Compiler-Directed Page Coloring for Multiprocessors

Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry*,
Mendel Rosenblum and Monica S. Lam

Computer Systems Laboratory
Stanford University,
Stanford, CA 94305, USA
http://www-flash.stanford.edu/SimOS
http://suif.stanford.edu

* Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada M5S 3G4
http://www.eecg.toronto.edu/~tcm

Abstract

This paper presents a new technique, *compiler-directed page coloring*, that eliminates conflict misses in multiprocessor applications. It enables applications to make better use of the increased aggregate cache size available in a multiprocessor. This technique uses the compiler's knowledge of the access patterns of the parallelized applications to direct the operating system's virtual memory page mapping strategy. We demonstrate that this technique can lead to significant performance improvements over two commonly used page mapping strategies for machines with either direct-mapped or two-way set-associative caches. We also show that it is complementary to latency-hiding techniques such as prefetching.

We implemented compiler-directed page coloring in the SUIF parallelizing compiler and on two commercial operating systems. We applied the technique to the SPEC95fp benchmark suite, a representative set of numeric programs. We used the SimOS machine simulator to analyze the applications and isolate their performance bottlenecks. We also validated these results on a real machine, an eight-processor 350MHz Digital AlphaServer. Compiler-directed page coloring leads to significant performance improvements for several applications. Overall, our technique improves the SPEC95fp rating for eight processors by 8% over Digital UNIX's page mapping policy and by 20% over a page coloring, a standard page mapping policy. The SUIF compiler achieves a SPEC95fp ratio of 57.4, the highest ratio to date.

1 Introduction

Recent advances in parallelizing compilers have made it possible to automatically generate parallel programs from sequential numeric applications. This technology has the promise of making parallel processing accessible to a much broader range of users. However, the performance of the parallel codes is highly sensitive to its memory subsystem behavior.

We used the SimOS machine simulation environment [21] to study the performance of the SPEC95fp benchmark suite [22] parallelized by the SUIF compiler [24]. A close look at the behavior of the applications shows that the compiler is good at making use of the additional processors and at eliminating unnecessary communication between processors. However, the parallelized codes are not taking advantage of the increase in aggregate cache size available to the parallel computation.

Unlike sequential applications, where a single processor accesses all of the data used by the application, each processor involved in a parallel computation typically accesses only a subset of the data. This results in sparse memory access patterns that are not normally found in sequential applications. The operating system's page mapping policy determines the location of an application's memory pages in physically indexed caches, such as the large external cache of current processors. The page mapping policies of current operating systems were developed for sequential applications, and not for the sparse access patterns found in parallel programs. As a result, some regions of the processors' external caches go unused while other regions are over-utilized.

To improve cache utilization and eliminate cache conflicts in parallelized code, we have developed a new technique, *compiler-directed page coloring (CDPC),* that involves the cooperation of the compiler and the operating system. CDPC is fully automatic as it uses information available within the compiler to predict the access patterns of a compiler-parallelized application. This information is used to customize the application's page mapping strategy. The suggested page mapping is then treated as a hint by the operating system [17].

We show the importance of page mapping policies to the performance of a set of compiler-parallelized workloads. We then compare our compiler-directed page coloring technique with two existing page mapping policies used by commercial operating systems. We show that neither existing page mapping policy dominates the other. However, our technique consistently outperforms both policies and can lead to significant performance improvements, as much as a factor of two in some cases.

The rest of this paper is organized as follows. In Section 2, we provide background information on existing operating system and compiler techniques aimed at improving memory system performance. After describing our experimental setup in Section 3, we analyze the base performance of the workloads in Section 4, and show that memory subsystem performance is their primary bottleneck. In Section 5, we describe the implementation of compiler-directed page coloring in the SUIF compiler and in two existing operating systems. In Section 6, we use SimOS to show the impact of compiler-directed page coloring and a complementary latency hiding technique, compiler-inserted

To appear in the *Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems, (ASPLOS VII),* October, 1996.

prefetching. Finally, we validate our results in Section 7 by comparing the performance of compiler-directed page coloring with two existing page mapping policies on a Digital AlphaServer 8400 multiprocessor.

2 Background

A large amount of research has been performed on hardware- and software-based techniques for improving memory subsystem performance. In this section, we briefly discuss existing techniques that are performed by the operating system and the compiler.

2.1 Operating System Techniques

Operating systems generally group the pages of physical memory into *colors,* where two pages have the same color if they map to the same location in a physically-indexed cache. Hence, cache conflicts only occur between pages of the same color. The number of colors is the cache size divided by the product of the page size and the associativity. For example, in a system with a 1MB cache and 4KB page size, there are 256 colors if the cache is direct-mapped, and 128 if the cache is two-way set-associative. When a page fault occurs, the operating system must allocate a physical page for a given address in the application's virtual address space. The operating system uses a page mapping policy to determine a preferred color and attempts to allocate a page of that color.

Most current operating systems support one of two page mapping policies, *page coloring* and *bin hopping* [16]. Page coloring maps consecutive virtual pages to consecutive colors. Page coloring exploits spatial locality by ensuring that conflicts only occur between pages whose virtual addresses differ by a multiple of the cache set size. Current operating systems that use page coloring include IRIX (SGI's version of UNIX) and Windows NT. A bin hopping strategy cycles through the different colors to allocate pages in the order page faults occur. Bin hopping exploits temporal locality as conflicts never occur between pages that were first touched close in time. Digital UNIX (formerly DEC OSF/1) implements a bin hopping policy. Bin hopping results in a non-deterministic coloring decision if multiple processors involved in a parallel computation suffer concurrent page faults. While such concurrent accesses are both legal and common, they result in a race in the kernel to determine the color of each page and can lead to unpredictable performance.

Both page coloring and bin hopping policies are *static policies* in that they do not change the color of a page after a fault. Recently *dynamic policies* have also been proposed that recolor a page by copying its contents to a newly allocated page of a different color and simultaneously changing the virtual-to-physical mapping of the page. The operating system uses either some custom hardware in the form of a cache-miss lookaside buffer [4] or a combination of TLB state information and cache miss counters to detect conflicts [20]. When such a conflict is detected, one of the pages involved in the conflict is recolored.

To our knowledge, the performance of dynamic policies for multiprocessors has not been studied. However, detecting conflict misses in a multiprocessor is harder than on a uniprocessor as one must be able to differentiate conflict misses from coherence misses. The overheads of a recoloring operation are also likely to be significantly larger than on uniprocessors [23]. The TLB state of each processor must be individually flushed and the recoloring operation may generate significant inter-processor communication.

The page mapping policies, and more generally the memory management policies, of an operating system can lead to poor performance for applications with non-standard requirements. To overcome these limitations, operating systems such as V++ [13] and Exokernel [10] allow sophisticated applications to manage their allocated portions of physical memory. These applications can implement their own customized page replacement and page mapping policies. In contrast to these approaches, CDPC is fully automatic and does not require applications to manipulate physical addresses directly. As a result, our technique can be easily and safely integrated in existing operating systems while providing the necessary flexibility in page mapping decisions.

2.2 Compiler Techniques

Compiler-based techniques for improving memory subsystem performance fall broadly into two categories: those that attempt to *reduce* the number of cache misses, and those that attempt to *tolerate* miss latency. It is preferable to reduce memory latency before tolerating it, since this reduces the demand for memory bandwidth that is often a precious resource in multiprocessors.

The first category includes loop transformations such as blocking that reorder the computation to enhance data locality [7,26]. Recently, there has also been research to minimize communication between processors by clever partitioning of the computation across processors [3,11,15]. Transformations that make data elements accessed by the same processor contiguous in the shared address space have been shown to be useful for enhancing spatial locality and minimizing false sharing [2]. These transformations make the data accessed by each processor contiguous within an individual data structure. Techniques that merge data structures in the virtual address space have also been proposed to make data accessed by each processor contiguous across data structures [14]. However, such transformations are not applicable to all programs. In contrast, compiler-directed page coloring achieves a similar effect but is completely transparent to the application.

