect of memory usage that has received significant attention is the characterization of the locality of memory accesses in an application [7, 30]. This approach can be combined with APEX map characterizations (described in [28]), where benchmarks are developed to set data locality to a desired level. The resulting methodology allows the analysis of spatial and temporal locality of memory accesses of an application in an architecture-independent way [32].

Another area of active research is the memory and cache analysis [26]. These results are typically generated using cache simulators running both on instructions traces and using execution-driven techniques widespread in the industry. In addition, there are a number of analytical techniques, such as Cache Miss Equations [16], that are applicable for compile-time cache analysis.

A related approach for memory access characterization is the dynamic analysis of the working set size of a workload [10, 11]. This methodology is related to both cache utilization analysis and memory paging algorithms.

In many cases, the effect of cache misses can be largely mitigated by data prefetching, although the impact of this technique has limited impact on performance of workloads where data access patterns are irregular or unpredictable. The efficiency of prefetching is reported in [17]. Additional studies, in particular on the impact of instruction prefetching on the performance of commercial workloads, were presented in [27].

For parallel workloads, the amount and types of data sharing have significant implications on the coherency traffic and ultimately on the performance of SMP systems. Some results quantifying data sharing on SMPs are presented in [15, 20].

In this paper, we apply workload characterizations to a large set of computational HPC workloads. For complex workloads that do not exhibit steady-state behavior, the focus is on one or several computationally-intensive phases. The goal of this paper is to analyze similarities and differences of various workload characteristics and to compare the properties of benchmarks to the properties of real applications and draw conclusions about the comparisons.

3. WORKLOADS

For this study both standard high performance computing benchmarks and real numerically-intensive applications were used. This diversity allowed for the examination of workloads that stress a particular resource in the our internal product designs as well as those representing a balanced usage scenario. While only certain benchmarks were selected out of the HPC and NAS suites, we feel that the range of workloads from these simple kernels to those of the real world applications provides valuable data. The inclusion of simple kernels also helped validate our results, as we could often predict application behavior to compare with measured results.

This section provides a brief description of the workloads used in our studies as well as a discussion of the problem size tested with each. To make the studies practical for our tools infrastructure, we study in detail only a single dataset size for each code. However, we validated our results against larger data set sizes on reference hardware to ensure that our data is representative.

3.1 HPC Challenge Benchmarks

HPC Challenge[1] is a suite of benchmarks for comparing the performance of HPC architectures using simple kernels. It extends the High Performance Linpack (HPL) traditionally used in ranking the systems on the Top 500 List[2] by providing additional kernels with challenging memory access patterns.

- High Performance Linpack [13, 14] is a linear system of equations solver. The factoring portion of the workload is examined using a 4K problem size.

[1]http://icl.cs.utk.edu/hpcc
[2]www.top500.org

- RandomAccess [13] (also known as GUPS, or Giga Updates Per Second) measures the rate of integer random updates of memory. For this testing, a 1 GB problem size was implemented.

- STREAM [13] is a benchmark that is used to find the maximum sustainable memory bandwidth possible in a machine. 16 million elements were used for testing.

- FFT is a Fast Fourier Transform implemented in the FFTE[3] package [13]. We used a two dimensional, single-precision, complex Fast Fourier Transform that was performed on a 1K problem size. Both the computation and communication phases of this workload were examined together.

3.2 NAS Parallel Benchmarks

NAS Parallel Benchmarks[4] were developed in the 1990s to analyze performance and scalability of hardware platforms. These benchmarks are synthetic compact representations of applications used in computational fluid dynamics.

- NAS BT [2] represents a computation on a regular 3D grid and performs a solution of a block-tridiagonal system of equations. For our experiments we used a fairly small Class A problem size to make instruction tracing practical.

- NAS CG [2] is a conjugate gradient solver that functions on a large, sparse, symmetric, positive definite matrix. Similar to BT, this was tested with a Class A problem size.

3.3 HPC Applications

We balance the standard benchmarks, that typically have very homogenous properties, with more irregular workloads by including real HPC applications. The three selected applications represent particle physics, computational chemistry, and structural analysis domains.

- GTC [25] (Gyro-Kinetic Toroidal Code) is a scientific workload that simulates the particle interactions within a Tokamak fusion reactor. A problem containing 3 million particles was used in these studies.

- NAB [4, 22] (Nucleic Acid Builder) is open-source software that permits the building of models of biomolecules, specifically nucleic acids and proteins. This code is used to measure the forces that seek to restore the atoms to their equilibrium positions after the individual particles have been given random, initial velocities. A molecule with 1957 atoms was studied.

- LS Dyna[5] is an implicit, finite-element solver that is used to simulate and analyze highly nonlinear physical phenomenons obtained in real world problems. The portion analyzed covers the forward and backward solve portions of the elimination tree that is used to find the displacements and element properties of timesteps for a 1024 element impeller problem.

[3]www.ffte.org

[4]www.nas.nasa.gov/Software/NPB

[5]www.lstc.com

3.4 Commercial Applications

To contrast with the HPC workloads, two applications from the commercial space were also examined.

- TPC-C[6] is a benchmark that is designed to test online transaction processing. It is centered around users executing transactions related to warehouse operations against a database. In this case 1600 warehouses were simulated.

