Cacheminer: A Runtime Approach to Exploit Cache Locality on SMP

Yong Yan, *Member*, *IEEE Computer Society*, Xiaodong Zhang, *Senior Member*, *IEEE*, and Zhao Zhang, *Member*, *IEEE*

Abstract—Exploiting cache locality of parallel programs at runtime is a complementary approach to a compiler optimization. This is particularly important for those applications with dynamic memory access patterns. We propose a memory-layout oriented technique to exploit cache locality of parallel loops at runtime on Symmetric Multiprocessor (SMP) systems. Guided by application-dependent and targeted architecture-dependent hints, our system, called Cacheminer, reorganizes and partitions a parallel loop using the memory-access space of its execution. Through effective runtime transformations, our system maximizes the data reuse in each partitioned data region assigned in a cache, and minimizes the data sharing among the partitioned data regions assigned to all caches. The executions of tasks in the partitions are scheduled in an adaptive and locality-preserved way to minimize the execution time of programs by trading off load balance and locality. We have implemented the Cacheminer runtime library on two commercial SMP servers and an SimOS simulated SMP. Our simulation and measurement results show that our runtime approach can achieve comparable performance with the compiler optimizations for programs with regular computation and memory-access patterns, whose load balance and cache locality can be well optimized by the tiling and other program transformations. However, our experimental results show that our approach is able to significantly improve the memory performance for the applications with irregular computation and dynamic memory access patterns. These types of programs are usually hard to optimize by static compiler optimizations.

Index Terms—Cache locality, nested loops, runtime systems, simulation, symmetric multiprocessors (SMP), and task scheduling.

1 Introduction

The recent developments in circuit design, fabrication technology, and Instruction-Level Parallelism (ILP) technology have increased microprocessor speed about 100 percent every year. However, memory-access speed has only improved about 20 percent every year. In a modern computer system, the widening gap between processor performance and memory performance has become a major bottleneck to improving overall computer performance. Since the increase in memory-access speed cannot match that of the processor speed, memory-access contention is increased, which results in a longer memory-access latency. This makes memory-access operations much more expensive than computation operations. In multiprocessor systems, the effect of the widening processor-memory speed gap on performance becomes more significant due to the heavier access contention on the network and the shared memory and to the cache coherence cost. Recently, Symmetric Multi-Processor (SMP) systems have emerged as a major class of parallel computing platforms, such as HP/Convex Exemplar S-class [3], SGI Challenge [7], Sun SMP servers [5], and DEC AlphaServer [19]. SMPs dominate the server market for commercial applications and are used as desktops for scientific computing. They are also important building blocks for large-scale systems. The improvement on the memory performance of applications is critical to the successful use of SMP systems for applications.

In order to narrow the processor-memory speed gap, hardware caches have been widely used to build a memory hierarchy in all kinds of computers, from supercomputers to personal computers. The effectiveness of the memory hierarchy for improving performance of programs comes from the locality property of both instruction executions and data accesses of programs. In a short period of time, the execution of a program tends to stay in a set of instructions close in time or close in the allocation space of a program, called the *instruction locality*. Similarly, the set of instructions executed tends to access data that are also close in time or in the allocation space, called the *data locality*. Using a fast and small cache close to a CPU is expected to hold the working set of a program so that memory accesses can be avoided or reduced.

Unfortunately, the memory hierarchy is not a panacea for eliminating the processor-memory performance gap. Low-level memory accesses are still substantial for many applications and are becoming more expensive as the processor-memory performance gap continues to widen. The reasons for possible slow memory accesses are:

- Applications may not be programmed with an awareness of the memory hierarchy.
- Applications have a wide range of working sets which cannot be held by a hardware cache, resulting in capacity misses at cache levels of the memory hierarchy.

- *Y. Yan is with the Computer Systems Labs, Hewlett Packard Laboratories, Palo Alto, CA 94304. Email: yongyan@hpl.hp.com.*
- *X. Zhang and Z. Zhang are with the Computer Science Department, College of William and Mary, Williamsburg, VA 23187-8709. E-mail: {zhang, zzhang}@cs.wm.edu.*

Manuscript received 1 Oct. 1997; revised 24 June 1999; accepted 6 Jan. 2000. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 105746.

- The irregular data-access patterns of applications result in excessive conflict misses at cache levels of the memory hierarchy.
- In a time-sharing system, the dynamic interaction among concurrent processes and the underlying operating system causes a considerable amount of memory accesses as processes are switched in context. This effect cannot be handled by the memory hierarchy on its own.
- In a cache coherent multiprocessor system, false data sharing and true data sharing result in considerable cache coherent misses.
- In a CC-NUMA system, processes may not be perfectly co-located with their data, which results in remote memory accesses to significantly degrade overall performance.

Due to the increasing cost of memory accesses, techniques for eliminating the effect of long memory latency have been intensively investigated by researchers from application designers to hardware architects. In general, the proposed techniques fall into two categories: *latency avoidance* and *latency tolerance*. The latency tolerance techniques are aimed at hiding the effect of memory-access latencies by overlapping computations with communications or by aggregating communications. The latency avoidance techniques, also called locality optimization techniques, are aimed at minimizing memory accesses by using software and/or hardware techniques to maximize the reusability of data or instructions at cache levels of the memory hierarchy. In a SMP system, reducing the total number of memory accesses is a substantial solution to reduce cache coherence overhead, memory contention, and network contention. So, the locality optimization techniques, i.e., the latency avoidance techniques, are more demanding than the latency tolerance techniques. In addition, because instruction accesses are more regular than data accesses, designing novel data-locality optimization techniques is more challenging and more important for performance improvement.

The objective of this work is to propose and implement an efficient technique to optimize the data cache locality of parallel programs on SMP systems. The contributions of this work are: 1) We propose an effective cache locality exploitation method on SMP multiprocessors based on simple runtime information, which is complimentary to compiler optimizations. To our knowledge based on the existing literature, our design method and its implementations on SMPs are unique. 2) Abstracting the estimated physical memory-access patterns of program loops into internal data structures, we are able to partition and schedule parallel tasks by optimizing cache locality of the program. 3) We have built an application programming interface (API) to collect simple program hints and automatically generate memory-layout oriented parallel programs for users.

1.1 The Problem

In a SMP system as shown in Fig. 1, each processor has a hierarchy of local caches (such as the on-chip cache and the off-chip cache in the figure) and all the processors share a global memory. When a processor accesses its data, it first looks up the cache hierarchy. If the data is not found in caches, the processor reads the memory block that contains the required data from the shared memory and brings a copy of the memory block in an appropriate cache block in the cache hierarchy. Data is copied into the cache hierarchy so that the subsequent accesses to the data can be satisfied from the cache and memory accesses can be avoided. The cache locality optimization is aimed at optimizing the cache-access pattern of an application so that memory accesses can be satisfied in the cache as often as possible (or in other words, cache data can be reused as much as possible). To increase the chance of cache data to be reused, we must reduce the interference that would replace or invalidate the cache data. In a SMP system, there are two types of interference that would affect the reuse of cache data: the interference from the local processor which refills a cache block with new data, and the interference from a remote processor which invalidates stale data copies to maintain data consistency. The two types of interference in a parallel computation are determined by how the data is accessed in the computation, called the *data-access pattern*, and how the data is mapped into a cache, called the *cache-mapping pattern*. Hence, it is essential for a cache locality optimization technique to obtain and use the information on the data-access pattern and the cache-mapping pattern of a parallel program.

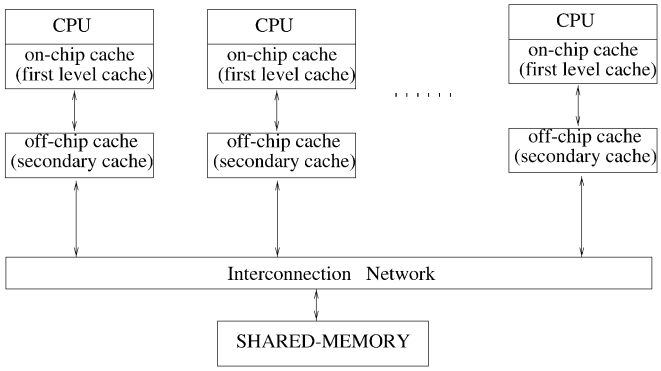


Fig. 1. SMP shared memory system model.

The data-access pattern of a program is determined by program characteristics. Because the compilation time of an application is not a part of its execution time, a compiler can use sophisticated techniques to analyze the data-access pattern of a program. However, there is a large class of real world applications whose data-access patterns cannot be analyzed at compile-time. The data-access patterns of many of these functions are dependent on runtime data. In addition, many real-world applications have indirect data accesses [22], which are difficult for a compiler to analyze. For example, pointers may point to different objects during the execution of a program, and the subscripts of an array variable may be given by another array variable. The existence of these complicated applications recommands runtime techniques for analyzing data-access patterns.

In Fig. 2, we present a sparse matrix multiplication algorithm where two sparse source matrices have dense representations. In the innermost loop, the two elements to

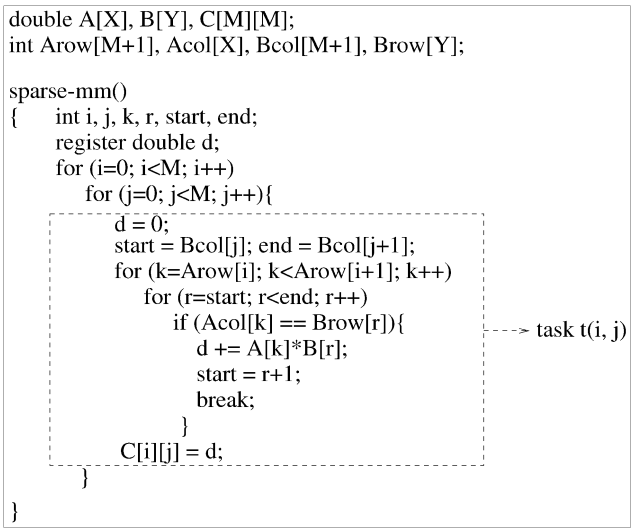


Fig. 2. A Sparse Matrix Multiplication (SMM) which has a dynamic data-access pattern and an irregular computation pattern.

be multiplied, `A[k]` and `B[r]`, are indirectly determined by the data in indexing arrays `Arow`, `Acol`, `Bcol`, and `Brow`. The data-access pattern of this program can only be determined at runtime when input data is available.

1.2 Comparisons with Related Work

Because the cache locality of an application is affected by program characteristics, by parallel compilation methods, by the interference of the underlying operating system, and by the architectural features of the cache, many locality exploitation methods have been proposed to improve the memory performance of applications at different system levels (see, e.g., [2], [4], [6], [8], [10], [11], [12], [13], [14], [15], [16], [20], [25]).

