DRAFT VERSION: PLEASE DO NOT DISTRIBUTE

Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie,
Jaswinder Pal Singh, and Anoop Gupta
Computer Systems Lab
Stanford University

DRAFT VERSION

Abstract

We have recently released the SPLASH-2 suite of parallel applications for the study of centralized and/or distributed shared-address-space multiprocessors. In this regard, this paper has two main goals. One is to quantitatively characterize the SPLASH-2 programs in terms of fundamental properties that are important to understanding these parallel programs, and to describe how these properties vary with problem and machine parameters. The properties we study include the concurrency and load balance behavior, the communication-to-computation ratio, the sizes and scaling of the important working sets, and issues related to spatial locality. The other, perhaps more important, goal is methodological: to assist people who will use the programs for architectural evaluations to prune the design space of machine parameters in an informed and meaningful way. For example, by characterizing the working sets of the applications, we describe which operating regions in terms of cache size and problem size are representative of realistic situations and which are not (as well as which are redundant). Using SPLASH-2 as an example, we also hope to convey the importance of understanding the interplay of problem size, number of processors, and working sets in designing experiments and interpreting their results.

1 Introduction

Many architectural studies use parallel programs as workloads for the quantitative evaluation of ideas and tradeoffs. In shared-address-space multiprocessing, early research typically used small workloads consisting of a few simple programs. Often, different programs and different problem sizes were used, making comparisons across studies difficult. Many recent studies have used the Stanford ParalleL Applications for SHared memory (SPLASH) [SWG92], a suite of parallel applications written for cache-coherent shared-address-space machines. While SPLASH has provided a degree of consistency and comparability across research studies, like any other suite of applications it has many limitations. In particular, SPLASH consists of only a small number of programs (5 full applications and 1 computational kernel) and does not provide broad enough coverage even of scientific and engineering computing. Also, the SPLASH codes are not implemented for optimal interaction with memory system characteristics (such as long cache lines and physically distributed memory) or for machines that scale beyond a relatively small number of processors.

Given these limitations, and with the increasing use of SPLASH for architectural studies, we have recently expanded and modified the suite to include several new programs as well as improved versions of the original SPLASH programs. The resulting SPLASH-2 suite contains programs that (i) represent a wider range of interesting computations in the scientific, engineering, and graphics domains; (ii) use better algorithms and implementations than the programs in SPLASH; and (iii) are more architecturally aware. For moderate sized shared-address-space machines with a contemporary memory organization (physically distributed memory, long cache lines, long memory latencies relative to processor speed), we believe that these programs qualify as well-written parallel applications. Working under the assumption that SPLASH-2 may be as widely used as the original SPLASH suite, this paper has two related main goals.

- To characterize the programs in terms of the basic properties and architectural interactions that are important to understand them well.
- To help people who will use the SPLASH-2 programs to prune the architectural design space in an informed and meaningful manner.

The motivation for the first goal is obvious. The motivation for the second goal is more complex, though in many ways more important than the first. In most research studies new architectural ideas are evaluated using simulations; since simulations typically have substantial slowdowns and there are many architectural parameters to vary, people use scaled-down data sets for programs. In using scaled-down data sets for simulation, architects are invariably faced with questions about how to scale machine parameters (such as cache size) to obtain meaningful results that are representative of full-sized problems running on the full-sized machine. Even when the problem size issue can be addressed, one is often left wondering how variations in basic machine parameters (such as cache line size and associativity) and architectural decisions (such as coherence protocol) would affect the validity of their results. Thus our second goal is to provide a comprehensive characterization of SPLASH-2 programs that can be used as a guide by people for choosing architectural parameters for their studies, without each such group having to repeat this work themselves. Since the design space for such a characterization is itself huge, we provide quantitative data where feasible, and we use our knowledge of the programs to provide a qualitative understanding where providing quantitative data is infeasible (due to length or simulation-time limitations). Overall, we hope that this paper will encourage a sound methodology when using the SPLASH-2 programs for workload-driven architectural evaluation.

In addition to the above primary goals, we believe the methodology used to characterize and understand the SPLASH-2 programs is broadly applicable, and people may find it useful to apply it to other application suites as well. Also, several of the architectural characteristics we examine in this paper have been studied before with the older SPLASH suite and/or other parallel applications; the results in this paper for SPLASH-2 programs are of interest in how they corroborate or differ from previous results. We do not describe the SPLASH-2 programs in any detail in this paper, but only to the extent necessary for understanding the results.

1.1 Axes of Characterization

We characterize the SPLASH-2 applications along five axes that we believe are key to understanding their behavior. We examine (i) concurrency and load balancing, (ii) working sets (temporal locality), (iii) communication-to-computation ratios, (iv) data sharing patterns, and (v) spatial locality for the application programs, and how these characteristics change with number of processors and with important application parameters. We discuss the relevance of these characteristics below:

- The *concurrency and load balancing* characteristics of an application program, given a particular data-set, indicate how many processors can be effectively utilized by that program. It is important to characterize these as it does not make sense to use an application to evaluate a multiprocessor system in a regime where the speedups are essentially flat or going down as number of processors is increased.
- The *working sets* of a program [Den68, RSG93] correspond to points where the cache-miss-rate drops sharply as the cache size is increased, and they are indicative of the temporal locality in an application. A detailed characterization of the working sets and how they scale with number of processors and data-set size is key to determining what cache sizes to use in simulation studies for an application.
- The *communication-to-computation ratio* for a program indicates the amount of communication bandwidth needed as a function of processing rate. Other than this data being intrinsically interesting and a fundamental aspect of the application, our goal in characterizing this ratio is so that people in their simulation studies do not make assumptions that are inconsistent with the intrinsic application bandwidth requirements.
- The *data sharing patterns* indicate how many processors read a data item before it is modified by some other processor. This data has no major implications for the methodology we propose, but is presented mainly for completeness of characterization.
- The *spatial locality* for an application program indicates how well the program can exploit long cache lines to prefetch subsequently needed data. In shared-memory multiprocessors, however, large cache lines are a two-edged sword. While they can reduce misses due to prefetching, they can also introduce false-sharing. The data we present on variations of miss-rates and data traffic as a function of cache line size is intended to characterize the SPLASH-2 programs along this important dimension.

1.2 Organization of the Paper

The rest of the paper is organized as follows. Section 2 provides a brief description of the SPLASH-2 applications and the key data structures and algorithms used by each program. Section 3 discusses the simulation environment and the application input data sets used in this paper. Sections 4 through 8 then characterize the programs along the five axes discussed above. Finally, in Section 9, we present our concluding remarks.

2 The SPLASH-2 Application Suite

Like the original SPLASH suite, the SPLASH-2 suite consists of a mixture of complete applications and computational kernels. Currently it consists of 8 complete applications and 4 kernels with a few more to be added in the near future. The new programs include a wide variety of computations that are relevant to multiprocessing in the domains of scientific, engineering, and graphics computing. Some of the original SPLASH codes have been removed, either because of their poor formulation for medium-to-large scale parallel machines (e.g., MP3D), or because they are no longer maintainable (e.g., PTHOR). In addition, some of the programs have been improved either in fundamental algorithmic ways or in data restructuring to improve spatial locality, reduce false-sharing, and allow for efficient data distribution on distributed memory multiprocessors. We now briefly describe the kernels and applications. (Note to reviewers: In the absence of the SPLASH-2 report, we provide short application descriptions here. If accepted, the final version will contain a reference to the SPLASH-2 report instead.)

FFT: The FFT kernel is a complex 1-D version of the radix-$\sqrt{n}$ *Six-step* FFT algorithm described in [Bai90], which is optimized to minimize interprocessor communication. The data set for the FFT consists of the n complex data points to be transformed, and another n complex data points referred to as the *roots of unity*. Both sets of data are organized as $\sqrt{n}$ x $\sqrt{n}$ matrices, and the matrices are partitioned so that every processor is assigned a contiguous set of rows. Communication occurs in the three matrix transpose steps, which require all-to-all interprocessor communication. Every processor transposes a contiguous submatrix of size $\sqrt{n}$/p x $\sqrt{n}$/p from every other processor, and transposes one submatrix locally. In order to exploit cache line reuse, a blocking technique is performed in the transpose phases. Finally, in order to avoid hot-spotting, submatrices are communicated in a staggered fashion, with processor i first transposing a submatrix from processor $i+1$, then one from processor $i+2$, etc. The SPLASH-2 implementation is described more fully in [WSH94].

LU: The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. It can be performed efficiently if the dense n x n matrix A is divided into an N x N array of B x B blocks ($n = NB$) to exploit *temporal* locality on submatrix elements. To reduce interprocessor communication, block ownership is assigned using a 2-D scatter decomposition, with blocks being updated by the processors that own them. The block size B is chosen to be large enough to keep the cache miss rate low, and small enough to reduce the time spent in the less parallel parts of the computation and to maintain good load balance. In practice, relatively small block sizes (B=8 or B=16) strike a good balance. Contiguous elements within a block are allocated in contiguous memory locations to reduce false sharing and obtain spatial locality benefits from long cache lines. The SPLASH-2 implementation is described more fully in [WSH94]

Cholesky: The blocked sparse Cholesky factorization kernel factors a sparse matrix into the product of a lower triangular matrix and its transpose. It is similar in structure and partitioning to the LU factorization kernel, but has two major differences: (i) it operates on sparse matrices, which makes the communication-to-computation ratio much larger, and (ii) to achieve good load balance and performance, it is not globally synchronized between steps like LU.

Radix: The integer radix sort kernel is based on the method described in [BLM+91]. The algorithm is iterative, performing one iteration for each radix r digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram of values. Next, the local histograms are accumulated into a global histogram, which is then used to permute the keys for the next iteration. All-to-all communication occurs in the permutation step. The permutation is inherently a sender-determined one, so that keys are communicated through writes rather than reads. The SPLASH-2 implementation is described more fully in [WSH93].

