To appear in the Proceedings of the 22nd Annual International Symposium on Computer Architecture, June 1995

The SPLASH-2 Programs: Characterization and Methodological Considerations

Steven Cameron Woo[†], Moriyoshi Ohara[†], Evan Torrie[†],
Jaswinder Pal Singh[‡], and Anoop Gupta[†]

[†]Computer Systems Laboratory
Stanford University
Stanford, CA 94305

[‡]Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

The SPLASH-2 suite of parallel applications has recently been released to facilitate the study of centralized and distributed shared-address-space multiprocessors. In this context, this paper has two goals. One is to quantitatively characterize the SPLASH-2 programs in terms of fundamental properties and architectural interactions that are important to understand them well. The properties we study include the computational load balance, communication to computation ratio and traffic needs, important working set sizes, and issues related to spatial locality, as well as how these properties scale with problem size and the number of processors. The other, related goal is methodological: to assist people who will use the programs in architectural evaluations to prune the space of application and machine parameters in an informed and meaningful way. For example, by characterizing the working sets of the applications, we describe which operating points in terms of cache size and problem size are representative of realistic situations, which are not, and which re redundant. Using SPLASH-2 as an example, we 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 programs written for cache-coherent shared address space machines. While SPLASH has provided a degree of consistency and comparability across studies, like any other suite of applications it has many limitations. In particular, it consists of only a small number of programs and does not provide broad enough coverage even of scientific and engineering computing. The SPLASH programs are also not implemented for optimal interaction with modern memory system characteristics (long cache lines, high latencies 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, the suite has recently been expanded and modified 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 computations in the scientific, engineering and graphics domains; (ii) use better algorithms and implementations; and (iii) are more architecturally aware.

This paper has two related goals.

- To characterize the SPLASH-2 programs in terms of the basic properties and architectural interactions that are important to understand them well.
- To help people who will use the programs for system evaluation to choose parameters and prune the experimental space in informed and meaningful ways.

While the first goal is clearly useful—it provides data about the behavior of new parallel programs and allows us to compare the results with those of previous studies—the second is in many ways more important. Architectural evaluations are faced with a huge space of application as well as machine parameters, many of which can substantially impact the results of a study. Performing a complete sensitivity analysis on this space is prohibitive. In addition, most architectural studies use software simulation, which is typically very slow and compels us to scale down the problem and machine configurations from those we would really like to evaluate. Finally, many points in the parameter space (scaled down or original) lead to execution characteristics that are not representative of reality, so blind sensitivity sweeps may not be appropriate anyway. For these reasons, it is very important that we understand the relevant characteristics of the programs we use for architectural evaluation, and how these characteristics change with problem and machine parameters. The goal is to avoid unrealistic combinations of parameters, choose representative points among the realistic ones, and prune the rest of the space when possible.

In this paper, we provide the necessary quantitative characterization and qualitative understanding for the SPLASH-2 programs. We also identify some specific methodological guidelines that emerge from the characterization. By doing this, we hope to help people prune their parameter spaces, and also contribute to the adoption of sound experimental methodology in using these (and other) programs for architectural evaluation.

The next section discusses the particular program characteristics that we measure, and our motivations for choosing them. It also describes our overall approach to gathering and presenting results. In Section 3, we provide a very brief description of each of the SPLASH-2 programs, concentrating on the features that will later be relevant to explaining the effects observed. Sections 4 through 7 characterize the programs along the dimensions discussed in Section 2. Finally, we present some concluding remarks in Section 8.

2 Characteristics and Approach

2.1 Axes of Characterization

We characterize the programs along four axes which we consider most important to understanding shared address space programs from the viewpoint of choosing experimental parameters. They are: (i) speedup and load balancing, (ii) working sets, (iii) communication to computation ratios and traffic needs, and (iv) issues related to spatial locality. We also discuss how these characteristics change with important application parameters and the number of processors, since understanding this is very important for using the programs appropriately.

- The *concurrency and load balancing* characteristics of a program indicate how many processors can be effectively utilized by that program, assuming a perfect memory system and communication architecture. This indicates whether a program with a certain data set is appropriate for evaluating the communication architecture of a machine of a given scale. For example, if a program does not speed up well, it may not be appropriate for evaluating a large scale machine.
- The *working sets* of a program [Den68, RSG93] indicate its temporal locality. They can be identified as the knees in the curve of cache miss rate versus cache size. Whether or not an important working set fits in the cache can have a tremendous impact on local memory bandwidth as well as on communication needs. It is therefore crucial to understand the sizes and scaling of the important working sets, so that application and machine parameters can be chosen in ways that represent realistic situations. As we shall see, knowledge of working sets can help us prune the cache size dimension of the parameter space.
- The *communication to computation ratio* indicates the potential impact of communication latency on performance, as well as the potential bandwidth needs of the application. The actual performance impact and bandwidth needs are harder to predict, since they depend on many other characteristics such as the burstiness of the communication and how much latency is hidden. Our goal in characterizing this ratio and how it scales is to guide simulation studies against making unrepresentative bandwidth provisions relative to bandwidth needs. In addition to the inherent communication in the application, we also characterize the total communication traffic and local traffic for a set of architectural parameters.
- The *spatial locality* in a program also has tremendous impact on its memory and communication behavior. In addition to the uniprocessor tradeoffs in using long cache lines (prefetching, fragmentation and transfer time), cache-coherent multiprocessors have the potential drawback of *false sharing*, which causes communication and can be very expensive. We therefore need to understand the spatial locality and false sharing in our programs, as well as how they scale.

There are two important program characteristics that we do not examine quantitatively in this paper: the patterns of data sharing or communication, and contention. While the first is useful in understanding a program and the second can be performance-critical, they are not as important as the other issues from the viewpoint of choosing application and memory system parameters.

2.2 Approach to Characterization

Experimental Environment: We perform our characterization study through execution-driven simulation, using the Tango-Lite reference generator [Gol93] to drive a multiprocessor cache and memory system simulator. The simulator tracks cache misses of various types according to an extension of the classification presented in [DSR+93] developed to handle the effects of finite cache capacity. We simulate a cache-coherent shared address space multiprocessor with physically distributed memory and one processor per node. Every processor has a single-level cache that is kept coherent using a directory-based 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.

All instructions in our simulated multiprocessor complete in a single cycle. The performance of the memory system is assumed to be perfect (PRAM model [FoW78]), so that all memory references complete in a single cycle as well regardless of whether they are cache hits, or whether they are local or remote misses. There are two reasons for this. First, for non-deterministic programs it is otherwise difficult to compare data (e.g., miss rates, bus traffic) when architectural parameters are varied, since the execution path of the program may change. Second, the focus of this study is not absolute performance but the architecturally-relevant characteristics of the programs. While these can sometimes be affected by the interleaving of instructions from different processors and hence the timing model (in both deterministic and particularly in nondeterministic programs), a given timing model is not necessarily any better than another from this perspective. In fact, we believe that the effect of the timing model on the characteristics we measure is small for our applications, including the nondeterministic ones.[1]

Data are distributed among the processing nodes according to the guidelines stated in each SPLASH-2 application. In most cases, we begin our measurements just after the parallel processes are created. The exceptions are cases where the application would in practice run for many more iterations or time-steps than we simulate. In these cases, we start our measurements after initialization and cold start. All programs were compiled with cc compiler version 3.18 on Silicon Graphics Indy machines with the -O2 optimization level.

Data Sets and Scaling: Default input data sets are specified for the programs in the SPLASH-2 suite. For almost all applications, larger data sets are either provided or automatically generated by the programs. These data sets are by no means large compared to practical data sets likely to be run on real machines. Rather, they are intended to be small enough to simulate in a reasonable time, yet large enough to be of interest in their problem domains in practice. While data set size and number of processors can have tremendous impact on the results of characterization experiments, due to space constraints we present most of our quantitative data for the default problem configuration and a fixed number of processors. We fix the number of processors at 32 for most of our characterizations, (except the communication to computation ratio), and discuss the effects of scaling the number of processors qualitatively.

