Incorporation of OpenMP Memory Consistency into Conventional Dataflow Analysis

Ayon Basumallik and Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285
http://www.ece.purdue.edu/ParaMount

Abstract. Current OpenMP compilers are often limited in their analysis and optimization of OpenMP programs by the challenge of incorporating OpenMP memory consistency semantics into conventional data flow algorithms. An important reason for this is that data flow analysis within current compilers traverse the program's control-flow graph (CFG) and the CFG does not accurately model the memory consistency specifications of OpenMP. In this paper, we present techniques to incorporate memory consistency semantics into conventional dataflow analysis by transforming the program's CFG into an OpenMP Producer-Consumer Flow Graph (PCFG), where a path exists from writes to reads of shared data if and only if a dependence is implied by the OpenMP memory consistency model. We present algorithms for these transformations, prove the correctness of these algorithms and discuss a case where this transformation is used.

1 Introduction

OpenMP [1] has established itself as an important method and language extension for programming shared-memory parallel computers. With multi-core architectures becoming the commodity computing elements of the day, OpenMP programming promises to be a dominant mainstream computing paradigm.

OpenMP is supported by several vendors by means of compilers and runtime libraries that convert OpenMP programs to multi-threaded code. However, current compilers are often limited in the extent to which they use the memory consistency semantics of OpenMP to optimize OpenMP programs. A reason for this is that most data flow analysis employed by state-of-the-art optimizing compilers are based on the traversal of a conventional control-flow graph – a program representation for sequential programs. In sequential programs, data always flows as expressed by the control-flow graph (CFG) and data flow algorithms infer dependences by traversing this CFG. In parallel programs, this is no longer accurate, as data can flow into a thread potentially at any read from a shared variable. To understand such flow, the specific memory consistency model of the employed parallel programming paradigm must be considered.

In this paper, we present techniques to incorporate OpenMP memory consistency semantics into conventional control-flow graphs. Our proposed techniques

R. Eigenmann and B.R. de Supinski (Eds.): IWOMP 2008, LNCS 5004, pp. 71–82, 2008.
© Springer-Verlag Berlin Heidelberg 2008

transform a conventional control-flow graph (CFG) into an “OpenMP Producer-Consumer Flow Graph” (PCFG), which resembles a conventional CFG and incorporates OpenMP memory consistency semantics into its structure. This enables the use of conventional data flow algorithms for OpenMP programs.

Related approaches to internally representing OpenMP programs for compiler analysis have proposed techniques to incorporate OpenMP control-flow semantics into a program’s control-flow graph [2,3]. The present paper is meant to complement, rather than compete with, these related approaches. The focus of this paper is more specifically on techniques to incorporate the memory consistency semantics of OpenMP programs into the internal representation. We shall illustrate why simply incorporating OpenMP control-flow information into the CFG may not be sufficient to account for the effects of the OpenMP memory consistency model. We shall then present formal algorithms to transform a conventional CFG into a representation that accurately reflects OpenMP memory consistency.

The rest of the paper is organized as follows. Section 2 describes the OpenMP Memory Model and introduces transformations that can be applied to a program’s CFG to incorporate OpenMP semantics. Section 3 presents algorithms to accomplish the transformations required to create the PCFG, presents a formal proof of correctness of these algorithms and discusses an application of the PCFG. Section 4 discusses related work. Section 5 concludes the paper.

2 The OpenMP Memory Consistency Model

The OpenMP memory consistency model is roughly equivalent to *Weak Consistency* [4]. Writes to shared data by one thread are not guaranteed to be visible to another thread till a synchronization point is reached. OpenMP has both implicit and explicit memory synchronization points. Examples of explicit synchronization points include *barrier* and $flush$ directives. Implicitly, there are memory synchronization points at the end of work sharing constructs (unless they have explicit *nowait* clauses) and at the end of synchronization directives like $master$ and *critical*. This means, for example, that writes to shared data in one iteration of an OpenMP for loop by one thread are not guaranteed to be visible to another thread executing a different iteration of the same loop till the implicit synchronization at the end of the loop is reached.

