Computing Per-Process Summary Side-Effect Information

Tor E. Jeremiassen and Susan J. Eggers
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195
(206) 543-9349 (206) 543-2118
{torerik,eggers}@cs.washington.edu

Abstract

This paper presents a static algorithm, based on standard iterative data-flow techniques, for computing per-process memory references to shared data in coarse-grained parallel programs. The algorithm constructs control flow graphs for families of processes by recognizing predicates used in control statements whose values are invariant relative to any one process, but vary across processes. It is used in conjunction with interprocedural flow-insensitive side-effect analysis techniques to identify data objects accessed by each process. Dynamically allocated objects and descriptors for sections of arrays are incorporated into the general framework.

The motivation for the work was to reduce coherency overhead in shared memory multiprocessors. However, the algorithm can also be applied to local/global memory allocation decisions in distributed systems.

1 Introduction

It is well known that parallel programs executing on shared memory multiprocessors fail to achieve linear speedup. In bus-based systems one of the major hindrances is excessive coherency overhead, i.e., additional bus traffic generated by coherency-related operations. This overhead is largely determined by the pattern of memory references to write-shared data. Several studies have documented that the pattern is bimodal [1, 5, 9]: under *processor locality* [1], a single processor makes multiple, consecutive accesses, most importantly, writes, to the words within a cache block, uninterrupted by accesses from other processors; in *inter-processor contention*, multiple processors contend for one or more words within the block, and the number of per-processor sequential writes is very low. Shared memory reference patterns have a bearing on the choice of coherency protocol: write-invalidate protocols[1] perform well when there is good processor locality, write-broadcast[2] during periods of inter-processor contention [9].

The pattern of sharing may vary across data structures within a program, and even over time for a single structure. Therefore neither write-invalidate nor write-broadcast may perform consistently well over the entire program, since each is optimized for a particular behavior, and cannot respond to changes in the inter-processor reference pattern. What is needed is a static technique for determining the cross-processor reference pattern and monitoring its shifts throughout the program.

This work was supported by IBM Contract No. 18830046 and NSF PYI Award #MIP-9058-439.

[1] Write-invalidate protocols maintain coherency by requiring a writing processor to invalidate all other cached copies of the data before updating its own. Coherency overhead is caused by (1) invalidation signals that guarantee the writing processor the only cached copy of a shared address and (2) cache misses for rereferenced, invalidated data, called *invalidation misses*.

[2] Write-broadcast protocols maintain coherency by requiring the writing processor to broadcast shared writes, so that all caches and memory have access to the most current value.

Once the reference pattern is analyzed, three avenues would be open for eliminating coherency overhead.

The first is appropriate when the program exhibits one dominant sharing pattern. This alternative would choose a type of coherency protocol and then restructure the shared data to obtain a sharing pattern that matches it. For example, restructuring shared data to change the processors' access pattern from inter-processor contention to processor locality would improve program performance when executed on a multiprocessor with a write-invalidate protocol [8].

The second avenue would keep the existing shared data structure, but generate different coherency operations, depending on the sharing pattern of the portion of the program currently being analyzed. Invalidations would be issued when the program exhibits processor locality; data broadcasts for inter-processor contention. The mechanism for stipulating the coherency policy could be either setting a cache controller mode bit [18] or generating coherency instructions [7]; its scope could be fine-grain (a single shared structure) or coarse (a page).

Lastly, shared data could be copied into a structure that is compatible with the memory reference pattern exhibited in a particular phase of the program, just before the pattern shifts. Whether the copying is cost effective would require a comparison to the overhead that would occur if the the shared data were not restructured.

All alternatives require a static algorithm that computes cross-process memory reference patterns for shared data. This paper presents such an algorithm for coarse-grained parallel programs in which parallelism is created explicitly. The algorithm focuses on detecting the references for each process in the program, i.e., it must differentiate between the processes in terms of the particular shared data they access. Two stages of analysis are involved: differentiating the control flow of the processes, and examining the data accesses by each process. They correspond to the two mechanisms in our model of parallel programming by which processes may access different portions of shared data. First, a program statement that references shared data may be executed by only one process or a subset of the processes. Second, a statement executed by several processes may access different portions of shared data, one per process (e.g., arrays with per-process private index variables).