Inherent versus Practical Characteristics: A question that arises in such a study is whether to characterize the *inherent* properties of the applications or to characterize properties that arise with realistic machine parameters. For example, the best way to measure inherent communication is by using infinite caches with a line size of a single word, and the best way to measure inherent working sets is with a one-word cache line and fully-associative caches. However, these memory system parameters are not realistic, and unlike timing parameters can change the observed characteristics substantially. Ideally, we would present both inherent properties and those obtained with realistic machine parameters, but space constraints prevent us from doing so. Since researchers are likely to use the SPLASH-2

[1] This may not be true for multiprogrammed workloads that exercise the operating system intensively, or for other workloads with a real-time component, since changes to the instructions executed by a processor (due to non-determinism) may affect memory system behavior substantially. The impact of the timing model on memory system behavior cannot be ignored so easily in these cases.

suite with realistic memory system parameters, we choose to focus on these while still trying to approach inherent properties and avoid too many artifacts. For example, the default line size for our characterizations (other than when we vary it to measuring spatial locality) is 64 bytes, which leads us away from inherent properties. On the other hand, our default cache associativity is 4-way, which is realistic but large enough to be relatively free of cache mapping artifacts.

We cannot present as much data as we would like in this paper, and since researchers may want to view this and other data in other ways, we have created an online database of characterization results. This database has a menu-driven interface to an interactive graphing tool that allows results for different combinations of machine and experiment parameters to be viewed. The tool and database are accessible via the World Wide Web at http://www-flash.stanford.edu/.

3 The SPLASH-2 Application Suite

The SPLASH-2 suite consists of a mixture of complete applications and computational kernels. It currently has 8 complete applications and 4 kernels, which represent a variety of computations in scientific, engineering, and graphics computing. Some of the original SPLASH codes have been removed 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) and some have been improved. We now briefly describe the applications and kernels. More complete descriptions will be available in the upcoming SPLASH-2 report. In these descriptions, p refers to the number of processors used.

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. It differs from the version in SPLASH in two respects: (i) it allows multiple particles per leaf cell [HoS95], and (ii) it implements the cell data structures differently for better data locality. Like the SPLASH application, it represents the computational domain as an octree with leaves containing information on each body, and 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 particle distribution and are quite unstructured. No attempt is made at intelligent distribution of body data in main memory, since this is difficult at page granularity and not very important to performance.

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 (see LU), but has two major differences: (i) it operates on sparse matrices, which have a larger communication to computation ratio for comparable problem sizes, and (ii) it is not globally synchronized between steps.

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 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 $\sqrt{n} \times \sqrt{n}$ matrices partitioned so that every processor is assigned a contiguous set of rows which are allocated in its local memory. Communication occurs in three matrix transpose steps, which require all-to-all interprocessor communication. Every processor transposes a contiguous submatrix of $\sqrt{n/p} \times \sqrt{n/p}$ from every other processor, and transposes one submatrix locally. The transposes are blocked to exploit cache line reuse. To avoid memory hotspotting, 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. See [WSH94] for more details.

FMM: Like Barnes, the FMM application also simulates a system of bodies over a number of timesteps. However, it simulates interactions in two dimensions using a different hierarchical N-body method called the adaptive Fast Multipole Method [Gre87]. As in Barnes, the major data structures are body and tree cells, with multiple particles per leaf cell. FMM differs from Barnes in two respects: (i) the tree is not traversed once per body, but only in a single upward and downward pass (per timestep) that computes interactions among cells and propagates their effects down to the bodies, and (ii) 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 quite unstructured, and no attempt is made at intelligent distribution of particle data in main memory.

LU: The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$) to exploit *temporal* locality on submatrix elements. To reduce 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 should be large enough to keep the cache miss rate low, and small enough to maintain good load balance. Fairly small block sizes (B=8 or B=16) strike a good balance in practice. Elements within a block are allocated contiguously to improve spatial locality benefits, and blocks are allocated locally to processors that own them. See [WSH94] for more details.

Ocean: The Ocean application studies large-scale ocean movements based on eddy and boundary currents, and is an improved version of the Ocean program in SPLASH. The major differences are: (i) it partitions the grids into square-like subgrids rather than groups of columns to improve the communication to computation ratio, (ii) grids are conceptually represented as 4-D arrays, with all subgrids allocated contiguously and locally in the nodes that own them, and (iii) it uses a red-black Gauss-Seidel multigrid equation solver [Bra77], rather than an SOR solver. See [WSH93] for more details.

Radiosity: This application computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method [HSA91]. A scene is initially modeled as a number of large input polygons. Light transport interactions are computed among these polygons, and polygons are hierarchically subdivided into patches as necessary to improve accuracy. In each step, the algorithm iterates over the current interaction lists of patches, subdivides patches recursively, and modifies interaction lists as necessary. At the end of each step, the patch radiosities are combined via an upward pass through the quadtrees of patches to determine if the overall radiosity has converged. The main data structures represent patches, 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. Parallelism is managed by distributed task queues, one per processor, with task stealing for load balancing. No attempt is made at intelligent data distribution. See [SGL94] for more details.

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. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication. The permutation is inherently a sender-determined one, so keys are communicated through writes rather than reads. See [WSH93, HHS+95] for details.

Raytrace: This application renders a three-dimensional scene using ray tracing. A hierarchical uniform grid (similar to an octree) is used to represent the scene, and early ray termination and antialiasing are implemented, although antialiasing is not used in this study. A ray is

Code	Problem Size	Total Instr (M)	Total FLOPS (M)	Total Reads (M)	Total Writes (M)	Shared Reads (M)	Shared Writes (M)	Barriers	Locks	Pauses
Barnes	16K particles	2002.79	239.24	406.85	313.29	225.05	93.23	8	34648	0
Cholesky	tk15.O	539.17	172.00	111.86	28.03	75.87	23.31	3	54054	4203
FFT	64K points	34.79	6.36	4.07	2.88	4.05	2.87	6	0	0
FMM	16K particles	1250.02	423.88	226.23	38.58	217.84	30.10	20	28088	0
LU	512 × 512 matrix, 16 × 16 blocks	494.05	92.20	104.00	48.00	93.20	44.74	66	0	0
Ocean	258 × 258 ocean	379.93	101.54	81.89	18.93	80.26	17.27	364	2592	0
Radiosity	room, -ae 5000.0 -en 0.050 -bf 0.10	2832.47	---	499.72	284.61	261.08	21.99	10	231190	0
Radix	1M integers, radix 1024	50.99	---	12.06	7.03	12.06	7.03	10	0	124
Raytrace	car	829.32	---	208.90	79.95	159.97	22.22	0	94471	0
Volrend	head	754.77	---	152.19	59.57	81.93	3.07	15	28934	0
Water-Nsq	512 molecules	460.52	98.15	81.27	35.25	69.07	26.60	10	17728	0
Water-Sp	512 molecules	435.42	91.50	72.31	32.73	60.54	22.64	10	353	0

Table 1. Breakdown of instructions executed for default problem sizes on a 32 processor machine. Instructions executed are broken down into total floating point operations across all processors for applications with significant floating point computation, reads, and writes. The number of synchronization operations is broken down into number of barriers encountered per processor, and total number of locks and *pauses* (flag-based synchronizations) encountered across all processors.

traced through each pixel in the image plane, and reflects in unpredictable ways off the objects it strikes. Each contact generates multiple rays, and the recursion results in a ray tree per pixel. The image plane is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used with task stealing. The major data structures represent rays, ray trees, the hierarchical uniform grid, task queues, and the primitives that describe the scene. The data access patterns are highly unpredictable in this application. See [SGL94] for more information.

Volrend: This application renders a three-dimensional volume using a ray casting technique. The volume is represented as a cube of voxels (volume elements), and an octree data structure is used to traverse the volume quickly. The program renders several frames from changing viewpoints, and early ray termination and adaptive pixel sampling are implemented, although adaptive pixel sampling is not used in this study. A ray is shot through each pixel in every frame, but rays do not reflect. Instead, rays are sampled along their linear paths using interpolation to compute a color for the corresponding pixel. The partitioning and task queues are similar to those in Raytrace. The main data structures are the voxels, octree and pixels. Data accesses are input-dependent and irregular, and no attempt is made at intelligent data distribution. See [NiL92] for details.

Water-Nsquared: This application is an improved version of the Water program in SPLASH [SWG92]. This application evaluates forces and potentials that occur over time in a system of water molecules. The forces and potentials are computed using an $O(n^2)$ algorithm (hence the name), and a predictor-corrector method is used to integrate the motion of the water molecules over time. The main difference from the SPLASH program is that the locking strategy in the updates to the accelerations is improved. A process updates a local copy of the particle accelerations as it computes them, and accumulates into the shared copy once at the end.