Figure 1 illustrates some ramifications of how the OpenMP consistency model affects analysis that are based on the program’s control-flow graph. The *nowait* clause in loop $L1$ denotes that writes to the array A by one thread in loop $L1$ are not guaranteed to be visible to another thread executing iterations of loop $L2$. However, any analysis based on the corresponding control-flow graph (which incorporates OpenMP control information) shown in the figure will find a path from vertex $v1$ to $v2$ and incorrectly infer that there is a dependence between the write to A in $V1$ and the read of A in $v2$.

On the other hand, the $flush$ and *atomic* directives denote that the atomic update to the scalar $tflag$ after loop $L2$ by one thread may be visible to another thread reading $tflag$ in loop $L1$. However, in the graph there is no path from

$v3$ to $v1$ and data flow analysis based on this graph will infer that there is no dependence between the two.

To correctly model these two cases, the control-flow graph needs to be adjusted so that there is a path from the write to a shared variable to a read if the write by one thread is visible to the read by another thread as per OpenMP specifications. A way of doing this for the graph shown in Figure 1 would be to add the edge $e1$ to account for the flush directives, delete the edge $e2$ to account for the nowait clauses and to add edges $e3$ and $e4$ to keep other paths in the graph unbroken even though the edge $e2$ has been deleted. By doing these edge additions and deletions, we create an *OpenMP producer-consumer flow graph* where there is a path from a write W to a read R in the program if and only if the write W occurring on one thread can be visible to the read R occurring on another thread. In certain cases, like the reads and writes connected by edge $e1$ in Figure 1, the read may be before the write in program order.

The next section of this paper presents formal algorithms to create such a *OpenMP Producer-Consumer Flow Graph*, starting from the sequential

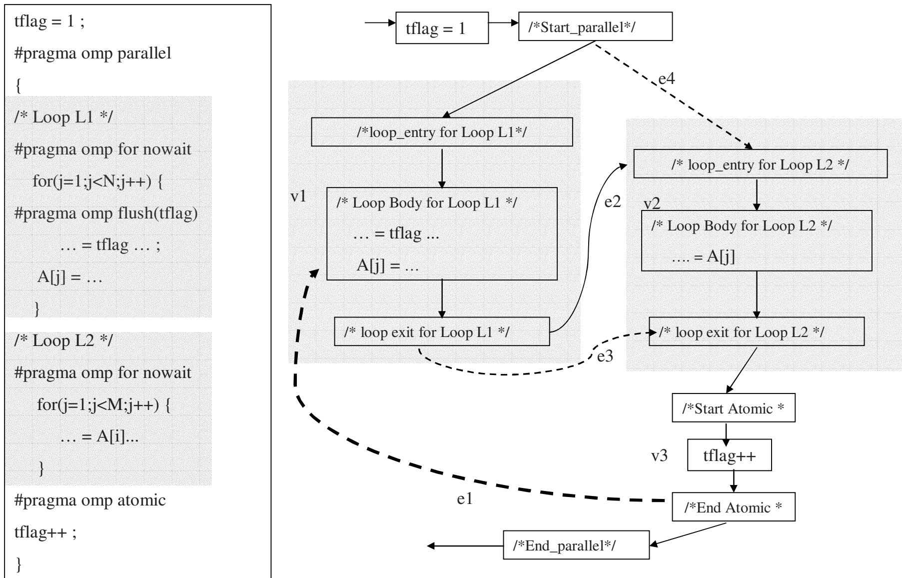

Fig. 1. Incorporation of OpenMP Memory Consistency Semantics: The graph edges drawn with solid arrows are present in the CFG for the program. The dependence implied between writes to array A in loop $L1$ and the read of A in loop $L2$ are relaxed by the *nowait* clause. Therefore, the path from vertex $v1$ to $v2$ in the sequential CFG must be broken to model this relaxation. The combination of *flush* and *atomic* directives imply a dependence between the update to $tflag$ in vertex $v3$ and the read of $tflag$ in vertex $v1$. Therefore, a path must be introduced between $v3$ and $v1$ in the sequential CFG to model this additional dependence. These path adjustments are accomplished by removing edge $e2$ and adding edges $e1, e3, e4$ to the CFG.