- SAP SD[7] (Sales & Distribution) is a two-tier ERP business test that represents full business workloads of order and invoice processing. It demonstrates the ability to run both the application and database software on a single system.

4. EXPERIMENTAL METHODOLOGY & INFRASTRUCTURE

To study these workloads in sufficient detail, instruction traces were gathered for each benchmark configuration. These traces provide a way to characterize the performance of an application by offering a means to explore the interaction between the workload and the underlying hardware. Instruction traces are collected by executing the reference workload inside a functional simulator of the target instruction set architecture (ISA). The simulator records a trace of the actual instructions executed by the system while running the workload together with other important events such as MMU, trap, I/O, and DMA activity.

Several steps are taken to ensure that valid instruction traces are gathered. First, the application of interest is set up on reference hardware and several benchmark runs are performed to insure correct execution. During this process, representative or interesting regions of the workload are identified and bracketed for tracing. In addition, various system and benchmark performance metrics are collected to later be used for trace validation. Once correct setup and execution of the benchmark are ensured, a disk image of the entire reference file systems is taken and ported to the functional simulation environment. For trace collection, the simulated machine is configured to mimic the exact hardware and software configurations of the reference hardware. This ensures that the benchmark will behave the same way in both environments. Once the simulation of the benchmark reaches the portion of interest, the internal state of the simulated machine is captured in a checkpoint. The simulation is resumed from the checkpoint, and the writing of the instruction trace begins.

An important step that follows trace collection is trace validation. This process insures that the behavior captured in the trace is identical to the reference hardware executing the same portion of the workload. Therefore, in addition to writing the trace, various system measurements (OS statistics, system calls, device interrupts, traps etc.) are collected together with benchmark output data both on the reference and the simulated hardware and validated against each other.

Library calls, specifically Solaris *libcpc*[8] calls, to reset and read hardware performance counters are placed in the benchmark to bracket the traced portion. By doing this, hardware

[6]www.tpc.org

[7]www.sap.com/solutions/benchmark

[8]http://developers.sun.com/solaris/articles/hardware_counters.html

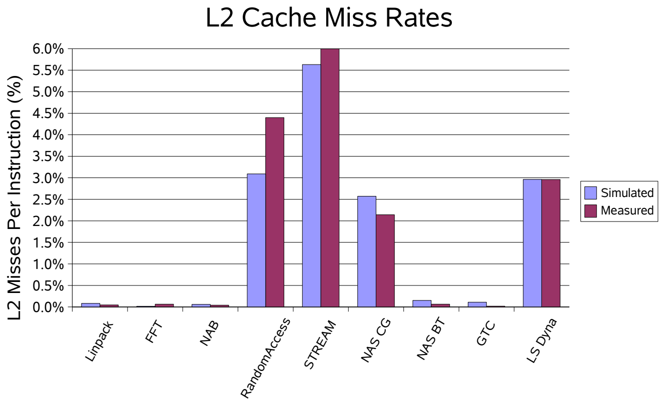

Figure 1: L2 Cache Miss Rate Validation: This figure compares the L2 cache miss rates per instruction measured through hardware counters on Sun Fire 6800 UltraSparc-III+ hardware versus those taken from cache simulations run on the gathered instruction traces.

counters can gather statistics, such as number of instructions or cache miss rates, for the exact traced portion of the application for accurate comparison. This information is later validated against data collected from memory hierarchy and performance simulators running the traces.

Figure 1 shows a sample validation between the reference hardware and the simulated machine from which the trace was collected. As an example, we selected the L2 cache miss rates as a basis for the comparison. The data shows that with the exception of RandomAccess, all workloads show a very good match between measured and simulated miss rates. This discrepancy is currently being investigated in detail. It should be noted that while there are differences in the relative size of some of the bars, the magnitude of the difference between the real and simulated hardware is on the order of 1 miss per 1000 instructions in most cases.

The reference platform used to evaluate these applications was a Sun Fire 6800 UltraSparc-III+[9], 900MHz running Solaris 10[10]. Single-processor instances of each workload were used for all experiments, except for memory sharing analysis, in order to isolate workload behavior from parallel overhead. For memory sharing analysis, 16-processor shared memory version of each benchmark were used. Two exceptions were: 1) TPC-C, where an 8-processor version was used because of the prohibitive overhead of tracing a larger processor version; and 2) LS Dyna, which also used an 8-processor version because of application scaling limitations with further parallelization.

Also relevant to our methodology are the following remarks: 1) results for only user code are reported for each experiment in order to isolate pure benchmark behavior from operating system kernel implementation; 2) most of the studies were performed with the prefetching enabled (it was only in the prefetch study that this feature was disabled for comparison purposes); 3) standard compiler optimizations were used for this study (-fast from SunStudio compiler suite).