Ocean: The ocean simulation studies large-scale ocean movements based on eddy and boundary currents, and is an enhanced version of the Ocean application in the SPLASH suite. The major differences between this version and the SPLASH version are: (i) it is written in C rather than FORTRAN, (ii) it partitions the grids into square-like subgrids rather than groups of columns to improve the communication-to-computation ratio, (iii) all subgrids are allocated contiguously and locally in the nodes that own them, and (iv) it uses a red-black Gauss-Seidel multigrid technique based on that presented in [Bra77], whereas the SPLASH version uses a relaxed Gauss-Seidel SOR solver. The SPLASH-2 implementation is described more fully in [WSH93]

Barnes: The BARNES application simulates the interaction of a system of bodies (galaxies or particles, for example) in three dimensions over a number of time-steps, using the Barnes-Hut hierarchical N-body method. While similar to the Barnes-Hut application in SPLASH, it differs in two respects: (i) it allows multiple particles per leaf cell [HoS95], and (ii) it implements

the data structures that represent the cell data structures differently to exploit more data locality. Like the SPLASH application, it represents the computational domain as an octree with the leaves containing records of information on each body and the internal nodes representing space cells. Most of the time is spent in partial traversals of the octree (one traversal per body) to compute the forces on individual bodies. The communication patterns are dependent on the distribution of particles and are relatively unstructured. No attempt is made at intelligent distribution of body data in main memory in the SPLASH-2 version.

FMM: Like Barnes, the FMM application also simulates a system of bodies over a number of time-steps. FMM differs from Barnes in that it simulates interactions in two dimensions using a different hierarchical N-body method called the adaptive Fast Multipole Method [Gre87]. As in Barnes-Hut, the major data structures are body and tree cells, which again are records of body attributes. Like Barnes, FMM also allows multiple particles per leaf cell. However, there are two main differences between them. First, the tree is not traversed once per body, but only in a single upward and downward pass that computes interactions among cells and propagates their effects down to the bodies. Second, the accuracy is not controlled by how many cells a body or cell interacts with but by how accurately each interaction is modeled. The communication patterns are dependent on the distribution of particles and are relatively unstructured. Again, no attempt is made at intelligent distribution of particle data in main memory.

Water-Nsquared: This code is an improved version of the original Water code in SPLASH, and has been renamed. This application evaluates forces and potentials that occur over time in a system of water molecules in the liquid state. Every time-step solves the Newtonian equations of motion for the water molecules, and a predictor-corrector method is used to integrate over time. The main change is that the locking strategy around the updates to the water accelerations is improved: a process updates a local copy of the relevant particle accelerations, and then accumulates into the shared copy once at the end. A more detailed description can be found in [SWG92].

Water-Spatial: This code solves the same molecular dynamics N-body problem as Water-Nsquared, but uses a different algorithm. In particular, it imposes a 3-d spatial data structure on the cubical domain, resulting in a 3-d grid of boxes. The advantage of the spatial grid is that a process that owns a box in the grid has to look at only neighboring boxes for molecules that might be within the cutoff radius of molecules in the box it owns, reducing the complexity of the algorithm from $O(n^2)$ to $O(n)$. For small problems (upto a few thousand molecules, depending on the cutoff radius), the overhead of the spatial data structure is not justified and Water-Nsquared might solve the problem faster. For larger systems of particles, this program is much better.

Radiosity: This application computes the equilibrium distribution of light in a scene using the hierarchical diffuse radiosity method [HSA91]. A scene is initially modeled as comprising a number of large input polygons. Light transport interactions are computed among these polygons, and polygons are hierarchically subdivided as necessary to improve accuracy. The algorithm is iterative, proceeding over several steps until the radiosity converges. In each step, it iterates over the current interaction lists of polygons, subdividing polygons recursively as a result of computing interactions, modifying interaction lists as necessary, and repeating. At the end of a step, the patch radiosities are combined via an upward pass through the quadtree polygon data structures to determine if the radiosity has converged. The main data structures represent polygons, interactions, interaction lists, the quadtree structures, and a BSP tree which facilitates efficient visibility computation between pairs of polygons. The structure of the computation and the access patterns to data structures are highly irregular, except to the BSP tree. Parallelism is managed by a system of distributed task queues, one per processor, with task stealing for dynamic load balancing. No attempt is made at smart data distribution in main memory, since this is very difficult. More detailed descriptions of the parallel method can be found in [SGL94].

Volrend: This code renders a three-dimensional volume onto a two-dimensional image plane using an optimized ray casting technique. A hierarchical octree data structure is used to represent the volume, and early ray termination and antialiasing are implemented. The volume to be rendered is represented by a cube of *voxels* (volume elements). The program is iterative, producing a sequence of viewing frames as the viewpoint changes. Rays are sampled along their path using trilinear interpolation, with a color for the corresponding pixel being computed from these samples. Rays are not reflected, but pass straight through the volume unless they encounter too much opacity, in which case they are terminated early. In the parallel version, the image plane is partitioned into as many contiguous rectangular blocks as there are processors. Every image block or partition is further subdivided into fixed-sized square image tiles, which become the *tasks* in the system. Since the computational load depends on the volume data and is unpredictable, dynamic load balancing is implemented through task stealing. The main data structures are the voxels, the octree and the pixels, and the data accesses are input-dependent and quite irregular, though they follow a ray's path through the voxel space. No attempt is made at smart data distribution, as it is very difficult due to page granularity

restrictions and because the viewpoint changes across frames. This algorithm is described more fully in [NiL92].

Raytrace: This application renders a three-dimensional scene onto a two-dimensional image plane using optimized ray tracing. A hierarchical uniform grid (similar to an octree) is used to represent the scene for efficient access, and early ray termination and antialiasing are implemented. Rays are shot through pixels into the scene and traced, just as in Volrend. However, in Raytrace the rays reflect in unpredictable ways off objects that they strike. Each contact generates multiple rays, resulting in a ray tree per input ray. This causes the application to be far more unpredictable in its data accesses patterns than Volrend. As in Volrend, partitioning is implemented by dividing the image plane among processors, and distributed task queues with task stealing are used for dynamic load balancing. The major data structures represent the rays, the ray trees, the hierarchical uniform grid, the task queues, and the primitives that describe the scene. The parallel algorithm is described more fully in [SGL94].

3 Simulation Environment and Application Data Sets

In this study, we simulate a distributed shared-address-space multiprocessor (DSM) with one processor per processing node. For our base characterizations, we simulate a 16-processor machine, although we vary the number of processors where important to our methodology. Each processor has a single level of cache memory that is kept coherent through the use of the Illinois protocol (dirty, shared, valid-exclusive, and invalid states) [PaP84]. Processors are assumed to send replacement hints to the home nodes when shared copies of data are replaced from their caches, so that the list of sharing nodes maintained at the home contains only those nodes which require invalidations when an invalidating action occurs. The memory system is assumed to be perfect, so that all memory references complete in a single cycle. The reasons for assuming a perfect memory system are two-fold: (i) for non-deterministic codes, such as Radiosity, it is otherwise difficult to compare data (e.g., miss rates, bus traffic) when architectural parameters are varied, and (ii) the focus of this study is not absolute performance of various applications, but intrinsic architecturally-relevant characteristics of the applications; we do not believe that the latter are qualitatively affected by the single-cycle latency model for these applications.[1] The simulator we use for this study is based on the Tango-Lite event reference generator [Gol93], and tracks both true and false sharing misses according to the classification presented in [DSR+93].

Table 1. Breakdown of Instructions Executed for Default Problem Sizes on 16 Processor Machine

Code	Problem Size	Total Instr (M)	Total FLOPS (M)	Total Refs (M)	Total Reads (M)	Total Writes (M)	Shared Reads (M)	Shared Writes (M)	Barriers	Locks
FFT	64K points	34.27	6.36	6.94	4.06	2.88	4.05	2.87	6	0
LU	512x512 matrix, 16x16 blks	489.52	92.20	151.07	103.09	47.99	92.79	44.74	66	0
Radix	256K integers, radix 1024	15.49	---	5.91	3.92	1.98	3.79	1.98	8	512
Cholesky	tk15.O	753.77	172.00	247.78	220.74	27.04	185.55	23.16	3	26136
Ocean	258x258 ocean	376.51	101.54	99.70	81.16	18.54	79.65	16.97	364	1296
Water-Nsq	512 molecules	456.91	98.15	115.09	80.07	35.02	67.94	26.38	10	9376
Water-Sp	512 molecules	435.41	91.50	105.03	72.30	32.73	60.54	22.64	10	177
FMM	16K particles	1245.65	423.88	262.68	224.21	38.47	215.86	30.02	19	27369
Barnes	16K particles	2002.74	239.24	720.13	406.84	313.29	225.04	93.23	7	34516
Volrend	head	755.17	---	211.88	152.31	59.57	82.06	3.06	14	27551
Radiosity	room, -ae 5000.0 -en 0.050 -bf 0.10	2297.19	---	769.56	486.84	282.72	249.67	21.88	10	210485
Raytrace	car	833.35	---	290.35	210.03	80.31	161.10	22.35	0	94456

[1]The assumption that a single-cycle latency model does not affect the intrinsic properties such as cache miss-rates and causes of misses is likely not true for operating-system intensive multiprogrammed workloads, or for other workloads with a real-time component. In such systems the occurrence of timing interrupts intrinsically changes the instructions executed in the workload. For such workloads a different strategy would have to be used.

Data is distributed among the processing nodes in each code according to the guidelines stated in each SPLASH-2 application. All references that occur during the initialization phases of the codes are not counted in the final statistics, so that only the parallel portions of the codes are studied. All codes, except Radix, were compiled with C compiler version 3.18 running on SGI Indy machines with the -O2 optimization level. Radix was compiled and run on a DecStation 5000/260 using the MIPS C compiler version 3.0 again with the -O2 optimization level.

The SPLASH-2 programs come with both default input data sets and optionally larger data set sizes. The default input data-set sizes provided with the SPLASH-2 suite are not meant to be the largest data-set sizes for these codes that are used in practice. Rather, they are meant to be small enough to simulate in a reasonable time, yet large enough to be of interest to researchers in the particular problem domains. We use these default input sets for the runs in this paper, and using knowledge of the applications themselves we qualitatively discuss the impact of larger problem sizes. Table 1 provides concrete information about each of the applications for a 16 processor run. It lists the application data sets, a breakdown of instructions executed, and synchronization information. The instructions executed are broken down into total floating point operations for those applications with significant floating point computation, and reads and writes. Synchronization information is broken down into the number of barriers encountered per processor, and the total number of lock operations across all processors.

4 Concurrency and Load Balancing

One fundamental characteristic of applications is their available concurrency and load balancing characteristics. Since concurrency and load balance are influenced by both the underlying algorithms and the data set size used, it is important to understand these properties and determine the usefulness of an application to a particular study. Algorithms that do not speed up well, for example, offer little insight into the scalability of a large-scale architecture, and should thus not be used for this purpose. In this study we approximate the amount of concurrency and the load balance characteristics of a program by measuring speedup on a PRAM architectural model, in which all memory references complete in a single cycle. Since a perfect memory system is assumed, miss rates do not influence overall performance. Instead, deviations from ideal speedup are attributed to load imbalances, contention due to sequential access in critical sections, and the overhead associated with managing the parallel computation.