control-flow graph for the program. We are incorporating these algorithms into the Cetus [5] infrastructure as part of an OpenMP to MPI translation pass.

3 Incorporation of OpenMP Memory Consistency into the Dataflow Analysis

Our compiler creates an *OpenMP producer-consumer flow graph* using four steps –

1. Identify Shared Data.
2. Incorporate OpenMP control and data synchronization.
3. Relax Sequential Consistency.
4. Adjust for flushes.

We start with the sequential control-flow graph (CFG) for the program. The first step distinguishes between *shared* and *private* data in the program. The second step incorporates OpenMP constructs into the CFG to create an *OpenMP control-flow graph (OpenMP CFG).* The third and fourth steps first relax and then tighten ordering constraints in the program based on OpenMP memory consistency semantics to transform the OpenMP CFG into an *OpenMP Producer-Consumer Flow Graph (PCFG).*

3.1 Identification of Shared Data

The very first step in the incorporation of OpenMP semantics is to distinguish between *shared* and *private* data in the program. *Shared data* is defined as data in the program that may be read or written by more than one thread. Private data, on the other hand, is written by only a single thread in the program. The only way that private data is affected by the OpenMP semantics is when it is classified using clauses such as *firstprivate, lastprivate* and *threadprivate.* Thus, dataflow analysis for private data can still use the sequential CFG for the program. It is for the shared data that the PCFG needs to be created before any dataflow analysis can be done. In the very first step, our compiler identifies data that *may* be shared between multiple threads using the algorithm in Figure 2.

At this point, our compiler has separated data in the program into two classes - *shared* and *private.* Dataflow analysis for private data can be now invoked using the sequential CFG for the program. Once that analysis is complete, our compiler transforms the graph to incorporate OpenMP memory consistency semantics prior to invoking dataflow analysis for shared data. For this, our compiler starts by making OpenMP constructs explicit in the CFG to create the OpenMP CFG.

3.2 Making OpenMP Constructs Explicit

Our starting point for this step is the sequential CFG $G =< V, E >$ for the program, where a vertex V represents a basic block in the program and an edge $E = V_1 \rightarrow V_2$ exists if the basic block denoted by V_2 is a successor of

Algorithm *list_shared_variables*
Input : A - An OpenMP program. **Output :** S - A List of *Shared Variables* in A.
Start *list_shared_variables*
1. Set $S = \Phi$
2. **do** $\forall R$, R is an OpenMP *parallel region* in A
3. Set V = Set of all variables used in R
4. Set L = Set of all variables declared locally within R
5. Set PV = Set of all variables explicitly declared *private* for R
6. Set SV = Set of all variables explicitly declared *shared* for R
7. Set $S = S \cup (V - L - PV) \cup SV$
8. **end do**
9. **if** $(S = \Phi)$, *exit*, **endif**
10. **do** $\forall F$, F is *function call* within program A
11. **do** $\forall Pa$, Pa is a parameter of F
12. **if** $(Pa \in S)$
13. Let FP be the *Procedure* that defines F
14. Let PA be the *Procedure Parameter* of FP corresponding to function parameter Pa
15. Set $S = S \cup PA$
16. **end if**
17. **end do**
18. **end do**
19. **if** (Steps 10 through 18 have added new elements to S)
20. Go to Step 10
21. **end if**
End *list_shared_variables*