Padding [5] is a simple, commonly-used technique for eliminating cache conflicts. With padding, the compiler or loader offsets the starting locations of data locations and increases the dimensions of arrays, usually by a small number of cache lines, to avoid cache conflicts. While this is a relatively small change, it is not always legal since constructs such as pointer arithmetic in C and common blocks in FORTRAN expose the data layout to the programmer. Padding cannot be legally applied unless every single access in the whole program has been successfully analyzed and updated to reflect the new data layout. Finally, padding is constrained by the fact that it operates on the virtual address space and not on the physical address space. For example, pads that are larger than a page size are ineffective if the operating system has a bin hopping policy for page mapping.

To tolerate memory latency, the compiler can insert prefetch instructions to move data into the cache before it is needed, thus overlapping memory accesses with computation [6,18]. Prefetch instructions are supported in many recent processors (e.g., the MIPS R10000, the DEC Alpha 21164). Previous studies have demonstrated the effectiveness of automatic compiler-inserted prefetching on sequential and hand-parallelized applications [19]. In this paper, we present the first results where automatic prefetching is combined with automatic parallelization.

3 Experimental Setup

3.1 The Workloads

We used the ten programs of the SPEC95fp benchmark suite for our experiments. The SPEC95fp suite is an industry standard set of programs designed to evaluate the performance of general-purpose processors. These benchmarks are representative of numerical computations and were not designed specifically for parallel systems. We run each application using the full "reference" data

Benchmark	Data set size (MB)
101.tomcatv	14
102.swim	14
103.su2cor	23
104.hydro2d	8
107.mgrid	7
110.applu	31
125.turb3d	24
141.apsi	9
145.fpppp	< 1
146.wave5	40

Table 1. Reference Data Set Sizes of SPEC95fp

set. As we see in Table 1, these data sets are too large to fit into the caches of current high-end workstations, but are small enough to fit in main memory.

3.2 The Compiler

We used the Stanford SUIF research compiler to automatically transform the applications into parallel programs. The compiler translates sequential programs into parallelized source code for shared-memory multiprocessors, which is then compiled with the native compiler of the target machine. The SUIF system includes standard parallelization techniques that form the basis for most of today's commercial parallelizing compilers. It uses data dependence analysis on array data structures to extract parallelism in loop nests. The compiler will recognize reduction operations (e.g. summations and products), convert scalar variables into private copies on each processor, and transform loop nests (e.g. interchanging inner and outer loops) to optimize for both parallelism and locality.

The SUIF compiler also includes several advanced optimizations that have been described and evaluated in [1] and [12]. The compiler performs inter-procedural analysis to parallelize large regions of code that may span multiple procedures. It performs reduction recognition and privatization analysis on arrays. SUIF also contains a suite of locality optimization techniques to make the multiprocessor caches more effective. To minimize inter-processor communication, the compiler partitions the computation across processors so that computations using the same data are co-located on the same processor. The compiler also transforms the data layout of individual arrays so that data elements accessed by each processor are contiguous within the shared address space. This transformation increases spatial locality and reduces the amount of false sharing within data structures.

The applications parallelized by SUIF follow a master/slave model of parallelism, as shown in Figure 1. The master process executes the sequential portions of the program while the slaves wait at a barrier. When the master reaches the start of a parallel region, it notifies the slaves. The slaves and the master then operate in parallel until they reach a barrier at the end of the parallel region. The code within each parallel region is statically scheduled; each processor executes a pre-determined portion of the computation. SUIF includes a feedback mechanism that suppresses the parallelization of loops that result in a net performance loss (e.g. when the amount of work in the loop is smaller than the overhead of the barrier). The feedback mechanism uses the training data sets of the benchmarks to determine which loops should be suppressed.

3.3 Simulation Methodology

The simulation platform is SimOS [21]. We used SimOS to study the performance of the workloads in detail and to analyze the

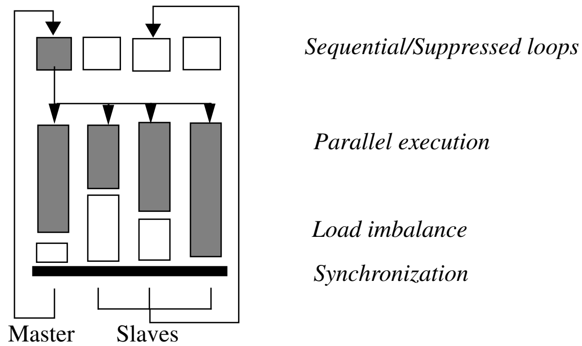

FIGURE 1. Master/Slave Structure of SUIF Applications. The gray regions correspond to useful execution. The white regions correspond to the overheads of parallel execution. The synchronization bar corresponds to the cost of the implementation of the barrier. The execution of the application consists of a sequence of sequential and parallel regions.

impact of various architectural parameters. SimOS models the hardware of an SGI multiprocessor in enough detail to boot and run IRIX 5.3, SGI's version of UNIX. SimOS contains a set of simulators that trade off different simulation speeds against the level of simulation detail.

For our studies, we model 400MHz single-issue R4400 processors with two-way set-associative split instruction/data caches of 32KB each. For the base configuration each processor has a 1MB direct-mapped external cache with 128 byte lines. The split-transaction bus can sustain up to 1.2 GB/s of fetch bandwidth in the absence of writebacks. The minimum latency of a cache miss is 500 nanoseconds when fetched from memory and 750 nanoseconds when fetched from another processor. This configuration is loosely based on the characteristics of current high-performance multiprocessors.

The long execution times of the SPEC95fp benchmarks—on the order of a couple of minutes on today's fastest machines—make it infeasible to simulate the entire benchmark suite to completion in enough detail to model a modern memory hierarchy. We estimate that the simulation time for all of the workloads and configurations presented in this paper would take more than one year.

To limit simulation requirements, previous studies have arbitrarily bounded the number of simulated instructions or used smaller data sets rather than the full data sets. However, the size of the data set has a strong effect on cache performance. To limit simulation time while preserving the characteristics of the program executing the full data set, we developed a technique based on the concept of *representative execution windows*.

A representative execution window is a portion of the execution that summarizes the overall behavior of the application. Representative execution windows are based on the observation that these benchmarks contain a short initialization section that includes I/O requests and pages faults, followed by a steady computation section. Overall, the steady state of the workloads accounts for more than 95% of the sequential execution time. We separate the application into phases that correspond to different paths in the control flow of the program. The steady state consists of a sequence of these phases. Some simple workloads contain a single phase that is repeated over and over again. Others have a more complicated control flow that alternates between phases. For example, the application *turb3d* contains four phases that each occur 11, 66, 100 and 120 times respectively during the steady state of the workload. The representative execution window is a part of the steady state that contains each phase at least once.

We used SimOS's high-speed simulator [25], configured to model a large external cache, to execute the sequential benchmarks to completion. For each benchmark, we analyze the variation in execution behavior between different occurrences of each phase. We found that in all but one case (*wave5*), the standard deviation of both the number of instructions and the miss rate is less than 1% of the mean. One of the phases of *wave5* showed a 4% variation in instructions and a 30% variation in cache misses.

To position the benchmark at its steady state point, we use SimOS's high-speed simulator in a mode where it can execute uniprocessor workloads at one tenth the speed of the underlying machine. We then switch to a more detailed simulator and use non-intrusive annotations to collect statistics separately for each phase. The results from the different phases are then weighted according to the number of occurrences during the steady state. To eliminate transient simulation effects such as cold misses, we discard the results from the first phases executed with the detailed simulator.

4 Performance Analysis of the Workloads

4.1 High-level Characterization

Figure 2 characterizes the performance of the parallelized benchmarks in four different ways. The first graph presents a high-level breakdown of the combined execution time of the benchmarks. The height of each bar is the sum of the execution times over all the processors. Using this metric, applications that show a constant combined execution time have linear speedup. We separate the application-specific code from the *overheads* that correspond to the time spent in the operating system and in the synchronization routines. The application time is further divided between the processing time (*execution*) and its associated *memory stall time*.

We first notice that seven of the ten benchmarks see near constant combined execution times (indicating near linear speedups), at least up to eight processors. The remaining three applications, *apsi*, *fpppp* and *wave5*, show little or no performance benefit from parallelization. Across all applications, we see increases in the parallelization overheads and general increases in the memory stall time, as the number of processors increases, especially with 16 processors.