Figure 1 shows the available concurrency for the SPLASH-2 suite for up to 32 processors. Most of the codes scale well through 32 processors, and are thus likely to be useful for studies involving larger numbers of processors. (Note to Reviewers: The final paper will also present results for 64 processor runs.) The exceptions are the LU, Cholesky, Radiosity, and Radix applications. In a broad sense, the reasons for the sub-linear speedups for all four of these applications have to do with the input data-set sizes rather than limited intrinsic concurrency. In LU and Cholesky, the default data-sets result in considerable load imbalance despite the block-oriented (in contrast to column-oriented) decompositions. Larger data sets will eliminate some of

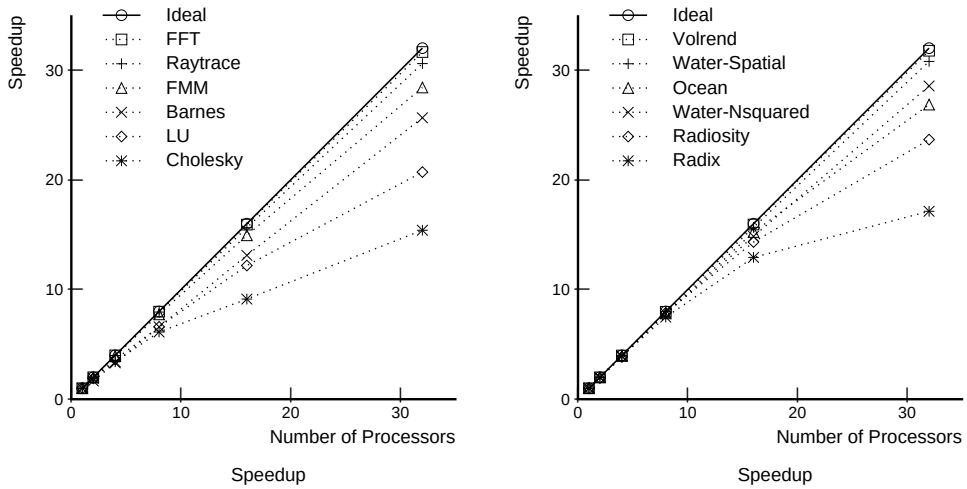

Figure 1: Speedups for the SPLASH-2 applications given default input data sets and a perfect memory system. The poor scalability of LU, Cholesky, and Radiosity is due in large part to small problem sizes. The poor scalability of Radix is due to the non-parallelized prefix computation in each step of the sort.

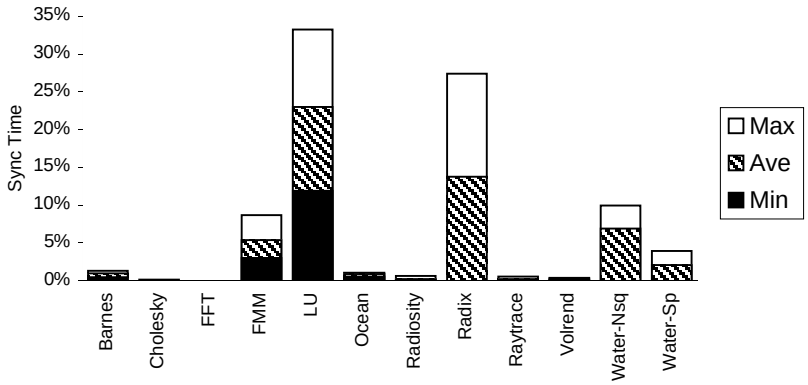

Figure 2: Synchronization characteristics of the SPLASH-2 applications for 16 processor runs. The graph shows a breakdown of minimum, maximum, and average execution time spent in explicit synchronization events across all processors (locks and barriers only; not flags). Note that LU and Radix have the largest average synchronization times, factors which contribute to their poor scalability.

the imbalance, and should be used when considering larger machines. For the Radiosity application, the sublinear speed-up is again due to poor load balance that partly results from use of the small default data set. Unfortunately, a larger data-set size is not currently available from the computer graphics community. Finally, for the Radix application, the poor speedup at 32 processors is due to a non-parallelized prefix computation that is performed in each sort phase. We have produced a new version of Radix that parallelizes this prefix computation. Throughout this paper, however, we will continue to use the results for the currently available of version of Radix; the final paper will contain results for the newer code.

Figure 2 shows a breakdown of synchronization time for the 16 processor runs of each application. The figure indicates the minimum, maximum, and average time spent at explicit synchronization points (locks and barriers, but not flag synchronization) as a fraction of overall execution time. Note that for LU and Radix, the maximum synchronization time exceeds 25% of overall execution time, indicating a load imbalance which results in the sublinear speedup seen in Figure 1. Although Figure 2 indicates that Cholesky and Radiosity spend little time in explicit synchronization, both codes utilize flag synchronization, which is not shown in the figure.

Overall, most of the codes in SPLASH-2 are quite suitable for studying 32-64 processor systems. With the larger data sets, many will quite easily scale to 128 processors.

5 Working Sets and Temporal Locality

The cache miss rate of applications is an important determinant of overall performance, as well as of the impact of architectural tradeoffs and innovations. Since cache size is an important determinant of miss rate, and since we often use scaled down data sets in simulation studies, it is important to understand what cache sizes to use in our experiments. One possbility is to maintain the ratio between the scaled-down and realistic data-sets to be the same as that between simulated and realistic cache sizes. However, this is obviously not a well founded strategy, since neither the miss rates nor the computation requirements of the application interact linearly with cache size or the data set size. In this section we propose a methodology based on working sets to help guide this selection process. Working sets are also a fundamental characteristic of an application program, and are thus intrinsically interesting.

To introduce the notion of working sets {Den68, RSG93], as cache capacity increases the miss rate typically decreases, but usually not in a smooth linear fashion. At particular points in the cache size domain, the miss rate is often observed to exhibit sharp decreases, while staying roughly constant between these points. These points define a hierarchy of *working sets* for an application, denoting cache sizes where critical portions of a program's data set start fitting into the cache. For example, the LU kernel has three working sets. The first working set consists of two columns of a block, the second of an entire block, and the third of all blocks owned by a processor. Some of these working sets are more important than others, where the importance is

primarily determined by the amount by which the miss rate falls at that point (e.g., a cache size that brings the miss rate down from 50% to 1% is much more important than one that brings it down from 1% to 0.02%). Given this perspective, the key points in our proposed methodology are: (i) it is often feasible to pick only one point on each side of the important working set to prune the choice of cache sizes that need to be studied; (ii) by exploit knowledge of how the important working sets scale with problem size and number of processors it is possible to pick cache sizes appropriately for scaled-down data sets. We discuss details below.

Depending on the cache configuration of a machine, a given working set may or may not fit completely in the cache. For a given working set, both of these situations may be reasonable operating points for the application. However, for some working sets one of these operating points may not be meaningful. For example, some applications, such as blocked matrix algebra, are designed to adjust their working sets to fit in the cache. Others have working sets that are always small enough to fit in a realistic cache regardless of problem size. At the other extreme, for some applications it is unreasonable to expect the problem to be run with a small enough data set that the working set does fit in the cache. It is important that we capture the meaningful operating points with regard to working sets and cache organizations that are relevant to our experiments, and not be corrupted by the ones that are unrealistic. Understanding working sets and how they scale with application and architectural parameters is therefore crucial to a good experimental methodology, and it is very important to choose problem and cache sizes based on working set characteristics.

We now examine the working sets of our applications for the base problem sizes, and discuss how they scale with application parameters and the number of processors. Understanding these working sets allows us to use the methodology we are proposing in this paper itself, choose cache sizes that capture the interesting operating points in terms of the important working set for the applications, and prune the rest of the cache size space when investigating other issues.

The inherent working sets of an application are best characterized by simulating fully associative caches with a one-word cache line size. Cache sizes should be varied at a very fine granularity (increments of a few kilobytes, perhaps) with “knees” in the resulting graph of miss rate versus cache capacity representing working sets. However, we are more interested here in using working sets to choose cache sizes of given realistic organizations (line size and associativity) in architectural studies. Thus, we are really interested only in cache sizes that are powers of two, and we plot miss rate for an application versus cache size at this granularity. We therefore do not characterize the knees precisely, but enough to determine what cache sizes to use for the base problem size and a 16-processor configuration. Since the cache size needed to hold a working set depends on cache associativity as well, we use three different realistic associativities, four-way, two-way and one-way, with four-way approximating a fully associative cache in most of our applications. For applications that exhibit poor spatial locality, the cache size needed to hold a working set can grow with line size, since much of every cache line may be wasted. However, particularly since many of our applications exhibit very good spatial locality for realistic cache lines, we ignore this effect and use only our standard 32-byte cache lines to characterize working sets. Finally, we use our knowledge of the applications to explain what constitutes the important working sets, and how they scale with problem size and the number of processors. This qualitative characterization of scaling assumes fully associative caches and a one-word line size, since we cannot qualitatively characterize artifacts of line size and associativity.

Figure 3 shows the plots of miss rate versus cache size for all the applications. We show results for all power-of-two caches between 16KB and 1MB. 1MB is chosen because it is a realistic size for second-level caches today, and it is large enough to comfortably accommodate the important working sets in almost all the applications. We do not examine caches smaller than 16KB since even on-chip caches are becoming about that large. The methodological danger in ignoring very small caches is that there may be an important working set which is smaller than 16KB for our problem size and number of processors, but which grows rapidly with these parameters so that there are realistic situations in practice for which that working set does not fit in the cache. If this were true, then we should also use a cache smaller than 16KB with our problem and machine configuration to model this situation. However, we shall argue based on our understanding and experimentation with the programs that this is not the case.