Fig. 2. Algorithm to create list of *Shared Variables* in an OpenMP Program. A key challenge in identifying shared variables is that function calls with shared variables as parameters may introduce additional shared variables that are not explicitly identified as *shared* by OpenMP directives. This algorithm addresses this challenge using the inter-procedural analysis shown in lines 10 through 21.

the basic block denoted by V_1. To incorporate OpenMP constructs into G for the OpenMP program, our compiler inserts vertices corresponding to OpenMP directives. Directives that refer to a set of statements in the program code are represented by *entry* and *exit* vertex pairs. For example, for each OpenMP parallel region in the program, there is a *parallel region entry* and a *parallel region exit* vertex. For each OpenMP critical section in the program, there is a *critical section entry* and a *critical section exit* vertex.

Stand-alone directives such as the $flush$ and *barrier* directives are represented with a single vertex in the program flow graph. Each OpenMP $flush$ vertex is associated with a *flush set*, which is a list of all shared variables that need to be *flushed* at that point. When the flush set is explicitly specified in the program, the corresponding flush vertex is annotated with this flush set. The *atomic* directive is represented with a pair of *atomic entry* and *atomic exit* vertices around the atomic statement.

Next, our compiler inserts an explicit *barrier* vertex wherever control synchronization is implicit in an OpenMP directive. Thus, *barrier* vertices are added to G at the entry to an exit from parallel regions and at the exit of worksharing regions that do not have *nowait* clauses. $flush$ vertices are inserted where a flush is implicit without a barrier, such as at entry to and exit from *critical*, *ordered* and *atomic* regions. Flush sets are derived by the compiler for these inserted $flush$ vertices. For example, for *critical* and *atomic* regions, flush sets include the shared variables accessed in these regions. For shared variables that have a *volatile* type, pairs of $flush$ vertices enclose every access to these variables.

Thus, at the end of this step, we have an *OpenMP Control Flow Graph* $\hat{G}$ that contains vertices corresponding to OpenMP constructs, *barrier* vertices where control synchronization is implied in the program and $flush$ vertices where a data coherence is implied in the OpenMP program.

3.3 Relaxation of Sequential Consistency

With the graph $\hat{G}$ now containing explicit synchronization vertices and vertices corresponding to OpenMP directives, our compiler proceeds to the next step of relaxing sequential consistency constraints using the algorithm *relax_sequential_consistency* shown in Figure 3.

In this algorithm, the compiler deletes edges from the program's control-flow graph, to break paths from writes to subsequent reads of shared data elements where the weak consistency model of OpenMP specifies that the write by one thread may not be visible to the read on another thread. Then the compiler adds edges from the previous synchronization points in the program to preserve paths to the read from writes before the previous synchronization.

At the end of this step, our compiler produces a control-flow graph where any path from a producer to a consumer for a shared variable exists only if this path exists in the original graph and the OpenMP directives in the program do not relax this dependence. In the next step, the compiler adds paths to account for producer-consumer relationships that are additionally introduced by OpenMP directives.

3.4 Adjustment for Flushes

Finally, our compiler uses the algorithm *Adjust_for_Flushes* shown in Figure 4 to adjust for explicit flushes in the program. For line 6 of this algorithm, two flushes are termed *concurrent* in our context if there is no execution order enforced upon them by the program structure. Thus, these may execute in any order, on different threads, between two synchronization points (*barriers*) in the program. To find concurrent flushes, the compiler uses a concurrency analysis for OpenMP [3] which has been used by other researchers as part of static race detection in OpenMP programs.

At this point, the compiler has a control-flow graph that reflects the OpenMP memory consistency model. In this graph, there is a path from a write statement $S1$ to a future read statement $S2$ if and only if the execution of $S1$ by one thread

Algorithm ***relax_sequential_consistency***
Input : 1. The OpenMP Control-Flow Graph $\hat{G}$ containing
explicit synchronization vertices for *barrier* and *flush*
and *entry* and *exit* vertices for OpenMP directives.
Output : 1. An OpenMP Control-Flow Graph $\hat{G}$ that models
the Relaxed Memory Consistency of OpenMP.