The first phase of the algorithm identifies when processes execute different code, i.e., have different control flow graphs. (We denote the collection of processes that have the same control flow graph a *process family*.) The algorithm constructs the control flow graph for each process family. This is done by considering predicates used in control statements whose values are invariant relative to any one process, but vary across processes.

The second phase performs an interprocedural flow-insensitive side-effect analysis [3, 2, 19, 6] on the section of the program corresponding to each discernible control flow graph. Heap allocated objects and descriptors for sections of arrays are integrated into our framework. The results reflect the sets of objects modified ($MOD(p)$) and used ($USE(p)$) by each process p. Following the side-effect analysis, each array descriptor is examined to detect when separate portions of the array are accessed by different processes.

The remainder of the paper is organized as follows: Section 2 presents the model of parallel computation that provides the framework for the algorithms that detect per-processor memory references. Section 3 describes the algorithm and pinpoints the sources and effects of inaccuracies. Section 4 applies the algorithm to various performance problems. Section 5 summarizes related work and section 6 concludes.

2 Model of Parallel Computation

Our work is based on a model of coarse-grained parallelism in which the granularity of parallelism is the process. The number of processes equals the number of processors and processes do not migrate. Each address space is divided into shared and private partitions. Processes are created explicitly, using a *fork()* operating system call that returns different values to the parent and the child processes. In addition, the address space is replicated upon creation of each separate process. The processes exist until explicitly killed or the program terminates. We target a workload written in a traditional programming language, C, enriched with process creation and synchronization primitives.

This model of parallel computation simplifies our analysis in several ways. First, since we are analyzing within the coarse-grained paradigm with equal numbers of processors and processes, the reference pattern of a processor is the same as that of a single process. Thus, we need not consider any partitioning and assignment of processes to processors. Second, since the processes are created explicitly, the points in the program where parallelism is created are easily identified. Hence, the control flow graphs of the processes have well-defined start nodes. Third, since the address space is replicated for the child process upon process creation, the only difference between the parent and the child process is the value that is returned from the *fork()* system call. Thus any differences in control flow or shared data accesses between processes can be traced back to the use of this value.

3 The Algorithm

The goal of our algorithm is to compute per-process memory references. We do this by constructing the sets of data objects that each process reads ($USE(p)$) and writes ($MOD(p)$). To arrive at this solution we divide the problem into two subproblems. The first is that of determining the control flow graphs for the different processes. The solution to this problem represents the main contribution of this paper. The second computes the $USE(p)$ and $MOD(p)$ sets on a per-process basis, given the different control flow graphs. To solve this, we use traditional interprocedural side-effect analysis.

3.1 Separating Control Flow Graphs

One important mechanism by which different processes of a program may access separate portions of shared data is by executing different code. The first step of our analysis therefore differentiates processes' control flow. Since the parallelism is not confined to loop nests, a general technique for separating control flow graphs over arbitrary control flow is presented.

In general, several processes may execute the same piece of code, i.e., have the same control flow graph, while still accessing different sections of shared data. Examples of this are programs that use a central work list accessed by multiple processes performing identical operations on different elements, and programs that use private variables in array subscript expressions to access different sections of shared arrays. Therefore, any control flow graph produced by the algorithm is assumed to correspond to a *family* of processes, unless it can be shown that it corresponds to a single process. Two process families may execute large sections of identical code. However, as long as they have one code segment that is dissimilar, they are considered to be different process families.

In order to compute the control flow for process families, it is necessary to determine two types of control points in the program. One is the set of locations that initiates separate process families.

As discussed in the previous section, our coarse-grain model of parallel processing identifies these locations. The second is all points in the program where the control flow of process families diverges.

To detect these points of divergence, the algorithm must identify control statements[3] that depend on a *process differentiating predicate* (*PDP*) for which the following two conditions hold: (1) there is at least one process for which the predicate holds true, and at least one for which it is false, i.e., the predicate is the mechanism by which processes are differentiated; (2) the truth value of the predicate is invariant throughout the lifetime of the processes. The latter condition guarantees that members of a process family will always take the same branch at the control statement in question; and the nodes dominated by the control statement along the other branch can be eliminated from that process family's control flow graph. The algorithm we present in section 3.1.1 assumes that the PDPs in the program have already been marked as such; PDP identification is discussed in section 3.1.2. (Note that each PDP may occur in several control statements.)