Water-Spatial: This application solves the same problem as Water-Nsquared, but uses a more efficient algorithm. It imposes a uniform 3-D grid of cells on the problem domain, and uses an O(n) algorithm which is more efficient than Water-Nsquared for large numbers of molecules. The advantage of the grid of cells is that processors which own a cell need only look at neighboring cells to find molecules that might be within the cutoff radius of molecules in the box it owns. The movement of molecules into and out of cells causes cell lists to be updated, resulting in communication.

Table 1 provides a basic characterization of the applications for a 32-processor execution. We now examine the four characteristics previously discussed for the SPLASH-2 suite.

4 Concurrency and Load Balance

As discussed in Section 2.1, the concurrency and load balance of a program—and how they change with problem size and number of processors—are very important to understanding whether an application and data set are appropriate for a study involving a machine with a given number of processors. For example, if a program is limited by computational load balance to a small speedup on a given input, it may not be appropriate for use in evaluating a large-scale machine. We study how the computational load balance scales with the number of processors by measuring speedups on a PRAM architectural model. When measured in this manner, deviations from ideal speedup are attributable to load imbalance, serialization due to critical sections, and the overheads of redundant computation and parallelism management.

Figure 1 shows the PRAM speedups for the SPLASH-2 programs for up to 64 processors. Most of the programs speed up very well even with the default data sets. The exceptions are LU, Cholesky, Radiosity, and Radix. To illustrate load imbalance further, Figure 2 shows the time spent waiting at synchronization points for 32-processor executions of each application. Figure 2 indicates the minimum, maximum, and average fraction of time, over all processes, spent at synchronization points (locks, barriers, and pauses). Note that for Cholesky, LU and Radiosity, the average synchronization time exceeds 25% of overall execution time.

The reasons for sub-linear speedups in the above four applications have to do with the sizes of the input data-sets rather than the inherent nature of the applications. In LU and Cholesky, the default

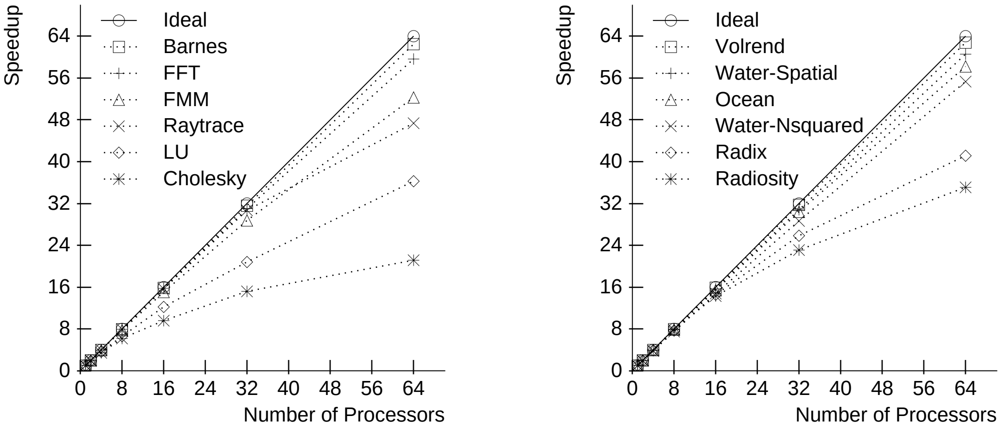

Figure 1: Speedups for the SPLASH-2 applications given the default input data sets shown in Table 1 and a perfect memory system. The poor scalability of LU, Cholesky, and Radiosity is due in large part to small problem sizes. The poor scalability of Radix is due to a prefix computation in each phase that is not completely parallelizable.

data sets result in considerable load imbalance for 64 processors, despite their block-oriented decompositions. Larger data sets reduce the imbalance by providing more blocks per processor in each step of the factorization. In LU, the number of blocks per processor in the k^{th} step (out of n/B total blocks) is $(1/p)((n/B)-k)^2$ (see Section 3 for an explanation of these parameters). Thus, larger input data sets should be used when studying larger machines. For Radiosity, the sublinear speedup is also due to the use of a small data set. The imbalance is difficult to analyze, and a larger data set is not currently available in an appropriate form. Finally, for Radix the poor speedup at 64 processors is due to a parallel prefix computation in each phase that cannot be completely parallelized. The time spent in this prefix computation is $O(\log p)$ while the time spent in the other phases is $O(n/p)$, so the fraction of total work in this unbalanced phase decreases quickly as the number of keys being sorted increases. All four applications can therefore be used to evaluate larger machines as long as larger data sets are chosen.

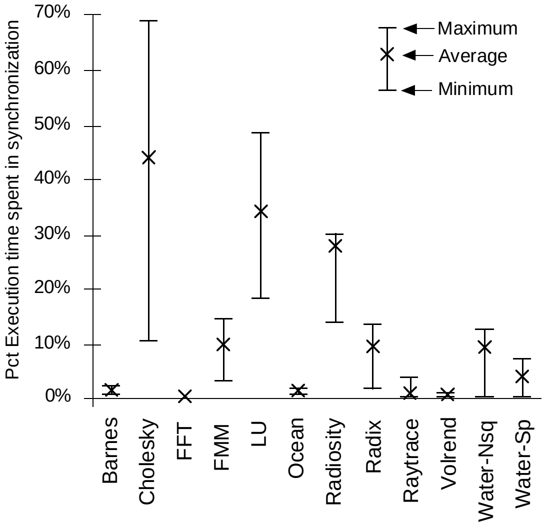

Figure 2: Synchronization characteristics of the SPLASH-2 suite for 32 processors. The graph shows a breakdown of minimum, maximum, and average execution time spent in synchronization across all processors (locks, barriers, and pauses) as well as user defined synchronization (for Radiosity only).

Overall, even with the default input data sets, most of the programs in SPLASH-2 scale well and are suitable for studying 32-64 processor systems. Because of this they are likely to be useful for studies involving larger numbers of processors as well.

The next characteristics we examine are the sizes and scaling of the important working sets of the applications. We study working sets before communication to computation ratio because in the latter section we examine not only inherent communication but also artifactual communication and local traffic, for which we use our understanding of working sets to pick parameters.

5 Working Sets and Temporal Locality

The temporal locality of a program, and how effectively a cache of given organization exploits it, can be determined by examining how a processor's miss rate changes as a function of cache size. Often, the relationship between miss rate and cache size is not linear, but contains points of inflection (or *knees*) at cache sizes where a working set of the program fits in the cache [Den68]. As shown in [RSG93], many parallel applications have a hierarchy of working sets, each corresponding to a different knee in the miss rate versus cache size curve. Some of these working sets are more important to performance than others, since fitting them in the cache lowers the miss rate more.

Methodological Importance: Depending on how data are distributed in main memory, the capacity misses resulting from not fitting an important working set in the cache may be satisfied locally (and increase local data traffic), or they may cause inter-node communication. Methodologically, it is very important that we understand the sizes of an application's important working sets and how they scale with application parameters and the number of processors, as well as how a cache's ability to hold them changes with line size and associativity. This can help us determine which working sets are expected to fit or not fit in the cache in practice. In turn, this helps us achieve our methodological goals of avoiding unrealistic situations and selecting realistic ones properly. This understanding is particularly important when scaling down problem sizes for ease of simulation, since cache sizes for the reduced problems must be chosen that represent realistic situations for full-scale problems running with full-scale caches.

When the knees in a working set curve are well defined knees, and particularly when they are separated by relatively flat regions, they present a valuable opportunity to prune the cache size dimension of the experimental space. Knowledge of the size and scaling of an important working set can indicate whether (i) it is unrealistic for that working set to fit in the cache for realistic problem and machine parameters, (ii) it is unrealistic for that working set to not fit in the cache, or (iii) both situations, fitting and not fitting, are realistic. This knowledge indicates which regions in the miss rate versus cache size curves are representative of practice and which are not. This helps us prune the space in two ways. First, we can ignore the unrealistic regions. Second, if the curve in a representative region is relatively flat, and if all we care about with respect to a cache is its miss rate, then a single operating point (cache size) can be chosen from that region and the rest can be pruned.