Start ***relax_sequential_consistency***
1. **do** $\forall L$, L is an OpenMP loop,
2. Remove the back edge from loop entry to loop exit for L.
3. **end do**
4. **do** $\forall V_x$, V_x is an OpenMP *exit* vertex in $\hat{G}$
5. **if** ($\hat{G}$ contains an edge $V_x \rightarrow V_y$ where
6. V_y is not an OpenMP *barrier* vertex) then
7. Delete edge $V_x \rightarrow V_y$
8. Let $V_e y$ be a *barrier* vertex reachable from V_y without intervening barriers
9. Let $V_d x$ be a *barrier* vertex that strictly dominates V_x
10. Add edge $V_x \rightarrow V_e y$ to $\hat{G}$
11. Add edge $V_d x \rightarrow V_y$ to $\hat{G}$
12. **end if**
11. **end do**
End ***relax_sequential_consistency***

Fig. 3. Algorithm to adjust the Control-Flow Graph to remove dependencies according to OpenMP's Memory Consistency specifications

produces an update to memory that is visible to the execution of $S2$ by another thread, as per OpenMP specifications. We refer to this adjusted control-flow graph as the *OpenMP Producer-Consumer Flow Graph (PCFG).*

3.5 Proof of Correctness

We now present a formal proof of the correctness of the two algorithms presented above.

Theorem 1. *For an OpenMP Producer-Consumer Flow Graph, a Read statement R is reachable from a $Write$ statement W $\Leftrightarrow$ the execution of W by one thread is guaranteed to be visible to the execution of R by another thread according to OpenMP specifications.*

Proof. We begin by first proving proposition in the forward direction – A *Read* statement R is reachable from a $Write$ statement W in the OpenMP PCFG $\Rightarrow$ the execution of W by one thread is guaranteed to be visible to the execution of R by another thread according to OpenMP specifications.

Consider two cases.

Case 1 – R occurs after W in program order.

Algorithm *Adjust_for_Flushes*
Input : 1. The OpenMP Control-Flow Graph $\hat{G}$ for the OpenMP program created by algorithm *relax_sequential_consistency.*
Output : 1. An OpenMP Producer-Consumer Flow Graph $\hat{G}$ for the program.

Start *Adjust_for_Flushes*
1. **do** $\forall V_f$, V_f is a $flush$ vertex in $\hat{G}$,
2. **do** $\forall V'_f$, V'_f is a $flush$ vertex in $\hat{G}$, $V_f \neq V'_f$
3. **if** ($V_f \neq V'_f$ and V'_f is not reachable from V_f) **then**
4. Let S be the *flush-set* of V_f
5. Let S' be the *flush-set* of V'_f
6. **if** (V_f and V'_f can be *concurrent* [3]
7. and $S \cap S' \neq \Phi$) **then**
8. Add edge $V_f \rightarrow V'_f$ to $\hat{G}$
9. **end if**
10. **end if**
11. **end do**
12. **end do**
End *Adjust_for_Flushes*

Fig. 4. Algorithm to adjust the Control-Flow Graph to incorporate dependencies created by explicit *flushes*

In this case, there can be three scenarios – (i) both R and W are in serial regions, (ii) either R or W is in a serial region and (iii) both R and W are in parallel regions.

If R and W are both in serial sections and there is a path from W to R, then the statement is trivially true.