At compilation phase, the main idea of locality optimization techniques is to conduct two types of transformations: program transformations and data layout transformations. Some program transformation-based techniques use a data-reuse prediction model to determine a sequence of program transformations in order to maximize the cache locality of applications (see, e.g., [11], [14]). These techniques are control-centric where transformations are mainly conducted based on control flow analysis. The other type of transformations reorganize program structures based on data layout (see, e.g., [6], [9]). These techniques are data-centric in the logic space of a program, and have only been used on sequential programs so far. Data transformation based techniques aim at restructuring data layout so that data locality is optimized (see, e.g., [8], [20]). More recently, some techniques are proposed to take into consideration both program transformations and data transformations (see, e.g., [1], [2]). The success of compiler based transformations in improving memory performance of applications depends on static analysis at compiler-time on control flow and data flow. For applications with irregular and/or dynamic data access patterns, it is difficult for compiler based techniques to conduct control flow analysis and data flow analysis. In this case, runtime analysis is necessary and important.

Exploiting cache locality at runtime has been done in loop scheduling. References [12], [13], [25] present dynamic scheduling algorithms that take into consideration the affinity of loop iterations to processors. The main idea is to let each iteration stay on the same processor while it is repeatedly executed. Although significant performance improvement can be acquired for some applications, the type of affinity exploited by this approach is not very popular because it does not take into consideration the relationship between memory references of different iterations. Reference [12] does give some consideration to the data distribution of a loop by allocating iterations close to their data. In a SMP shared-memory system, all shared data reside in a unique shared memory where the proposed method in [12] is not applicable. The proposed technique in this paper not only takes into consideration the affinity of parallel tasks to processors, it also uses information on the underlying cache architecture and memory reference patterns of tasks to minimize cache misses and false sharing.

In the design of the COOL language [4], the locality exploitation issue is addressed using language mechanisms and a runtime system. Both task affinity and data affinity are specified by users and then are implemented by the runtime system. A major limit with this approach is that the quality of locality optimizations heavily depends on a programmer. Our proposed technique uses a simple programming interface for a user or compiler to specify simple information about data, not about complicated affinity relations. The affinity relations will be recognized at runtime.

Regarding the runtime locality optimization of sequential programs, reference [16] proposes a memory-layout oriented method. It reorganizes the computation of a loop based on some simple hints about the memory reference patterns of loops and cache architectural information. Compared with a uniprocessor system, a cache coherent shared memory system has more complicated factors that should be considered for locality exploitation, such as data sharing and load balancing. Our proposed memory-layout oriented method is aimed at attacking the following two distinct tasks: 1) It uses a more precise multidimensional hash structure to reorganize tasks so that the task can be partitioned easily into groups to maximize data reuse in a group and minimize data sharing among groups. 2) It dynamically trades off cache locality with parallelism, load imbalance, and runtime scheduling overhead.

1.3 Models and Notations

The program structures addressed in this paper are nested loops in applications as shown in Fig. 3 (all the programs presented in this paper are in C-language format consistent with the C-language implementation of our runtime system). In Fig. 3, l_j and u_j are the lower bound and upper bound of loop index variable i_j for $j = 1, 2, \cdots, k(k \geq 1)$, which are usually functions of the outer loop index variables, $i_1, i_2, \cdots, i_{j-1}$, and are determined at runtime. The loop body **B** is a set of statements where the statements can also be loops. An execution instance of the loop body B can be considered as a fine-grained task, which can be

```
for (i1 = l1; i1 < u1; i1 ++)
    for (i2 = l2; i2 < u2; i2 ++)
        ⋮
        for (ik = lk; ik < uk; ik ++)
            B;
```

Fig. 3. Data-independent nested loop.

expressed as $\mathbf{B}(t_1, t_2, \cdots, t_k)$, where t_j is the value of index variable i_j for $j = 1, 2, \cdots, k$.

The condition that the above nested loop must satisfy is defined as follows: All the execution instances of the loop body **B** are data-independent, i.e., for any two instances, $\mathbf{B}(t_1^1, t_2^1, \cdots, t_k^1)$ and $\mathbf{B}(t_1^2, t_2^2, \cdots, t_k^2)$, the following condition is valid:

$$\begin{aligned} &out(\mathbf{B}(t_1^1, t_2^1, \cdots, t_k^1)) \cap out(\mathbf{B}(t_1^2, t_2^2, \cdots, t_k^2)) = \emptyset \wedge \\ &out(\mathbf{B}(t_1^1, t_2^1, \cdots, t_k^1)) \cap in(\mathbf{B}(t_1^2, t_2^2, \cdots, t_k^2)) = \emptyset \wedge \\ &in(\mathbf{B}(t_1^1, t_2^1, \cdots, t_k^1)) \cap out(\mathbf{B}(t_1^2, t_2^2, \cdots, t_k^2)) = \emptyset, \end{aligned} \quad (1)$$

where notations `out` and `in` represent, respectively, the output variable set and input variable set of an instance [22].

The rest of the paper is organized as follows: The next section describes our runtime optimization technique in detail. Section 3 presents our performance evaluation method. The performance results are given in Section 4. Finally, we conclude our work in Section 5.

2 Runtime Cache Locality Exploitation Method

A runtime system should be highly effective in order to prevent the benefit of cache locality optimizations from being nullified by the associated runtime overhead. Thus, a simple and heuristic runtime approach is the only realistic choice.

The basic idea of our method is to group and partition tasks through shrinking and partitioning the memory access space of parallel tasks. Shrinking the memory access space is to group tasks that have shared data accessing regions. These tasks are expected to reuse data in a cache when they execute as a group. Partitioning the memory access space is to divide tasks into several partitions so that tasks belonging to different partitions have minimal data sharing among their memory access regions. Finally, task partitions are adaptively scheduled to execute on the multiprocessor system for possible load balance, subject to minimize the execution time of the tasks. This runtime approach is primarily memory-layout oriented.

Our runtime technique has been implemented as a set of library functions. Fig. 4 presents a framework of the system. A given sequential application program is first transformed by a compiler or rewritten by a user to insert runtime functions. The generated executable file is encoded with application-dependent hints. At runtime, the encoded functions are executed to fulfill the following functionalities: estimating the memory-access pattern of a program,

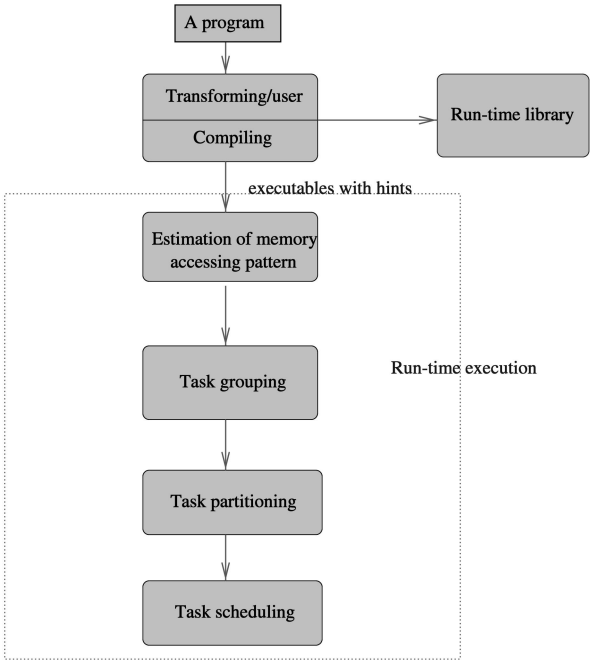


Fig. 4. Framework of the runtime system.

reorganizing tasks into cache affinity groups where the tasks in a group are expected to heavily reuse their data in the cache, partitioning task-affinity groups onto multiple processors so that data sharing among multiple processors is minimized, and then adaptively scheduling the execution of tasks.

In order to minimize runtime overhead, a multidimensional hash table is internally built to manage a set of task-affinity groups. Meanwhile, a set of hash functions are given to map into an appropriate task-affinity group in the hash table. Locality oriented task reorganization and partitioning are integratedly done in the task mapping. This section describes our information estimation method, internal data structures for task reorganization and partitioning, and our task scheduling algorithm.

2.1 Memory-Access Pattern Estimation

In a loop structure, data is usually declared and stored as arrays. Let $A_1, A_2, \cdots, A_n$ be the n arrays accessed in the loop body of a nested data-independent loop. Each array is usually laid out in a contiguous memory region, independent of the other arrays. Many operating systems attempt to map contiguous virtual pages to cache blocks contiguously, so that our virtual-address-based study is practically meaningful and effective. In rare cases, an array may be laid out across several uncontiguous memory pages. Although our runtime system may not handle these rare cases efficiently, the system works well for most memory layout cases in practice.

The memory regions of the n independent arrays can be represented by an n-dimensional memory-access space, expressed as $(A_1, A_2, \cdots, A_n)$, where arrays are arranged in any selected order by a user. This n-dimensional memory-access space practically contains all the memory addresses that are accessed by a loop.

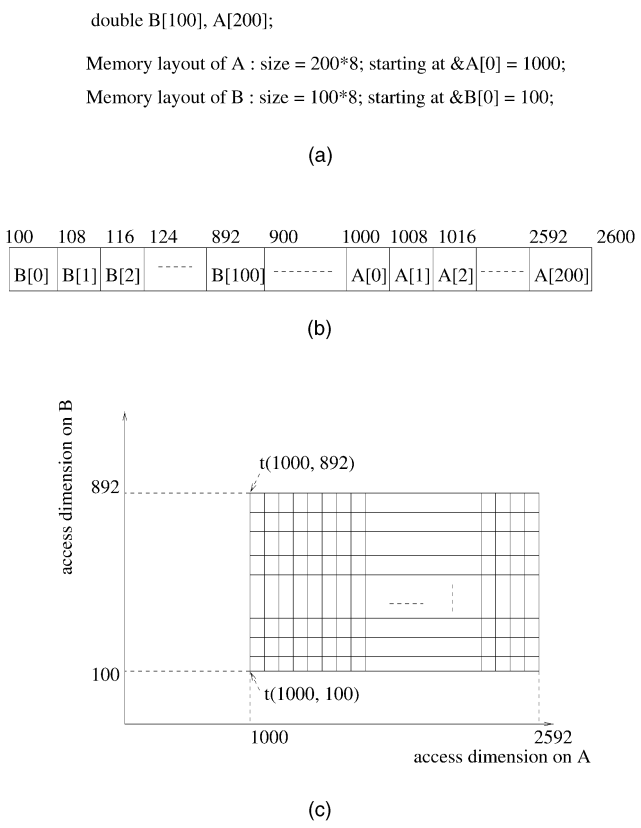


Fig. 5. Memory-access space representations of the SMM program. (a) Hints on memory layouts of two accessed arrays. (b) Physical memory layout. (c) A 2D memory-accessing space.