The inherent working sets of an application are best characterized by simulating fully associative caches of different sizes with one-word cache lines. Cache sizes should be varied at a fine granularity to precisely identify the sizes at which knees occur. Following our interest in realism (Section 2.2) however, we use a 64-byte line size, and examine only cache sizes that are powers of two. Since the cache size needed to hold a working set depends on cache associativity as well, we present results for three finite associativities, four-way, two-way and one-way. We also supply fully-associative miss rate information for comparison. We ignore the impact of line size on cache size required to hold working sets for space reasons. However, because most of the programs exhibit very good spatial locality the required cache size does not change much.

Figure 3 depicts the miss rate as a function of cache size for the SPLASH-2 suite. Results are shown for all power-of-two cache sizes between 1KB 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 our applications. The methodological danger in ignoring caches smaller than 1KB is that there may be an important working set smaller than this for our default problem sizes, but which grows rapidly at larger problem sizes so that it may no longer fit in the cache. However, this is not true for these applications.

We first examine the results for our default 4-way set associative caches (the bold lines). We see that for all the applications, the miss rate has either completely or almost completely stabilized by 1MB caches. The important working sets for these problem sizes (and for several applications the entire footprint of data that a processor references) are smaller than 1MB. From the fact that these are realistic problem sizes that can also yield good speedups (see Section 4), and from the sizes and growth rates of the working sets that we shall discuss, we infer that having the important working sets fit in the cache is an important operating point to consider for all applications. We therefore use 1MB caches as one of the operating points in the rest of our characterizations. The question is whether there is also a realistic operating point in practice where an important working set does not fit in a modern secondary cache. In this case we should also choose a cache size to represent this operating point. We examine this question next.

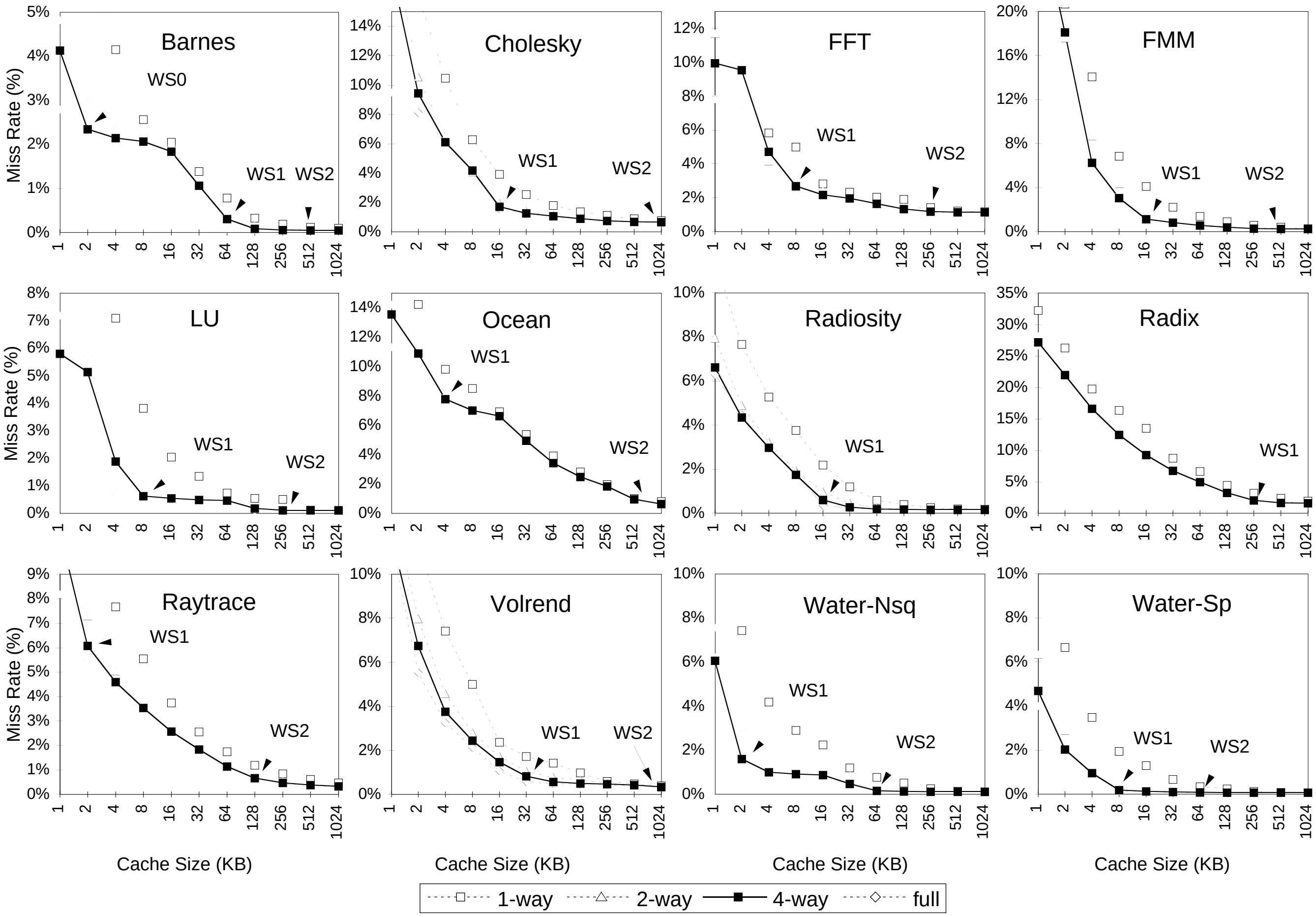

Figure 3: Miss rates versus cache size and associativity. Miss rate data assumes 32 processors with 64 byte line sizes. Note that for many applications the, difference in miss rate from 1-way to 2-way associativity is much larger than the difference from 2-way to 4-way associativity. WS1 and WS2 refer to the observed working sets as described in Table 2.

All of the applications have large miss rates at 1KB caches, which decrease dramatically by 1MB. For most of the applications (Barnes, Cholesky, FFT, FMM, LU, Radiosity, Volrend, Water-Nsquared and Water-Spatial) the most important working set is encountered quite early (in these cases by 64KB). However, if a working set grows quickly with problem size or number of processors, then there might be situations in practice with much larger problems or machines where it does not fit in a real second-level cache. To see if this is true, we examine the constitution and growth rates of the working sets, which are summarized in Table 2.[2] We see that the *important* working sets in the applications listed above do not grow with increasing numbers 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, and it would not make sense to simulate cache sizes smaller than these working sets. For example, in the LU and Cholesky factorization applications, the important working set is a single block which is always sized to fit in the cache. There are other working sets in all these programs that are larger and that may not fit in a cache in practice, but these usually amount to a processor's entire partition of the data set and turn out not to be very important to performance.

The applications that do have a realistic operating point in practice for which an important working set does not fit in the cache are Ocean, Raytrace, Radix and to a lesser extent FFT. Ocean streams through its partition of many different grids in different phases of the computation, and can incur substantial capacity and conflict misses as problem sizes increase. In Raytrace, unstructured reflections of rays result in large working sets and curves that are not so well defined into knees and flat regions until the asymptote. 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. This results in a working set that is also not sharply defined, and which may or may not fit in the cache. 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 partition of the data set). The most important working set in the SPLASH-2 FFT is proportional to a row of the $\sqrt{n} \times \sqrt{n}$ matrix. If it does not fit in the cache, the row-wise FFTs can be blocked to make the working set fit.

For these four applications we should also examine a cache size that does not accommodate the working sets described above. For Ocean and FFT, we choose a cache size that holds the first working set described in Table 2 (which we expect to fit in the cache in practice) but not the second. In Radix and Raytrace, the working sets are not sharply defined, so we might choose a range of cache sizes between 1KB and 1MB. We compromise by simply choosing a reasonable cache size that yields a relatively high capacity miss rate. A cache size of 8KB works out well in all cases, so for these four applications we shall present results in subsequent sections for both 1MB and 8KB caches.

Figure 3 also shows us how the miss rate, and potentially the desirable cache sizes to examine, change with associativity. In most cases, increasing cache associativity from 1-way (direct-mapped) to 2-way improves miss rates greatly, while increasing from 2-way to 4-way changes them much less. While direct-mapped caches sometimes change the power-of-two cache size needed to hold the working set compared to fully-associative caches, 2-way set associative caches do not for these problem sizes. In general, the impact of associativity on working set size is unpredictable,.

To summarize, the results and discussion in this section clearly show that understanding the relationship between working sets and cache sizes (i) is very important to good experimental methodology, and (ii) requires substantial understanding of how the important working sets in an application's hierarchy depend on the problem and machine parameters.