3.1.1 Flow Graph Construction

Our algorithm for constructing the set of flow graphs appears in Figure 1. The algorithm consists of two phases that together tag each node in the graph with the process families that may execute it. The first phase creates process families at points of divergence in the flow graph and propagates them over the graph using an iterative approach. At its conclusion, all nodes are tagged correctly for process families created upstream (toward the leaves). To correctly tag nodes for families created downstream (toward the root), it is necessary to propagate information backward over the control flow graph. The second phase takes care of this.

The data structures associated with each node in the flow graph are a set that contains the process families that execute that node, denoted X-Set, and a set of the process families that the algorithm considered for inclusion in the X-Set (denoted R-Set). R-Set is a superset of X-Set. Associated with each process family are two structures, a list of *PDP truth value* pairs and a pointer to its parent. Each truth value pair contains a PDP and the value true or false.

Process families are created either by process creation statements, in which case they are considered not to have a parent, or at points of divergence of control flow. The second case occurs when a process family has been propagated to a control statement that depends on a PDP that is not on the process family's list of truth value pairs. In this case a new child process family is created with a copy of the other's PDP truth value pair list. A new PDP truth value pair is entered into the list for both process families, but with the opposite truth value.

Figure 2 will be used to illustrate the algorithm. On the left is a control flow graph with two PDPs, at nodes A and E. Each node is tagged with *X-set* and *R-set* information; values outside the parentheses are created during the first phase of the algorithm, while those in parentheses are created during the second phase; together they make up the final set of values at the completion of the algorithm. On the right is a tree showing the parent-child relationships of the process families.

[3]The discussion assumes that any control statement has at most two paths leading from it. The approach can easily be extended to other cases such as switch statements.

```
FlowGraph     G;
Node          N,M,Start;
ProcFam       PF,NewPF,MPF,NPF;
ProcFamSet    temp-Set, temp-Set2;

 1.  for each node N of G do
 2.    X-Set(N) = R-Set(N) = empty set;
 3.  end for;
 4.  X-Set(Start) = R-Set(Start) = new_procfam(NULL,NULL,NULL);
 5.  change = TRUE;
 6.  while (change) do
 7.    change = FALSE;
 8.    for each node N in G do
 9.      temp-Set = X-Set(N);
10.      for each predecessor M of N in G do
11.        R-Set(N) = union(R-Set(N),R-Set(M));
12.        if (M contains a control statement with a PDP) then
13.          branch = which_branch(N,M); /* true or false */
14.          for each PF in X-Set(M) do
15.            if (<M.PDP,branch> in PF.PredList) then
16.              X-Set(N) = union(X-Set(N),{PF});
17.            else if (not (<M.PDP, (not branch)> in PF.Predlist)) then
18.              NewPF = new_procfam(M.PDP,branch,PF);
19.              NewPF.Predlist = NewPF.Predlist + PF.Predlist;
20.              PF.Predlist = PF.Predlist + <M.PDP,not(branch)>;
21.              X-Set(N) = union(X-Set(N),{NewPF});
22.              R-Set(N) = union(R-Set(N),{NewPF});
23.              X-Set(M) = union(X-Set(M),{NewPF});
24.              R-Set(M) = union(R-Set(M),{NewPF});
25.            end if;
26.          end for;
27.        else
28.          X-Set(N) = union(X-Set(N),X-Set(M));
29.        end if;
30.      end for;
31.    end for;
32.    change = (temp-Set != X-Set(N));
33.  end while;
34.  change = TRUE;
35.  while (change) do
36.    change = FALSE;
37.    for each node N in G do
38.      temp-Set = R-Set(N);
39.      temp-Set2 = R-Set(AnySuccesor(N));
40.      for each successor M of N in G do
41.        temp-Set2 = intersect(temp-Set2,R-Set(M));
42.        for each MPF in X-Set(M) in genealogical order do
43.          R-Set(N) = union(R-Set(N),{MPF});
44.          if (not in(MPF,R-Set(N)) then
45.            for each NPF in X-Set(N) do
46.              if (is_parent(NPF,MPF)) then
47.                X-Set(N) = union(X-Set(N),{MPF});
48.              end if;
49.            end for;
50.          end if;
51.        end for;
52.      end for;
53.      R-Set(N) = union(R-Set(N),temp-Set2);
54.      change = (temp-Set != R-Set(N);
55.    end for;
56.  end while;
```

Figure 1: Main algorithm for computing control flow graphs on a process family basis.