Let us focus first on the results for 4-way set associative caches (the black bars), since these are the least hampered by data mapping artifacts. One thing we notice is that for most of the applications, the miss rate has essentially stabilized by 1MB caches as stated above. This means that for all these applications the important working sets for this problem size (and for many applications the entire footprint of data that a processor references) are all smaller than 1MB. From the fact that our problem sizes deliver quite good speedups (see Section 4), and from our knowledge of the working sets and their growth rates, we can

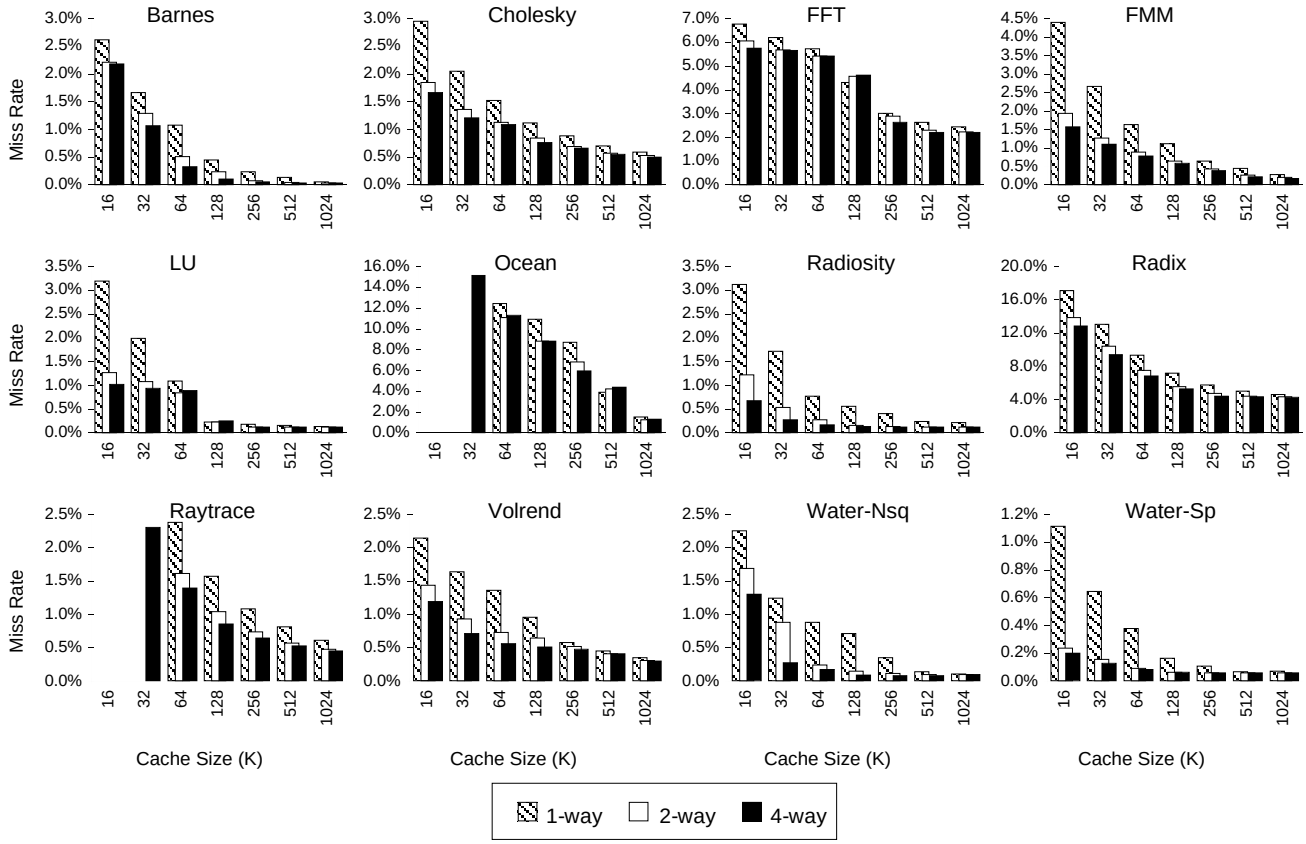

Figure 3: Miss Rates versus Cache Size and Associativity. Miss rate data assumes 16 processor runs with 32 byte line sizes. Note that for many applications the difference in miss rate from 1-way to 2-way is much larger than the difference from 2-way to 4-way. Some data points are missing, and if accepted, the final version will contain the missing data.

infer that having the important working sets fit in the cache is an important operating point to consider. The question is whether there is also a realistic operating point where an important working set does not fit in a modern secondary cache in practice. Let us examine this question next.

Many of the applications (LU, Cholesky, Barnes, FMM, FFT, Radiosity, Water-Nsquared, Water-Spatial) start out with a very low miss rate even with 16KB caches. If we looked to the left of 16KB, we would find that all of these applications have a sharp knee in their miss rate curves in that region after which the miss rates are low. This indicates that the most important working sets for these problem sizes is less than 16KB. In all of these cases, we know that these working sets are independent of the number of processors and grow either very slowly or not at all with the data set size. They are therefore expected to almost always fit in realistic second-level caches in practice. For example, in blocked LU and Cholesky factorization the important working set is essentially a single block which is always sized to fit in the cache. There are other working sets in these programs that are larger, but these usually amount to a processor's entire partition of the data set.

The applications that have an important operating point for which a working set does not fit in the cache are Ocean, Raytrace, Radix and to a lesser extent FFT. Ocean is an iterative regular-grid computation that streams through its partition of different grid in different phases (as such, it is an important representative of computational fluid dynamics computations). It can therefore incur substantial capacity and conflict misses as problem sizes become larger. Raytraces traces rays through potentially huge scenes, with somewhat large and diffuse working sets due to unstructured reflections of rays. Radix streams through different sets of keys with both regular and irregular strides in two types of phases, and also accesses a small histogram heavily in one phase. This results in a working set (though not the most important one) that is not very sharply defined. And in FFT, a processor typically captures its most important working set in the cache, but may or may not capture the next one (its set of rows in a single matrix) in a realistic configuration.

Table 2. Important Working Sets and Their Growth Rates for the SPLASH-2 Suite

Code	Important Working Set	Growth Rate of Working Set
FFT	One row of matrix	$\sqrt{DS}$
LU	One block	Constant
Radix	Histogram	Radix r
Cholesky	One block	Constant
Ocean	Individual partition	DS / P
Water-Nsq	Private data	Constant
Water-Sp	Private data	Constant
FMM	Number of terms in expansion	Constant
Barnes	Tree data needed by 1 body	$\log DS$
Volrend	Octree and part of 1 ray	K $\log DS$ + k $\sqrt[3]{DS}$
Radiosity	BSP tree	Constant
Raytrace	Unstructured	Unstructured

Following our methodology, for these four applications, here and in subsequent sections we also examine a cache size that does not accommodate the working sets described above in their entirety. If there were a sharp knee that characterized these working sets, we would choose a cache size below that knee. However, for three of the applications (Ocean, Raytrace and Radix) these working sets are not sharp, so we choose a reasonable cache size that gives us a representative effect of capacity misses. A cache size of 64KB works out well, and is also to the left of the knee for the better-defined FFT application. We therefore present results for these applications with both a 1MB cache and a 64KB cache.

Figure 3 also shows us how the miss rates, and potentially the desirable cache sizes to examine, change with associativity. We typically find that going from 1-way (direct-mapped) to 2-way associative caches improves miss rates greatly, while going from 2-way to 4-way changes them only a little. While direct-mapped caches sometimes change the power-of-two cache size required to hold the working set compared to 4-way associative, 2-way caches do not appear to do so for these problem sizes. The scaling of cache sizes needed to hold the working set is difficult to predict for caches with low associativity (due to data mapping artifacts). Researchers need to identify the desirable cache sizes for themselves when using low-associativity cache simulations with other problem sizes than the base ones used here, and to use the methodology outlined in this section when pruning the cache size space.

Finally, Table 2 shows the constitutions and growth rates of the important working sets for the different applications. In this table, DS refers to the data set size of the problem, P refers to the number of processors, and K and k are large and small constants, respectively (see Section 2.) The growth rates shown in Table 2 indicate that, in general, working sets for the SPLASH-2 suite are quite small and remain constant or grow only slowly with problem size. The most notable exceptions are iterative CFD computations like Ocean, ray tracing and radix sorting. While in the other cases the size of the important working set is independent of the number of processors, in Ocean and Radix the size of the important working set actually falls linearly with the number of processors. Overall, the above examples clearly indicate that a sound experimental methodology must involve a strong understanding of the interplay between working set hierarchies and architectural parameters.

6 Communication-to-Computation Ratios

Another key to understanding an application is understanding its communication behavior. Important aspects for researchers include the communication-to-computation ratio and sharing patterns (examined later in Section 7), and how these vary with problem size, number of processors, and other architectural parameters. The communication-to-computation ratio indicates the intrinsic communication requirements of an application, but when coupled with specific architectural parameters, such as processor speed, cache-line size, and the use of prefetching, it translates into bandwidth requirements for the machine.

An example of the importance of understanding the communication-to-computation ratio and its effects on bandwidth requirements is a previous study by Tullsen and Eggers [TuE93] which showed that if bandwidth is a constraint, then prefetching may actually hurt performance.

The inherent communication behavior of an application is best characterized by simulating infinite caches with one-word cache lines to eliminate artifacts due to cache capacity effects, mapping conflicts, and false sharing. As stated previously in the context of working sets, however, we are more interested in obtaining characteristics for realistic machine parameters. We rely on our previous working set analysis and choose a 1MB, 4-way set associative cache with 32-byte lines as the base for our communication-to-computation ratio characterization.

We use two metrics to evaluate the communication-to-computation ratio for the programs in the SPLASH-2 suite. For codes that perform large amounts of integer computation, we report bytes per instruction executed, averaged over all processors. For codes that are dominated by floating point computation, we present the communication traffic per floating point operation (FLOP), since the number of FLOPS is less influenced by compiler technology than the number of instructions. Our data for communication traffic are broken down into three categories: (i) *remote data*, which is the traffic caused by all data and associated headers being transmitted to fulfill remote processor requests and writebacks, (ii) *overhead*, which is the traffic associated with remote data requests, invalidations, acknowledgments, and replacement hints, and (iii) *local data*, which is the amount of data transmitted by accesses to local memory (and which are not dirty in another processor's cache). In all cases, we assume that headers for data packets, and all other overhead packets are 8 bytes long.

The two left graphs in Figure 4 show the traffic per instruction for the six applications in SPLASH-2 that are not floating-point intensive (note that the results for Radix have a different Y-axis scale). The results are shown for 8, 16, and 32 processor runs, all using the default input data-set sizes. Looking at the 16 processor results first (the middle bar for each application), we see that for all applications other than Radix, the overall traffic is less that 0.1 bytes per instruction. With 200 MIPS processors, this translates to less than 20 MB/sec of traffic per processor, which in the absence of contention effects is well under the per-processor network bandwidths found on shared memory multiprocessors today. For Radix, the traffic for a 200 MIPS processor approaches 120 MB/sec per processor. This is quite high, so evaluation studies using Radix should model both memory contention and network bandwidth limitations to provide confidence in results.