6 Comm-to-Comp Ratio and Traffic

In this section, we examine the communication-to-computation ratio and the traffic characteristics of applications, using our default cache parameters (4-way set associative with 64-byte lines) and the representative cache sizes that we identified for the default data-set size in the previous section (1MB in all cases and 8KB for four cas-

Code	Working Set 1	Growth Rate of Working Set 1	Impo-rtant?	Fits in Cache?	Working Set 2	Growth Rate of Working Set 2	Impo-rtant?	Fits in Cache?
Barnes	Tree data for 1 body	$\log DS$	Yes	Yes	partition of DS	DS/P	No	Maybe
Cholesky	One block	Fixed	Yes	Yes	partition of DS	DS/P	No	Maybe
FFT	One row of matrix	$\sqrt{DS}$	Yes	Yes	partition of DS	DS/P	Maybe	Maybe
FMM	Expansion terms	Fixed for n, P	Yes	Yes	partition of DS	DS/P	No	Maybe
LU	One block	Fixed	Yes	Yes	partition of DS	DS/P	No	Maybe
Ocean	A few subrows	$\sqrt{P}/\sqrt{DS}$	Yes	Yes	partition of DS	DS/P	Yes	Maybe
Radiosity	BSP tree	$\log(polygons)$	Yes	Yes	Unstructured	Unstructured	No	Maybe
Radix	Histogram	Radix r	Yes	Yes	partition of DS	DS/P	Yes	Maybe
Raytrace	Unstructured	Unstructured	Yes	Yes	Unstructured	Unstructured	Yes	Maybe
Volrend	Octree, part of ray	$K\log DS +$ $k\sqrt[3]{DS}$	Yes	Yes	partition of DS	approx DS/P	No	Maybe
Water-Nsq	Private data	Fixed	Yes	Yes	partition of DS	DS	No	Maybe
Water-Sp	Private data	Fixed	Yes	Yes	partition of DS	DS/P	No	Maybe

Table 2. Important working sets and their growth rates for the SPLASH-2 suite. DS represents the data set size, P represents the number of processors, and K and k represent large and small constants, respectively.

[2] The scaling expressions assume fully-associative caches and a one-word line size, since it is difficult to analyze artifacts of associativity and line size. This also makes it difficult to use analytic scaling models to predict working sets exactly with finite-associativity caches. Users are advised to use the base results and the working set growth rates we provide as guides to determine for themselves where the working sets fall for the problem and machine sizes they choose.

es). This approach is useful for characterizing the traffic for realistic cache parameters but not necessarily useful for characterizing the inherent communication in the algorithm itself, since the traffic includes artifacts of those cache parameters. In fact, we use *true sharing traffic*, which is the data traffic due to true sharing misses, as an approximation of the inherent communication. A true sharing miss is defined so that it is independent of finite capacity, finite associativity, and false sharing effects (see Section 7). The difference between the true sharing traffic and the inherent communication is that the true sharing traffic includes unnecessary traffic that occurs when there is not enough spatial locality to use the entire 64-byte communicated line. As we will discuss in Section 7, our applications generally have good spatial locality up to 64-byte lines, so that the true sharing traffic is a good approximation of the inherent communication.

We use two metrics to evaluate the different types of traffic needed by the programs. For programs that perform large amounts of 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. For programs that mostly perform integer computation, we report bytes per instruction executed. We break traffic down into three major categories: (i) *remote data*, which is the traffic caused by writebacks and all data transferred between nodes to satisfy processor requests, (ii) *remote overhead*, which is the traffic associated with remote data-request messages, invalidations, acknowledgments, replacement hints, and headers for remote data transfers, and (iii) *local data*, which is the amount of data transmitted by processor requests and writebacks to local memory. Remote data can be broken down further into four subcategories: remote shared, remote cold, remote capacity, and remote writeback. The first three subcategories are a decomposition of remote traffic excluding writebacks, by the cache miss type (*remote shared* consists of traffic due to remote true and false sharing). Figure 4 shows the traffic broken down into these categories as well as the true sharing traffic[3] for 1 to 64 processor runs using the default input data-set sizes and 1MB caches. In all cases, headers for data packets, and all other overhead packets are assumed to be 8 bytes long.

Traffic with 32 processors: Let us first examine the traffic for a fixed number of processors by focusing our attention on the second bar from the right for each application in Figure 4. For all integer applications other than Radix, the remote traffic (the bottom five sections of each bar) is less than 0.1 bytes per instruction. With processors executing at 200 MIPS, this translates into less than 20MB/sec of traffic per processor. In the absence of contention effects, this is well under the per-processor network bandwidths found on shared memory multiprocessors today. For Radix, the remote traffic for processors executing at 200 MIPS approaches 90MB/sec per processor. This is quite high, and the traffic is in fact bursty as well, so evaluation studies using Radix should model both memory contention and network bandwidth limitations to provide accurate results. Figure 4 also shows that the overhead traffic is moderate for 64-byte cache lines (we study the impact of larger cache line sizes in Section 7). The amount of local traffic is usually small as well, since the 1MB caches hold the important working sets and keep capacity misses low (a decomposition of misses by type is shown in Figure 7).

For the eight floating-point intensive applications, the remote traffic is again typically quite small with 1MB caches. For Cholesky, with processors executing at 200 MFLOPS, the required bandwidth is about 68MB/sec per processor, not unreasonable for networks found on multiprocessors today. (We also saw in Section 4 that Cholesky is dominated by load imbalance for this problem size, which further reduces bandwidth requirements.) The exception is FFT, in which the remote bandwidth requirement is close to 124MB/

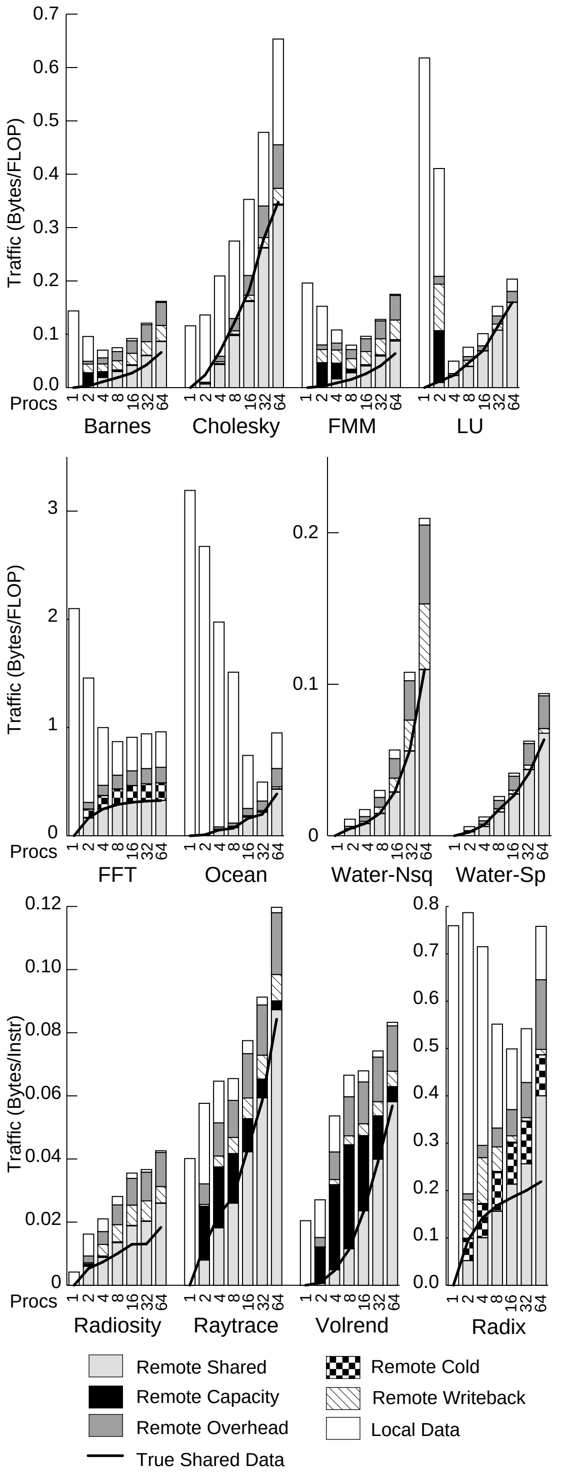

Figure 4: Breakdown of traffic generated in bytes per instruction and bytes per FLOP. The results are shown for 1 to 64 processors. The graphs assume 1MB, 4-way associative, 64-byte line caches. All overhead packets and data headers are assumed to be 8 bytes long.