In order for the runtime system to capture the memory space information, the following three hints are provided by the interface.

Hint 1. *n, the number of arrays accessed by tasks.*

Hint 2. The *size* in bytes of each array. Based on this, the runtime system maintains a Memory-access Space Size vector $(s_1, s_2, \cdots, s_n)$, denoted as the MSS vector, where s_i is the size of ith array $(i = 1, 2, \cdots, n)$.

Hint 3. The *starting memory address* of each array. From this, the underlying runtime system constructs a starting address vector $(b_1, b_2, \cdots, b_n)$, denoted the SA vector, where $b_i (i = 1, 2, \cdots, n)$ is the starting memory address of ith array.

Hint 1 is static and can be collected at the user or compiler level. **Hint 2**, the array size, may be static and is known at compiler-time, or dynamic and is determined at runtime. **Hint 3**, the starting addresses, are dynamic because memory addresses can only be determined at runtime.

After determining the global memory-access space of a loop, we need to determine how each parallel iteration accesses the global memory-access space so that we can reorganize them to improve memory performance. Here, we abstract each instance of a loop body of a parallel loop as a parallel task. The accessing region of a task in an array is simply represented by the starting address of the region. So, the following hint is provided by the runtime system.

Hint 4. *A memory-access vector of task t_j:*

$$(a_{j1}, a_{j2}, \cdots, a_{jn}),$$

where a_{ji} is the starting address of the referenced region on ith array by task t_j $(i = 1, 2, \cdots, n)$. In some loop structures, a parallel iteration may not contiguously access an array so that the access region may not be precisely abstracted by the starting address. In this case, the loop iteration should be further split into smaller iterations so that each iteration accesses a contiguous region on each array. In addition, the following hint is also provided to assist task partitioning.

Hint 5. p, the number of processors.

Based on the above hints, the memory-access space of the loop is abstracted as an n-dimensional memory-access space:

$$(b_1 : b_1 + s_1 - 1, b_2 : b_2 + s_2 - 1, \cdots, b_n : b_n + s_n - 1).$$

Task t_j is abstracted as point $(a_{j1}, a_{j2}, \cdots, a_{jn})$ in the memory-access space based on the runtime estimation on its memory-access pattern. Fig. 5 presents an example of the abstract representation of the memory accesses based on the physical memory layout of arrays `A` and `B` in the SMM given in Fig. 2. Fig. 5a gives the hints on the memory-access space. Fig. 5b illustrates the memory layout of two arrays where `B` and `A` are laid out at starting address 100 and 1,000, respectively. Each array element has size of 8 bytes. Then, the memory-access space is represented by a 2D space as shown in Fig. 5c where each point gives a pair of possible starting memory-access addresses on `A` and `B`, respectively, by a task. For example, `t(1,000, 100)` means task `t` will access array `A` at starting memory address 1,000, and access array `B` at starting physical address 100.

2.2 Task Reorganization

In the memory-access space, same or nearby task points access the same or nearby memory addresses. So, grouping nearby tasks in the memory-access space for execution take advantage of temporal locality and spatial locality of programs. This is achieved by shrinking the memory-access space based on the underlying cache capacity (size).

Let $\{t_i(a_{i1}, a_{i2}, \cdots, a_{in}) | i = 1, 2, \cdots, m\}$ be a set of m data independent tasks of a parallel loop, and $(b_1 : b_1 + s_1 - 1, b_2 : b_2 + s_2 - 1, \cdots, b_n : b_n + s_n - 1)$ be the memory-access space of the parallel loop. Conceptually, task t_i (i=1, $\cdots$, n) is mapped onto point $(a_{i1}, a_{i2}, \cdots, a_{in})$ in the memory-access space based on the starting memory addresses of their memory-access regions. In addition, let p be the number of processors and C be the capacity of the underlying secondary cache in bytes.

Task reorganization consists of two steps. In the first step, the memory-access space $(b_1 : b_1 + s_1 - 1, b_2 : b_2 + s_2 - 1, \cdots, b_n : b_n + s_n - 1)$ is shifted to origin point $(0, \cdots, 0)$ by subtracting $(b_1, b_2, \cdots, b_n)$ from the coordinates of all task points. In the second step, we use the *equal-shrinking* method to shrink each dimension of the shifted memory by fC/n. The n-dimensional space resulted from shrinking is called a n-dimensional bin space. Here, f is a weight constant in $(0, 1]$. When $f = 1$, the cache is fully utilized, otherwise the cache is partially used for the tasks. This gives an alternative to tune program performance. In the bin

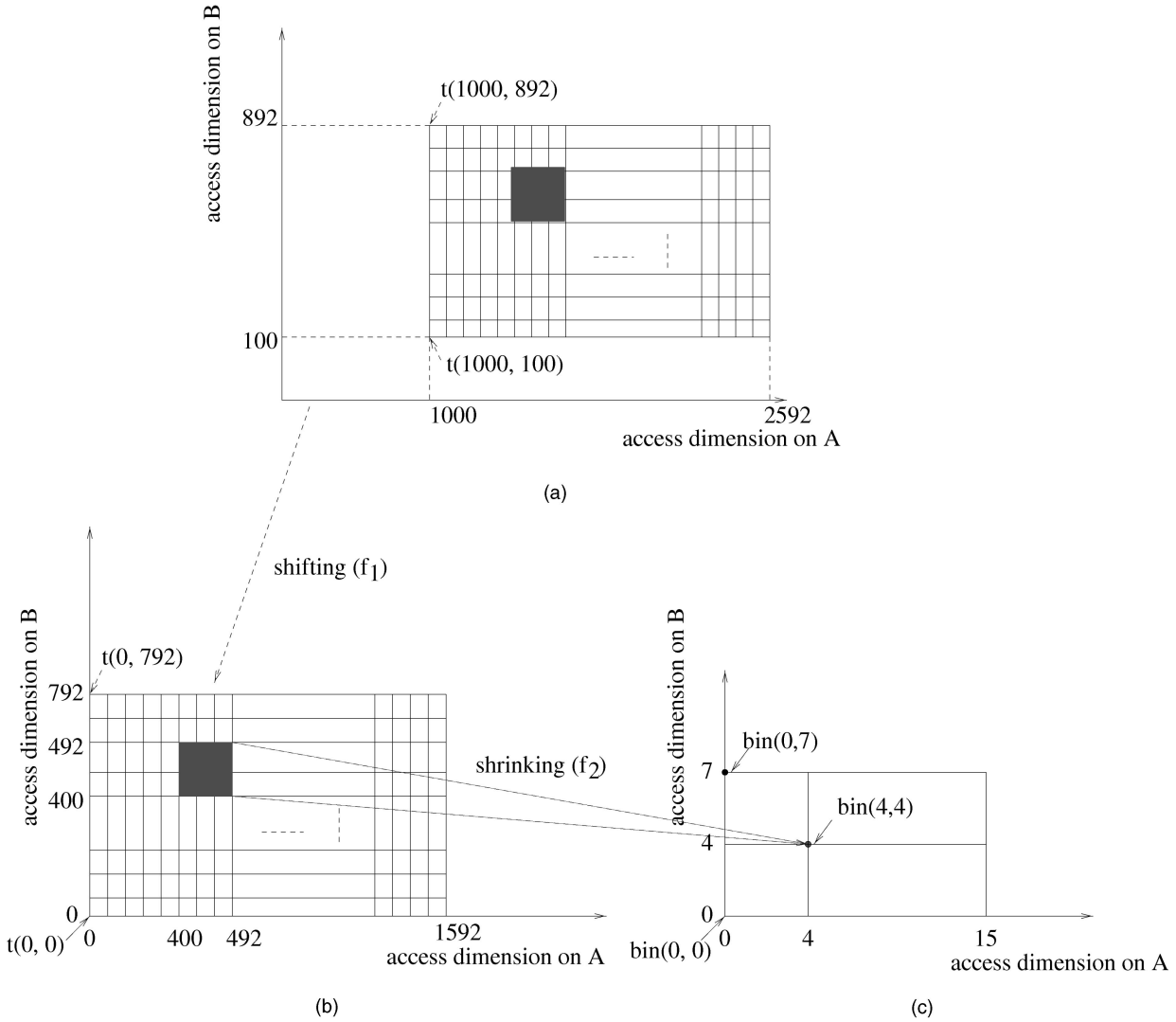


Fig. 6. Equally shrinking a memory-access space. (a) Abstract 2D memory-access space. (b) Shifting memory-access space by function $f_1 : f_1(x, y) = (x - 1,000, y - 100)$; (c) Shrinking memory-access space by function $f_2 : f_2(x, y) = (x/1,000, y/100)$ on a cache of 200 bytes.

space, each point is associated with a task bin which holds all the tasks that are mapped into it.

In Fig. 6, the shrinking procedure of the memory-access space is exemplified by the 2D memory-access space given in Fig. 5. Before shrinking, the original memory-access space is shifted to origin point $(0, 0)$ (see Fig. 6b). The shifting function is shown in Fig. 6b. Then each dimension of the shifted memory-access space is shrunk by C/2 into a new 2D bin space in Fig. 6c. The tasks in the shadow square in Fig. 6b would not access larger space than the cache size, and are mapped onto one point in the bin space. All the tasks in a bin can be grouped together to execute.

2.3 Task Partitioning

After shrinking the n-dimensional memory-access space, tasks have been grouped based on locality affinity information in an n-dimensional bin space. Task partitioning is aimed at partitioning the n-dimensional bin space into p partitions (p is the number of processors and each partition is an n-dimensional polyhedron) by achieving:

1. Optimally partitioning tasks into p parties: $P_1, P_2, ..., P_p$ so that

$$\sum_{(1 \le i,j \le p) \wedge (i \ne j)} |sharing(P_i, P_j)|$$

   is minimized, where $sharing(P_i, P_j)$ is called sharing degree measured by the boundary space among the partitions. This optimization is aimed at minimizing cache coherence overhead.
2. Balancing the task volumes among p partitions.

The major function of partitioning an n-dimensional bin space $B^n(0 : L_1, 0 : L_2, \cdots, 0 : L_n)$ is to find a partitioning vector $\vec{k}(k_1, k_2, \cdots, k_n)$ so that the above two conditions are satisfied. Because finding an optimal partitioning vector is an NP-complete problem, we propose a heuristic algorithm based on the following partitioning rules. Detailed proofs can be found in [24].