If W is in a serial region and R is in a parallel region, let E_R be the entry vertex for the parallel region that R is in. E_R dominates R and so any path from W to R must contain E_R. Since there is an implicit synchronization at the beginning and end of each parallel region, E_R is dominated by a *barrier* vertex which must also be in the path from W to R and thus, the execution of W will be visible to an execution of R on any thread. Similarly, if W is in a parallel region and R is in a later serial region, the barrier at the end of this parallel region must be in the path from W to R and thus the execution of W on any thread will be visible to the thread executing the serial region that contains R. If W and R are both in a parallel regions then let E_W be the exit vertex for the OpenMP construct that W is within. Since Algorithm 3 ensures that the successor of E_W is always a *barrier* vertex, there is always a *barrier* vertex in the path from W to R and thus the execution of W will be visible to an execution of R on any thread.

Case 2 – R occurs before W in program order.
In this case, the path from W to R must contain an edge not present in the original control-flow graph of the program. Additional edges are introduced by line 8 in algorithm *Adjust_for_Flushes* in Figure 4 and by lines 10-11 in algorithm

relax_sequential_consistency in Figure 3. Since these edges contain vertices which are either barriers or flush pairs, the execution of W must be visible to an execution of R on any thread.

We now prove the proposition in the reverse direction – The execution of a $Write$ statement W by one thread is guaranteed to be visible to the execution of a *Read* statement R by another thread according to OpenMP specifications $\Rightarrow R$ is reachable from W in the OpenMP PCFG.

If W is visible to R on all threads as per OpenMP specifications, then one of two cases must be true –

Case 1 – R is reachable from W in the original program flow graph.

In this case, there must be an intervening *barrier* vertex in the path from W to R since W is guaranteed to be complete before R is started on any thread. We call this *barrier* vertex V_b. The only transformation that deletes edges from the original control-flow graph is line 7 in algorithm *relax_sequential_consistency* in Figure 3. However, the additional edges introduced in lines 10 and 11 of this algorithm ensure that paths from OpenMP *entry* and *exit* vertices to preceding and succeeding barriers are not broken. Thus, a path from W to V_b and from V_b to R is unbroken by the two algorithms. Thus, R is reachable from W in the OpenMP PCFG.

Case 2 – R is not reachable from W in the original program flow graph.

In this case, W must become visible to R because of OpenMP $flush$ directives. Thus, there must be a $flush$ in the program after W that is reachable from W in the OpenMP CFG. There must also be a $flush$ in the OpenMP CFG from which R is reachable. Additionally, these two flushes must be *concurrent*. However, if these flushes are concurrent, then line 8 in algorithm *Adjust_for_Flushes* in Figure 4 will create an edge between them. Thus, R will be reachable from W in the OpenMP PCFG.

Thus, by combining the two propositions proved above, we get "A *Read* statement R is reachable from a $Write$ statement W in the OpenMP PCFG $\Leftrightarrow$ the execution of W by one thread is guaranteed to be visible to the execution of R by another thread according to OpenMP specifications."

3.6 Applications of the OpenMP Producer-Consumer Flow Graph

The OpenMP producer-consumer flow graph is just like a conventional control-flow graph except that it accurately represents producer-consumer relationships between writes to and reads of shared data. Thus, this graph can now form the basis for subsequent dataflow analysis passes for shared variables.

Consider, for example, a typical dataflow analysis pass to find *reaching definitions*. Consider again the program snippet shown in Figure 1. Let $B1$ be the basic block that contains the statement accessing the shared variable $tflag$ in loop $L1$ and let $B2$ be the basic block containing the statement $tflag++$ after loop $L2$. The algorithm *Adjust_for_Flushes* shown in Figure 4 creates a path from $B2$ to $B1$ in the graph and thus, the definition of $tflag$ in $B2$ would get included in the list of *reaching definitions* for the use of $tflag$ in $B1$.