[3] True sharing *misses* do not generate remote traffic if a processor requests a locally allocated cache block that has been written back after being modified by a remote processor. The true sharing *traffic* shown in Figure 4 consists of local and remote traffic due to true sharing misses.

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

Table 3. Growth rates of the communication to computation ratio for the SPLASH-2 suite. *DS* represents the data set size, and *P* represents the number of processors.

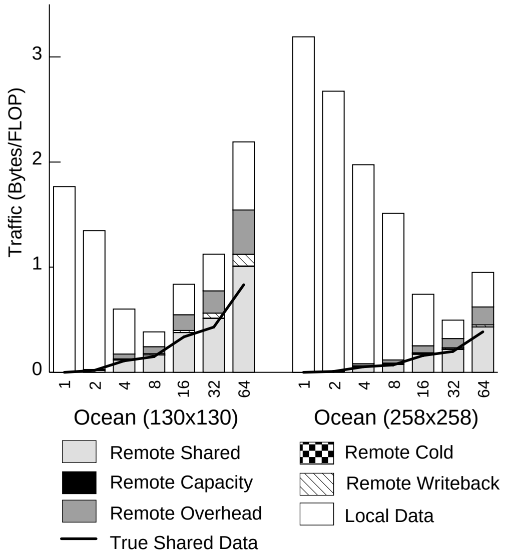

Figure 5: Breakdown of traffic for Ocean in bytes per floating point operation for two problem sizes. The graph assumes 1MB, 4-way associative, 64-byte line caches.

sec per node. Communication in FFT is also bursty, so like Radix, studies using FFT should be careful about modeling network and memory system bandwidth and contention.

Scaling with number of processors and data set size: Let us now look at the effects on traffic of changing the number of processors while keeping the problem size fixed (Figure 4). Typically, the communication-to-computation ratio increases with the number of processors due to a finer-grained decomposition of the problem. For example, in applications that perform localized communication in a two-dimensional physical domain, the amount of computation is often proportional to the area of a partition while the amount of communication is proportional to the perimeter. 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. DS in the table represents the data set size, and P is the number of processors. While FFT and Radix have high communication to computation ratios, their growth rates with P small, particularly at large values of P.

Let us first look at the results with 1MB caches in Figure 4. While true and false sharing traffic (and hence remote sharing traffic in the figure) increase with P, capacity-related traffic may decrease: A processor accesses less data, so more of these data may fit in its cache. For example, a working set that grows as DS/P may not fit in the cache for small P, but may fit for large P. This capacity effect reduces local traffic in almost all our applications as P increases, and can even reduce remote traffic due to capacity misses to nonlocal data (see *remote capacity* traffic for Raytrace and particularly Volrend, where the working set close to 1MB for the default data set with 32 processors is not completely unimportant, and does not fit in the 1MB cache with smaller P). How significant the change in capacity related traffic is depends on the importance of the working set that scales with P.

The impact of increasing data set size is usually just the opposite of that of increasing P: Sharing traffic decreases, while capacity-related traffic (local or remote) may increase. For example, Figure 5 shows the effect of using two different data set sizes in Ocean. The change in overall traffic, as well as in the total communication (remote) traffic, depends on how the different components scale.

To examine the representative cases where an important working set does not fit in the cache, we also present the traffic with 8KB caches for the four applications discussed in Section 5. The results are shown in Figure 6. The total traffic (including local) is of course much larger than for the 1MB caches. The increased capacity-related traffic may be local (as in Ocean and FFT) or cause communication (as in Raytrace). It is therefore more important to model contention when the working set does not fit in the cache.

Overall, the above results reaffirm the importance of understanding the interplay among problem size, number of processors and working set sizes for an application when using it in architectural studies.

7 Spatial Locality and False Sharing

The last set of characteristics we examine are those related to the use of multiword cache lines: spatial locality and false sharing. Programs with good spatial locality perform well with long cache lines due to prefetching effects. Those with poor spatial locality do better with shorter cache lines, because they avoid fetching unnecessary

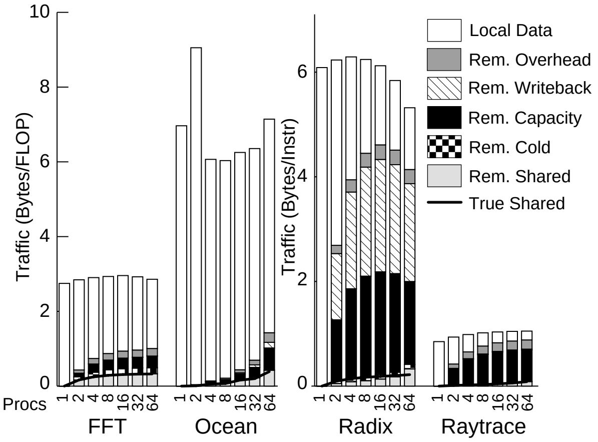

Figure 6: Breakdown of traffic generated for 1-64 processors using 8KB, 4-way associative, 64-byte line caches. All overhead packets and data headers are assumed to be 8 bytes long. Traffic is show in bytes/FLOP for FFT and Ocean and in bytes/instruction for the others.

data and undergoing more capacity misses due to fragmentation. On parallel machines, long cache lines can also be detrimental if they are used as the units of coherence (which we assume), since a program may then exhibit false sharing [EgK89]. While perfect spatial locality implies no false sharing, a program with quite good spatial locality in each processor's reference stream (e.g. a processor writes every other element of a contiguous array, thus having 50% locality) can suffer greatly from false sharing (e.g. another processor may write the intervening elements of the array at the same time). In this section, we characterize the behavior of the SPLASH-2 applications as a function of cache line size, looking both at miss rates (which translate to latency) and traffic (which translates to bandwidth). We explain results with respect to the data structures and access patterns of the applications, classify the applications in this regard, and discuss how the behavior changes with data set size and the number of processors. Methodologically, the characterization shows how the interaction with line size depends on these parameters and that one should therefore be aware of this dependence when performing architectural studies. This characterization also tells us for which applications we can predict the effects of line size and for which ones we must perform sensitivity analysis along this dimension.

If an application has perfect spatial locality, a *k*-fold increase in line size will reduce the miss-rate *k*-fold, while keeping total data traffic constant. In this case, there is no false sharing, and overall traffic will decrease since the relative impact of header overhead decreases as line size increases. To understand why an application falls short of this perfect interaction, and to obtain insights into program behavior, the misses incurred by an application are divided into four broad components (i) cold misses, (ii) capacity (replacement) misses, (iii) true sharing misses, and (iv) false sharing misses. We use the classification scheme presented by Dubois et al. [DSR+93], which we have extended to account for the effects of finite cache capacity.

In our classification, a miss is a true sharing miss if, during the lifetime of the line in the cache, the processor accesses a value that was written by a different processor since either (i) the last true sharing miss by the same processor to the same line, or (ii) the beginning of the execution if there is no such previous true sharing miss. Thus, a miss to a line that was replaced, but which would have incurred a sharing miss (true or false) had it not been replaced is still classified as a sharing miss. A miss is a false sharing miss if the line has been modified since the last time it was in the processor's cache (or the beginning of the execution if this is the first time in the processor's cache), but the processor does not use any of the newly defined values. All other misses are either cold misses (if the line has never been in this processor's cache before) or capacity misses (all others). This definition captures the true communication inherent in the application independent of cache size, and also recognizes the benefits of long cache lines in capturing required communication.

Following our cache size methodology, we present results for all applications with 1MB 4-way set associative caches (which fit the important working sets) and for four of the applications with 8KB caches as well. Here we encounter an interesting methodological question, since the cache size needed to hold a working set may depend on the line size if spatial locality is not perfect. Particularly for applications with poor spatial locality, then, it is important to ensure that as the line size changes, the cache sizes chosen still represent the same operating point with regard to fitting a working set. Figure 7 shows the breakdown of miss rate for the applications as line size is varied from 8 to 256 bytes with 1MB caches, and Figure 8 with 8KB caches for the relevant applications.[4] Besides Volrend, none of the applications start to incur many more capacity misses with 1MB caches as the line size changes. This is because the operating point does not shift to the other side of an important knee in the working set curve as the line size changes. The same is true for the 8KB caches.