The data in Figure 4 also shows that for 32-byte cache lines there is substantial overhead traffic. This indicates that it is important for studies to model overhead traffic when working with such small cache line sizes (we study the impact of larger cache line sizes in Section 8). The amount of local traffic is usually small as well, a result of the fact that the 1 Mbyte caches

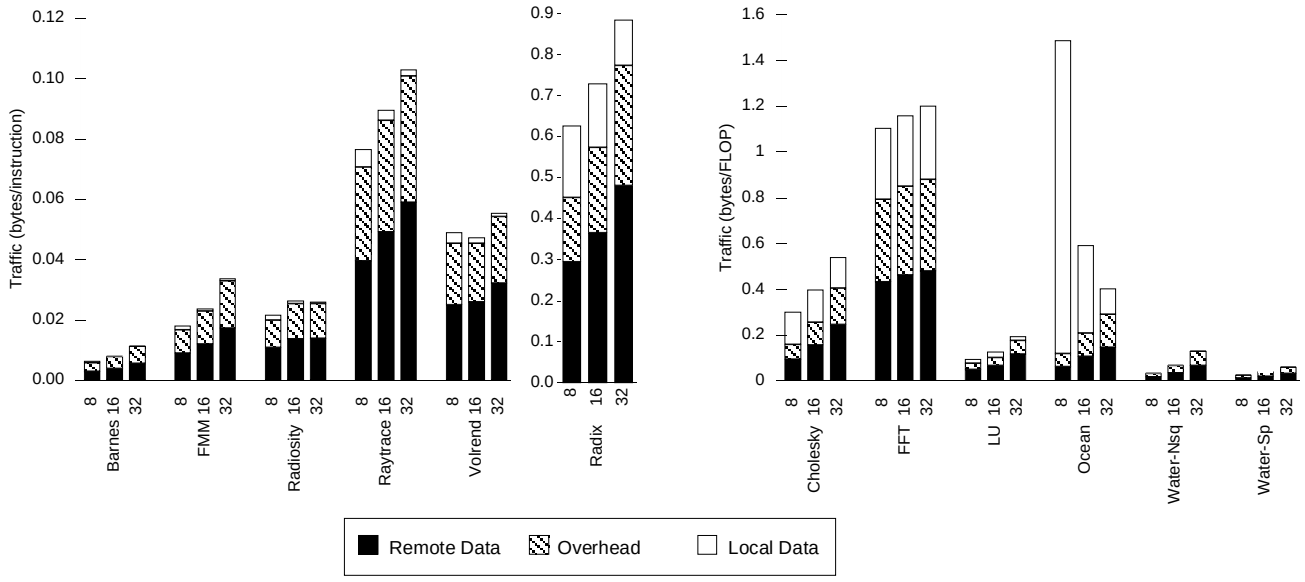

Figure 4: Breakdown of traffic generated (in bytes) per instruction and per FLOP. The results are shown for 8, 16, and 32 processors. The graphs assume 1 Mbyte, 4-way associative, 32-byte line caches. All overhead packets and data headers are assumed to be 8 bytes long.

capture most capacity misses. (A decomposition of misses by type is shown in Figure 8 in Section 8.) In a number of these applications, however, it is very difficult to do data distribution well among physically distributed memories. Consequently, for these applications, even capacity misses can generate substantial remote traffic (Raytrace and Volrend are perfect examples of this).

The right graph in Figure 4 shows the traffic-per-FLOP for the remaining six floating-point intensive applications. Again, with 1MB caches, the remote traffic (the bottom two sections of each bar) is typically quite small. For Cholesky, with a processor executing at 200 MFLOPS, the required bandwidth is about 60 MB/sec per processor, not unreasonable for networks found on multiprocessors today. The exception is FFT, in which case the remote bandwidth requirement is close to 160 MB/sec per node. In fact, in FFT communication is bursty, so studies using FFT should be especially careful about modeling network and memory system bandwidth and contention. Data in Figure 4 also shows that for 32-byte cache lines there is substantial overhead traffic, as was true for integer applications; such traffic should thus be modeled in studies operating close to network bandwidth limits.

Let us now look at the effects on traffic of changing the number of processors while keeping the problem size fixed. Typically, the communication-to-computation ratio increases with the number of processors due to a finer-grained decomposition of the problem. This effect can easily be seen in Figure 4 as traffic due to remote communication almost always increases as we go from 8 to 16 to 32 processors. Local traffic often stays relatively independent of the number of processors, but in many cases decreases since the amount of data that each processor references becomes smaller while the cache size per processor stays constant. This effect is often most clearly observed in applications like Ocean which have regular access patterns to large amounts of mostly local data, but it can also be seen in less regular applications like Volrend. In Ocean, unlike most other applications, local capacity misses dominate communication misses even at 1MB caches, and the overall traffic is reduced substantially as more processors are used. One important methodological implication is that when there are substantial capacity misses remaining at the default cache size, then it is crucial for architectural studies to explore the sensitivity of the results to variations in cache size.[2]

As we had stated earlier, for four of the applications (Ocean, Raytrace, Radix and FFT) we need to also look at the 64KB cache results. These are shown in Figure 5. The total traffic (including local) is generally much larger than for the 1 Mbyte caches, since the number of capacity misses increases due to an important working set not fitting in the cache. In some applications

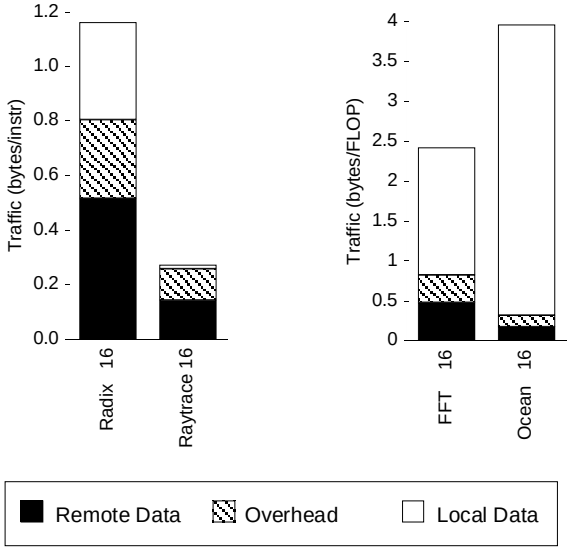

Figure 5: Breakdown of traffic generated for 16 processors using 64Kbyte, 4-way associative, 32-byte line caches. All overhead packets and data headers are assumed to be 8 bytes long.

[2]The 1 Mbyte caches we are using in the base runs are close to what are used in real machines today. Thus, given that the data sets that we are using are interesting and relevant to the application domain people, the 1Mbyte results are important, regardless of the amount of capacity misses that we may observe.

(e.g. Ocean) these capacity misses are mostly to local data. For other applications (such as Raytrace) the capacity misses are not satisfied locally, and thus the communication traffic increases. In these cases, the nonlocal traffic volume per node can become quite large on modern processors when an important working set does not fit in the cache, even though it is fairly small when the working set fits. Sometimes the capacity misses by different processors can be to lines that are allocated on the same processor's node, making it all the more important to also model contention.

Finally, we examine how the communication-to-computation ratio and remote bandwidth requirements change with increase in problem size. There are two opposing effects that we need to take into account. Table 3 shows how the communication-to-computation ratio changes with data set size and number of processors for the applications in which these growth rates can be modeled analytically, where DS represents the data set size, and P represents the number of processors. On one hand, communication-to-computation ratio for *sharing* traffic usually goes down, the reason being that a larger problem size means larger-grained tasks. For example, in applications that perform localized communication in physical domains, computation is often proportional to volume while communication is proportional to surface area. On the other hand, given a fixed cache size, a larger problem size can mean more capacity misses in cases where a working set grows with problem size, and depending on whether data can or can not be placed in local memory, these capacity misses manifest themselves as local misses or remote (communication) misses. Also, depending on how close 1MB is to the knee of the working set, it may behoove the experimenter to re-estimate the working sets and communication-to-computation ratios as problem sizes increase.

As an illustration of the dependence of total traffic and its composition on problem size, number of processors, and working sets, Figure 6 illustrates the impact of increasing problem size on overall traffic with three different numbers of processors for FFT. Comparing the communication-to-computation ratio for input data set sizes of 2^{16} and 2^{18} points, we see that the communication-to-computation ratio (and hence remote data traffic) is smaller for the larger problem, since the FFT algorithm is O(n) in communication, but O(n log n) in computation. As the number of processors is increased, the remote traffic increases for both problem sizes. However, the local traffic decreases for the 2^{18} point FFT, because the data set fits in the cache at some point as we discussed for Ocean. Although not shown for reasons of space, a similar effect is seen for both local and remote traffic with smaller caches (64KB).

Table 3. Growth Rates of the Communication-to-Computation Ratio for the SPLASH-2 Suite

Code	Growth Rate of Comm/Comp Ratio	Code	Growth Rate of Comm/Comp ratio
FFT	$1 / \log DS$	Water-Sp	$\sqrt{P} / \sqrt{DS}$
LU	$\sqrt{P} / \sqrt{DS}$	FMM	approximately $\sqrt{P} / \sqrt{DS}$
Radix	constant	Barnes	approximately $\sqrt{P} / \sqrt{DS}$
Cholesky	$\sqrt{P} / \sqrt{DS}$	Volrend	constant
Ocean	$\sqrt{P} / \sqrt{DS}$	Radiosity	unpredictable
Water-Nsq	1 / DS	Raytrace	unpredictable

Overall, the above results reaffirm the importance of understanding the interplay between sizes and scaling of the important working sets of the application, program data-set size, number of processors, as well as interactions with other machine parameters when determining communication-to-computation ratios and designing experiments to evaluate architectural ideas.

7 Data Sharing Patterns

Another important characteristic of applications is the degree of sharing they exhibit. An analysis of the invalidation behavior of an application reveals insight into its sharing properties, and such studies have been used in the past to determine tradeoffs for specific implementations of directory-based coherence protocols. While an analysis of sharing patterns does not have methodological implications, we include it here for completeness of characterization.