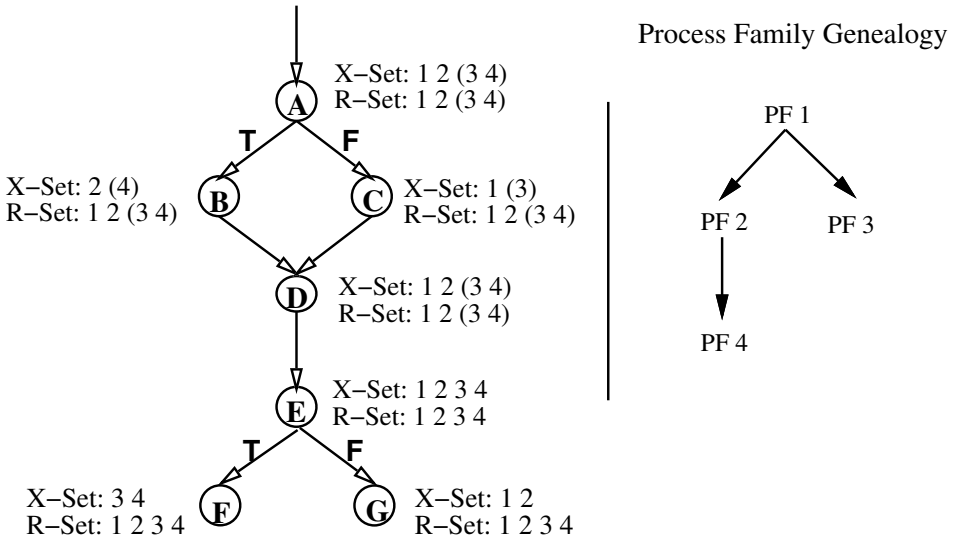

Figure 2: Illustration of how the algorithm propagates information and tags nodes

Prior to the first phase of the algorithm, the X-Set and R-Set of all nodes in the flow graph are initialized to the empty set. The X-Set and R-Set of the start node are set to contain a new process family with an initially empty list of PDP truth value pairs.

Phase one works by visiting each node of the flow graph repeatedly, beginning with the root, until no change in the X-Set of any node occurs. At each node X-Sets and R-Sets are constructed in different ways, depending on whether or not the node's predecessor is a point of divergence.

- If a predecessor does not contain a control statement with a PDP, and thus cannot cause control flow of process families to diverge, every process family in its X-Set and R-Set is included in the X-Set and R-Set of the current node (lines 11 and 28). For example, in Figure 2 node E inherits the X-Set and R-Set of node D, and node D inherits from nodes B and C.

- If the predecessor contains a control statement with a PDP, making it a point of divergence, each process family, PF, in its X-Set is examined for inclusion in the current node's X-Set. Inclusion occurs if the PDP truth value pair corresponding to the edge from the control statement in the predecessor node also occurs in the PDP truth value pair list associated with PF. Nodes A and B, and A and C illustrate this. If the PDP truth value pair does not occur in the current node's list at all, a new process family is created as described above (lines 13-26). The R-Set is set to include the R-Set of the predecessor. If a process family is created, it is included in the R-Set of both nodes.

The second phase analyzes the control flow graph in reverse, visiting each node repeatedly until no change in the R-Set of any node occurs. For each node visited the algorithm examines each of its successors. Each process family, PF, in a successor's X-Set is considered for inclusion in the current node's X-Set. This is done in genealogical order, that is parent before child. Inclusion occurs if: (1) PF is not in the current node's R-Set (line 44) and (2) there is another process family PPF in the current node's X-Set that is a parent of PF (line 46). The first condition guarantees that the process family was created at a node downstream in the flow graph from the current node. The second condition guarantees that PF will only be propagated backwards along the path of its parent. For example, in Figure 2 process family 4 is included in the X-Set for node B, since it is a

child process family of process family 2 and was created upstream from B. Regardless of whether PF is included in the current node's X-Set, it is added to the current node's R-Set (line 43). Also added is the intersection of the R-Set of all its successors (lines 39, 41 and 53).