The second graph of Figure 2 breaks down the overheads into five categories. The kernel overhead corresponds to the time spent in the operating system, primarily servicing TLB faults and processing timer interrupts. The other four categories result from the structure of SUIF applications, as shown in Figure 1:

- The difference in arrival times of the processors at the barrier at the end of the parallel regions is counted as *load imbalance*. The most common cause of load imbalance occurs when the number of iterations of a parallel loop is not a multiple of the number of processors. For example, the parallelized loops of *applu* consist of only 33 iterations. As a result, 16 processors do not execute such loops more efficiently than 11 processors.
- *Sequential time* is the time when the slave processors spin while the master process executes code that could not be parallelized by the compiler. The sequential time increases with the number of processors because more processors are idle. For example, the figure shows that *fpppp* has essentially no loop-level parallelism.
- *Suppressed time* corresponds to time spent by slave processes while the master alone executes a parallelizeable loop. Both *apsi* and *wave5* have fine-grain loop-level parallelism that is suppressed since it cannot be exploited effectively on the multiprocessors available today because of their high synchronization and communication costs.
- The *synchronization* time corresponds to the cost of the software implementation of barriers and locks. Since the parallel execution of the smaller loops is already suppressed by the compiler, this overhead is relatively small.

The third and fourth graphs of Figure 2 present two complementary views of the memory system behavior of the applications. The memory system behavior graph gives a processor-centric view of the performance of the memory system, while the bus utilization graph shows the load on the shared hardware. The memory system behavior is shown in terms of memory cycles per instruction (MCPI). An MCPI of 1.0 means that the application is spending half of its time in the memory subsystem. The MCPI indicates the performance of the processor during useful execution only. The time spent in overheads, for example lost to load imbalance, does not cause many cache misses because the processor is spinning in a tight loop. The graph separates the stall time due to on-chip cache misses from off-chip cache misses and classifies the latter. Our classification separates replacement misses from communication misses. Communication misses are classified according to [8].

The memory system behavior graph shows that the memory stall time of most applications is dominated by replacement misses caused by the limited size (capacity misses) and associativity (conflict misses) of the cache. The graph also shows that the data transformation and scheduling algorithms used by the SUIF compiler are effective in eliminating most cases of true and false sharing communication.

The bus utilization graph shows the bus cycles occupied because of data transfers (requests and replies), writebacks and upgrades from a shared state to an exclusive state. The graph shows that the performance of these parallel applications is often limited by the contention on the memory bus. With 16 processors, the average occupancy of the bus ranges from 50% to over 95% for five of the ten benchmarks. In such cases, the bandwidth requirements of the application exceed the 1.2 GB/s that the bus can offer. This results in a significant increase in the average latency of cache misses, as indicated by the increase in the MCPI of these benchmarks. For example, although the miss rate of *tomcatv* decreases by 3% from one to 16 processors, the MCPI more than doubles.

Finally, *fpppp* is limited entirely by instruction cache misses fetched from the external cache and puts no load on the shared bus.

4.2 Cache Conflicts in Parallel Programs

Figure 2 shows that the primary performance bottleneck of half the benchmarks is the memory traffic due to replacement misses. By comparison, processors communicate significantly less with each other than with main memory.

Parallel applications have an advantage over their sequential counterparts. For a constant problem size, each processor typically operates on a subset of the data. When the data is evenly partitioned between the different processors and processors communicate only a small amount with each other, each processor's data set size is reduced proportionally to the number of processors that take part in the computation.

The workload runs, whose results are shown in Figure 2, used 1MB caches. With 16 processors, the aggregate cache size of 16 MB exceeds the data set size of many of the benchmarks, as listed in Table 1. Contrary to our expectations, we generally do not observe a reduction in the number of replacement misses.

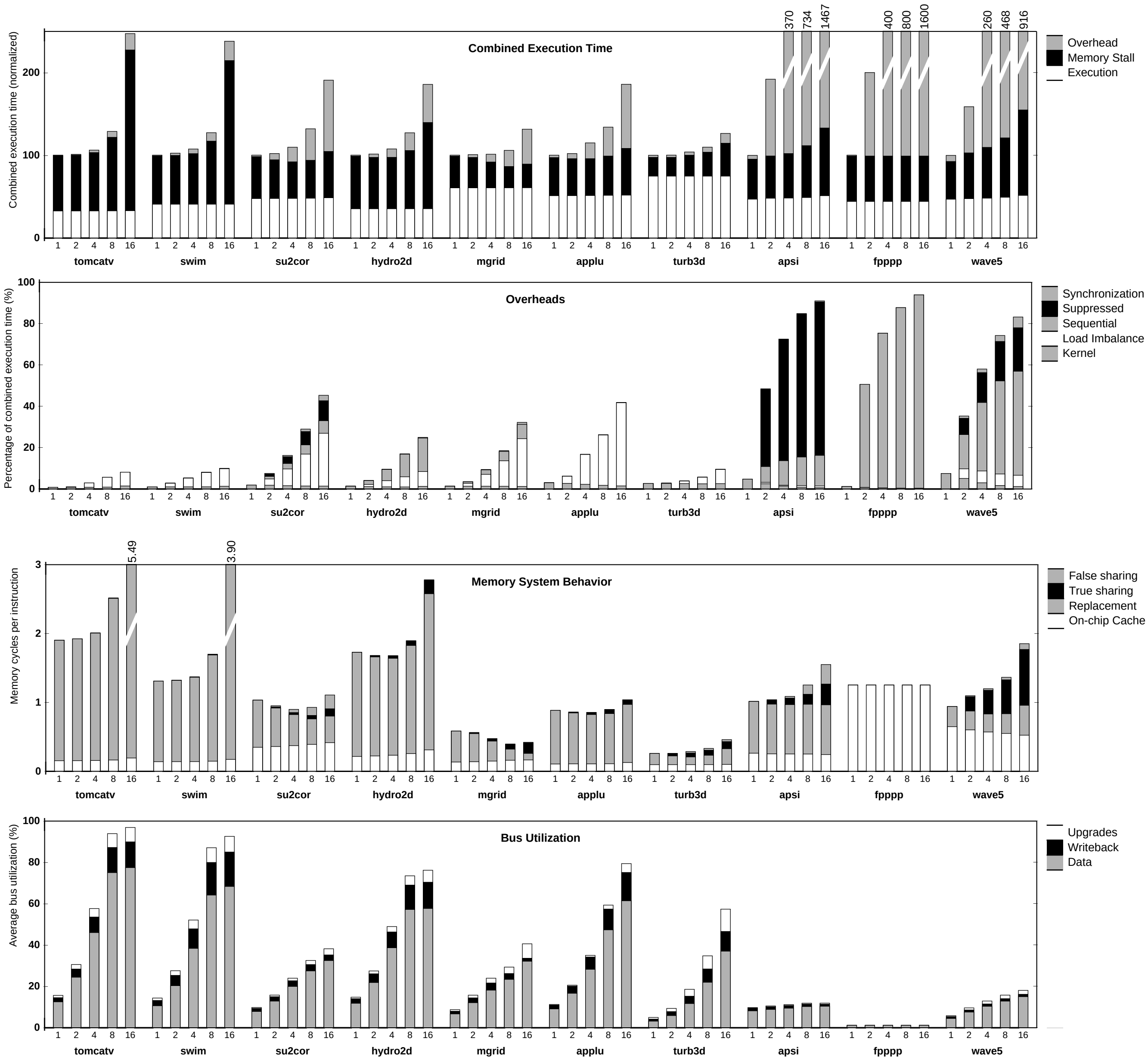

FIGURE 2. High Level Characterization of the Workloads

This figure shows four complementary views of the performance of all ten benchmarks of the SPEC95fp suite, for 1 to 16 processors. The *combined execution time* is the sum of the execution time over all processors. Using this metric, bars of same height for a given application correspond to a linear speedup. The *overhead* graph corresponds to the time spent in the operating system and in synchronization routines. The *memory system behavior* graph quantifies and classifies the memory system behavior. The memory system behavior is only reported for the useful execution of each processor. We use the definition of true and false sharing based on inter-processor word communication from Dubois et al. [8]. The last graph shows the occupancy of the bus. The results were generated with a 1MB direct-mapped cache. The operating system is IRIX and uses a page coloring policy for page mapping.