Figure 7 shows the invalidation distributions for each of the SPLASH-2 applications for 16 processor runs. The x-axis

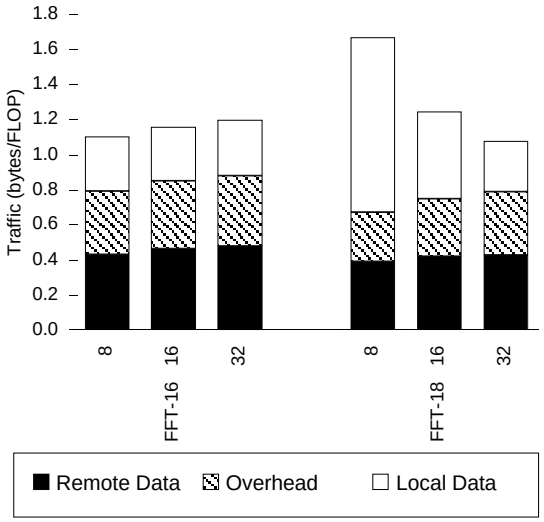

Figure 6: Breakdown of traffic for FFT in bytes per floating point operation for two problem sizes f for 8, 16, and 32 processor runs. The graph assumes 64Kbyte, 4-way associative, 32-byte line caches.

represents the number of invalidations sent out to other processors, and the y-axis represents the fraction of invalidating writes (write misses or write-hits to shared lines) causing that many invalidations. Looking at Figure 7, it is possible to group the applications into three categories.

The first category of applications is those where writes cause mostly 0 or 1 invalidations, such as Cholesky, FFT, LU, Ocean, Radix, Raytrace, and Volrend. In these applications writes cause a small number of invalidations for a variety of reasons. In FFT, Radix, Ocean, Raytrace, and Volrend the majority of data sharing happens pair-wise between processors (e.g., at grid-

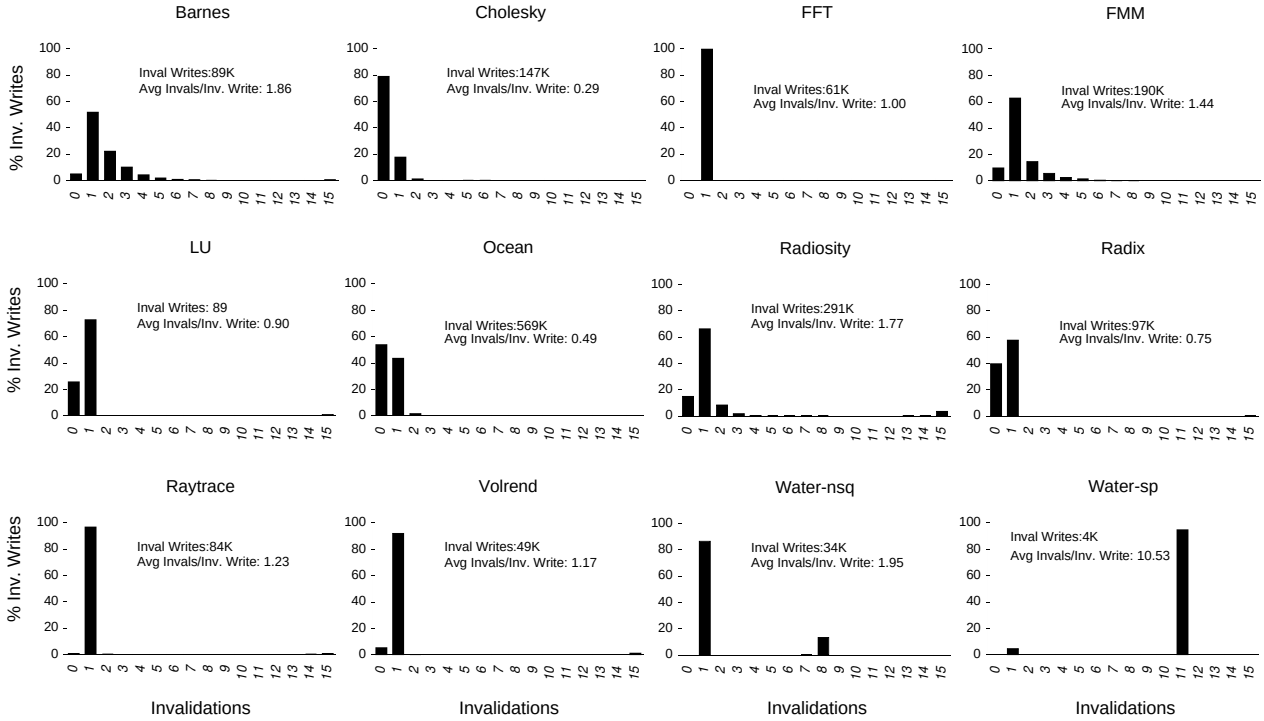

Figure 7: Invalidation histograms for 16 processors with 1MB, 4-way associative, 32 byte line caches. Applications can be grouped into 3 categories based on the patterns exhibited in these histograms: (i) those with little sharing, (ii) those with varying amounts of sharing, and (iii) those with large amounts of sharing.

boundaries in Ocean) and hence the mostly 0/1 invalidations (Note that zero invalidations may occur because our protocol uses replacement hints and shared data may be replaced from the cache before it is written). In LU, a given block of the matrix is written to by only one processor, and only when it is complete is it read by all other processors. Thus, ideally, there should be no invalidation causing writes, as is borne out by the small number (89). Finally, Cholesky is similar to LU, except that there are some queues that are shared between the processors, and hence the somewhat larger number of invalidating writes. The invalidation patterns in these applications are not expected to change as either number of processors or problem size is increased.

The second category of applications is exemplified by Barnes and FMM where a majority of writes cause 0 or 1 invalidations, but there are many that cause higher number of invalidations. (The LocusRoute application in the old SPLASH suite was similar.) In both these applications, the observed invalidation patterns result from the shared-tree structure that is built on every time step. Information from the root node is shared more widely than information at lower levels of the tree, and this is reflected in the "tapering" nature of the invalidation histograms. As number of processors is increased, the tail of the histograms is expected to extend somewhat more. As problem size is increased, the peaks at lower invalidations will become higher, as an increasing number of writes will correspond to modifications to lower levels of the tree, where there are fewer sharers.

The third category of applications is exemplified by Radiosity, Water-Nsquared, and Water-Spatial. In these applications, while most writes cause small invalidations there are peaks at much higher points. In Radiosity there is a peak at 15 because in the initial stages of the algorithm, key polygon data is shared and read by all processors, and this data is later modified. In Water-Nsquared, pairwise molecular interactions are computed for each molecule with half of the other molecules, which in turn are owned by half of the other processors. For this reason there is always a peak at one half of the number of processors. In Water-Spatial, the peak at 11 invalidations roughly corresponds to the number of neighboring blocks that a given block may have in a dense 3-D grid of blocks. Overall, as the number of processors is increased, the number of invalidations caused by writes is expected to scale-up for Radiosity and Water-Nsquared, and it is expected to stay roughly constant for Water-Spatial, with some additional invalidations occurring to smaller numbers of processors. As problem size is increased, the peaks at lower invalidations should become higher for Radiosity, and the patterns should remain the same for Water-Nsquared and Water-Spatial.

So far we have looked at the invalidation patterns for 1Mbyte caches. Our methodology suggests that we also examine the patterns for FFT, Ocean, Radix, and Raytrace for 64Kbyte caches. Since all four of these applications fall in the first category discussed above, the invalidation patterns remain essentially the same. The only difference is that the fraction of writes causing 0 invalidations increases, because there is now greater chance that shared data is replaced from a processor's cache before it is written to by some other processor.

8 Spatial Locality

Spatial locality in applications is a key property exploited by computer architects in designing memory systems, and in particular, in choosing the cache line size for machines. Programs which exhibit good spatial locality perform well with long cache lines due to prefetching effects [Smi82]. Programs with poor spatial locality do better with shorter cache lines because they avoid wasting bandwidth fetching extraneous data. Another issue concerning cache line sizes that is particular to multiprocessors is false sharing [EgJ91, TLH94]. As line size is increased, distinct variables being accessed by different processors are more likely to reside on the same cache line. When using an invalidation based protocol, this leads to frequent invalidations and subsequent cache misses, although no useful values are being communicated between the processors. Given these concerns about spatial locality and false sharing, our goal here is to characterize the behavior of SPLASH-2 applications as a function of cache line size, discuss implications of changing number of processors and problem size, and how other parameters such as working set size and communication to computation ratios are affected by line size. In particular, we hope the data presented here will enable people to decide when and for which SPLASH-2 applications it is critical to study a range of cache line sizes in their experiments.

We investigate the spatial locality and sharing behavior of the SPLASH-2 application suite by investigating cache miss-rates and traffic as the cache line size is varied. If there were perfect spatial locality, if we were to increase the line size k-fold, we would expect the miss-rate to drop k-fold and we would expect total memory traffic to remain the same. In order to understand reasons for less than perfect reduction of miss rates in practice (which may be due to lack of spatial locality, or increased interference in a small cache, or true/false sharing misses), we divide the miss rate of applications into four constituent compo-

nents: (i) cold misses, (ii) capacity (replacement) misses, (iii) true sharing misses, and (iv) false sharing misses. We use the definition of true and false sharing as presented by Dubois et al. [DSR+93]. In this classification, cold misses account for the first reference to a memory block by each processor (thus cold misses per instruction can grow as number of processors increases). Capacity misses have their usual meaning. A true sharing miss is any miss where the processor subsequently uses values defined by other processors since the last true, cold or replacement miss by this processor. This definition recognizes the beneficial effects of long cache lines in capturing required communication. In contrast, a false sharing miss is one where the values defined by other processors are not used by the time the line is replaced from the processor's cache.[3]

Following our methodology, we first present results for all applications for 1 MB 4-way set associative caches. By using a cache large enough to capture the working sets we are able to focus on an operating point where for most applications overall miss-rate is not dominated by capacity misses. From a methodological point, we note that there is a little bit of chicken-and-egg problem here, since the working set size is itself impacted by the cache line size. At a high-level though, if using the default cache size we find that the application has good spatial locality, then we can infer that the working set size is unlikely to change with line size. However, if we observe poor spatial locality, then methodologically it behooves us to recompute the working set for different line sizes.

Figure 8 shows the breakdown of miss rate for the SPLASH-2 application suite as line size is varied from 8 to 256 bytes. (Note: The bars contain an extra category, Upgrades. Upgrades are writes that find the memory block in the cache but in shared-state, and thus have to send an "upgrade" request for ownership. We discount upgrades in our discussion below; they are used in the traffic computations we do later.)