3.1.2 Identifying PDPs

The algorithm for separating the control flow for different process families hinges upon the ability to decide whether a predicate used in a control statement is a PDP. We base that determination on information collected about variables in the program.

The second condition for PDPs (see Section 3.1) is that their truth values be invariant with respect to the processes. If they were not, the algorithm would erroneously eliminate nodes from the process family control graphs (when the truth values of the predicates change) that the processes actually execute. To guarantee invariance, we can only consider predicates in which the program variables used are not redefined after their first use.

The first condition requires that each predicate evaluate to true for some processes and false for others. The only variables that can have different values across processes are private variables. Thus, to be a PDP, the predicate must use private variables.

A PDP can therefore be defined as a predicate that (1) only contains variables that are not redefined after their first use and (2) contains at least one private variable whose value varies across the processes. There are several ways in which private variables can have values that vary across processes. For example, the values could depend on the private induction variable for a loop that creates processes, one for each value of the induction variable. Alternatively, a private variable could be assigned the value of a shared variable which is then incremented or decremented in a critical section.

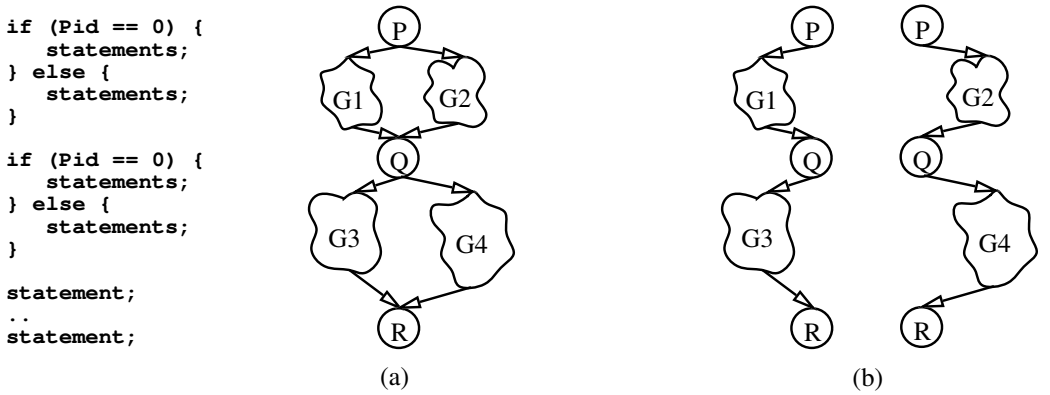

Figure 3: Original flow graph and flow graphs for process families after correct PDP identification

As an example, consider the control flow graph in Figure 3(a). Given that P and Q are if-statements whose predicates[4] are equi-partitioning PDPs and that there are two sets of processes that execute $G1$ and $G3$, and $G2$ and $G4$, respectively, the algorithm will construct the two control flow graphs in Figure 3(b).

Inaccuracies in constructing the control flow graphs can be introduced in two ways. One is when the algorithm fails to recognize PDPs. This causes a more conservative estimate for the control flow graphs of the affected process families to be made. Figure 4 shows the control flow

[4]Hereafter we will refer to the predicates themselves as P and Q.

```
if (Pid == 0) {
   statements;
} else {
   statements;
}

if (is_equal(Pid,0)) {
   statements;
} else {
   statements;
}

statement;
..
statement;
```

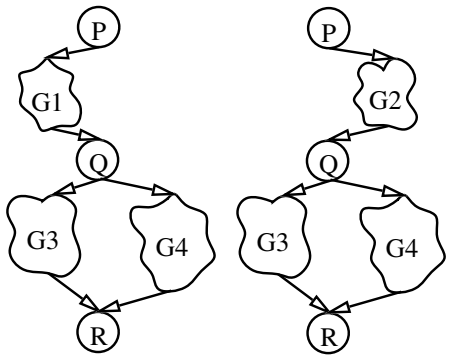

Figure 4: Flow graphs for the process families when a PDP (Q) is not recognized.

graphs that would be constructed from Figure 3(a), given that the algorithm could recognize P, but not Q. (Should neither P nor Q be recognized as PDPs, the resulting control flow graph would be that of Figure 3(a)). Overestimating the control flow graph for a process family inflates the sets of data objects that are read and written by its processes, since code they will not execute is included during side-effect analysis. The consequence is a corresponding overestimation of process contention, since identical data accesses now appear for several process families.

```
if (Pid == 0) {
   statements;
} else {
   statements;
}

if (Pid < 1) {
   statements;
} else {
   statements;
}

statement;
..
statement;
```