[9]www.sun.com/processors/ultrasparc-iii
[10]www.sun.com/software/solaris

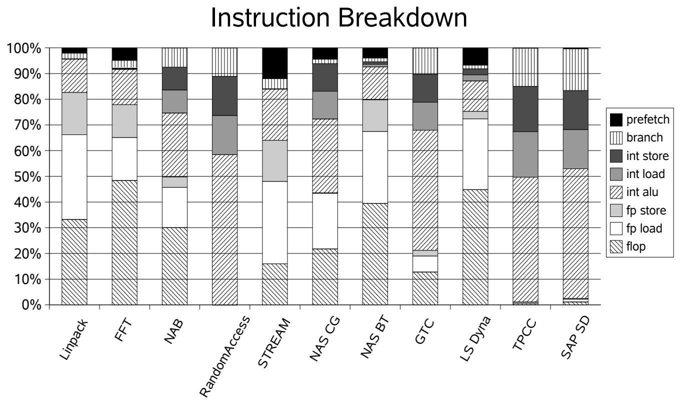

Figure 2: Dynamic Instruction Decomposition: This figure shows the percentage of dynamic instructions of several different categories.

5. INSTRUCTION DECOMPOSITION

Understanding what types of instructions make up the dynamic instruction stream of a program is essential to understanding its performance. Figure 2 shows the dynamic instruction breakdown for each of the studied workloads. The breakdown includes floating-point instructions (floating point additions and multiplications, loads, stores), integer instructions (simple ALU instructions, loads, stores), branches and software prefetch instructions.

While the two enterprise workloads (TPCC and SAP SD) exhibit nearly-identical instruction mix, the HPC codes show a remarkable variability in their instruction decomposition. This diversity makes it difficult to characterize computationally-intensive applications as a single group of workloads. It also shows that it is not possible to capture the dynamic code behavior of HPC workloads with one or few benchmarks.

Another observation is that the relative amount of floating point operations in the codes we examined is fairly low. In fact, for several of the computationally-intensive applications, the share of floating point instructions is 20% or less, with none being over 50%. This contradicts a common belief that the performance of HPC codes can be effectively measured in floating-point operations per second (FLOPS).

Similarly, the integer component of HPC applications and its contribution to application performance is often ignored. Figure 2 shows that each of the HPC applications has a sizeable integer component, with several (RandomAccess, STREAM, NAS CG, GTC) consisting of more simple integer ALU than FLOP instructions. In many cases, even if the computation operates on predominantly floating point data, arithmetical operations are matched by index or array offset computations, which explains a sizable fraction of integer instructions in linear algebra computations. Given these observations, we conclude that HPC applications should be analyzed with a more general methodology without overemphasizing the FLOP component's importance to overall application performance.

The data shown in Figure 2 also allows us to analyze the relative balance between the loads and the stores. If the focus is just on memory operations, we observe that integer loads and stores are well balanced for both HPC and enterprise workloads. At the same time, comparing the fre-

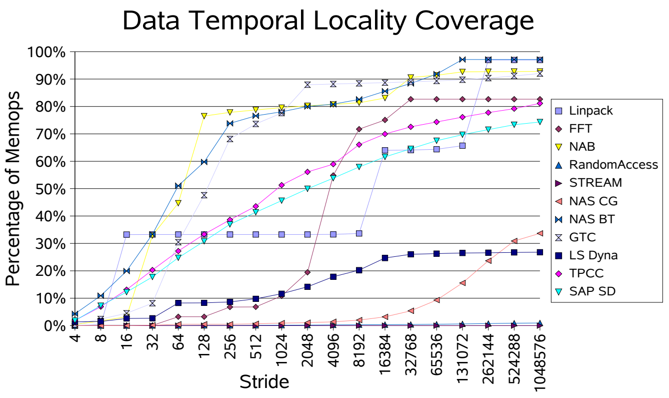

Figure 3: Data Temporal Locality: This figure shows the cumulative percentage of memory references that have a temporal stride of less than the threshold given on the x-axis. The temporal stride is defined by the number of memory operations between references to the same address.

quencies of floating point memory operations, the loads can be seen to dominate in all computational workloads. That can be attributed to the fact that many floating point instructions have two operands loaded from memory and a single stored result. Modern computer systems use this property and provision higher incoming than outgoing processor bandwidth.

6. LOCALITY ANALYSIS

The performance of computational applications is highly dependent on memory access properties of these workloads. In this and the following sections we study temporal and spatial locality, caching properties, data sharing in parallel applications and performance sensitivity to data prefetching. The results reported in these sections are generated by post-processing workload traces.

6.1 Temporal Locality

Temporal locality describes an application's tendency to reference memory addresses that it had referenced recently [18]. Typically this is characterized by measuring the reuse distance in terms of the number of intervening memory references between consecutive accesses to a given address.

Similar to the work by Weinberg, et. al. [32], we present reuse distances obtained from analysis of the selected workloads. An example of this study can be seen in Figure 3, which demonstrates the temporal locality cumulative fractions versus the reuse distance for the workloads described in Section 3. This type of data can be very useful in analyzing overall application profiles [12]. At each reuse distance, these plots show the cumulative percentage of the user code data access operations that have a distance less than or equal to the given stride on the graph. Since these distances are counted by the number of intervening memory accesses, analysis can be done without regard to a particular memory architecture. However, this study could yield insight into the optimal sizing for the various portions of the cache memory hierarchy.

While this analysis is independent of a particular memory hierarchy, there were some considerations made to streamline the overall process. On the low end, strides of less than 4 are not be tabulated, since it is unlikely that the given data associated with that address would be evicted from any reasonably sized cache in that span of time. Conversely, values with a stride distance above 1M were ignored since stride distributions are to be used in providing a weighted locality score based on distance. Strides at that end of the spectrum would have little impact on the score. By using these bounds, a range of 'reasonably-sized' caches can be studied with minimal simulation overhead.