This disappointing result is due to the poor cache utilization that results from these applications' access patterns. Figure 3 illustrates the access patterns of the data segments of three of the applications, *tomcatv*, *swim* and *hydro2d*, executing on a 16-processor system. The figure plots the virtual pages that are accessed at least once during the steady state of the computation by the different processors.

If a processor accesses a single contiguous region of memory that is smaller than the cache, the page coloring policy used by the operating system successfully eliminates all conflicts. However, we see in Figure 3 that the access patterns of individual processors are rather sparse. Even though each processor accesses less than 1 MB of data, it does so in a range that is significantly larger than the cache size. The unfortunate consequence is that portions of the external cache will be under-utilized while other portions will suffer many cache conflicts.

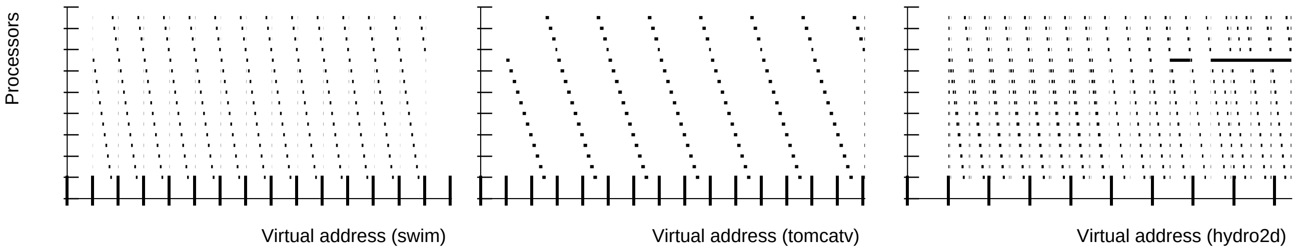

FIGURE 3. Page-level Virtual Access Patterns of 16-processor Runs.
The horizontal axis represents the data segment of the application's address space. The vertical axis corresponds to the 16 processors involved in the computation. The graph plots the virtual pages accessed at least once during the steady state of the workloads by the different processors. The sequential component of *hydro2d* gives a different access pattern for the master process, which runs on processor 11. Measurements were taken with a 4KB page size. The horizontal ticks correspond to 256 pages or 1MB of data. A page coloring policy will cause conflicts between pages that differ by a multiple of the cache size. By comparison, a uniprocessor access pattern would consist of a single horizontal line that spans the address space.

5 Compiler-Directed Page Coloring

Compiler-directed page coloring (CDPC) reduces cache conflicts by using information available in the compiler to direct the operating system's page mapping policy. CDPC extends the default mapping policy to allow applications to request a preferred mapping for a particular region in the virtual address space. This extension is compatible with both page coloring and bin hopping. We implemented compiler-directed page coloring with both IRIX 5.3 (which has a page coloring policy) and Digital UNIX (which has a a bin hopping policy).

Compiler-directed page coloring consists of three stages:

1. The compiler creates a summary of the array access patterns. The compiler generates function calls that pass the array access patterns to a run-time library, along with information known only at program start-up time (e.g. the exact dimensions of data structures).
2. The run-time system uses machine-specific parameters (i.e. the number of processors, the cache configuration and the page size) along with the array access information to generate a preferred color for each virtual page. These hints are passed to the operating system through a single system call.
3. The operating system uses the hints and tries to honor them as much as possible. For example, it may not be able to honor the hints if the machine is under memory pressure.

5.1 Creating Access Pattern Summaries

To create the access pattern summaries, the compiler uses information that is directly derived from its parallelization and locality analysis. It also takes advantage of several properties of compiler-parallelized codes. First the compiler uses optimizations to assign the computation and restructure the arrays so that the data within each individual data structure accessed by one processor is contiguous in the virtual address space when possible [2]. Second, to optimize for locality and minimize parallelization overhead, the compiler statically schedules the parallel computation across the processors. The access patterns of individual processors are therefore predictable.

The compiler extracts three kinds of information from the program:

Array Partitioning. The partitioning information consists of the starting address of the array, its total size, the size of the data partition unit and the data partitioning policy. The data partitioning unit is the amount of data that is operated on in each iteration of a parallel loop. For example, if the parallel loops iterate over the columns of a 2D array, then the partitioning unit is the size of a column. Our current implementation supports several common partitioning policies, including even partitions (where each processor is allocated a number of iterations that is as close to equal as possible) and blocked partitions (where processors are allocated $\lceil N/p \rceil$ iterations where N is the size of the distributed array dimension and p is the number of processors). We support both forward partitions, with iterations assigned starting at processor 0 up to processor p-1, and reverse partitions, with iterations assigned from processor p-1 down to 0.

Overlapping array partitions occur when the same array is accessed differently in different loops of a program or when there are unions of data structures.

Communication Patterns. In a shared-memory parallel application, communication occurs when a processor accesses parts of an array previously written by another processor. The compiler records information that combines array partitioning information with a communication type. Our current implementation supports shift and rotate communication of data on the boundaries between neighboring processors.

Group Access Information. The compiler records pairs of arrays that are accessed within the same loops.

5.2 Generating Page Coloring Hints at Run Time

Our algorithm generates page coloring hints based on two simple objectives. The first objective is to map the data accessed by each processor as contiguously in the physical address space as possible. This strategy will eliminate all conflict misses if the data accessed by a single processor fits in the cache since all pages will have a different color. Even when the data accessed by a single processor does not fit, the load is spread out evenly across the cache.

The second objective is to assign different colors to the starting locations of arrays that are used by the same processor at the same time. The rationale behind this objective is that element i of one array is often accessed with element *(i+c)* of a different array, where c is a small constant. Separating the starting locations of arrays in the cache is especially important when the working sets do not fit in the cache as it allows the program to exploit spatial locality.

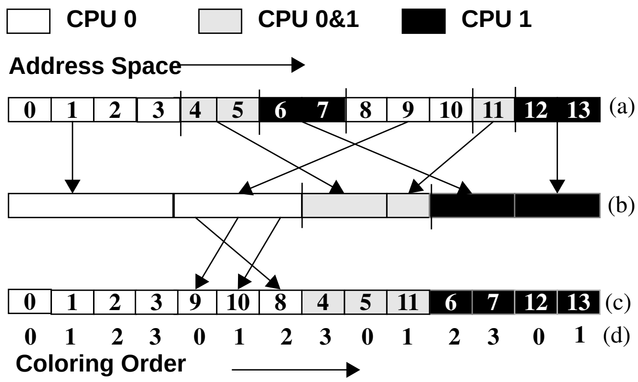

FIGURE 4. Illustration of the Algorithm.
This example illustrates the compiler-directed page coloring algorithm for two data structures (virtual pages 0-7 and 8-13, respectively), and two processors. The virtual address space is first split into uniform access segments (a). Segments that are accessed by the same processor set are grouped into uniform access sets (b). Pages within a segment are cyclically assigned to avoid a conflict on CPU0 (c). This produces the page coloring hints for a cache configuration with four colors (d).

We first define a *uniform access segment* to be a set of contiguous pages within a single array that is accessed by the same set of processors. Note that a page may be shared by multiple processors, e.g. when there are overlapping partitioning declarations. We partition the virtual address space of an array into maximal uniform access segments. Maximal uniform access segments with the same processor set from different arrays are grouped together to form *uniform access sets*.

Generating the page coloring hints consists of five steps:

1. **Create the Uniform Access Sets.** The maximal uniform access segments are easily be computed using the compiler-supplied array partitioning and communication information, along with parameters known only at start up time such as the number of processors. The algorithm starts by treating the entire virtual address space as a single uniform access segment. It processes each array partitioning and communication pattern summary in turn, by splitting segments at boundaries of arrays and whenever the access pattern within the array changes. Figure 4 illustrates the different steps of the CDPC algorithm for a simple example; Figure 4 (a) shows the uniform access segments.