Theorem 1. Ordering Rule. *For a given partitioning vector $\vec{k}(k_1, k_2, \cdots, k_n)$ not in decreasing order, by sorting $\vec{k}$ in*

decreasing order, the sharing degree of the resulting vector is at least as low as that of $\vec{k}$.

Theorem 2. Increment Rule 1. *For an* n*-dimensional bin space* B^n*, and partitioning vectors* $\vec{k}(k_1, k_2, \cdots, k_i, k_{i+1} \times q, 1, \cdots, 1)$ *and* $\vec{k'}(k_1, k_2, \cdots, k_i \times q, k_{i+1}, 1, 1, \cdots, 1)$*, where* $q > 1$*,* $\vec{k}$ *has smaller shared data regions than that of* $\vec{k'}$ *if and only if*

$$k_i \times L_{i+1} > k_{i+1} \times L_i.$$

Corollary 1. Increment Rule 2. *For an* n*-dimensional bin space* B^n*, and partitioning vectors* $\vec{k}(k_1, k_2, \cdots, k_i, k_{i+1}, 1, \cdots, 1)$ *and* $\vec{k'}(k_1, k_2, \cdots, k_i \times k_{i+1}, 1, 1, \cdots, 1)$*, where* $k_{i+1} > 1$*,* $\vec{k}$ *has smaller shared data regions than that of* $\vec{k'}$ *if and only if*

$$k_i \times L_{i+1} > \times L_i.$$

Based on the above three rules, we design an efficient heuristic algorithm as follows:

1. Factoring p, the number of processors, we generate all the prime factors of p in decreasing order. Assume that there are q prime factors: $r_1 \geq r_2 \geq \cdots \geq r_q$. Initially, the n-dimensional partitioning vector $\vec{k}$, stored in $k[1:n]$, is $(1, 1, \cdots, 1)$ for the bin space $B^n(0 : L_1, 0 : L_2, \cdots, 0 : L_n)$.
2. Let index $last$ be a chosen position in $k[1:n]$ where $k[i] > 1$ for $i < last$ and $k[i] = 1$ for $i \geq last$. Initially, $last = 1$. For each prime factor r_j where j increases from 1 to q, do the following:
   a. When ($last \leq n$), use the increment rule 2 to determine whether r_j should be put in $k[last]$. Based on the ordering rule, the best place to put r_j must be in $k[1 : last]$. So, we use increment rules to find a better place in $k[1 : last]$. If so, $last$ is increased by 1 and go back; otherwise, use the increment rule 1 to put r_j together with $k[last - 1]$ or $k[last - 2]$, then reorder $k[1 : last - 1]$ in decreasing order and go back.
   b. Otherwise, use the increment rule 1 to put r_j together with $k[last - 1]$ or $k[last - 2]$, then reorder $k[1 : last - 1]$ in decreasing order and go back.

The above algorithm has a computational complexity $O(n + \sqrt{p})$. After the determination of a partitioning vector, the bin space is partitioned into multiple independent spaces that are further reconstructed in an $(n+1)$-dimensional space. An example of this procedure is shown in Fig. 7, where the bin space produced in Fig. 6 is partitioned by vector $(2, 2)$. The partitions in Fig. 7a are first transformed into four independent spaces in Fig. 7b, which are further transformed into a 3D space shown in Fig. 7c. The 3D space in Fig. 7c is implemented as a 3D hash table where task bins in each partition are chained together to be pointed by a record in a Task Control Linked (TCL) list. The hashing of tasks into the hash table is performed by the space transformation functions. For detailed information of these functions, the interested readers may refer to [24].

2.4 Locality-Preserved Adaptive Scheduler

The dynamic scheduler in the runtime system is aimed at minimizing the parallel computing time of a set of data-independent tasks created in the task reorganization step. The task groups generated in the task reorganization step are locality oriented. This may not guarantee that all the partitions in the second step have the same execution time, due to the following possible reasons:

1. Irregular data access patterns of programs will generate different amount of tasks in each partition;
2. irregular computation patterns will directly result in different execution times among the partitions;
3. possible interference of a paging system in the operating system will also generate different amount of tasks in each partition; and
4. different data locality exploited in different partitions may speed up their executions at different rates.

The scheduling problem of parallel loops in shared memory systems has been studied in a wide range (e.g, [13], [17], [25]). In [25], we proposed an adaptive algorithm to balance workload among multiple processors while exploiting the affinity of parallel tasks to processors, which was shown to outperform many existing algorithms for a wide range of applications. The task scheduling problem here is different from the traditional loop scheduling problem, because the tasks are locality dependent. To schedule a set of locality dependent tasks, the scheduler must take advantage of the locality exploited in the task reorganization phase as much as possible, while balancing workload to minimize the parallel computing time. Here, we extend our linearly adaptive algorithm proposed in [25] to address this issue in the runtime system. The extended algorithm is called the Locality-preserved Adaptive Scheduling algorithm, denoted as LAS. Because the number of processors in the targeted SMP system is in the range of small scale to medium scale, the linearly adaptive algorithm is aggressive enough to reduce synchronization overhead.

Initially, the ith task group chain in the TCL list is considered as the local task chain for processor i, for $i = 1, 2, \cdots, p$ (p is the total number of processors). This initial allocation maintains the minimized data sharing among processors. The remaining number of tasks in the local chain of processor i is tracked by the corresponding TCL counter variable, denoted as $Count_i$, which is used in the LAS algorithm to estimate load imbalance, just like what the processor speed variables do in [25]. However, the TCL counter variables are different from the processor speed variables. A processor speed variable records the number of tasks that have been finished in the corresponding processor, which can precisely estimate how many tasks remain in the processor because the tasks are evenly partitioned among processors initially. It does not work in the runtime system, because the task chains in the TCL list may contain imbalanced numbers of tasks. In addition, each processor has a chunking control variable of initial value of p, denoted K_i for processor i, to determine how many tasks to be executed at each scheduling step.

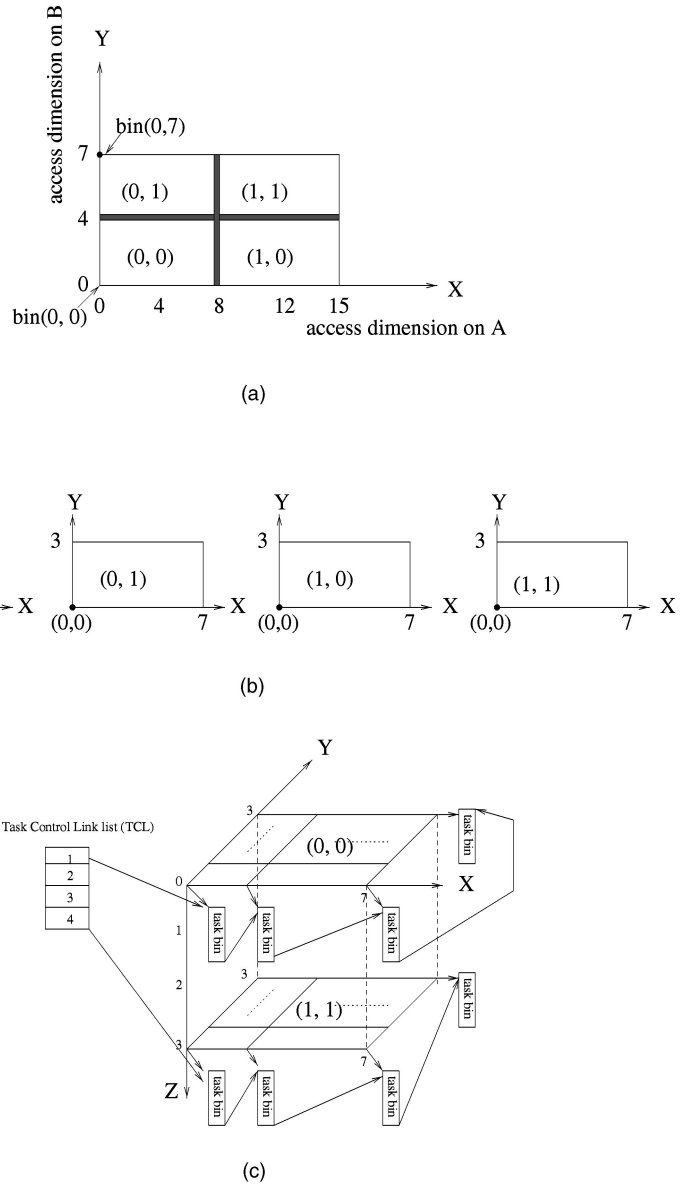


Fig. 7. Partitioning: the bin space is evenly divided into 4 partitions from X and Y dimensions. (a) Indexing of partitions. (b) Independent address space of each partition. (c) 3D internal representation of the memory access space.

The LAS algorithm still works in two phases: the local scheduling phase and the global scheduling phase. However, it considers a task group as a minimal schedulable unit, never breaking up the tasks in a group. The LAS algorithm can take good advantage of locality-oriented task assignments while achieving a good load balance because the tasks are usually fine-grained. All the processors start at the local scheduling phase. The algorithm is described for processor $i (i = 1, 2, \cdots, p)$ as follows.

Local scheduling. Processor i first calculates its load status relative to the other processors as follows:

$$\text{heavy} \quad \text{if } Count_i > \sum_{j=1}^{p} Count_j/p + \alpha \tag{2}$$

$$\text{light} \quad \text{if } Count_i < \sum_{j=1}^{p} Count_j/p - \alpha \tag{3}$$

$$\text{normal} \quad \text{otherwise.} \tag{4}$$

Here, α is $\lceil \sum_{j=1}^{p} Count_j/p)/(2p) \rceil$, which decreases with execution to control the load distribution more closely.

During the above computation, if the number of remaining tasks in one processor's local chain is found to be 0, i.e., $\exists_{j \in [1,p]}(Count_j = 0)$, processor i sets its chunking control variable, K_i, to p, then goes to the global scheduling phase. Otherwise, processor i linearly adjusts its chunking control variable according to its load status as follows:

$$K_i = \begin{cases} \max\{p/2, K_i - 1\} & \text{if its load is light} \\ \min\{2p, K_i + 1\} & \text{if its load is heavy} \\ K_i & \text{otherwise.} \end{cases} \tag{1}$$

The varying range $[p/2, 2p]$ for the chunking control variables has been shown to be safe for balancing the load [13], [17], [25]. Then processor i gets the following number of task groups from its local chain:

$$\lceil Count_i/(K_i * g_s) \rceil,$$

where g_s is the size of a task group. Having finished the allocated tasks, processor i goes back to repeat the local scheduling.