With that said, the temporal locality score is given by Equation 1.

$$\frac{\sum_{i=0}^{log(N)-1}((reuse_{2^{i+1}} - reuse_{2^i}) * (log_2(N) - i))}{log_2(N)} \quad (1)$$

This formula is similar to that derived in [32] with a correction to the terms in the numerator to ensure that the overall score is between zero and one. With these changes, the terms have the appropriate weighting and the smaller strides have a larger impact. As discussed in [32], values close to zero indicate little temporal locality, and a value of one would indicate that data accesses occur in the smallest stride. Discussion of temporal scores for these workloads will be tied with the spatial scores as a part of Section 6.3.

The first takeaway from these results is that the STREAM and RandomAccess show the expected behavior. RandomAccess uses random reads/writes to memory and STREAM is used to test sustainable memory bandwidth. Neither one of these has a data footprint where it repeatedly accesses the same memory addresses, and thus a value of zero across the range of reuse distances applies. RandomAccess actually does a load from a memory address followed immediately by a store to the same address, but this happens within the minimum distance window and is not recorded.

NAS CG and LS Dyna both exhibit a low level of temporal reuse due to the sparse nature of the address spaces for these benchmarks. LS Dyna was traced for the backward/forward solve elimination tree of the implicit solver, so it is reasonable that there would be little temporal locality as it goes from branch to branch of the tree in the first part of the solver. As it progresses further up the tree, there would be more locality, but the bulk of the work is during the initial levels of the calculations. NAS CG is a kernel that tests a conjugate solver on a sparse matrix, so references to a particular element in the matrix would be infrequent for this particular type of calculation.

The other applications exhibited reuse patterns where over 70% of the memory operations were captured by the upper bound of the x-axis. The HPC workloads have points at which there are rather sharp increases in the accumulation of memory operations. Conversely, the commercial applications of SAP SD and TPC-C both have a more gradual rise. The nature of the former set of applications is to solve one problem, and the observed behavior indicates that the programs loop back regularly to perform calculations one or more arrays of elements. Linpack is particularly interesting with a stair-step pattern indicating three tiers of nested loops used for cache blocking of data.

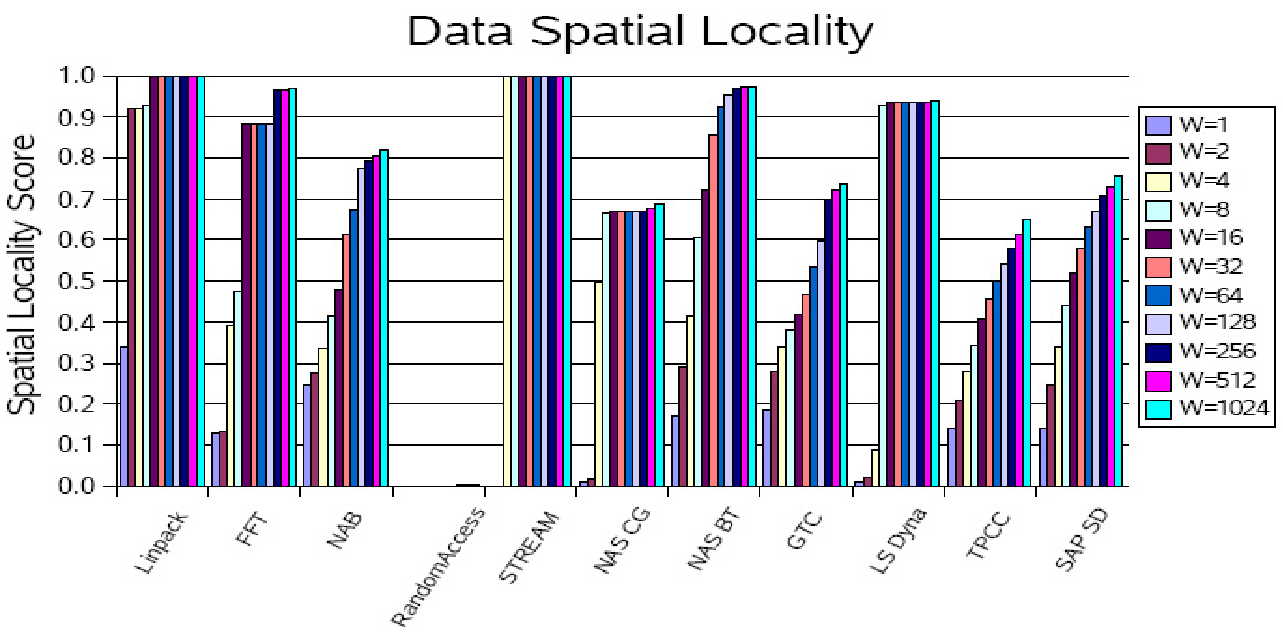

Figure 4: Data Spatial Locality: This figure shows the spatial locality score derived from Equation 2 for a variety of lookback window sizes.