2. **Order the Uniform Access Sets**. We represent each uniform access set as a node in an undirected graph, and edges are inserted between two nodes whenever their processor sets intersect. We wish to build a path that visits each node once. While the path need not include only edges in the graph, the objective is to have the path include as many edges as possible. The uniform access sets are sorted according to their order in the path. We have developed a simple heuristic for this step. We start by considering a subgraph with uniform access sets that only have either one or two members in their processor sets. We use a greedy algorithm that starts with a node with a singleton processor set, and extends the path by choosing an adjacent node that has not been visited whenever possible. For the remaining nodes, we simply insert them next to the node with the maximum overlap of processor set members. This heuristic clusters the pages accessed by each processor. We illustrate this step in Figure 4(b) where the pages accessed by both CPUs are put between the pages accessed by only CPU 0 and only CPU 1.

3. **Order the Segments within a Uniform Access Set.** The second step lays out the segments within each uniform access set. To increase locality, we use the group access information supplied by the compiler to put arrays used together close to each other. Again, we represent the segments as nodes in an undirected graph, and edges are inserted whenever the compiler specifies that they are used together in the group access information. The objective is the same as in Step 1, and that is to include as many edges in the graph in a path connecting all the nodes together. The segments are sorted according to their order in the path. We again use a simple greedy algorithm that starts with an arbitrary node and extends the path by choosing an adjacent node that has not yet been visited whenever possible. If there is a choice, we simply pick the segment with the smallest virtual address. In the example, each set contains only two segments (one from each data structure) that are trivially ordered.

4. **Order the Pages within a Uniform Access Segment.** Instead of laying down the pages within a segment in ascending virtual address order, we use a cyclic assignment strategy. We choose a starting point within the segment and lay out the pages in ascending order up to the segment boundary. At that point, the pages are wrapped around and placed in the first part of the segment. The starting points are chosen so as to space out *conflicting* segments as much as possible across the range of possible colors. Two segments may cause a conflict when (1) the two arrays are used together in the same loop, (2) the intersection of the processor sets of the two segments is non-empty, and (3) the two segments partially overlap in the cache. We illustrate this step in Figure 4 (c) where the pages 8, 9 and 10 are cyclically assigned to avoid such a conflict on CPU 0. The starting pages of the two data structures (pages 0 and 8) no longer have the same color.

5. **Assign Colors to the Pages.** At this point, the algorithm has sorted the virtual pages. Individual pages are assigned colors in a round robin order, as shown in Figure 4 (d).

Figure 5 shows the impact of this reordering on the locality of the access patterns. By comparison with Figure 3, we see that the access patterns are significantly denser.

5.3 Implementing Page Coloring Hints

The interface to the operating system consists of a sequence of virtual pages with their associated preferred color. Applications do not request particular pages of memory, but only suggest a particular coloring for a range of pages. The information is treated as a hint by the operating system.

In IRIX 5.3, we implemented this mechanism as an extension of the `madvise` system call. The page mapping information is stored in a table that is accessed by the virtual memory subsystem during page faults. If no mapping has been defined for the page, the virtual memory manager uses IRIX 5.3's native page coloring policy to determine a preferred color. In both cases, a page allocation request is passed on to the physical memory manager that ultimately chooses a free page, thereby determining its color.

With Digital UNIX, we take advantage of the operating system's bin-hopping policy to selectively touch the pages in a specific order that will generate the desired mapping. The advantage of such a scheme over a kernel implementation is that it requires no operating system modification. One drawback is that all page faults are serialized at the beginning of the application. Another

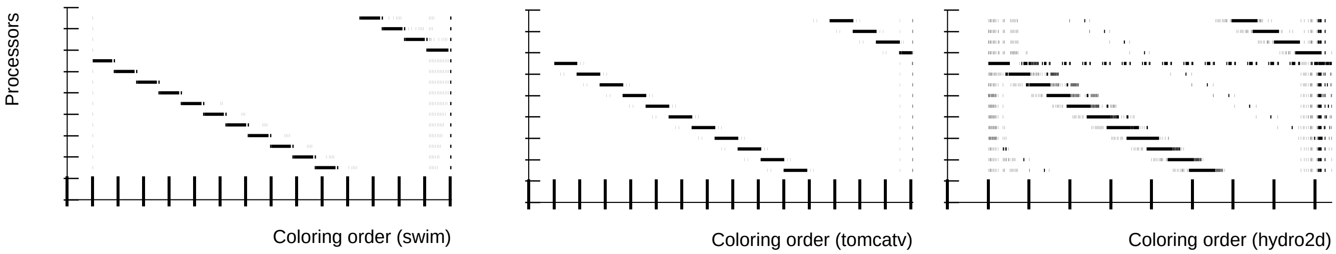

FIGURE 5. Effect of Compiler-Directed Page Coloring on Page-level Access Patterns.
The figure plots the access patterns in the coloring order for the same workloads as in Figure 3. The coloring order is determined by CDPC. Each tick of the horizontal axis corresponds to color zero in a 1MB cache. Conflicts occur only between pages that have the same color. The figure plots all page-level accesses that occur during the steady state of the computation.

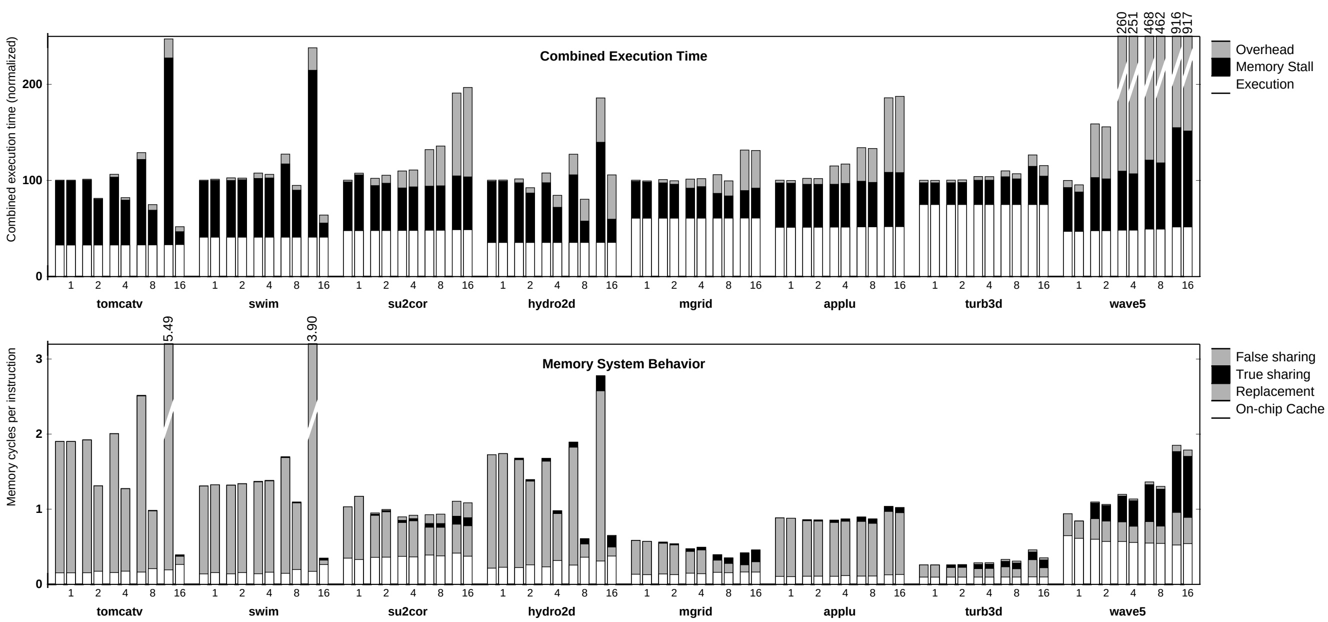

FIGURE 6. Impact of Compiler-Directed Page Coloring
For each application and a given number of processors, we compare the performance of a standard page coloring policy (left bar) with compiler-directed page coloring (right bar). CDPC has no effect on the other two applications, *apsi* and *fpppp*, so we omit them for space considerations.

drawback of the implementation is the delay between the initialization of the page, done by the operating system at page-fault time, and the first effective use of the page by the application. In the absence of delay, the application benefits from the likely presence of the page in the cache, due to the kernel's initialization.

5.4 Aligning Data Structures

Page mapping policies are only effective at eliminating conflicts in physically indexed caches, not in the virtually indexed caches that are usually found on-chip. Page mapping policies also cannot reduce the number of communication misses suffered by the application. However, a simple policy that aligns all data structures to start on a cache line boundary is effective at eliminating false sharing problems between data structures and within data structures when each processor operates on a multiple of the cache line size. In our current implementation, all data structures are dynamically allocated and aligned at start-up time. Such a policy could also be implemented at link time.