In our compiler, the OpenMP producer-consumer graph is a key element in a pass to transform OpenMP programs directly to MPI programs [6]. A summary

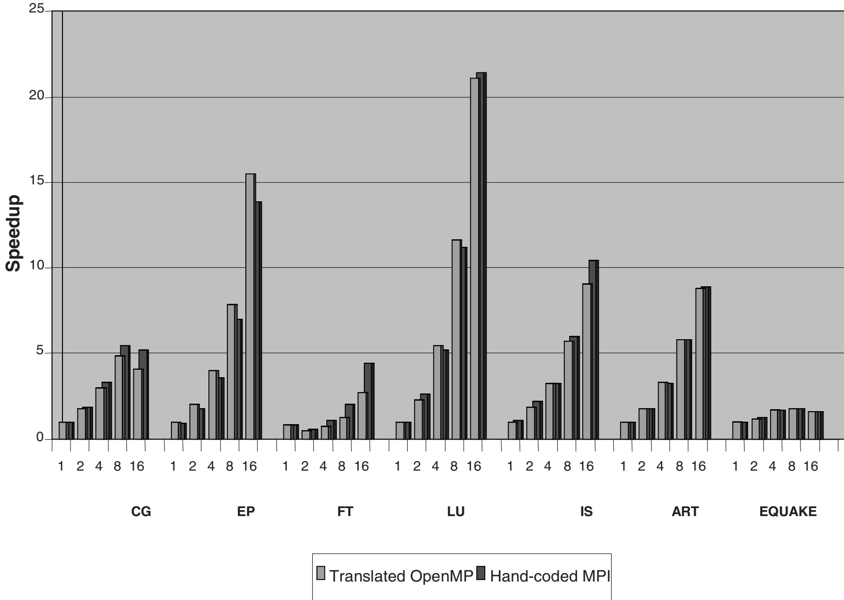

Fig. 5. Performance of seven representative OpenMP applications translated to MPI, compared with their hand-coded MPI counterparts on 16 WinterHawk nodes of an IBM SP2 system. A key step in the OpenMP to MPI translation is the creation of the OpenMP Producer-Consumer Flow Graph.

of the performance of this transformation is shown in Figure 5. This pass performs a whole program analysis of accesses to shared variables and communicates shared data, as it is produced, to all *potential* future consumers. This analysis needs to be conservative. Therefore, accurately representing the relaxation of constraints implied by weak consistency and OpenMP clauses such as *nowait* enables performance optimizations by eliminating certain producer consumer relationships. On the other hand, it includes additional constraints introduced by *flush* operations into the dataflow analysis framework, thereby preserving the correctness of the derived producer-consumer relationships.

4 Related Work

Previous research into the compiler analysis for programs with relaxed consistency models have focused on *delay set analysis* [7,8,9] to ensure correct execution of programs. Others have proposed techniques to use the compiler to hide or abstract the effects of the memory model from the programmer [10,11] and in doing so have relied on specialized graph representations such as the *Concurrent Static Single Assignment* form to represent parallel programs. Our techniques do not have to incorporate delay set analysis since the requisite fences or barriers in our program are already present in the form of OpenMP synchronization

statements. Also, rather than modifying the dataflow analysis passes in any way, our technique modifies the sequential control-flow graph for the program, adding and deleting edges to incorporate the effects of weak memory consistency and the additional constraints introduced by OpenMP flushes.

Our work complements recent research to analyze the synchronization structure of OpenMP programs [3]. To build the PCFG our algorithm starts from the sequential CFG. A first addition is to incorporate control-flow semantics for the parallel program to create the OpenMP CFG. This extension has been addressed in recent work [2,3]. In addition, we consider data flow that results from cross-thread communication at *flush* operations and dependences excluded by *nowait* type directives using the techniques proposed in this paper. Recent work on data flow analysis for OpenMP programs [12] has proposed the incorporation of OpenMP semantics by creating *Super Nodes* and *Composite Nodes* in the control flow graph to encapsulate OpenMP constructs and then use different data flow equations for these to incorporate OpenMP semantics. In this paper, we present a different approach that adjusts predecessor and successor relationships between basic blocks to incorporate OpenMP semantics and thus, we do not need to introduce any special data flow equations into the framework.