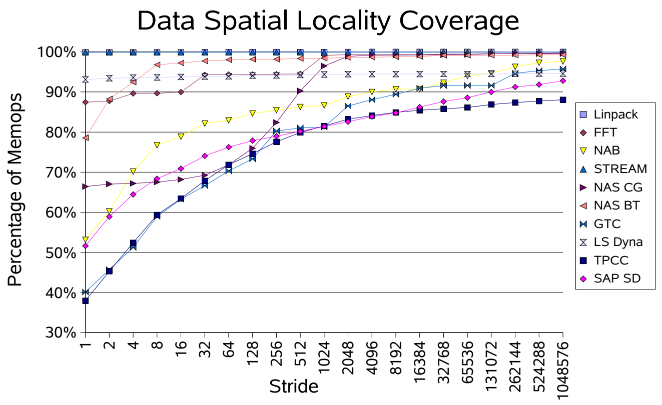

Figure 5: Data Spatial Locality: This figure shows the cumulative percentage of memory references that have a stride of less than the threshold given on the x-axis for a lookback window of *W=32* memory operations. To clear up the graphic, the y-axis starts at 30% and the curve for RandomAccess is not shown, as it is virtually 0% for all points.

6.2 Spatial Locality

Spatial locality is defined as a measure of a program's tendency to reference memory addresses near one another close together in time [18]. The spatial locality of a program can be examined by computing the difference in the addresses of neighboring memory accesses.

Similar to their work on temporal locality, Weinberg, et. al. [32] defined a singular metric for the spatial locality of a program. This metric is given by Equation 2, where $stride_i$ is the fraction of memory references whose minimum stride in words between itself and all memory references in the lookback window of length W was equal to i. By this metric, a program with all memory references of stride one would have a spatial locality score of one, while a program with all memory references of stride two would have a spatial locality score of 0.5.

$$\sum_{i=1}^{\infty} \frac{stride_i}{i} \tag{2}$$

Figure 4 shows the spatial locality score for each benchmark for window sizes (W) ranging from 1 to 1024 previous memory references. As expected, STREAM shows a locality score of virtually 1.0 and RandomAccess shows no spatial locality. Also notable is the fact that several HPC applications (NAS BT, GTC, NAB), as opposed to HPC Challenge kernels [13], show similar spatial locality patterns to the commercial applications. In addition, benchmarks with a significant linear algebra portion (LS Dyna, NAS CG and Linpack) all behave similarly, with each code exposing all of its spatial locality within a window of *W=32* memory operations corresponding to the cache blocking in the linear algebra libraries used in these applications.

In [32], Weinberg, et. al. present results for this spatial locality metric only for *W=32*, assuming that a window of the last 32 memory references would capture all reasonable sized loops and would therefore be a representative score for all workloads. Our data demonstrates that several codes do not show plateaus in their spatial locality score for reasonable window sizes. Therefore, presenting a spatial locality score for many window sizes (Figure 4) is a more complete metric for locality and also more informative, as additional information about the application can be gained from the examination of all scores.

Figure 5 shows the cumulative fraction of memory refer-

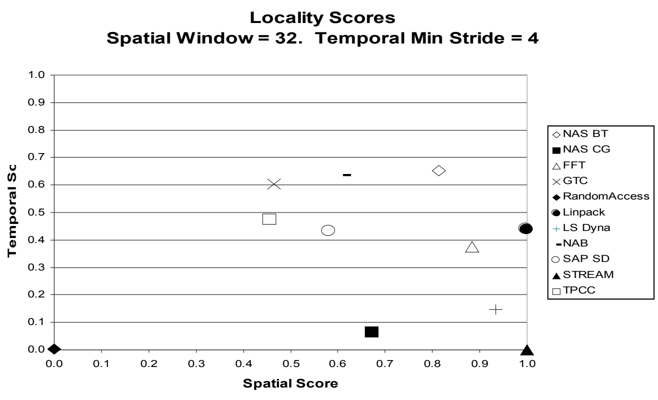

Figure 6: Data Spatial vs. Temporal Locality: This figure encapsulates both spatial and temporal locality scores for each benchmark.

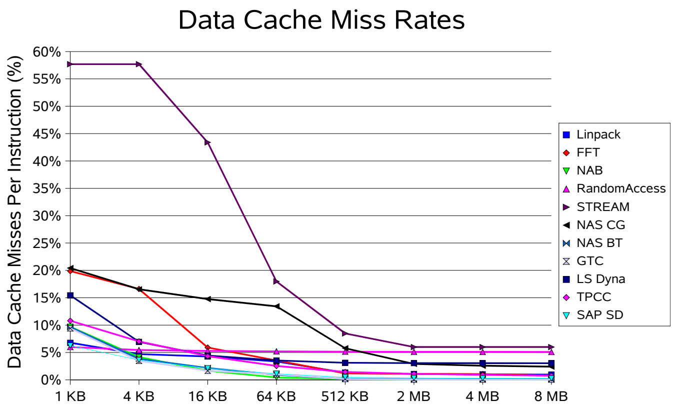

Figure 7: Data Cache Size Sensitivity: This figure shows the cache miss rate per instruction for direct mapped caches of increasing sizes (64 byte line sizes).