To avoid a common source of conflicts in the on-chip cache, we also use the group access information generated by the compiler to insert small amount of padding between data structures in the virtual address space. As a result, the starting addresses of data structures that are used together never map to the same location in the on-chip cache.

6 Simulation Results

In this section we use SimOS to evaluate the performance of compiler-directed page coloring. We compare it with IRIX's page coloring policy. We then study the impact of a complementary technique, compiler-inserted prefetching, that hides the latency of cache misses.

6.1 Impact of Compiler-Directed Page Coloring

Figure 6 shows the performance results of compiler-directed page coloring for our base configuration, a 1MB direct-mapped cache.

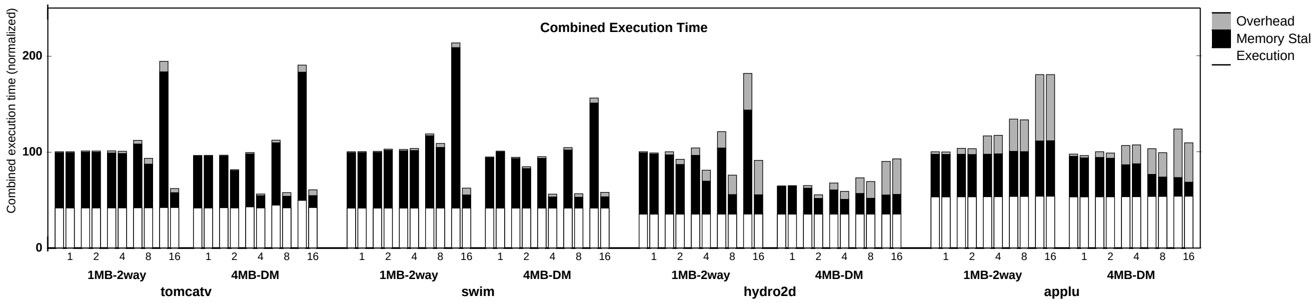

FIGURE 7. Impact of CDPC on Two-way Set-associative Caches and a larger 4MB Direct-mapped Cache. The combined execution time with the default page coloring policy (left bar) is compared with the performance with compiler-directed-page coloring (right bar). We only show the results for the four benchmarks where CDPC is most significant.

We see that CDPC can have a significant impact on performance by greatly reducing the number of replacement misses. Also, it is generally more effective as the number of processors is increased.

For *tomcatv, swim*, and *hydro2d*, CDPC shows large performance improvements. For *tomcatv* and *hydro2d*, performance improvements start with two processors and for *swim* the gains begin with eight processors. When the applications' working sets (8MB for *hydro2d*, 14MB for *tomcatv* and *swim*) fit into the caches, CDPC eliminates nearly all of the conflict misses. In our classification, the stall time associated with on-chip cache misses corresponds only to the stall time associated with hits in the off-chip cache, and not to misses serviced by memory. As a result of the better utilization of the off-chip cache, we therefore see an increase in the on-chip stall time.

Because the number of replacement misses in *turb3d* and *mgrid* is small, CDPC shows only a slight improvement in performance above four and eight processors. The performance of *su2cor* actually degrades slightly with CDPC. In this case, each processor does not access contiguous regions of some important data structures. CDPC is only applied to the remaining data structures, but the mapping happens to conflict with the other data structures. Finally, CDPC does not improve the performance of *applu,* which suffers from capacity misses due to its large (31MB) data set.

Two-way set-associative caches are known to be effective at eliminating many conflict misses. We therefore study the impact of CDPC with such a cache configuration (Figure 7). We see that the improvements for CDPC on two-way set-associative caches are similar to that on direct-mapped caches. Although set-associative caches reduce conflict hot spots, they do not address the issue of under-utilized caches. For example, *tomcatv* has seven large data structures and only an eight-way set-associative cache of size 1MB would eliminate all conflicts for 16 processors. For some applications such as *swim* and *tomcatv*, the benefits of using a two-way set-associative cache are small since SUIF's policy of aligning data structures on cache line boundaries already eliminates most of the conflict misses that would show up in a direct-mapped cache and not in set -associative caches.

Figure 7 also shows the impact of a larger 4MB cache on the same workloads. The benefits of CDPC appear with fewer processors, e.g. for *tomcatv* and *swim*. With larger caches, it takes less processors to fit the data set into the aggregate cache size. We see that *hydro2d*, whose data set is only 8 MB, benefits significantly from the increase in cache size, even in the sequential case. Although CDPC is still effective at reducing replacement misses, this does not translate into a large performance gain since the default page mapping policy is already effective at eliminating the problem. On the other hand, *applu*, whose data set is 31 MB, showed no benefit of CDPC with the 1MB cache configuration, but shows benefits from CDPC with the larger 4MB configuration.

6.2 Compiler-Inserted Prefetching

While CDPC effectively avoids mapping conflicts, it is not applicable to other types of cache misses—i.e. those caused by communication between processors or by limited cache capacity. As we saw earlier in Figure 6, the compiler has effectively parallelized the code such that true sharing and false sharing communication misses are typically small. However, when the data set size exceeds the aggregate cache size—which is often true for these applications with eight or fewer processors—the replacement misses can remain high. To hide the latency of the misses that remain after CDPC, we used the SUIF compiler to automatically insert *prefetch* instructions into the compiler-parallelized code. Our algorithm uses *locality analysis* to insert prefetches only for those references that are likely to suffer misses [19].

Although the MIPS R4400 processors do not support prefetching, the SimOS CPU simulators model the prefetch instruction of the MIPS R10000. The processor supports up to four outstanding prefetches (a fifth prefetch stalls the processor), and prefetches for pages that are not mapped in the TLB are ignored and do not cause exceptions. Prefetched lines are inserted into the external cache but not the on-chip cache. As before, the system bus bandwidth is 1.2 GB/s. Due to space considerations, Figure 8 shows the results of prefetching combined with CDPC.

As we see in Figure 8, compiler-inserted prefetching effectively hides memory latency in three of the applications: *tomcatv*, *swim*, and *hydro2d*. For *su2cor*, prefetching degrades the performance for runs with a larger number of processors, due to an increase in false sharing. In *applu*, the loop tiling introduced during parallelization to reduce synchronization inhibits the software pipelining of prefetches (i.e. they are not scheduled early enough). In addition, since the prefetches often have large access strides, they often reference pages that are not mapped in the TLB, in which case they are dropped[1].

The relative advantages of prefetching and CDPC vary with the number of processors. With fewer processors, capacity misses tend to dominate conflict misses, and hence prefetching offers more of an advantage than CDPC. With increased numbers of processors, the aggregate cache size increases, and CDPC becomes more important.

1. This suggests that a version of a prefetch that is *not* dropped on a TLB miss may be desirable for large matrix-based codes where TLB faults are common.