We focus first on the results for 1MB caches, which are shown in Figure 7. In this case, all important working sets fit in the cache, so sharing and cold misses are most prominent. Figure 7 shows that the impact of long cache lines varies greatly across applications. Some applications, like LU, almost halve the miss ratio with every doubling of line size in the range we study, while others like Radiosity don't improve much. Still others, like FMM, improve early on but then become worse. We first examine the applications in a few different categories according to the interaction of their data structures and access patterns with respect to long cache lines. During this process, we raise some important points that architects should keep in mind when performing evaluations that depend on line size effects. Finally, we summarize these observations at the end of this section.

The first class of applications consists of those whose data access patterns are regular, and whose data structures are organized so the access patterns use good stride through contiguously allocated data structures. These include LU, Cholesky, FFT, and Ocean. LU

[4] The bars in these figures contain a new category called Upgrades. These are writes that find the memory block in cache but in shared state, and have to send an "upgrade" request for ownership. We put upgrades in white at the top of the bars, so readers can easily ignore them visually if necessary; we do not discuss them with miss rates, but they are used to compute traffic later.

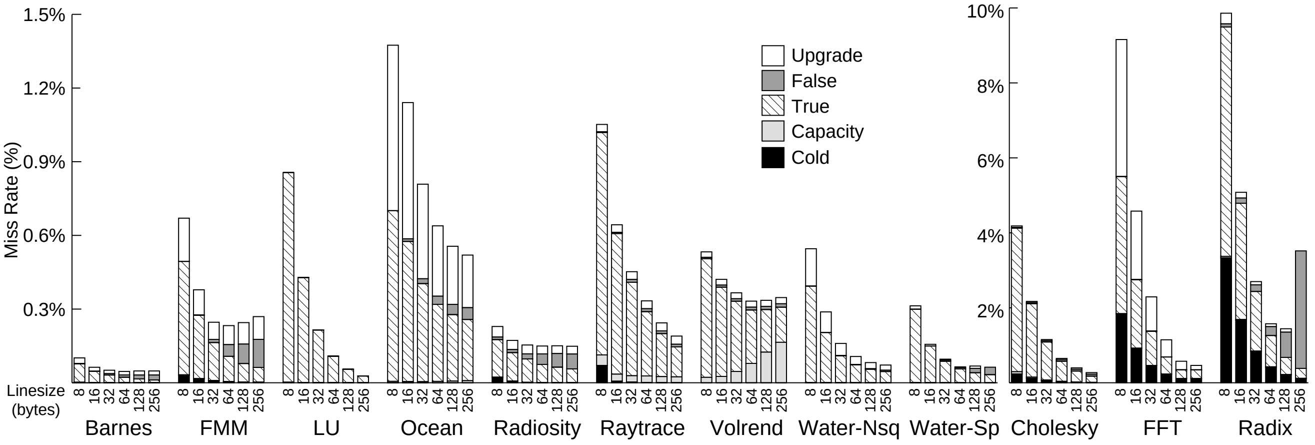

Figure 7: Breakdown of miss rate versus cache line size. The 4-way associative caches have capacities fixed at 1MB, and line sizes that are varied from 8 to 256 bytes.

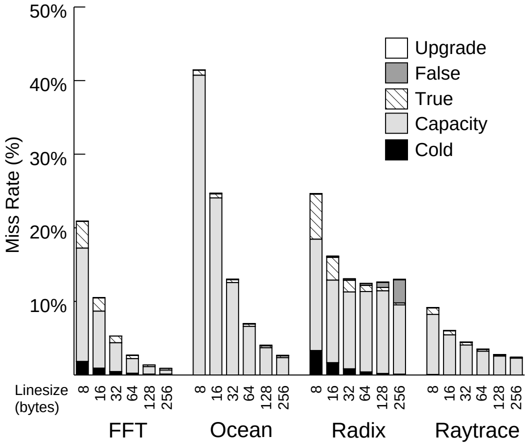

Figure 8: Breakdown of miss rate versus cache line size with 8KB, 4-way associative caches for applications with important working.

and Cholesky are blocked matrix codes that use data structures such that a block is contiguous in the address space. FFT maintains good stride during the row-wise FFT computations, and uses a blocked submatrix transpose to ensure good cache line utilization. In Ocean, processors stream through grid partitions which are allocated contiguously and locally. The result is that in LU the miss rate drops almost linearly with increasing line size. In Cholesky, both the cold and true sharing misses (but not the few false sharing misses) fall almost linearly as well. In FFT, the cold and true sharing misses fall almost linearly until 256 bytes (this is explained shortly). In Ocean, although the access patterns have very good spatial locality, the miss rate with 1MB caches does not fall linearly with increasing line size. The reason is that the best spatial locality is obtained on the references within a processor's own partition. While these are the majority of the references, they do not cause misses in a 1MB cache for this problem size, so only sharing misses are observed. Thus, we would expect to see a larger influence with long cache lines when a processor's local partition(s) does not fit in the cache, as shown in Figure 8. As for sharing misses, they occur in Ocean when a processor tries to access the elements in the nearest-neighbor subrows and subcolumns of its adjacent partitions. The accesses to the neighboring subrows have unit stride and good locality, while those to the subcolumns have non-unit stride and no spatial locality.

These results bring us to an important methodological point. In many applications the impact of long cache lines depends on application parameters (particularly data set size) and the number of processors. We have seen an example of this for Ocean, where whether a processor's partition fits in the cache depends on these parameters. Another example is provided by FFT. With our default 64K-point FFT (256×256 matrix of complex doubles) and 32 processors, a processor reads a submatrix of 8×8 points from every other processor during the transpose phase. Since each element is 16 bytes, a subrow of the submatrix is 128 bytes. Up to 128 byte lines, we see very good spatial locality for this problem size in Figure 7. But going from 128 to 256 byte lines does not reduce either the cold miss rate (for the first transpose) or the sharing miss rate (next two transposes), although it does reduce the number of upgrades required.

Finally, a much more dramatic example is provided by Radix. In the permutation phase of this program, a processor reads keys contiguously from its partition in one array and writes them in scattered form (based on histogram values) to another array. The pattern of writes by processors to the second array is such that on average, writes by different processors are interleaved in the array at a granularity of $n/(r \times p)$ keys, where n, r, and p are the number of keys being sorted, the radix, and the number of processors, respectively. While the exact pattern is dependent on the distribution of keys, whether or not we have substantial false sharing clearly depends on how $n/(r \times p)$ compares with the cache line size. We therefore see the sharing miss rate drop with line size, until this ratio is less than a line. At this point, the true sharing miss rate continues to drop while the false sharing miss rate rises dramatically, making large cache lines hurt performance.

The next class of applications are those whose data structures are records representing independent logical program units (e.g. molecules), and in which the fields of these records are accessed differently in different phases of computation. Examples are Water-Nsquared, Water-Spatial, Barnes-Hut and FMM. A processor in these applications may only read certain fields of particles not owned by it that are written by other processors. If these fields do not constitute an integer multiple of the cache line size, then sharing misses will not have perfect spatial locality. We see this in all these applications beyond about a 64-byte line size, with the Water programs having better spatial locality than Barnes and FMM. Also, if these fields are located on the same cache line with other fields of a particle that is owned and updated by another processor—in either the same or another phase of computation—then we will see false sharing with long cache lines. This effect is seen in Barnes and FMM. In both these programs, true sharing misses continue to drop with larger lines (though not linearly), and false sharing misses start to grow and outweigh the true-sharing reduction by about 128-byte lines. If cache lines are larger than a single record, false sharing across records may result. This is more likely in Water-Nsquared than in Barnes or FMM, since in the former a processor's particles are contiguous in the array of records while in the latter the assignment of particles to processors changes dynamically so that a processor's particles usually are not contiguous.

The third class of applications has highly unstructured access patterns to irregular data structures. The graphics programs Raytrace, Volrend and Radiosity fall into this class. In Radiosity the main data structures are written as well as read. However, access patterns are unstructured, making it is difficult to analyze or predict the impact of line size and how it changes with problem size and the number of processors. Our limited experiments (not shown) indicate that the spatial locality of true shared data does not change its locality patterns much, while the relative impact of false sharing increases with the number of processors. However, the miss rate is low enough that the false sharing may not matter very much.