Global scheduling. First, processor i always gets $Count_i/(K_i \times g_s)$ groups of tasks from its local task chain to execute until its local task chain is empty. Then, processor i gets $1/(K_i \times g_s)$ groups of the remaining tasks in the local task chain of the most heavily loaded processor until all the processors empty their local task chains.

In the local scheduling phase of the LAS algorithm, tasks are executed in a bin-by-bin order. Only when a processor is trying to help some processors in the global phase are tasks in a bin allocated by groups to execute remotely. In the global scheduling phase, emphasis is put on load balancing.

2.5 The Application Programming Interface (API)

The interface functions are mainly used to provide application-dependent hints for the run-time system. The current system is implemented in C language. The API of the system is simple, and consists of the following three functions:

1. `void cacheminer_init(int csize, float f, int p, int n, long` s_1`, void` $* b_1, \cdots,$ `long` s_n`, void` $* b_n$). This function provides the following types of hints:
   - Cache size `csize` gives the size in Kbytes of the secondary cache on each processor.
   - Task control parameter `f` is a floating point number in $(0, 1]$, giving the usage percentage of a cache to cluster tasks.
   - `p` is the number of processors.
   - `n` is the total number of arrays, on which hints are provided.
   - Hints on arrays are given in pairs. Each pair of s_j and b_j $(j = 1, 2, \cdots, n)$, give respectively the size and the starting physical addresses of a referenced array by tasks. All the arrays are arranged in a user-defined order. (There is no specific requirement on the array order.)

   Based on the information provided by this function, the runtime system builds a $(n+1)$-dimensional hash structure.
2. `task_create(void (* fun), int m, int` $t_1, \cdots,$ `int` t_m, `void` $*a_1, \cdots,$ `void` $*a_m$). This function creates a task with its computing function, denoted as `void fun` $(t_1, t_2, \ldots, t_m)$, and carries hints $a_1, a_2, ..., a_m$ on the access pattern of the task into the runtime system. Here a_i $(i = 1, \cdots,$ m) is the starting access address of the task on the ith array.

   If the number of hints, m, is larger than the number of hinted data arrays, n, only the first n hints are used. This flexibility would allow a task function to have a larger number of parameters than that of the accessed data arrays. However, m cannot be smaller than n, which is easily achieved in programming by using dummy parameters in a task function. The order of hints here must be the same as that of those hints presented in `cacheminer_init`, i.e., a_j, s_j, and b_j are hints on the same array for $j = 1, 2, \cdots, n$.
3. `void task_run(int repeat)` This function starts the runtime system to execute the tasks in the hash structure in parallel. If the tasks are going to execute at second time, the variable `repeat` is set to 1 so that the runtime system can keep the hash structure in order to eliminate the overhead of rebuilding it. In this situation, the runtime system exploits cache locality by using existing partitions of tasks. Otherwise, the variable `repeat` is set to 0.

2.6 Classifications of Application Programs

In order to reflect all types of applications while not getting into exhaustive investigation which is not necessary, we classify applications based on three factors: computation pattern, memory-access pattern, and data-dependence pattern. Computation patterns can be classified as two types: regular, where the computation tasks of an application are naturally balanced, and irregular, where the computation tasks of an application are not naturally balanced. Furthermore, memory-access patterns and data-dependence patterns can be respectively classified into static patterns that can be determined at compile-time, and dynamic patterns which can not be determined at compile-time. Intuitively, based on these patterns, applications can be classified into eight possible types. However, the computation pattern, the memory-access pattern, and the data-dependence pattern interact one another. Usually, when an application has a dynamic memory-access pattern or a dynamic data-dependence pattern, it has an irregular computation pattern. In addition, the memory-access pattern of an application is affected by its data-dependence pattern. When the data-dependence pattern can not be determined at compile-time, its memory-access pattern must not be determined, either. Considering these effects, applications finally fall into the following four types, listed in increasing difficulty degree for locality optimization.

- **Type 1.** Applications with regular computation patterns, static memory-access patterns, and static data dependence.
- **Type 2.** Applications with irregular computation patterns, static memory-access patterns, and static data dependence.
- **Type 3.** Applications with irregular computation patterns, dynamic memory-access patterns, and static data dependence.
- **Type 4.** Applications with irregular computation patterns, dynamic memory-access patterns, and dynamic data dependence.

Based on the above classification, we choose each benchmark from the first three application types that fit into our programming model. For the most difficult

```
double C[N][N], A[N][N], B[N][N];

dense_mm()
{       int i, j, k;
        for (i=0; i<N; i++)
           for (j=0; j<N; j++)
             for (k=0; k<N; k++)
                C[i][j] += A[i][k]*B[j][k];
}
```

```
double C[N][N], A[N][N], B[N][N];

dmm_rt(float f, int p)
{       int i, j, k;
        long s = N*N*sizeof(double);
        cacheminer_init(Csize, f, p, 4, s, a[0], s, b[0]);
        for (i=0; i<N; i++)
            for (j=0; j<N; j++)
                task_create(fun,4, i, j, A[i], B[j]);
        task_run(0);
}
fun(int i, int j)
{       int k;
        for (k=0; k<N; k++)
           C[i][j] += A[i][k]*B[j][k];
}
```

Fig. 8. Dense matrix multiplication: the sequential program is given on the left and the parallelized version on the runtime system is given on the right.

```
int A[N*N], B[N*N], C[N*N];

ac(int x)
{       int i, j;
        for (i=0; i<N*N; i++)
         for (j=i; j<N*N; j++)
          A[i] += x*B[j]*C[j-i];
}
```

```
int A[N*N], B[N*N], C[N*N];

ac(int x, float f, int p)
{       int i;
        long s = N*N*sizeof(double);
        cacheminer_init(Csize, f, p, 4, s, &B[0], s, &C[0]);
           for (i=0; i<N*N; i++)
             task_create(fun, 4, i, x,& B[i], &C[0]);
           task_run(0);
   }
   fun(int i, int x)
   {       int j;
             for (j=i; j<N*N; j++)
               A[i] += x*B[j]*C[j-i];
     }
```

Fig. 9. Adjoint convolution (AC): the sequential program is given on the left and the parallelized version on the Cacheminer system is given on the right.

applications in Type 4, our technique can also be used to improve memory performance by combining some data-dependence detection techniques. Next, we will show how to use the Cacheminer to rewrite ordinary C programs to exploit different types of cache localities in a SMP system. These applications are used to evaluate the runtime system.

2.6.1 Programming Examples

The first program is dense matrix multiplication (DMM) with a regular memory access pattern, which belongs to program Type 1. The sequential program and its parallelized version on the runtime system are presented in Fig. 8. The array `B` is assumed to have been transposed so that the cache locality of the sequential program is optimized. In the parallelized version, the innermost loop is treated as a task function with two parameters, i and j. The two outer loops are data independent, which create N^2 tasks with the same computation load. Each task reads, respectively, a row of A and B.

The second application is adjoint convolution (AC), which belongs to program Type 2. The sequential program is given on the left in Fig. 9. The outer loop is data independent and the inner loop is data dependent. So, we parallelize the outer loop by creating $N \times N$ tasks, each task executes the inner loop. The parallelized version on the Cacheminer system is shown on the right in Fig. 9. Each task accesses respectively a contiguous partion of arrays B and C. Different tasks have different sizes of work sets so that the tasks have imbalanced load. Hence, cache locality must be traded off with load balance to minimize parallel execution time. This application is more difficult than the DMM for a compiler to exploit its cache locality.

The third application is Sparse Matrix-Matrix (SMM) multiplication with dense representation, which belongs to program Type 3. The elements of two source matrices are input at runtime. To simplify the presentation, we assume that two source $M \times M$ sparse matrices have been stored in one-dimensional arrays, `A` and `B`, respectively. The elements in `A` are stored by rows and the elements in `B` are stored by columns. Array `Arow` gives the position in `A` of the first nonzero element of each row for the first source matrix. Array `Acol` gives the column number of each nonzero element in `A`. Similarly, array `Bcol` gives the position in `B` of the first nonzero element of each column for the second source matrix. Array `Brow` gives the row number of each element in `B`. The sequential sparse matrix multiplication is described on the left side of Fig. 10. The two outer loops are data independent and are parallelized on the runtime system as described by the program presented on the right side of Fig. 10. The created tasks have irregular memory access patterns so that tasks have an imbalanced workload,

```
double A[X], B[Y], C[M][M];
int Arow[M+1], Acol[X], Bcol[M+1], Brow[Y];

sparse_mm()
{       int i, j, k, r, start, end;

        for (i=0; i<M; i++)
         for (j=0; j<M; j++){
          start = Bcol[j]; end = Bcol[j+1];
          for (k=Arow[i]; k<Arow[i+1]; k++)
           for (r=start; r<end; r++)
            if (Acol[k] == Brow[r]){
              C[i,j] += A[k]*B[r];
              start = r + 1;
              break;
            }
         }
}
```

```
double A[X], B[Y], C[M][M];
int Arow[M+1], Acol[X], Bcol[M+1], Brow[Y];

sparse_mm(float f, int p)
{       int i, j, s = sizeof(double);

        cacheminer_init(Csize, f, p, 4, X*s, &A[0], Y*s, &B[0]);
        for (i=0; i<M; i++)
          for (j=0; j<M, j++)
            task_create(fun, 4, i, j, &A[Arow[i]], &B[Bcol[j]]);
        task_run(0);
}

fun(int i, int j)
{       int k, r, start, end;
        start = Bcol[j]; end = Bcol[j+1];
        for (k=Arow[i]; k<Arow[i+1]; k++)
           for (r=start; r<end; r++)
             if (Acol[k] == Brow[r]){
                   C[i][j] += A[k]*B[r];
                   start  = r + 1;
                   break;
             }
}
```

Fig. 10. Sparse matrix-matrix multiplication: the left side is the sequential program version and the right side is the rewritten version on the runtime system.

which is closely related to the input matrices. The elements of arrays A and B accessed by each task are determined at runtime. This application is hard for a compiler to exploit cache locality. In our experiments, two sparse matrices, A and B, are set to have 30 percent nonzero elements, which are randomly generated from a uniform distribution.

3 Performance Evaluation Method and Environments

This section introduces our performance evaluation method and evaluation environments. The runtime system was first implemented and tested on an event-driven simulator built on MINT [21]. The preliminary results were reported in [26]. We have recently implemented the system on an SimOS simulated SMP, a much more reliable environment for performance evaluation. We have also measured the performance of the system on two commercial SMP multiprocessors.