The data in Figure 8 show that FFT, LU, Cholesky, and Ocean have both good spatial locality and little false sharing. There is good spatial locality because the miss-rates drop proportionately to increases in cache line size, and there are essentially no false sharing misses. (Note as per the footnote on the previous page, many of the cold misses in these applications are really

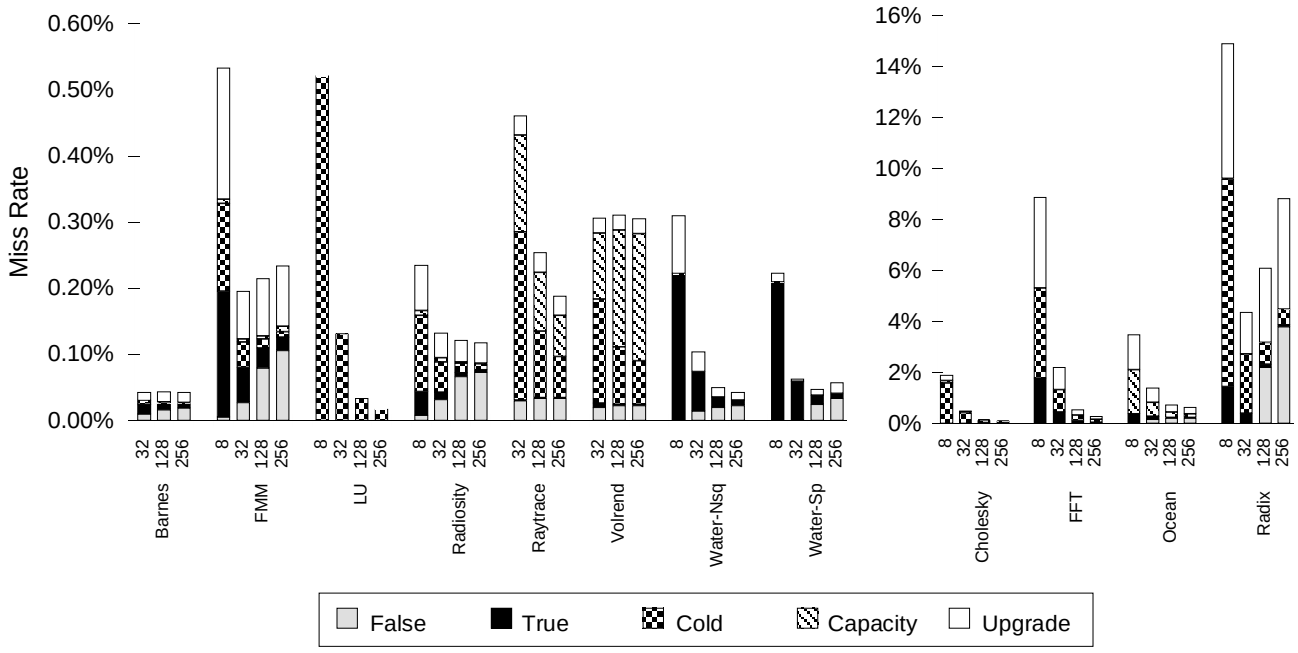

Figure 8: Breakdown of miss rate versus cache line size. The 4-way associative caches have capacities fixed at 1MB, and line sizes that are varied from 8 to 256 bytes.

[3]As per our current definitions and implementation, we sometimes end up classifying true sharing misses as cold misses. For example, in LU after a pivot block is completed, it is read by all other processors. Since the other processors are reading this block for the first time, these misses are all labeled cold misses. Intuitively, these are really sharing misses. In the final version of the paper, this non-intuitive classification will be fixed. It will, however, not change the nature of results or the conclusions we draw in the section.

Similar to the above case, if a processor replaces a line from its cache, and that same line is subsequently written by another processor, any ensuing miss by the first processor is counted as a capacity miss and not a sharing miss. This case represents a genuine dilemma about how to classify such a miss.

true sharing misses.) This is in large part due to the fact that these are matrix codes and that architecturally-aware data-structuring has been done for these applications. For example, a grid in Ocean is not represented as a single 2-D array (which can introduce substantial false-sharing), but as a 2-D array of blocks each of which is itself a 2-D array. Such structuring, by programmers or compilers, ensures that most accesses are unit-stride and over small blocks of data, and thus the nice behavior. As for scaling, the spatial locality for these applications is expected to remain good and false-sharing non-existent both as the problem size is increased and as the number of processors is increased. This should be true even for cache lines much larger than 256 bytes. In some sense, the cache line size for these applications becomes an almost "uninteresting" parameter since the resulting behavior is very predictable across all line sizes.

Water-Nsquared and Water-Spatial are also regular array-based codes, like those just discussed, and see good spatial locality for cold, capacity, and true-sharing misses. However, a substantial fraction of misses starting at 128 byte lines are false-sharing misses. In Water-Nsquared the false-sharing misses arise because the particle data structure (record) brings together both (i) fields that are being modified by the owner of that particle (e.g., the current force on this particle), and (ii) fields that are read-only by other processors and that are not being modified in this phase (e.g., the current position of the particle). Since these two fields may fall in the same cache line for large line sizes, false-sharing results. It is possible to eliminate such false-sharing by splitting the particle data structure but that is not currently done, as the absolute magnitude of the miss-rate is small. As for scaling, both spatial locality and false-sharing are unlikely to be affected by larger data-set size or more processors. In Water-Spatial the source of false-sharing is the head of the linked list of particles that are in each box. As particles move from box to box, they are taken off of one list and placed on the next. The pointers themselves are small, thus causing false sharing. As for scaling, both the spatial locality and false-sharing behavior are expected to improve for larger problem sizes, as each processor will have more particles and contiguous boxes (in 3-D) allocated to it. The effect of increasing number of processors for fixed problem size is exactly the opposite.

Barnes, FMM, and Radiosity employ complex data structures, including trees encoding spatial information and arrays where records assigned to each processor are not contiguous in memory. For example, both Barnes and FMM operate on particle records stored in an array. As the application proceeds and particles move in space, particle records get reassigned to different processors, with the result that the processor's data set becomes fragmented. True sharing misses then seldom see good spatial locality because adjacent records are often unused. For the same reason, false sharing becomes a problem at large line sizes, as different processors write to records that are adjacent within a cache line. The reason why we saw false-sharing in Water-Nsquared is also applicable to Barnes and FMM. As problem size is scaled and number of processors is scaled, the behavior of Barnes and FMM applications is not expected to change much. This is because the working set size changes very slowly (as log of number of particles), spatial locality is determined by the record size of one particle and thus remains the same, and finally because the sources of false sharing are not sensitive to number of processors. Radiosity is, unfortunately, a much more complex application whose behavior is difficult to predict with larger data sets or more processors; empirical data showing the growth trends appears to be the only alternative for now.

The graphics codes Raytrace and Volrend show only minimal false sharing but quite poor spatial locality. The false sharing is small because for both applications the main data structures (collection of polygons/voxels) are read-only. The primary sharing happens at the image plane, which is well controlled, and thus only a small factor. The reason for these application's poor spatial locality is that they use non-uniform data structures with irregular access patterns. The set of objects that a ray will bounce off of is unpredictable. In fact, Volrend actually has its capacity miss rate increase with line size due to cache conflicts. From a methodology perspective, it appears that with 1 Mbyte caches we are working too close to the working set determined earlier for 32-byte lines, and we need to reevaluate the working set for larger cache lines. As for scaling, as problem size is increased (most likely in the form of more polygons/voxels) the primary effect is likely to be larger capacity miss rates; the spatial locality and false-sharing should not change. A larger number of processors is in most ways similar to having a smaller problem size, except that we may see slightly more false sharing in the image-plane data structure.

The poorest sharing behavior is exhibited by Radix, which not only has a very high miss rate (due to cold and true sharing misses), but which gets significantly worse due to false sharing misses for line sizes of 128 bytes or more. The false-sharing in Radix arises as follows. Consider sorting 256K keys, using a radix of 1024, and 16 processors. On average, this results in 16 keys per radix per processor (64 bytes of data), which are then written to a contiguous portion of a global array but (conceptually) at a random starting point. Since 64 bytes is smaller than the 128 byte cache lines, the high potential for false-sharing is clear. As the problem size is increased (e.g., to 4 million keys above), it is clear that we will see much less false sharing. The

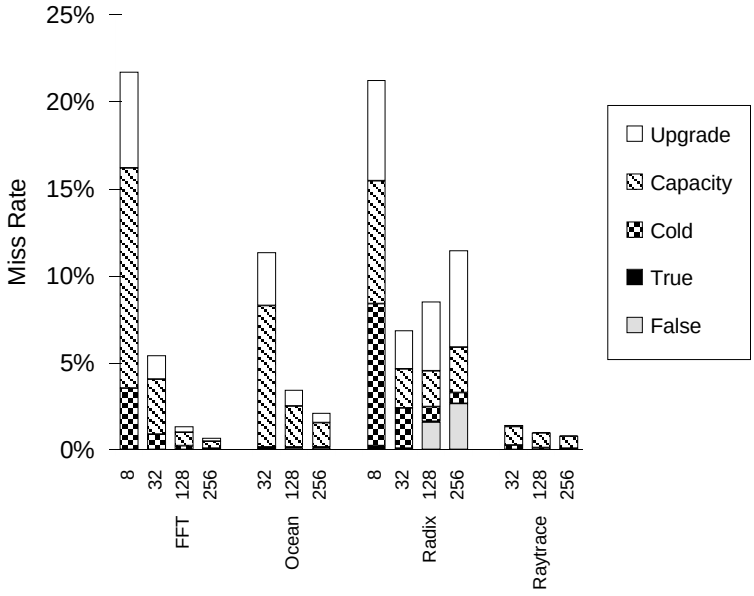

Figure 9: Breakdown of miss rate versus cache line size with 64Kbyte, 4-way associative caches for applications with important working sets larger than this.

effect of increasing number of processors is exactly the opposite.

As per our discussion of working sets, it is also interesting to look at the behavior of FFT, Ocean, Radix, and Raytrace for 64Kbyte caches. The miss-rate results are shown in Figure 9. As one may have expected, the overall miss rates are higher and capacity misses have increased substantially. However, the spatial locality and false-sharing trends have not changed significantly as compared to the results for 1 Mbyte caches, mostly because these properties are fairly fundamental to data structure and algorithms used by a program and are not too sensitive to cache size. The key new interaction that we see is for the Ocean application, where the capacity misses show somewhat poor spatial locality. The reason is that because of the small cache size data blocks are being thrown out of the cache due to interference before the processor has had a chance to reference all of the words in a cache line.