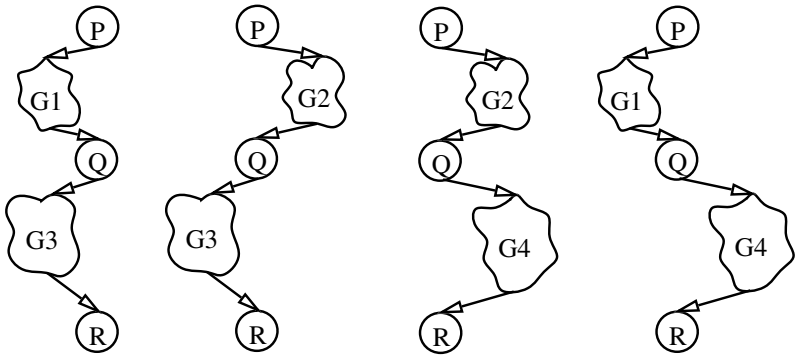

Figure 5: Flow graphs for the process families when the equi-partitioning of P and Q is not recognized.

The second inaccuracy occurs when the algorithm fails to recognize that a PDP is equi-partitioning with another PDP that has already been considered (i.e., it is on the PDP truth value pair list associated with the control flow graph), and generates a *ghost* process family that does not correspond to any of the processes of the program. Figure 5 shows the result of the analysis if the algorithm fails to detect that P and Q are equi-partitioning. Overestimating the number of control flow graphs (process families) also increases the estimated processor contention. Data accesses of the ghost process families, are now included in the side-effect analysis, implying sharing that does not occur in reality.

3.2 Performing Side Effect Analysis

Once the control flow graphs have been computed for each process family, we compute the sets of elements that each process in a process family reads (USE) and writes (MOD). We do this by

employing standard inter-procedural side-effect analysis techniques over the parallel portion of the program control flow graph. We use bounded regular section descriptors [11] to describe sections of arrays. The side-effects of individual process families are accumulated as the basic blocks they execute are processed. Since the techniques for side-effect analysis have been discussed extensively in the literature [3, 2, 19, 6], we will only address two issues.

3.2.1 Detecting Per-Process References of Shared Data in Arrays

The first issue is detecting the distribution of shared arrays across processes within a process family. In our model of parallel computation the partitioning of shared arrays is done implicitly, using private variables containing process-specific values in the array index expressions (the same variables that cause PDPs to partition the set of processes). The partitioning is therefore defined by the elements that each process accesses.

The USE and MOD sets contain regular section descriptors that describe portions of arrays that each process family accesses. If a descriptor contains a private variable that varies in value across the processes, we test whether it identifies disjoint sections of the array for different values of that variable. The array is implicitly partitioned across processes if the sections are disjoint. If they are not, or if the expressions in the regular section descriptors are too complex, the array is considered shared among the processes in the process family. The consequence is that the amount of sharing may be overestimated and locality enhancing transformations may be precluded; however, program correctness is maintained.

3.2.2 Pointers and Heap Allocated Objects

The second issue concerns detecting cross-processor accesses to (shared) heap-allocated objects in linked structures, such as lists and graphs. In the general case the set of objects that are linked by pointers may be arbitrarily complex, and it is difficult to detect their dynamic relationships. The presence of pointer arithmetic further complicates the analysis and can result in a conservative, worst-case estimate that may not reflect the actual execution behavior of the program.

To date our workload does not contain programs that use the full generality of pointers in C. Pointers to structures always access the same type of object as that associated with it in the declaration, and each element is accessed by name rather than a numeric offset. There is also no pointer arithmetic. Subject to these constraints, several techniques can be used to analyze accesses to heap-allocated structures and compute the set of objects that can be accessed by each pointer in the program [25, 17, 22, 13, 12, 4]. However, due to their time and space cost we elect to use a simpler approach. Instead of attempting to differentiate between heap-allocated objects of the same data type, we create a meta-object for the data type and summarize all accesses to heap-allocated objects of that data type as if they were accesses to the meta-object. In other words, during side effect analysis we treat all accesses to heap-allocated objects of a given data type as if they had occurred to a single static object. Based on this analysis our algorithm can detect per-process family accesses to different data types and to components of objects of the same data type. The algorithm cannot determine whether different process families access different objects of the same type. Therefore, it must conservatively conclude that all objects of a given data type are accessed by any process family that accesses any object of that type.