3.1 Evaluation Method and Metrics

The performance evaluation of the runtime system was conducted using simulation and measurement. Simulation was used to study the effectiveness of the runtime system in exploiting the cache locality of applications with respect to the changes of the cache miss rate, bus traffics, execution time, and cache interference. Measurements were used to further verify the effectiveness of the runtime technique on given commercial SMP systems. Miss rate, load coefficiency (i.e., the ratio of deviation to mean), and execution time are three metrics used in the performance evaluation. Cache misses are classified as the following three types:

Compulsory misses. Misses caused by reads or writes on data that have never been brought into the cache before.

Replacement misses. Misses caused by reads or writes on data that were brought into the cache but replaced by other block data at the most recent time.

Coherence misses. Misses caused by reads or writes on data that were brought into the cache but invalidated by other processors at the most recent time.

The total number of the first two types of misses is a good measure of the data reuse in caches. The last type of misses evaluate the data sharing degree among processors.

The selected benchmark programs are the DMM, the AC, and the SMM, which are described in Section 2.6.1. Their optimized versions, which exploit locality on the runtime system, are denoted as DMM_LO, AC_LO, and SMM_LO, respectively. We parallelized the three programs by using well-known compiler optimizations or runtime techniques as follows:

1. For the dense matrix-matrix multiplication program, we used the sequential block algorithm proposed by Wolf and Lam [23]. This algorithm has been shown to effectively exploit cache locality. The locality of the block algorithm is further improved by transposing one matrix so that the innermost loop accesses contiguous memory regions on the two arrays. Based on this transposed block algorithm, a parallelized algorithm, denoted as DMM_WL presented in Fig. 11, is given by uniformly partitioning computations on multiple processors, so that good load balance is achieved.
2. For the adjoint convolution program, each iteration of the outermost loop accesses a contiguous segment of arrays A and C, respectively. In order to investigate the effect of locality optimization, we assume that arrays A and C are too large to fill in a cache. Two iterations of the outer loop that have

```
int b = n/p; /* here matrix size n is assumed to be evenly divided
                by the number of processors. */

for (kk=0; kk<n; kk+=bf)
    for (jj=0; jj<n; jj+=bf)
        for (i=pid*b; i<(pid+1)*b; i++) /* task partition */
            for (j=jj; j<min(jj+bf, n); j++){
                d = A[i][j]; /* d is register type */
                for (k=kk; k<min(kk+bf, n); kk++)
                    d += B[i][k] * C[j][k];
                A[i][j] = d;
            }
```

Fig. 11. A well-tuned parallel version of the DMM program published in [23]. Here, `pid` is a thread id of value from 0 to `p-1`. `p` is the number of threads.

closer values of index `i` have larger overlap between their data sets. From the standpoint of optimizing cache locality, the AC program should be parallelized by using the blocking technique to chunk the outermost loop. By this, each processor will be allocated with a set of adjacent outermost iterations. Because the iterations of the outermost loop have decreasing workload as index `i` decreases, a varying-sized blocking technique should be used to optimize both locality and load balance. For any given `N` and the number of processors, it is difficult or impossible to choose a set of different block sizes to balance load among processors. Here, we have integrated several compiler optimizations to solve this problem. At first, we equally split the outermost loop into two loops and reversed the computations of the second loop, as shown in Fig. 12b. Then, the second loop was aligned and fused with the first loop to make a new loop with balanced iterations, which is illustrated by Fig. 12c and Fig. 12d. The new loop can be equally blocked onto multiple processors to maintain both load balance and cache locality. By comparing with this parallel program, denoted AC_BF, we evaluated the quality of our runtime technique which optimizes load balance and exploits cache locality.

3. The SMM program has an irregular memory access pattern that is determined by runtime input data. This type of application is very hard for compiler-based techniques to effectively conduct partition and cache locality optimization. Here we use the linearly adaptive scheduling technique, proposed in [25], to schedule the executions of parallel iterations in the SMM where parallel iterations are initially created as parallel tasks in multiple task queues. The adaptive runtime scheduling technique has been shown to outperform previous runtime scheduling techniques for imbalanced parallel loops [25]. Hereafter, We denote this parallelized version as SMM_A. Although the SMM_A has a similar execution procedure to the SMM_LO, three significant differences are: 1) Initially, the SMM_LO groups and partitions tasks with the objective of minimizing data sharing between partitions and maximizing data reuse in a partition. The SMM_A just cyclically puts tasks in local queues of processors. The SMM_LO has higher runtime scheduling overhead than the SMM_A. 2) Although both the SMM_LO and the SMM_A use the linearly adaptive scheduling algorithm, the scheduling in the SMM_LO is locality-oriented, which has a better chance to reduce its number of memory accesses. Compared with the SMM_A, the SMM_LO is expected to further reduce execution time by optimizing memory performance on modern computers.

The last issue is how to select the problem sizes of the tested programs. Because our goal is to evaluate the effectiveness of the runtime system in exploiting cache locality, we select the problem sizes based on the underlying cache size so that the data set of a program is too large

```
for (i=0; i<N*N; i++)
    for (j=i; j<N*N; j++)
        A[i] += x*B[j]*C[j-i];
```

(a)

```
for (i=0; i<N*N/2; i++)
    for (j=i; j<N*N; j++)
        A[i] += x*B[j]*C[j-1];
for (i = N*N/2; i<N*N; i++) {
    k = 3*N*N/2 - i - 1;
    for (j=k; j<N*N; j++)
        A[k] += x*B[j]*C[j-k];
}
```

(b)

```
for (i=0; i<N*N/2; i++)
    for (j=i; j<N*N; j++)
        A[i] += x*B[j]*C[j-i];
for (i=0; i<N*N/2; i++){
    k = N*N - 1 - i;
    for (j=k; j<N*N; j++)
        A[k] += x*B[j]*C[j-k];
}
```

(c)

```
for (i=0; i<N*N/2; i++){
    for (j=i; j<N*N; j++)
        A[i] += x*B[j]*C[j-i];
    k = N*N - 1 - i;
    for (j=k; j<N*N; j++)
        A[k] += x*B[j]*C[j-k];
}
```

(d)

Fig. 12. A well-tuned parallel version of the AC application. (a) Original AC. (b) Splited AC with reversed execution order in the second loop. (c) Align the second loop with the first loop. (d) Fused AC.

enough for the cache to hold. In order to shorten simulation time without losing the confidence of simulation results, we selected relatively small problem sizes for the programs. We scaled down the cache size accordingly for these programs. These selections can prevent the advantage of the runtime system from being shadowed by hardware caches, because on a given commercial system, cache capacity is fixed but the problem size of an application varies.

3.2 The SimOS Simulation Environment

The SimOS [18] is used to simulate a bus-based cache coherent SMP system. SimOS is a machine simulation environment developed by Stanford University. SimOS simulates the complete hardware of a computer system booting a commercial operating system and running a realistic workload, such as our application programs supported by the Cacheminer runtime system, on top of it. The simulator contains software simulation of all the hardware components of a computer system: processors, memory management units, caches, memory systems, as well as I/O devices such as SCSI disks, Ethernets, hardware clocks, and consoles. The current version of SimOS simulates the hardware of MIPS-based multiprocessors to run Silicon Graphics' IRIX operating system. The cache coherence protocol of the simulated SMP is write-invalidate.

The simulated SMP is very similar to popular commercial SMPs. The simulated SMP consists of a number of 200 MHz R10000 processors, each of which has 8 KB instruction cache, 8 KB data cache, and a unified 64 KB L2 cache. The size of the memory is set to 64 MB. The access times to L1, L2 caches and the shared-memory are 2, 7, and 100 cycles, respectively. As the memory size is scaled down eight times, so is the workload. This has significantly reduced the simulation time.

3.3 HP/Convex S-class and Sun Hyper-SPARC Station-20

The HP/Convex S-class [3] is a crossbar-based cache coherent SMP system with 16 processors, while the Sun Hyper-SPRACstation-20 is a bus-based cache coherent SMP system with four processors. The architectural differences of these two SMP systems provide the runtime system with different opportunities/challenges to improve the performance of applications.

The HP/Convex S-class has 16 PA8000 processors of 720 peak MFLOPS. A PA8000 is a four-way super-scalar RISC processor, supporting 64-bit virtual address space, which operates at 180MHz. A PA8000 processor has a single level primary cache with separate instruction cache and data cache of size 1 Mbytes each. The cache is direct-mapped using a write back policy, which has cache line size of 32 bytes and cache hit time of three cycles (about 16.7 nanoseconds). Cache coherence is maintained by a distributed directory-based hardware cache coherent protocol. The S-class has a pipelined, 32-way interleaved shared memory of eight memory boards, which is interconnected with processors by a 8×8 nonblocking crossbar. The data path from the crossbar to the memory controller is 64-bits wide and operates at 120 MHz. The access latency of a 32-byte cache line from the shared memory to a cache is 509 nanoseconds. The ratio of cache miss time to cache hit time is about 30.

TABLE 1
Architectural Parameters of the Simulation

Parameter	Value
1-Level instruction cache size	8k
1-Level data cache Size	8k
2-Level cache size (uniform)	64K
2-Level hit time	7 cycles
Memory size	64M
Memory access latency	100 cycles
Bus bandwidth	528 MB/s

Our Sun Hyper-SPARCstation-20 has four hyperSPARC processors operating at 100MHz. Each processor has a two-level cache hierarchy: a 64 Kbyte on-chip cache and a 256 Kbyte virtual secondary cache where the cache line size is 64 bytes. Compared with the S-class, the larger cache line of the Hyper-SPARCstation-20 exploits better spatial locality for applications. The cache hit time is about 300 nanoseconds. A cache miss time is about 13,360 nanoseconds. The ratio of cache miss time to cache hit time is about 36. Cache coherence is maintained by the well known bus-based snooping protocol.

Compared with HP/Convex S-class with respect to instructions issuing rate and memory access latency, the Sun Hyper-SPARCstation-20 is much slower. In measurement, we focused on the comparisons of relative performance results.

4 Performance Evaluation

4.1 Simulation Results

The architectural parameters of the simulation are shown in Table 1. The selected array sizes of programs DMM, AC, and SMM are 256×256, 16,384, and 512×512, respectively, which correspond to working set sizes 1,536 Kbytes, 384 Kbytes, and about 1,350 Kbytes, respectively. The cache block size is 32 bytes.

4.1.1 Cache Performance

Table 2 comparatively presents the cache performance of the parallel programs optimized by different techniques. Regarding regular program DMM, the locality optimized parallel version (DMM_LO) using the runtime technique is 10 percent higher than the well-tuned version (DMM_WL) in the number of cache misses. Both versions had similar performance with respect to their compulsory misses and invalidations. As the number of processors increases, the numbers of compulsory misses and invalidations in both optimized versions increase, but cache replacement performance is also improved. The former is due to the increase in sharing degree between caches. The latter is due to the use of more caches. This is consistent with previous research work. The measured speedup for SMM_A is 1.81, 3.04, and 4.67 on two, four, and eight processors, respectively, compared with the sequential time of DMM_WL.