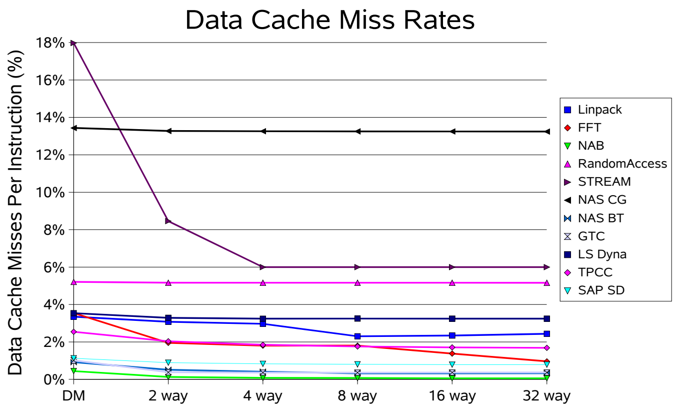

Figure 8: Data Cache Associativity Sensitivity: This figure shows the cache miss rate per instruction for 64 KB caches (64 byte line sizes).

ences that have a stride of less than a certain number within a window of *W=32*. As expected, a large portion of memory references for all workloads, with the exception of RandomAccess, have a sequential reference within the window. The step function corresponding to FFT can be attributed to a multi-scale nature of this benchmark. Another interesting case is NAS CG, which shows an S-shaped locality curve with many references falling in a fairly narrow band of strides. This happens because this benchmark traverses a randomly populated sparse matrix and the offsets between adjacent accesses are predominantly of the same order of magnitude.

6.3 Overall Benchmark Data Locality

Figure 6 shows memory locality for each benchmark over both the temporal and spatial dimensions. In this figure, spatial scores are shown for a lookback window of *W=32* to correlate with [32].

Comparisons were made with the data presented in Figure 1 in [32]. The scores for RandomAccess, STREAM, and FFT closely correlate and match our expectations. For example, because of the random nature of the RandomAccess benchmark, both its spatial and temporal locality scores are zero. STREAM shows perfect spatial score, but its lack of data reuse leads to its temporal score of zero.

Linpack and NAS CG, on the other hand, demonstrate some differences between the two studies. The version of Linpack used in characterization performed here indicates a higher level of spatial locality while trading off temporal when compared to [32]. Our version of NAS CG shows lower temporal locality, while the spatial locality scores are similar. Possible reasons for these deltas could be related to issues such as problem size and density variations, compilation, and various optimizations.

We highlight two conclusions drawn from Figure 6. First, that many HPC microbenchmarks and kernels are contrived to bound the data locality space, both in the temporal and spatial dimensions. Second, that many real computationally-intensive codes behave much more like standard commercial applications with respect to data locality than like these contrived HPC benchmarks.

7. CACHE ANALYSIS

This section extends the locality studies that are independent of any details of the memory hierarchy and examines the properties of workloads with respect to specific cache configurations. The results presented in this section were generated by trace-driven cache simulations.

7.1 Cache Size Sensitivity

Figure 7 shows each application's sensitivity to different cache sizes, showing the miss rates for a single-level direct-mapped cache of several sizes (all with 64 byte lines). As expected, the miss rate of RandomAccess is not helped at any cache size. The data also shows that all benchmarks, for the data sets that we chose, do not benefit from data caches larger than 2 MB. In addition, all benchmarks have a miss rate of less than 6% for cache sizes of 16 KB and larger, with two exceptions: STREAM and NAS CG.

7.2 Cache Associativity Sensitivity

Figure 8 shows each application's sensitivity to different cache associativities, showing the miss rates for a single-level

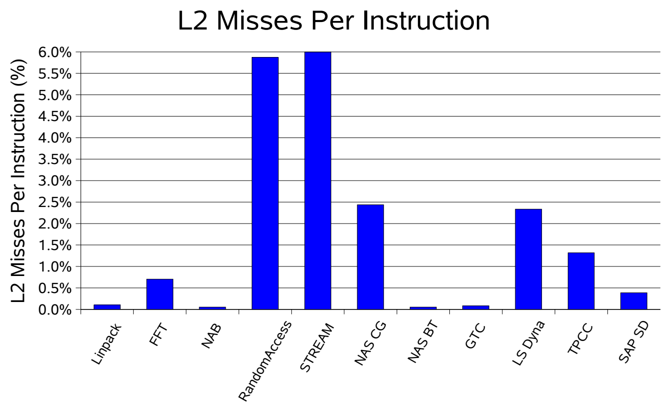

Figure 9: Cache Miss Breakdown: Magnitude: This figure shows the L2 cache miss rate per instruction for each workload.

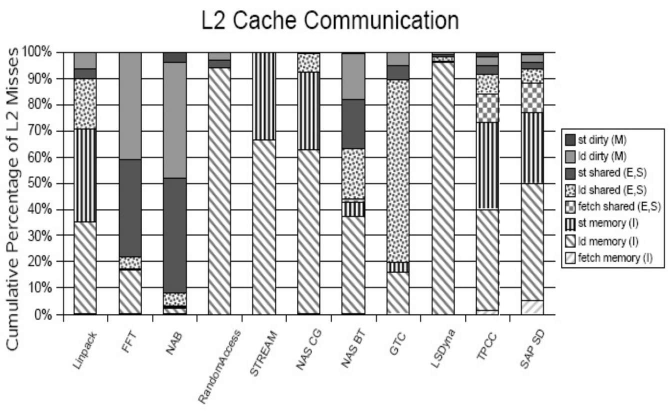

Figure 10: Cache Miss Breakdown: Percentages: This figure presents the same data as Figure 9, but each category is presented as a fraction of L2 cache misses for each benchmark.