4 Implementation and Applications

We have implemented the detection algorithm, extended to cover the interprocedural case, using Parafrase-2 [21] as the underlying foundation. Approximately 5500 lines of C code has been added to the original Parafrase-2 source code. To date, the algorithm correctly identifies all but one PDP in our benchmarks and creates one ghost process family in one benchmark program and two in another[15] under the following restrictions. First, the predicates in the PDPs must be equal rather than equivalent. Second, detection of private variables that vary across processes is limited to induction variables for the loop that creates processes. Equivalent PDPs and private variables that are assigned in critical sections have not yet been implemented.

We have coupled the detection algorithm with compiler-directed data transformations to eliminate false sharing in bus-based, shared memory multi processors [14]. The results show that the transformations reduce the overall miss rate by up to 76% and the false sharing miss rate by up to 92%. In several cases these results surpass those obtained through manual analysis and restructuring [8].

The algorithm can also be applied to several other problems. First, it can determine the dominant sharing pattern in a program and consequently can guide the choice of coherency protocol. For instance, if the analysis concludes that a program exhibits inter-processor contention, a write-broadcast protocol would perform better that a write-invalidate protocol.

Second, the algorithm can be augmented to detect sharing patterns in separate phases of a program. This will involve detecting the frequency of access to all elements in the $USE(p)$ and $MOD(p)$ sets, and identifying barriers and other synchronization points. Incorporating the notion of sequences of access to shared data will facilitate other solutions for reducing coherency overhead. The first would dynamically restructure the shared data just before the pattern shifts. A second would keep the original shared data structure, but cause different coherency operations to be generated, depending on the sharing pattern of the portion of the code currently being analyzed. This alternative would allow coherency to be provided by software alone, extending the domain of the algorithm to higher performance multiprocessors with multi-path rather than bus-based interconnects.

Third, the detection algorithm and compiler-directed restructuring can enhance the effectiveness of prefetching in shared memory machines. Traditional prefetching algorithms do not consider invalidation misses when predicting which data objects should be prefetched[24]. Our algorithm alleviates the problem in two ways: first it eliminates false sharing as described above, thereby making the additional misses caused by prefetching tolerable; second, the results of the detection algorithm can direct the prefetching strategy for write-shared data.

Last, our algorithm can also also be applied to the problem of data placement in distributed shared memory machines with non-uniform memory access time (NUMA).

5 Related Work

Related work concentrates on reducing inter-processor communication by reorganizing the control structures rather than detecting cross-processor memory references and restructuring the shared data. Their workload consists of either loops or library routines that have fine-grain parallelism, and their studies record performance improvements only for the code fragments that have been transformed. Therefore the results are overly optimistic with regard to the expected performance of executing entire programs.

Ju and Dietz[16] restructured a program fragment of several loops accessing array elements. Their restructuring algorithm was based on applying loop transformations (such as loop distribution) and data layout transformations (accessing arrays in row or column major order), according to a coherency cost function. The restructuring provided a 25% to 300% improvement in execution time and bus utilization, depending on the block size of a 64KB cache.

Gupta and Padua[10] also examined sequential programs that were automatically parallelized at the loop level. They strip mined the loops to the size of the cache block and assigned each strip to a different processor. The decline in invalidations ranged from approximately 200% to 300% and in miss ratios from 10% to over 100%, as block size was increased to 128 bytes.

Peir and Cytron[20] partitioned loops to minimize inter-processor communication when processing recurrences. Their mechanism for partitioning utilizes loop unrolling and dependence vectors. Partitions are then scheduled on different processors. Partitioning data (to make an analogy to our work) would require that the compiler be aware of the size of the cache block. Our solution does not require this.