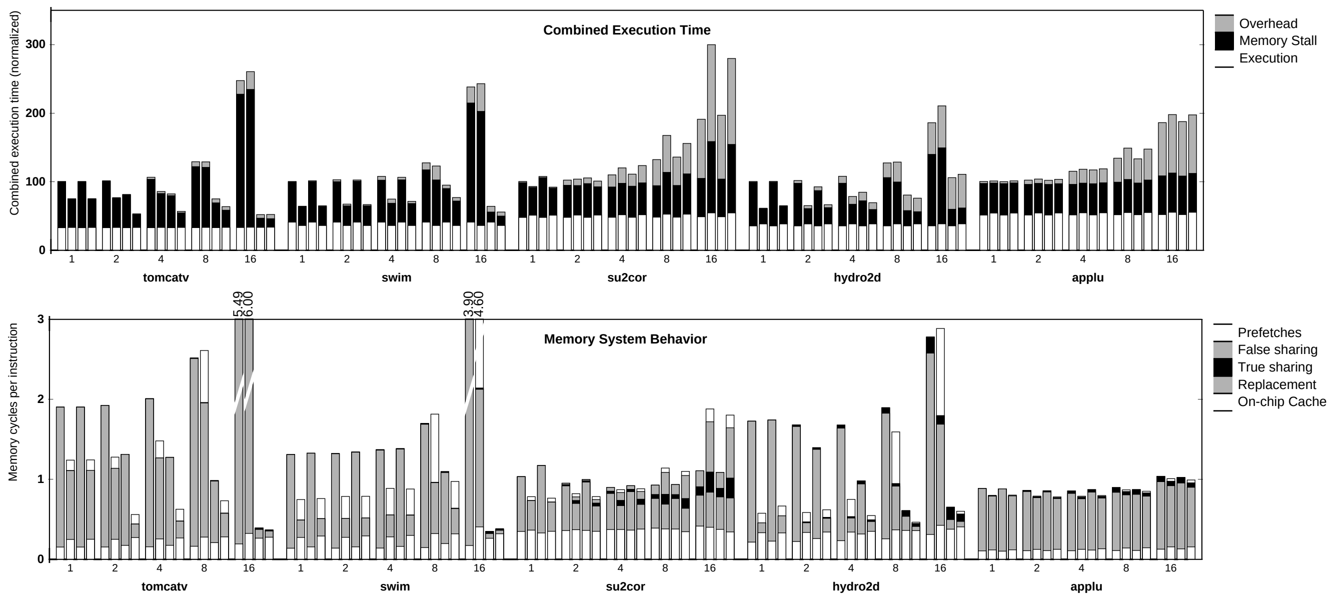

FIGURE 8. Impact of Prefetching and Compiler-directed Page Coloring

For five workloads and different numbers of processors, these graphs compare from the left bar to the right bar: (1) page coloring without prefetching, (2) page coloring with prefetching, (3) CDPC without prefetching and (4) CDPC with prefetching. The results are presented only for the applications where the memory system is a primary performance bottleneck.

Prefetching and CDPC are complementary. Prefetching improves the performance of CDPC by hiding the latency of misses that CDPC does not eliminate. By eliminating cache conflicts, CDPC improves the performance of prefetching in two ways. First, it reduces the probability that prefetched data will be displaced from the cache before it can be referenced. Second, it frees up bus bandwidth, which is crucial for any latency tolerance scheme. The complementary interaction between CDPC and prefetching is illustrated by *tomcatv* with four processors: taken individually, CDPC and prefetching each accelerate performance by 29% and 24%, respectively—when combined, however, they yield a total speedup of 88%.

7 Performance Results

To validate our technique on a real machine, we performed experiments on an 8-CPU Digital AlphaServer 8400 with 350 MHz 21164 processors. Each processor has two levels of on-chip caches and an external 4MB direct-mapped cache.

Figure 9 shows the performance of the applications with three page mapping policies: bin hopping, page coloring and CDPC. We implement both page coloring and CDPC by using Digital UNIX's native bin hopping policy and selectively touching pages in the desired order. We also show the performance of bin hopping when the data structures are neither aligned nor padded in the virtual address space (as described in Section 5.4).

We first notice that the results are consistent with the simulation results of Section 4 and Section 6. The figure also shows that neither bin hopping nor page coloring dominates the other one for all applications. Two benchmarks, *swim* and *tomcatv*, stand out as being the most sensitive to the alignment and page mapping policy. For *swim*, the limited speedup measured with page coloring is due primarily to contention on the AlphaServer's bus. For both benchmarks, bin hopping leads to better performance than page coloring. CDPC, however, significantly outperforms both bin hopping and page coloring. With eight processors, *swim* is 1.4 times faster with CDPC than with bin hopping and 2.6 times faster than with page coloring. For the same number of processors, *tomcatv*, is 1.3 times faster with CDPC than with bin hopping and 2.2 times faster than with page coloring! Finally, *applu* speeds up by 1.2 times over bin hopping and by 1.06 over page coloring.

For *turb3d*, *hydro2d* and *mgrid*, CDPC leads to noticeable, yet less significant, performance gains over either or both page mapping policies. Finally, for *su2cor, wave5, apsi* and *fpppp*, we see little variance in the performance between any of the page mapping policies. As shown in Figure 9 and Table 2, CDPC performs at least as well as the best of the two static policies in most cases.

Table 2 presents the AlphaServer results and their SPEC95fp ratio[2]. Overall, compiler-directed page coloring improves the SPEC95fp rating for eight processors by 8% over Digital UNIX's bin hopping policy and by 20% over page coloring. With this technique, the geometric mean of the performance improves over the uniprocessor execution by 2.9 with four processors and by 4.2 with eight processors.

8 Conclusions

This paper describes a new technique, *compiler-directed page coloring*, that reduces conflict misses in compiler-parallelized applications. Compiler-directed page coloring relies on the interaction of the parallelizing compiler and the operating system. The technique is transparent to the application as it does not require any transformations of the virtual address space.

This technique is fully automatic and simple to implement. The information generated by the compiler is directly derived from the analyses required for parallelization and locality optimization. The operating system extension is simple to integrate in commercial operating systems.

2. Since fpppp has no loop-level parallelism, we use the native compiler directly for this workload.

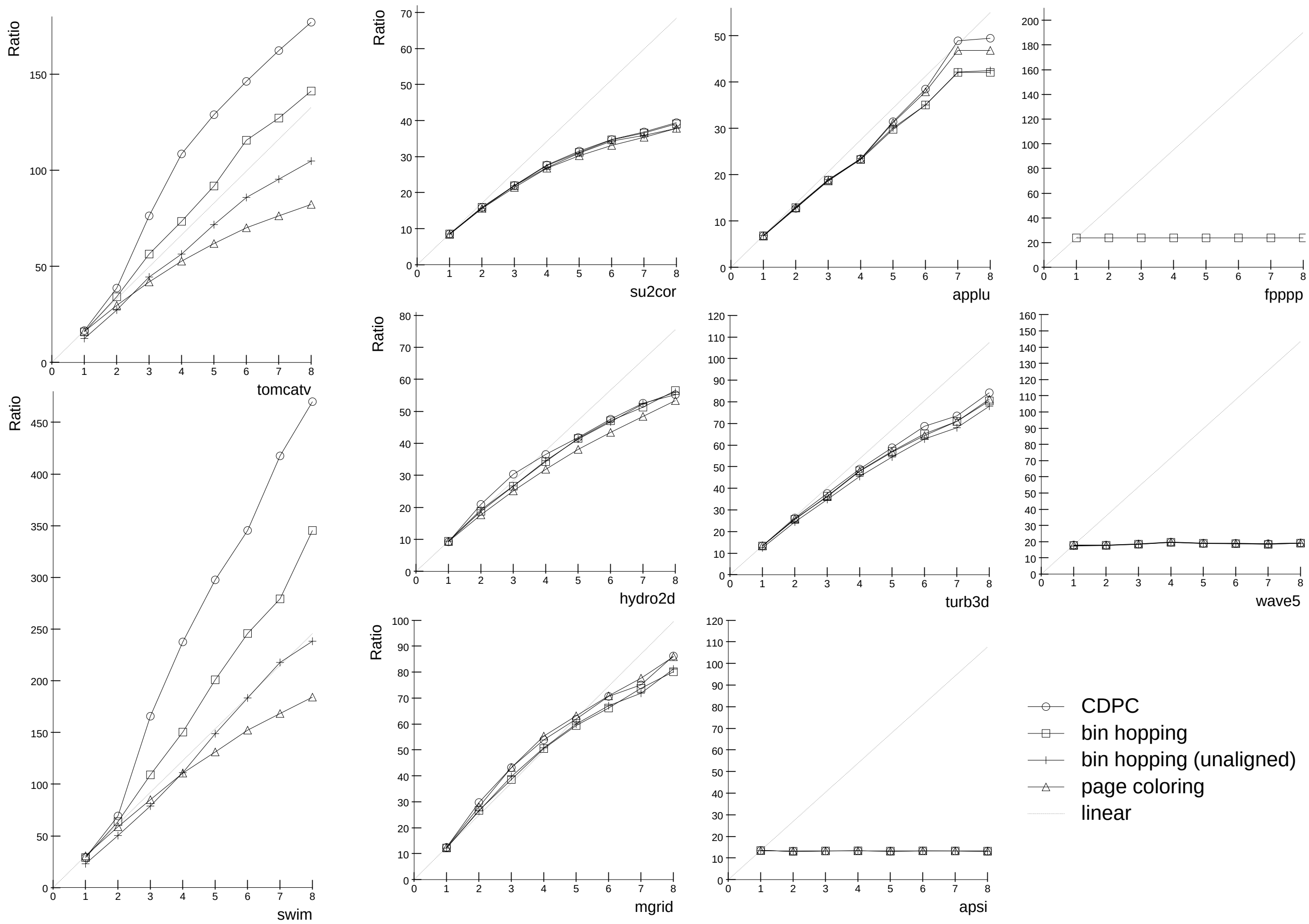

FIGURE 9. Performance of the SPEC95fp Benchmark Suite parallelized by the SUIF Compiler on 350 MHz AlphaServer 8400. For each of the ten benchmarks, we compare the performance of the benchmark with one to eight processors using the page coloring, bin hopping and CDPC page mapping policies. Additionally, we report the time with bin hopping when data structures are not aligned. The performance is compared with an ideal speedup curve based on the best of the four uniprocessor times. All results are expressed as a SPEC95fp ratio. The ratio is the speedup over the reference execution time of a SparcStation10.