Raytrace and Volrend incur little false sharing, but also have mediocre spatial locality. False sharing is small because the main data structures are read-only. The primary sharing happens at the image plane, which has relatively few accesses. The reason for poor spatial locality is that the access patterns to the read-only data are highly unstructured, and the processor that touches one small field of a voxel or polygon may be different than the one that touches the next field. Volrend is the one example in which the capacity miss rate increases with line size even in 1MB caches, due to increased fragmentation and cache conflicts. Methodologically, this indicates that, especially for applications like Volrend, working set issues should in fact be re-evaluated with the larger line sizes. As for scaling, as problem size is increased (most likely in the form of more polygons/voxels) the primary effect is a larger capacity miss rate. The spatial locality on scene data does not change much, though it improves on the image since the image becomes larger. The opposite effect is obtained by reducing the problem size or increasing the number of processors.

Figure 8 shows the results for FFT, Ocean, Radix, and Raytrace for 8KB caches, an operating point where an important working set does not fit in the cache. As expected, the overall miss rates are higher since capacity misses increase substantially. The spatial locality

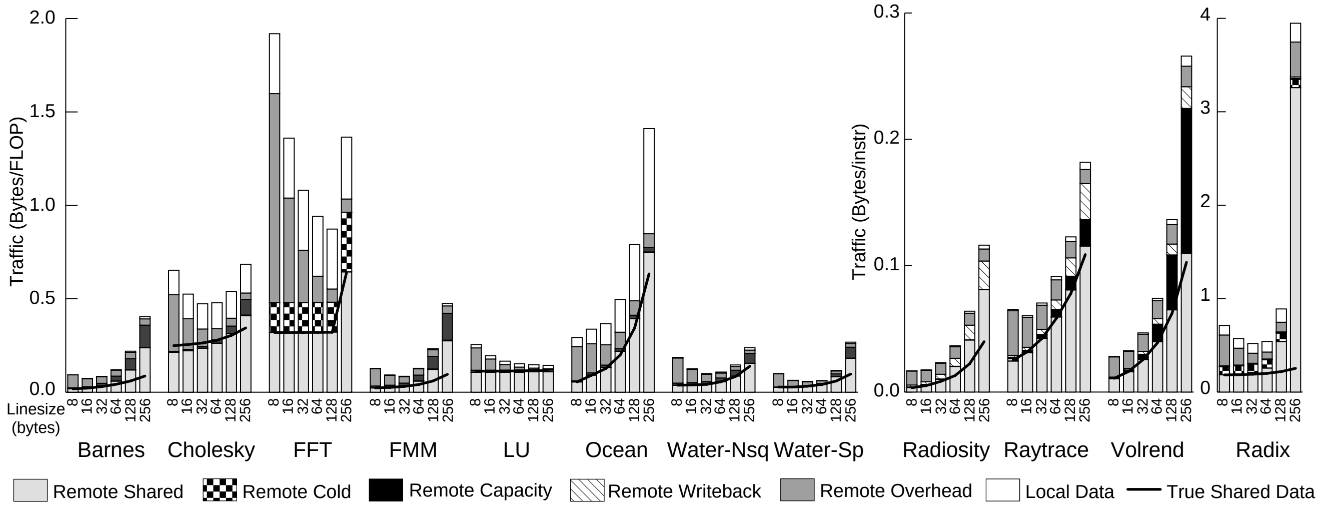

Figure 9: Breakdown of traffic versus line size for 1MB, 4-way associative caches.

and false sharing trends do not change significantly as compared to results for 1MB caches, since these properties are fundamental to the data structures and access patterns of the 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 much better spatial locality than true sharing misses in the 1MB cache case. These capacity misses dominate, so the overall effect of long cache lines is much more positive.

Finally, Figure 9 indicates the impact of cache line size on network traffic. Remote (network) traffic can be seen from the results in Figure 9 by ignoring the top component (local traffic) of each bar. Figure 9 shows the traffic in bytes per instruction as the line size is varied. There are three points to observe from this graph. First, although the traffic decreases with line size for LU and FFT (miss rate decreases linearly, and per-miss overhead is amortized over larger lines), for most of the applications the overall network traffic increases substantially as line sizes become larger. Thus, bandwidth assumptions for a machine may have to be re-examined as line size changes. Whether the reduction in miss rate is more important than the increased traffic depends on the latency and bandwidth provided by the machine, and on how much latency can be hidden. Second, the overall network traffic requirements for the SPLASH-2 suite are still small even for large line sizes, with the exception of Radix. The 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 the line size is increased, the total traffic is usually a minimum at between 32 and 128 bytes. Our results reconfirm previous studies such as [GuW92], which shows that the overall network traffic in a distributed shared address space multiprocessor is usually a minimum for cache line sizes of 32 bytes.

To summarize, in addition to showing which programs have high bandwidth requirements with long cache lines, which have good spatial locality and which incur false sharing, the above characterization emphasizes some methodological points:

- It is important that users of the SPLASH-2 suite understand the behavior of individual applications when choosing a line size for their studies. Line size effects can be predicted in some programs that have excellent spatial locality, but not in others that do not (e.g. Volrend and Raytrace) or that can have significant false sharing relative to other types of misses (e.g., Radix, FMM, Barnes, Radiosity).
- The spatial locality and false sharing in a program often depend on problem size, number of processors, and whether working sets fit in the cache (capacity misses may have different spatial locality than sharing misses). These effects must be understood, and it is not sufficient to evaluate the effects of spatial locality with a single set of these parameters.
- While there is a thresholding effect in the relationship among line size, problem size, and number of processors, in many cases (where this relationship is acute), line size is not as easy a parameter to prune in architectural studies as cache size.

8 Concluding Remarks

The SPLASH-2 application suite is designed to provide parallel programs for the evaluation of architectural ideas and tradeoffs. However, performing such evaluation well is a difficult task owing to the large number of interacting degrees of freedom. For example, many memory system parameters such as cache size, associativity, and line size can both quantitatively and qualitatively impact the results of a study, as can application parameters and the number of processors used. It is extremely time consuming to perform complete sensitivity analyses on all these parameters. Since evaluation is often done through simulation which is expensive, we are forced to use smaller problem and machine sizes than we would really like to evaluate. Finally, many combinations of application and machine parameters that we might choose to evaluate might not be representative of realistic usage of the programs, so blind sweeps through the space may not be appropriate. To use these programs effectively for architectural evaluation, it is therefore very important that we understand their relevant characteristics well, particularly with regard to determining what are realistic and unrealistic regions in the parameter space, and how these regions change as important parameters are scaled.

In this paper, we have tried to provide the necessary understanding of the SPLASH-2 programs, as well as some methodological guidelines for their use. We have characterized the programs along several important behavioral axes, and described how the characteristics scale with key application and machine parameters. These axes are concurrency and load balancing, working-set sizes, communication-to-computation ratio and traffic, and spatial locality. Our hope is that this characterization will allow people to understand the necessary growth rates, decide where the effects of changing certain parameters can be predicted and where they must be determined experimentally, and prune the design space by avoiding unrealistic and redundant operating points. We have provided some specific guidelines for pruning the space, for example looking for knees and flat regions in characteristic curves where these can be found (e.g. working sets in the miss rate versus cache size curve, as well as knees in curves for bandwidth and associativity), understanding how the parameter values where knees occur scale, and using this understanding to prune entire regions when possible. Of course, we must be careful in our pruning and ensure that the characteristic that displays the knees is the only one we care about with regard to that parameter.

9 Acknowledgments

This work was supported under ARPA Contract Number DABT63-94-C-0054. Steven Woo is supported by a Hughes Aircraft Company Doctoral Student Fellowship. Moriyoshi Ohara is supported by the Overseas Scholarship Program of IBM Japan, Ltd. The authors gratefully acknowledge these funding sources.

References

[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.

[EgK89] Susan J. Eggers and Randy H. Katz. The Effects of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pp. 257-270, April 1989.

[FoW78] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Tenth ACM Symposium on Theory of Computing*, May 1978.

[Gol93] Stephen Goldschmidt. Simulation of Multiprocessors: Accuracy and Performance. Ph.D. Thesis, 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.

[HHS+95] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Stanford University Technical Report No. CSL-TR-95-660. January 1995.

[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 Seventh SIAM International Conference on Parallel Processing for Scientific Computing*, pp. 313-318, Feb 1995**.**

[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.

[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.

[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.

[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.

[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.

[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.