Lastly, Sarkar and Hennessy[23] partition and schedule code to obtain a good combination of low communication costs and a high degree of parallelism. They assume that parallelism has already been detected; they then identify sequential tasks that can be run in parallel, and schedule these tasks on a number of processors (which is less than the number of tasks) in a way that minimizes communication between them. As discussed above, we are doing neither partitioning nor scheduling of code; instead we assume that both partitioning (i.e., forking, in our context) and scheduling will be done via the runtime system. In addition, rather than using given communication costs, our aim is to reduce them still further by decreasing inter-processor communication.

6 Summary

This paper presents a static algorithm for computing per-process memory references to shared data in coarse-grained parallel programs in which parallelism is created explicitly. The algorithm constructs separate control flow graphs for each process family by recognizing predicates used in control statements whose values are invariant relative to any one process, but vary across processes. It then performs interprocedural flow-insensitive side-effect analysis on the sections of the program that correspond to each discernible control flow graph to identify shared data objects accessed by each process. Dynamically allocated objects and descriptors for sections of arrays are incorporated into the general framework.

7 Acknowledgments

We owe many thanks to Craig Chambers, David Callahan and Brian Koblenz for their invaluable comments on an earlier draft of this paper.

References

[1] A. Agarwal and A. Gupta. Memory-reference characteristics of multiprocessor applications under mach. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 215–225, 1988.

[2] J.P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Sixth Annual Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[3] J. Barth. A practical interprocedural data flow analysis algorithm. *CACM*, 21(9):724–736, September 1978.

[4] D.R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation*, pages 296–310, White Plains, NY, June 1990.

[5] D.R. Cheriton, A. Gupta, P.D. Boyle, and H.A. Goosen. The vmp multiprocessor: Initial experience, refinements and performance evaluation. In *15th Annual International Symposium on Computer Architecture*, pages 410–421, 1988.

[6] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Conference on Programming Languages Design and Implementation*, pages 57–66, June 1988.

[7] R. Cytron, S. Karlovsky, and K.P. McAuliffe. Automatic management of programmable caches. In *International Conference on Parallel Processing*, volume II, pages 229–238, August 1988.

[8] S.J. Eggers and T.E. Jeremiassen. Eliminating false sharing. In *International Conference on Parallel Processing*, volume I, pages 377–381, August 1991.

[9] S.J. Eggers and R.H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *16th Annual International Symposium on Computer Architecture*, pages 2–15, Jerusalem Israel, May 1989.

[10] M. Gupta and D. A. Padua. Effects of program parallelization and stripmining transformations on cache performance in a multiprocessor. In *International Conference on Parallel Processing*, volume 1, pages 301–304, August 1991.

[11] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[12] L.J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *ACM Transactions on Parallel and Distributed Systems*, 1(1):35–47, 1990.

[13] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Conference on Programming Language Design and Implementation*, pages 28–40, Portland OR, June 1989.

[14] T.E. Jeremiassen and S.J. Eggers. Eliminating false sharing through compiler-based shared data restructuring. Submitted for publication, 1992.

[15] T.E. Jeremiassen and S.J. Eggers. Experimental results of per-process side-effect analysis. Submitted for publication, 1992.

[16] Y. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fourth Workshop on Languages and Compilers for Parallelism*. Springer Verlag, August 1991.

[17] J.R. Larus and P.N. Hilfinger. Restructuring lisp programs for concurrent execution. *Parallel Programming: Experience with Applications, Languages and Systems*, 23(9):100–110, September 1988.

[18] MIPS Computer Systems, Inc., Sunnyvale, CA. *MIPS R4000 Microprocessor User's Manual*, first edition, 1991.

[19] Eugene Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the Eight ACM Symposium on Principles of Programming Languages*, pages 219–230, Williamsburg, VA, January 1981.

[20] J.K. Peir and R. Cytron. Minimum distance: A method for partitioning recurrences for multiprocessors. *IEEE Transactions on Computers*, 38(8):1203–1211, August 1989.

[21] C. Polychronopoulos, M. Girkar, M. Haghighat, C.L. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *International Conference on Parallel Processing*, volume II, pages 39–48, August 1989.

[22] C. Ruggieri and T.P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Symposium on Principles of Programming Languages*, pages 285–293, January 1988.

[23] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Symposium on Compiler Construction*, pages 17–26, June 1986.

[24] D.M. Tullsen and S.J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. Submitted for publication, 1992.

[25] W. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Symposium on Principles of Programming Languages*, pages 83–94, January 1980.