Finally, we briefly discuss the impact of cache-line size on traffic. Figure 10 shows the total traffic for a DSM machine in bytes per instruction/FLOP (as described in Section 6) as line size is varied. There are three points to observe from this graph. Firstly, although overall miss rate decreases for most of the SPLASH-2 applications for the line sizes presented, network traffic behaves quite differently. Only FFT and LU shows monotonically decreasing traffic for these line sizes. Most other applications see a doubling or tripling of network traffic as line size becomes large. Thus, for studies using long line sizes it may be necessary to recompute the communication to computation ratio for that particular line size to be able to justify decisions on what is sufficient bandwidth. Secondly, given the above, the overall traffic requirements for the SPLASH-2 suite are still small for very large line sizes, with the exception of Radix. Radix's large bandwidth requirements reflect its false sharing problems at large line sizes. If bandwidth is a possible concern, then, Radix makes a good stress test. Finally, the constant overhead for each network transaction comprises a significant fraction of total traffic for small line sizes. Hence, although actual data traffic increases as we increase the line size, due to poor spatial locality, the total traffic is usually a minimum at between 32 and 128 bytes. This confirms previous studies [GuW92] which show that the overall message traffic is usually a minimum for cache line sizes of 32 bytes, and remphasizes that overhead traffic should not be excluded in such studies for DSM machines.

To summarize, given the above characterizations, it is important that users of the SPLASH-2 suite consider the behavior of each application carefully when choosing a line size. Specifically, studies using the SPLASH-2 suite should include a range of line sizes for those programs which have poor spatial locality (e.g., Raytrace, Volrend) or are prone to false sharing (e.g., Radix, FMM, Radiosity) to demonstrate the influence of these effects on the design. Furthermore, we stress that it is not sufficient to look at a given problem size, a given number of processors, and based on that draw conclusions of whether false-sharing or spatial locality are or are not a problem. It is very important to understand how the results are dependent on the particular

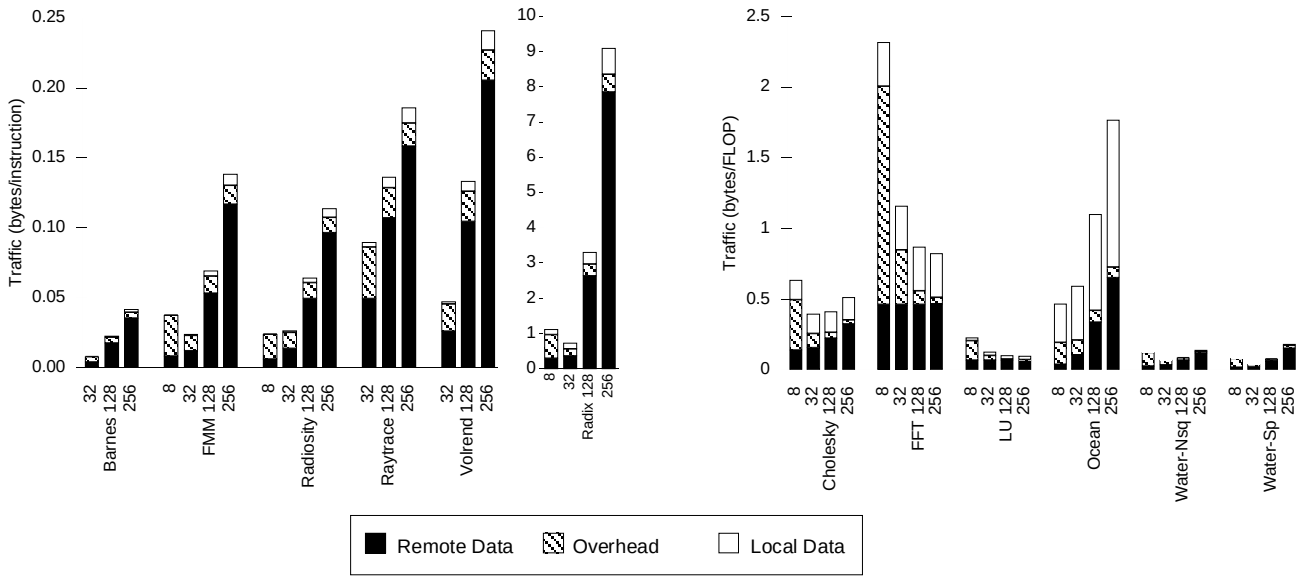

Figure 10: Breakdown of traffic versus line size for 1MB, 4-way associative caches.

parameters chosen in the experiment and how these parameters may vary in reality.

9 Concluding Remarks

A thorough workload-driven evaluation of architectural tradeoffs for multiprocessors is a difficult task due to the large number of interacting degrees of freedom. Many machine parameters such as cache size, associativity, and line size can impact the results, as can application parameters and the number of processors used. Since it is extremely time consuming to perform complete sensitivity analyses on all these parameters, and because the problem sizes used in simulation studies are typically forced to be either unrealistically small or at least at the small end of what one might encounter in practice, it is very important that a good methodology be developed to evaluate tradeoffs through simulation. It seems clear that such a methodology must include an understanding of relevant characteristics of the programs being used, and the implications of pruning the parameter space. In particular, we believe that it is very important to understand the interplay of problem size, number of processors, working set size, and cache size, and how they affect the results of architectural evaluations. It is also useful for us to understand the interaction of data structuring techniques and program optimization on performance, so as to be able to infer what effects are fundamental and what are artificial and/or short-term glitches.

To this end, we have provided a detailed characterization of the SPLASH-2 application suite across five axes. These axes are concurrency and load balancing, working-set sizes, communication-to-computation ratio, data sharing patterns and spatial locality. Characterizing the individual applications across these axes provides insight into the sensitivity of the applications to changes in various architectural parameters, thus helping to prune the design alternatives that need to be examined.

In addition to this characterization, we have outlined a methodology which we believe should also be useful for studies involving other application suites. Our methodology is based on the notion of hierarchy of working sets, and relies on the observation that the miss-rate vs. cache-size graph for applications has sharp knees and otherwise flat portions. We suggest that for most studies it is necessary to pick only one point on each side of the important working set, thus pruning the choice of cache sizes that need to be studied. Similarly, for other parameters, such a cache associativity, we suggest that the experimenters determine (either through experimentation or application-level knowledge) flat regions in the parameter space, and then pick only one point from each flat region to prune the design space.

Finally, while we believe our methodology is useful in many situations, its limitations, where relevant should also be understood. For example, the working-set model we have proposed for picking cache sizes to simulate does not work for comparing update versus invalidate protocols. While our model captures the importance of miss-rate of processor-issued references, it does not capture the fact that an update protocol (e.g., Dragon [McC84] or Firefly [ArB86]) keeps generating update traffic on the bus as long as some other cache has a copy of that memory block, even if the processor associated with that cache does

not refer to that memory block. As cache size increases, larger numbers of "dead" lines occupy the processor caches, requiring increasing amounts of bandwidth to service the useless updates. Thus, while our model would suggest that we pick only one point in the flat region below a knee, the traffic for an update protocol would be quite different for the various points in that flat region.

References

[ArB86] James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273-298, November 1986.

[Bai90] David H. Bailey. FFT's in External or Hierarchical Memory. *Journal of Supercomputing*, 4(1):23-35, March 1990.

[BLM+91] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pp. 3-16, July 1991.

[Bra77] Achi Brandt. Multi-Level Adaptive Solutions to Boundary-Value Problems. *Mathematics of Computation* 31(138):333-390.

[Den68] Peter J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323-333.

[DSR+93] Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, Krishnan Ramamurthy, and Per Stenstrom. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 88-97, May 1993.

[EgJ91] Tor E. Jeremiassen and Susan J. Eggers. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pp. 377-381.

[Fox88] Geoffrey Fox, et. al. *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems.* Prentice Hall, 1988.

[Gea71] C. W. Gear. *Numerical Initial Value Problems in Ordinary Differential Equations.* Prentice Hall, 1971.

[Gol93] Stephen Goldschmidt. Simulation of Multiprocessors: Accuracy and Performance. Stanford University, June 1993.

[Gre87] Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* ACM Press, 1987.

[GuW92] Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794-810, July 1992.

[HSA91] Pat Hanrahan, David Salzman and Larry Aupperle, "A Rapid Hierarchical Radiosity Algorithm", In *Proceedings of SIGGRAPH* 1991.

[HoS95] Chris Holt and Jaswinder Pal Singh. Hierarchical N-Body Methods on Shared Address Space Multiprocessors. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, Feb 1995, to appear.

[McC84] McCreight, E. The Dragon Computer System: An Early Overview. Technical Report, Xerox Corporation, September 1984.

[NiL92] Jason Nieh and Marc Levoy, "Volume Rendering on Scalable Shared-Memory MIMD Architectures", In *Proceedings of the Boston Workshop on Volume Visualization*, October 1992.

[PaP84] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 348-354, 1984.

[Rot92] Edward Rothberg. Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization. Ph.D. Thesis, Stanford University, November 1992.

[RSG93] 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 International Symposium on Computer Architecture*, pp. 14-25, May 1993.

[Sco91] David S. Scott. Efficient All-to-All Communication Patterns in Hypercube and Mesh Topologies. In *Proceedings of the 6th Distributed Memory Computing Conference*, pp. 398-402, 1991.

[Sin93] Jaswinder Pal Singh. Parallel Hierarchical N-body Methods and Their Implications for Multiprocessors. Ph.D. Thesis, Stanford University, February 1993.

[SGL94] Jaswinder Pal Singh, Anoop Gupta and Marc Levoy, "Parallel Visualization Algorithms: Performance and Architectural Implications", *IEEE Computer* 27(7):45-55, July 1994.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5-44, March 1992.

[Smi82] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473-530, September 1982.

[TLH94] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651-663, June 1994.

[TuE93] Dean M. Tullsen and Susan J. Eggers. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 278-288, May 1993.

[VeF92] Jack E. Veenstra and Robert J. Fowler. A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 149-159, October 1992.

[WSH94] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pp. 219-229, October 1994.

[WSH93] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Message Passing in Cache-Coherent Multiprocessors. Stanford University Technical Report No. CSL-TR-93-593, December 1993.