TABLE 2
Cache Performance Comparisons

Dense matrix-matrix multiplication

Processors	DMM_WL					DMM_LO				
	misses	miss-rate	comp.	rep.	inv.	misses	miss-rate	comp.	rep.	inv.
2	856K	0.25%	7K	849K	4.4K	910K	0.27%	52K	858K	3.0K
4	881K	0.25%	55K	826K	10.3K	923K	0.27%	93K	829K	8.2K
8	952K	0.27%	103K	849K	17K	1017K	0.30%	180K	837K	16.8K

Adjoint convolution

Processors	AC_BF					AC_LO				
	misses	miss-rate	comp.	rep.	inv.	misses	miss-rate	comp.	rep.	inv.
2	24.5M	0.73%	13K	24.5M	8.4K	22.2M	0.71%	17K	22.2M	9.2K
4	24.5M	0.73%	27K	24.5M	15.6K	22.2M	0.71%	28K	22.2M	19.4K
8	24.5M	0.75%	53K	24.5M	19K	22.2M	0.71%	48K	22.2M	26.4K

Sparse matrix-matrix multiplication

Processors	SMM_A					SMM_LO				
	misses	miss-rate	comp.	rep.	inv.	misses	miss-rate	comp.	rep.	inv.
2	5.19M	0.53%	127K	5.06M	6.9K	2.97M	0.30%	143K	2.82M	6.7K
4	5.16M	0.53%	230K	4.93M	14.7K	2.97M	0.30%	250K	2.69M	18.3K
8	5.16M	0.53%	393K	4.77M	22.8K	2.97M	0.30%	410K	2.61M	28.1K

`misses`, `comp.`, and `inv.`, respectively, give the numbers of misses, compulsory misses, and replacement misses. K=10^3 and M=10^6. DMM_LO was measured with shrinking factor f = 1.

The memory access pattern of the AC program is not as regular as that of the DMM program. The AC_LO, a locality optimized version using the runtime technique, is shown to achieve moderately better cache performance than the AC_BF, a well-tuned version. The numbers of replacement misses were improved by the runtime technique. The cache performance of both parallel versions do not show a significant change when the number of processors is increased. This is different from the execution of the DMM program. The measured speedup of AC_LO is 1.99, 3.94, and 7.73 on two, four, and eight processors, respectively.

Regarding program SMM, the runtime locality technique is shown to be highly effective in reducing cache misses. This reduction is mainly contributed by a significant reduction in replacement cache misses. Both parallel versions present similar invalidation performance. These results show the great potential of optimizing the cache locality using runtime techniques for applications with dynamic memory access patterns. The measured speedup of SMM_LO is 1.88, 3.33, and 5.00 on two, four, and eight processors, respectively.

4.1.2 Execution Performance and Bus Traffic

In Table 3, the execution performance of different parallel versions are presented. The overall performance of a program was measured by its execution time. The performance differences between different parallel versions can be clarified by the differences in bus traffic and load balance quality.

Regarding program DMM, the DMM_WL slightly outperformed the DMM_LO. This is mainly because the DMM_LO had worse load balance and longer data transfer time. Although the parallel iterations were perfectly partitioned among multiple processors in the DMM_WL, slight load imbalance was also observed. Regarding program AC, the AC_LO outperformed the AC_BF from 5 percent to 7 percent in terms of execution time. This improvement was also mainly contributed by certain reductions in load traffic. The AC_LO had worse load balance than that of the AC_BF because the AC_LO was trying to balance load based on pre-grouped tasks. However, this imbalance does not impact the overall performance significantly. This also shows that locality optimization is more important in this case. Regarding program SMM, the SMM_LO performed much better than the SMM_A by reducing 8 to 16 percent execution time using the runtime technique.

4.1.3 Runtime Overhead

Runtime overhead is another important factor which affects the effectiveness of a runtime technique. In our proposed runtime technique, runtime overhead is mainly caused by task organization and task runtime scheduling. The task organization overhead is affected by the number of tasks created at runtime and the number of arrays accessed. The runtime scheduling overhead is affected by the imbalance in the initial task partition and in the runtime executions of multiple processors. Table 4 gives the percentage of the runtime overhead in the total execution time. For both the DMM_LO and the SMM_LO, the runtime overhead had a bigger influence on execution performance than the AC_LO. This difference is mainly due to the difference in the computation granularities of tasks. The tasks in the

TABLE 3
Execution Performance and Bus Traffic

Dense matrix-matrix multiplication

Processors	DMM_WL					DMM_LO				
	execution	balance(%)	load	writeback	upgrade	execution	balance(%)	load	writeback	upgrade
2	3.31	0.3%	164M	20M	2.0M	3.45	0.4%	176M	21M	2.4M
4	1.98	0.7%	173M	20M	2.1M	2.05	0.9%	181M	21M	2.5M
8	1.25	3.9%	185M	20M	2.3M	1.34	4.2%	201M	21M	3.0M

Adjoint convolution

Processors	AC_BF					AC_LO				
	execution	balance(%)	load	writeback	upgrade	execution	balance(%)	load	writeback	upgrade
2	10.6	0.0%	3135M	25M	11M	10.1	0.2%	3311M	34M	19M
4	5.4	0.1%	3123M	23M	11M	5.1	0.4%	3297M	32M	17M
8	2.8	0.3%	3114M	23M	11M	2.6	0.9%	3405M	33M	17M

Sparse matrix-matrix multiplication

Processors	SMM_A					SMM_LO				
	execution	balance(%)	load	writeback	upgrade	execution	balance(%)	load	writeback	upgrade
2	3.8	0.1%	802M	36M	4.6M	3.2	0.1%	501M	37M	4.9M
4	2.1	0.1%	796M	36M	4.5M	1.8	0.2%	506M	37M	4.8M
8	1.3	1.9%	803M	36M	4.9M	1.2	3.4%	526M	37M	5.4M

All the timing results are given in simulation second; $M = 10^6$. Load balance was measured by the division of time derivation by mean time. The locality optimized programs using our runtime approach use shrinking factor $f = 1$.

AC_LO had the largest computation granularity and the tasks in the SMM_LO had the smallest computation granularity.

4.2 Measurements

4.2.1 Measurements on HP/CONVEX S-class

Measurement results of the different parallel versions on HP/CONVEX S-class are presented in Table 5. Regarding program DMM, the DMM_WL consistently performed a little bit better than the DMM_LO. The better load balance in the DMM_WL is a reason for this. For program AC, the AC_LO performed significantly better than the AC_BF on two processors. When more processors were applied, the execution times were close. But, the AC_BF always balanced loads better due to its perfect initial partition. But, the load imbalance that occurred in the AL_LO was not larger than 1 percent. For the SMM, the SMM_LO had achieved a significant performance improvement over the SMM_A. This further confirms the effectiveness of the runtime technique in improving the performance of applications with dynamic memory access patterns. However, the SMM_A still achieved better load balance than the SMM_LO. One reason for this is that the SMM_LO used a locality preserved scheduling algorithm, which tried to keep the tasks in a group to execute together on a processor. This can increase data reuse in a cache. But, it also tended to cause more imbalance.

TABLE 4
Runtime Overhead in Percentage of Total Execution Time

Applications	Processor		
	2	4	8
DMM_LO	4.0	3.3	2.7
AC_LO	0.2	0.2	0.2
SMM_LO	4.5	4.0	3.3

Table 5 also gives the runtime overhead of the task reorganization. Among all the applications, the SMM_LO had the largest runtime overhead in term of the percentage in the total execution time, and the AC_LO had the lowest. This is consistent with the simulation results. As mentioned before, this is mainly affected by the task granularity. Regarding the effect of different values of f on performance, Table 6 presents the measurement results. For the DMM_LO, the execution time decreased as f decreased, resulting in groups with a smaller number of tasks. The AC_LO is not sensitive to the change of f, which is consistent with our simulation results. The SMM_LO had longer execution time when a smaller f was used.

4.2.2 Measurements on HyperSPARC Station-20

Table 7 gives the execution times of the parallel versions on HyperSPARC station-20, a much slower multiprocessor workstation than the S-class. The DMM_LO still achieved a close performance to the DMM_WL, not worse than 9 percent in execution time. The runtime overhead in the DMM_LO was about 10 percent of its execution time. For program AC, the AC_LO outperformed the AC_A for 8.5 percent in execution time reduction, although it had worse load balance. Compared with the SMM_A, the SMM_LO reduced execution time up to 40 percent. These measurements are consistent with that on the S-class, although the absolute performance results are different.

The effects of different values of f are presented in Table 8. The DMM_LO, the AC_LO, and the SMM_LO achieved the best performance, respectively, at f = 0.25, f = 0.5, and f = 1.

5 Conclusion

The locality of a program is affected by a wide range of performance factors. The design of efficient locality-optimization techniques relies on an insightful

TABLE 5
Execution Time (in Seconds) Based Comparison on HP/Convex S-class

Application: Dense matrix multiplication

size	proc.	DMM_WL		DMM_LO		
		time	balance	time	overhead	balance
1024	2	11	0.0026	13	0.83	0.024
	4	5.7	0.0052	6.6	0.52	0.021
	8	3.0	0.0095	3.9	0.34	0.038
	16	1.8	0.010	2.2	0.24	0.040

Application: Adjoint convolution

size	proc.	AC_BF		AC_LO		
		time	balance	time	overhead	balance
400	2	180	0.0007	144	0.398	0.003
	4	102	0.0010	91	0.235	0.004
	8	65	0.0018	60	0.174	0.006
	16	39	0.0031	38	0.107	0.010

Application: Sparse matrix matrix multiplication

size	proc.	SMM_A		SMM_LO		
		time	balance	time	overhead	balance
1024	2	4.1	0.02	2.2	0.12	0.03
	4	2.5	0.03	1.3	0.11	0.05
	8	1.4	0.04	0.5	0.08	0.06
	16	0.8	0.06	0.5	0.01	0.06

Columns `time` and `overhead`, respectively, give total execution time and task organization overhead in second. `Balance` presents load balance measurements in term of the rate of the time deviation to the mean time. (f = 1).

understanding of these performance factors. This paper presents a runtime approach to exploit cache locality on SMP multiprocessors. We have built a runtime library including the following three functionalities:

1. *Information acquisition*, which collects information on the cache-access pattern of a program. The information on the data-access sequence of a program is essential for locality optimization. Higher precision in information acquisition is achieved at the cost of higher runtime overhead.
2. *Optimization*, which reorganizes the data layout and execution sequences of a program to maximize data reuse in caches and to minimize data sharing among caches.