64 KB cache of several different associativities with pseudo-LRU replacement policy. Our data shows that STREAM suffers from low cache associativities as the streaming data would fill every set of a direct-mapped cache replacing any array and loop indexing variables that could be reused through higher associativities. In general, since the working sets of these applications are larger than the 64 KB cache that we tested, most applications are not sensitive to cache associativity. In addition to this data point, we performed similar studies for different cache sizes, ranging from 16 KB to 128 KB, which yielded almost identical results.

8. DATA SHARING ANALYSIS

In highly parallel applications, caching behavior, along with the memory coherence traffic that it causes, is as important to performance as the capacity, compulsory and conflict misses that have already been examined. To analyze the caching effects of parallelizing these applications, we ran a shared-memory parallelized (OpenMP) version of each benchmark through our multi-processor cache simulator whose configuration is described in Table 1.

Table 1: M.P. Cache Simulation Configuration

Parameter	Configuration
L1 ICache	32 KB, 4 way, 64 B lines, PLRU
L1 DCache	32 KB, 4 way, 64 B lines, PLRU, WT, nWA
L2 Cache (I & D)	2 MB, 8 way, 64 B lines, PLRU, WB, WA, Includes L1 DCache
Coherency	MESI

Figure 9 shows the L2 cache miss rate per instruction for each benchmark. Figure 10 shows the same data as Figure 9 broken down into several categories. The categories are as follows (from bottom to top): 1) misses whose target line is not present in any other processor's cache and must be filled from memory (fetch, load, store); 2) misses whose target line is present in one or more caches in an unmodified state (fetch, load, store). These lines may be acquired via cache-to-cache transfer or filled from memory depending on implementation; 3) misses whose target line is in another processor's cache in modified state and must be acquired via cache-to-cache transfer (load, store).

In this data, most benchmarks behave as expected, with RandomAccess and STREAM having the highest miss rates, with the majority of misses coming from memory. Parallel shared-memory versions of FFT, NAB, GTC exhibit the same data sharing patterns, with a majority of accessed memory being shared between threads. FFT and NAB communicate significantly between threads, both loading from and storing to shared data. GTC only loads from shared data, sharing constants between threads. LS Dyna, RandomAccess, STREAM, and NAS CG show similar behavior with most misses being to non-shared data.

9. DATA PREFETCHING

Given that the performance of HPC applications is often governed by cache miss latency and the ability to stream data into the processor, it is not surprising that properly placed prefetch instructions can have a tremendous effect on application performance. Table 2 shows the performance gains for each HPC application when software prefetching into the prefetch cache of the UltraSparcIII+ micro-architecture is enabled.

Not surprising is the fact that highly predictable streaming applications benefit greatly from data prefetch, with STREAM showing almost a 2X performance boost, while RandomAccess shows virtually no speedup. Interesting data points and conclusions from this data include: 1) the lack of speedup for GTC, where memory access patterns are irregular and sufficiently complex that there is no effective software prefetching present in the code; 2) the lack of speedup for FFT, resulting from the problem size fitting in the processor's cache thus making prefetching irrelevant; 3) LS Dyna, Linpack, NAS CG, NAB represent linear algebra applications and the results show that there is a very high performance impact of prefetching due to the regular and predictable nature of memory accesses for these workloads.

Table 2: Prefetch Effect on Benchmark Performance

Benchmark	CPI Without Prefetch	CPI With Software Prefetch	% Improvement
Linpack	1.07	0.47	126.25%
FFT	1.15	1.10	4.94%
NAB	4.99	1.62	207.39%
RandomAccess	11.01	11.01	0.01%
STREAM	9.58	3.38	183.37%
NAS CG	6.52	2.45	166.12%
NAS BT	0.92	0.84	9.41%
GTC	1.15	1.15	-0.08%
LS Dyna	8.09	2.70	199.44%

10. CONCLUSIONS

In this work, we present detailed characterizations for a group of HPC applications that represent the space of computationally-intensive scientific workloads. To make our analysis more comprehensive, we include two common enterprise benchmarks in our studies. Several aspects of each workload are examined including dynamic instruction decomposition, memory locality, cache sensitivity, data communication decomposition, and the performance impact of data prefetch.

While the techniques described within this paper are accepted within the industry, the main contribution is that we apply a wide range of characterization techniques to both HPC and commercial workloads. Generating this type of data is labor and infrastructure intensive and, especially in regards to TPC-C and SAP, can be difficult to scale to represent a realistic size. As a result, it was primarily the intent to share the results from these studies such that others could use the information to do their own design analysis.

Reinforcing conclusions in [24], we observe that full-scale HPC applications, similar to commercial codes, show large fractions of integer code that are not appropriately represented in the studied kernels. This suggests the use of floating point efficiency as a singular metric may not be the most representative measure of benchmark performance of this application space. Also, we found the data locality properties of smaller HPC benchmarks and kernels to match our expectations; however, we unexpectedly discovered that the more complete HPC applications share similar properties with commercial applications. Both NAB and GTC fall into this category.