Benchmark	Execution Time (seconds)								
	Bin Hopping			Page Coloring			CDPC		
	1P	4P	8P	1P	4P	8P	1P	4P	8P
101.tomcatv	232.5	50.4	26.2	229.7	70.2	45.0	223.0	34.1	20.9
102.swim	296.2	57.2	24.9	280.1	77.7	46.7	295.3	36.2	18.3
103.su2cor	164.2	50.9	35.8	166.9	52.3	37.0	163.7	50.6	35.5
104.hydro2d	254.9	70.0	42.5	256.8	75.3	45.0	260.0	65.7	43.5
107.mgrid	205.2	49.5	31.2	201.2	45.2	29.1	201.2	46.4	29.0
110.applu	324.1	94.6	52.3	327.3	94.4	47.0	324.0	93.9	44.5
125.turb3d	309.2	85.2	51.0	305.5	85.7	50.5	307.3	83.9	48.7
141.apsi	156.4	157.8	159.6	156.0	157.8	159.6	156.1	157.7	159.5
145.fpppp	403.7	403.7	403.7	403.7	403.7	403.7	403.7	403.7	403.7
147.wave5	169.9	153.1	157.4	167.2	151.6	156.2	167.4	152.6	155.9
SPEC95fp	13.7	36.0	53.0	13.8	33.8	47.6	13.8	39.8	57.4

Table 2. Execution Time and SPEC95fp rating on 350MHz AlphaServer with bin hopping, page coloring and CDPC.

Our study of the SPEC95fp benchmark suite revealed that numerical applications are sensitive to page mapping policies. Compiler-directed page coloring is effective on both direct-mapped and two-way set-associative caches and is complementary to latency-hiding techniques such as compiler-inserted prefetching. Compiler-directed page coloring performs better than page coloring and bin hopping (two standard page mapping policies implemented in commercial operating systems). By improving the performance of some applications significantly, we obtain a SPEC95fp ratio of 57.4, which is the highest to date.

Acknowledgments

The authors wish to thank Evan Torrie, Chris Wilson, Dave Heine and Kinshuk Govil for their help in the preparation of this paper. We also wish to thank the SUIF and SimOS teams for their contributions to the infrastructure used in this study. We thank Robert Cohn, Zarka Cvetanovic, and Richard Grove from DEC. This study is part of the Stanford FLASH and SUIF projects, funded by ARPA grant DABT63-94-C-0054 and ARPA and Air Force Material Command Grant F30602-95-C-0098. This research is also supported in part by an equipment grant from Digital. Ed Bugnion is supported in part by a NSF Graduate Research

Fellowship. Monica Lam and Mendel Rosenblum are partially supported by NSF Young Investigator Awards. Todd Mowry is partially supported by a Research Grant from the Natural Sciences and Engineering Research Council of Canada, and by a Faculty Development Award from IBM. Jennifer Anderson is currently a researcher with Digital Equipment's Western Research Lab.

References

[1] Saman P. Amarasinghe, Jennifer M. Anderson, Christopher S. Wilson, Shih-Wei Liao, Robert S. French, Mary W. Hall, Brian R. Murphy and Monica S. Lam. The Multiprocessor as a General-Purpose Processor: A Software Perspective. *IEEE Micro,* 16(3), Jun. 1996.

[2] Jennifer M. Anderson, Saman P. Amarasinghe and Monica S. Lam, "Data and Computation Transformations for Multiprocessors," In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jul.1995, pp. 166-178.

[3] Jennifer M. Anderson and Monica S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines", In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, Jun. 1993, pp. 112-125.

[4] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen, "Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches", In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994, pp. 158-170.

[5] David F. Bacon, Susan L. Graham and Oliver J. Sharp, "Compiler Transformations for High-Performance Computing", In *Computing Surveys*, 26 (4), Dec. 1994.

[6] David Callahan, Ken Kennedy and Allan Porterfield, "Software Prefetching", In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991, pp. 40-52.

[7] Steve Carr, Kathryn S. McKinley and Chau-Wen Tseng, "Compiler Optimizations for Improving Data Locality", In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994, pp. 252-262.

[8] 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*, May 1993, pp. 88-97.

[9] Susan J. Eggers and Randy H. Katz, "The effect of sharing on cache and bus performance of parallel programs", In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr.1989, pp. 257-270.

[10] Dawson R. Engler, M. Frans Kaashoek and James O'Toole Jr. "Exokernel: An Operating System Architecture for Application-Level Resource Managment", In *Proceedings of the 15th ACM Symposium on Operating System Principles,* Dec. 1995, pp 251-266.

[11] Manish Gupta and Prith Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers." In *IEEE Transactions on Parallel and Distributed Systems*, 3(2), Mar. 1992, pp. 179-193.

[12] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao and Monica S. Lam, "Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler," In *Proceedings of Supercomputing '95*, Dec. 1995.

[13] Kieran Harty and David R. Cheriton, "Application-controlled Physical Memory using External Page-Cache Management", In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* , Apr. 1991.

[14] Tor E. Jeremiassen and Susan J. Eggers, "Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations", In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jul. 1995, pp. 179-188.

[15] Ken Kennedy and Ulrich Kremer, "Automatic Data Layout for High Performance Fortran", In *Proceedings of Supercomputing '95,* Dec. 1995.

[16] Richard E. Kessler and Mark D. Hill, "Page Placement Algorithms for Large Real-indexed Caches", In *ACM Transactions on Computer Systems*, 10(4), Nov. 1992.

[17] Butler W. Lampson, "Hints for Computer System Design", In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles,* Oct. 1983, pp. 33-48.

[18] Todd C. Mowry, Monica S. Lam and Anoop Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992, pp. 62-73.

[19] Todd C. Mowry, "Tolerating Latency through Software-controlled Data Prefetching", Ph.D. thesis, *Technical Report CSL-TR-94-626,* Stanford University, Mar. 1994.

[20] Theodore H. Romer, Dennis Lee, Brian N. Bershad and J. Bradley Chen, "Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware", In *Proceedings of the First Symposium on Operating Systems Design and Implementation,* Nov. 1994, pp. 255-266.

[21] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel and Anoop Gupta, "Complete Computer Simulation: The SimOS Approach", In *IEEE Parallel and Distributed Technology,* 3(4), Fall 1995.

[22] Standard Performance Evaluation Corporation, The SPEC95fp benchmark suite. *http://www.spechbench.org*.

[23] Ben Verghese, Scott Devine, Anoop Gupta and Mendel Rosenblum, "Operating System Support for Improving Locality on CC-NUMA Compute Servers", In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems,* Oct. 1996.

[24] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steven W.K. Tjiang, Shi-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam and John L. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers", In *ACM SIGPLAN Notices*, 29(12), Dec. 1994.

[25] Emmett Witchel and Mendel Rosenblum, "Embra: Fast and Flexible Machine Simulation", In *Proceedings of the ACM SIGMETRICS '96 Conference on Measurement and Modeling of Computer Systems*, May 1996, pp. 68-79.

[26] Michael E. Wolf and Monica S. Lam, "A Data Locality Optimizing Algorithm", In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation,* June 1991, pp. 30-44.