TABLE 6
The Effect of Different Values of f on Execution Time (in Seconds) for the DMM_LO and the AC_LO on Four Processors of HP/Convex S-Class

Application	value of f			
	1	0.5	0.25	0.125
DMM_LO (N=1024)	6.6	6.1	5.8	5.8
AC_LO (N=400)	91	90	91	90
SMM_LO (N=1024)	1.3	1.3	1.4	1.5

TABLE 7
Execution Time (in Seconds) Based Comparison on HP/Convex S-Class

Application: Dense matrix multiplication

size	proc.	DMM_WL		DMM_LO		
		time	balance	time	overhead	balance
1024	2	108	0.01	115	10	0.06
	4	57	0.02	63	7	0.03

Application: Adjoint convolution

size	proc.	AC_A		AC_LO		
		time	balance	time	overhead	balance
256	2	763	0.002	698	0.67	0.003
	4	390	0.003	349	0.67	0.005

Application: Sparse matrix multiplication

size	proc.	SMM_A		SMM_LO		
		time	balance	time	overhead	balance
1024	2	37	0.012	23	2.0	0.035
	4	20	0.022	12	1.3	0.038

Columns `time` and `overhead`, respectively, give total execution time and task organization overhead in second. `Balance` presents load balance measurements in term of the rate of the time deviation to the mean time. (f=1).

3. *Integration*, which trades off locality with other performance factors to improve overall performance when the tasks are scheduled on an SMP.

Our locality optimization technique targets applications with dynamic memory-access patterns. We have shown that the multidimensional internal structure is effective to integrate both static and dynamic hints. We have shown that the runtime overhead is acceptable, which, in most of our test cases, is not larger than 10 percent of the total execution time of a program.

We have also shown that there is a good potential for the runtime locality optimization technique to improve the performance of application programs with irregular computation and dynamic memory access patterns. The runtime technique reduces the number of memory accesses to alleviate increasing demand on memory-bus bandwidth. In comparison with a regular application which was well-optimized by compiler-based techniques, we have shown that the runtime optimizations could perform competitively as well. Our run-time system was implemented as a set of simple and portable library functions. It can be conveniently used by users on commercial SMPs. The run-time system is not aimed at replacing compiler-based techniques, but at

TABLE 8
The Effect of Different Values of f on Execution Time (in Seconds) for the DMM_LO and the AC_LO on Four Processors of HyperSPARC Station-20

Application	value of f			
	1	0.5	0.25	0.125
DMM_LO (N=1024)	63	64	58	59
AC_LO (N=400)	349	347	352	373
SMM_LO (N=1024)	12	13	14.6	14.2

complementing a compiler to optimize those applications that are beyond of its optimization capability.

Our runtime technique has the following limits:

- We assume that the array data elements are contiguously allocated. Many operating systems attempt to do so. Our experiments using SimOS with SGI's IRIX operating system also show consistent results. However, if this assumption is not true in rare cases, a task may access an array in different memory regions, and the estimation of our method would not be accurate.
- When the task memory-access space is reduced to the bin-space by the cache capacity, a uniform scalar is used in each dimension for the reduction. However, a task may access different arrays in different sizes. The reduction by such a uniform scalar is the cheapest way to predict, but may not fully use the cache capacity in some cases.
- In some loop structures, a parallel iteration may not contiguously access an array. In order to fully utilize the cache, the loop iteration may be split further into smaller iterations so that each iteration accesses a contiguous region in each array.

The runtime technique only takes into consideration the nested loops without data-dependence. In the program behavior classifications given in Section 2.6, Type 4 has irregular computational patterns, dynamic memory patterns, and dynamic data-dependence patterns. This type of program is the most difficult for the locality optimization, because both the data-dependence and the locality optimization must be resolved at runtime. We are developing new methods to address this problem by combining our runtime technique with some existing methods on data dependence recognization.

ACKNOWLEDGMENTS

This work is supported in part by the U.S. National Science Foundation under grants CCR-9400719, CCR-9812187, and EIA-9977030, by the Air Force Office of Scientific Research under grant AFOSR-95-1-0215, and by Sun Microsystems under grant EDUE-NAFO-980405. We thank Craig Douglas for sending us his thread library. We are grateful to Greg Astfalk for his constructive suggestions and help in using the HP/Convex S-class. Dr. Neal Wagner carefully read the manuscript and made constructive comments. Finally, we appreciate the insightful comments and critiques from the anonymous referees, which are helpful to improve the quality and readability of the paper.

REFERENCES

[1] J.M. Anderson, S.P. Amarasinghe, and M.S. Lam, "Data and Computation Transformations for Multiprocessors," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 166–178, July 1995.

[2] J.M. Anderson and M.S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," *Proc. ACM SIGPLAN'93 Conf. Programming Language Design and Implementation,* pp. 112–125, June 1993.

[3] G. Astfalk and T. Brewer, "An Overview of the HP/Convex Exemplar Hardware," technical report, Hewlett-Packard, Mar. 1997.

[4] R. Chandra, A. Gupta, and J.L. Hennessy, "Data Locality and Load Balancing in COOL," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 249–259, May 1993.

[5] A. Charlesworth, N. Aneshansley, M. Haakmeester, D. Drogichen, G. Gilbert, R. Williams, and A. Phelps, "The Startfire SMP Interconnect," *Proc. Supercomputing '97,* Nov. 1997.

[6] S. Coleman and K.S. McKinley, "Tile Size Selection Using Cache Organization and Data Layout," *Proc. SIGPLAN '95 Conf. Programming Language Design and Implementation,* pp. 279–289, June 1995.

[7] M. Galles and E. Williams, "Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor," *Proc. 27th Hawaii Int'l Conf. System Sciences,* vol. 1, pp. 134–143, Jan. 1994.

[8] T.E. Jeremiassen and S.J. Eggers, "Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 179–188, July 1995.

[9] I. Kodukula, N. Ahmed, and K. Pingali, "Data-Centric Multi-Level Blocking," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 346–357, May 1997.

[10] A.R. Lebeck and D.A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," *Proc. 22nd Annual Int'l Symp. Computer Architecture,* pp. 48–59, 1995.

[11] M.S. Lam, E.E. Rothberg, and M.E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proc. ASPLOS '91,* pp. 63–74, Apr. 1991.

[12] H. Li, S. Tandri, M. Stumm, and K.C. Sevcik, "Locality and Loop Scheduling on NUMA Multiprocessors," *Int'l Conf. Parallel Processing* vol. II, pp. 140–144, 1993.

[13] E.P. Markatos and T.J. LeBlanc, "Using Processor Affinity in Loop Scheduling Scheme on Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, no. 4, pp. 379–400, Apr. 1994.

[14] K.S. McKinley, S. Carr, and C.W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Trans. Programming Languages and Systems,* vol. 18, no. 4, pp. 424–453, July 1996.

[15] K.S. McKinley and O. Teman, "A Quantitative Analysis of Loop Nest Locality," *Proc. ASPLOS '96,* pp. 94–104, Oct. 1996.

[16] J. Philbin, J. Edler, O.J. Anshus, C.C. Douglas, and K. Li, "Thread Scheduling for Cache Locality," *Proc. ASPLOS '96,* pp. 60–71, Oct. 1996.

[17] C. Polychronopoulos and D. Kuck, "Guided Self-Scheduling: A Practical Self-Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. Computers,* vol. 36, no. 12, pp. 1,425–1,439, Dec. 1987.

[18] M. Rosenblum, E. Bugnion, S. Devine, and S.A. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Trans. Modeling and Computer Simulation,* vol. 7, no. 1, pp. 78-103, 1997.

[19] M.B. Steinman, G.J. Harris, A. Kocev, V.C. Lamere, and R.D. Pannell, "The AlphaServer 4100 Cached Processor Module Architecture and Design," *Digital Technical J.* vol. 8, no. 4, pp. 21–37, 1996.

[20] J. Torrellas, M.S. Lam, and J.L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," *IEEE Trans. Computers,* vol. 43, no. 6, pp. 651–663, June 1994.

[21] J.E. Veenstra and R.J. Fowler, "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors," *Proc. MASCOTS '94,* pp. 201–207, Jan. 1994.

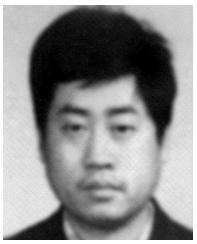
[22] M. Wolfe, *High Performance Compilers For Parallel Computing.* Addison-Wesley, 1996.

[23] M.E. Wolf and M. Lam, "A Data Locality Optimizing Algorithm," *Proc. ACM SIGPLAN '91 Conf. Programming Language Design and Implementation,* pp. 30–44, June 1991.

[24] Y. Yan, "Exploiting Cache Locality on Symmetric Multiprocessors: A Run-Time Approach," PhD dissertation, Dept. Computer Science, College of William and Mary, June 1998.

[25] Y. Yan, C.M. Jin, and X. Zhang, "Adaptively Scheduling Parallel Loops in Distributed Shared-Memory Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 1, pp. 70–81, Jan. 1997.

[26] Y. Yan, X. Zhang, and Z. Zhang, "A Memory-Layout Oriented Runtime Technique for Locality Optimization," *Proc. 1998 Int'l Conf. Parallel Processing (ICPP '98),* pp. 189-196, Aug. 1998.



Yong Yan received the BS and MS degrees in computer science from Huazhong University of Science and Technology, China, in 1984 and 1987, respectively, and his PhD in computer science from the College of William and Mary in 1998. He is currently a member of the technical staff of the Computer Systems Labs in the Hewlett Packard Laboratories. He was a system architect in HAL Computer Systems Inc. from 1998 to 1999. His current research objectives and interests are to build effective multiprocessor servers for both scientific and commercial workloads, and to provide efficient system support to the servers. He has extensively published in the areas of parallel and distributed computing, computer architecture, performance evaluation, operating systems, and algorithm analysis. He is a member of the IEEE and ACM.

Xiaodong Zhang received the BS in electrical engineering from Beijing Polytechnic University, China, in 1982, and the MS and PhD degrees in computer science from the University of Colorado at Boulder in 1985 and 1989, respectively. He is a professor of computer science at the College of William and Mary. His research interests are parallel and distributed systems, computer system performance evaluation, and scientific computing. He is an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*, and has chaired the IEEE Computer Society Technical Committee on Supercomputing Applications. He is a senior member of the IEEE.

Zhao Zhang received his BS and MS degrees in computer science from Huazhong University of Science and Technology, China, in 1991 and 1994, respectively. He is a PhD candidate of computer science at the College of William and Mary. His research interests are computer architecture and parallel systems. He is a member of the IEEE and ACM.