Our studies of the caching properties of HPC applications, including the detailed breakdown of data communication and each workload's sensitivity to data prefetch, reveal perhaps the most valuable finding of this work: the performance characteristics of HPC workloads are very diverse. No single application, or a small group of applications, fully represent this space. Therefore, it is imperative to analyze a large portfolio of these workloads when collecting performance data to be used for system analysis and design. We intend to expand this characterization work to encompass a larger, more diverse group of applications, though some components of our analysis, such as trace collection and certain simulations, will limit the rate at which new characterizations can be added.

11. ACKNOWLEDGMENTS

The authors would like to acknowledge insight, comments and assistance provided by following collegues: Lodewijk Bonebakker, Russ Brown, John Busch, Guy Delamarter, Theresa Do, John Fredricksen, Ilya Gluhovsky, Sriram Gummuluru, George Herman, Robert Kroeger, James Laudon, Man Leung, Xiaojun Liu, Eugene Loh, Tariq Magdon-Ismail, David Miller, Roy Moore, Brian O'Krafka, Pramod Rustagi, Sreenivas Vadlapatla and Eddine Walehiane.

12. REFERENCES

[1] http://www.spec.org.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. *International Journal of Supercomputing Applications*, 27(2):63–73, 1991.

[3] K. Beyls and E. Hollander. Reuse distance as a metric for cache behavior. In *International Conference on Parallel and Distributed Computing Systems*, pages 617–662, 2001.

[4] R. Brown and I. Sharapov. Parallelization of a molecular modeling application: Programmability comparison between OpenMP and MPI. In *Workshop on Productivity and Performance in High-End Computing*, February 2006.

[5] R. Bunt and J. Murphy. Measurement of locality and the behaviour of programs. *The Computer Journal*, 27(3):238–245, 1984.

[6] R. Bunt, J. Murphy, and S. Majumdar. A measure of program locality and its application. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 28–40, August 1984.

[7] R. Bunt and C. Williamson. Temporal and spatial locality: A time and place for everything. In *International Symposium in Honour of Professor Guenter Haring's 60th Birthday*, 2003.

[8] L. Carrington, A. Snavely, X. Gao, and N. Wolter. Performance prediction framework for scientic applications. In *Lecture Notes in Computer Science, 2659*, pages 926–935. Springer, January 2003.

[9] F. Darema-Rogers, G. Pfister, and K. So. Memory access patterns of parallel scientific programs. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 46–58. ACM Press, 1987.

[10] P. J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, 1968.

[11] P. J. Denning and S. C. Schwartz. Properties of the working-set model. *Commun. ACM*, 15(3):191–198, 1972.

[12] C. Ding and Y. Zhong. Predicting wholeprogram

locality through reuse distance analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 2003.

[13] J. Dongarra and P. Luszczek. Introduction to the HPCChallenge benchmark suite. *http://icl.cs.utk.edu/hpcc/pubs/*.

[14] J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: Past, present and fugure. *Concurrency: Practice and Experience*, 15:803–820, 2003.

[15] S. J. Eggers. Simulation analysis of data sharing in shared memory multiprocessors. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1989.

[16] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[17] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *ICS '90: Proceedings of the 4th international conference on Supercomputing*, pages 354–368, New York, NY, USA, 1990. ACM Press.

[18] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[19] T. Johnson, M. Merten, and W. Hwu. Runtime spatial locality detection and optimization. In *30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 57–64, 1997.

[20] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad pentium pro SMP using OLTP workloads. In *ISCA*, pages 15–26, 1998.

[21] S. Kumar and S. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368, 1998.

[22] T. Macke. Nab, a language for molecular manipulation. PhD Thesis, The Scripps Research Institute, 1996.

[23] J. Peachey, R. Bunt, and C. Colbourn. Towards an intrinsic measure of program locality. In *16th Annual Hawaii International Conference on System Sciences*, pages 128–137, 1983.

[24] K. Rupnow, A. Rodrigues, K. Underwood, and K. Compton. Scientific applications vs. spec-fp: A comparison of program behavior. In *ICS'06: Proceedings of the 20th ACM International Conference on Supercomputing*, Cairns, Australia, 2006.

[25] I. Sharapov, R. Kroeger, G. Delamarter, R. Cheveresan, and M. Ramsay. A case study in top-down performance estimation for a large-scale parallel application. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 2006.

[26] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[27] L. Spracklen, Y. Chou, and S. G. Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 225–236, Washington, DC, USA, 2005. IEEE Computer Society.

[28] E. Strohmaier and H. Shan. Architecture independent performance characterization and benchmarking for scientific applications. In *International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, 2004.

[29] J. Torrellas, M. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[30] P. Trancoso, J.-L. Larriba-Pey, Z. Zhang, and J. Torrellas. The memory performance of DSS commercial workloads in shared-memory multiprocessors. In *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture (HPCA-3)*, 1997.

[31] R. Uhlig and T. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, 1997.

[32] J. Weinberg, M. McCracken, A. Snavely, and E. Strohmair. Quantifying locality in the memory access patterns of HPC applications. In *Supercomputing*, 2005.