In our work, we use a simple conservative approach for differentiating between shared and private data. Recent work on autoscoping of data in OpenMP programs [13] proposes alternative approaches to accomplish this. Our work has also benefited from recent efforts to further elucidate the OpenMP memory consistency specifications [14] and to formalize the OpenMP memory model [15].

5 Conclusions

In this paper, we have presented techniques for incorporating OpenMP memory consistency semantics into conventional dataflow analysis. Instead of modifying dataflow analysis in any way, our method is to distinguish between shared and private data in the program, use the sequential control-flow graph of the program to perform dataflow analysis for private data, then transform the control-flow graph into an OpenMP producer-consumer flow graph that reflects the effects of OpenMP memory consistency and to use this graph to perform data flow analysis for shared data.

Our transformation has three essential steps - (*i*) distinguishing shared and private data, (*ii*) incorporating relaxed consistency semantics into the control-flow graph and (*iii*) incorporating any additional constraints introduced by the programming model (by *flush* operations). Thus, our transformations are broadly applicable for any parallel programming paradigm whose memory consistency model can be specified by (*i*) how it differentiates shared and private data, (*ii*) ordering constraints that are relaxed and (*iii*) additional ordering constraints that are introduced.

We use the techniques presented in this paper in the Cetus compiler as part of a set of transformations to translate OpenMP programs to MPI programs. These techniques are essential both for preserving the correctness of the translation and for performance optimization. We believe that these techniques hold promise in a

broad spectrum of transformations for a variety of parallel programming models when the memory consistency semantics differ from sequential consistency.

References

1. OpenMP Forum. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. Technical report (October 1997)
2. Satoh, S., Kusano, K., Sato, M.: Compiler Optimization Techniques for OpenMP Programs. In: Proc. of the Second European Workshop on OpenMP (EWOMP 2000) (September 2000)
3. Lin, Y.: Static Nonconcurrency Analysis of OpenMP Programs. In: Proceedings of the first International Workshop on OpenMP (IWOMP 2005) (2005)
4. Adve, S.V., Hill, M.D.: A Unified Formalization of Four Shared-Memory Models. IEEE Trans. on Parallel and Distributed Systems 4(6), 613–624 (1993)
5. Lee, S.-I., Johnson, T.A., Eigenmann, R.: Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 539–553. Springer, Heidelberg (2004)
6. Basumallik, A., Eigenmann, R.: Towards automatic translation of openmp to mpi. In: ICS 2005: Proceedings of the 19th annual International Conference on Supercomputing, Cambridge, Massachusetts, USA, pp. 189–198. ACM Press, New York (2005)
7. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. 10(2), 282–312 (1988)
8. Krishnamurthy, A., Yelick, K.: Analyses and optimizations for shared address space programs. Journal of Parallel and Distributed Computing 38(2), 130–144 (1996)
9. Fang, X., Lee, J., Midkiff, S.P.: Automatic fence insertion for shared memory multiprocessing. In: ICS 2003: Proceedings of the 17th annual international conference on Supercomputing, pp. 285–294. ACM Press, New York (2003)
10. Lee, J., Padua, D.A.: Hiding relaxed memory consistency with a compiler. IEEE Trans. Comput. 50(8), 824–833 (2001)
11. Midkiff, S.P., Lee, J., Padua, D.A.: A compiler for multiple memory models. Concurrency and Computation: Practice and Experience 16, 197–220 (2004)
12. Huang, L., Sethuraman, G., Chapman, B.: Parallel Dataflow Analysis for OpenMP Programs. In: Proceedings of the International Workshop on OpenMP (IWOMP 2007) (June 2007)
13. Lin, Y., Terboven, C., an Mey, D., Copty, N.: Automatic Scoping of Variables in Parallel Regions of an OpenMP Program. In: Chapman, B.M. (ed.) WOMPAT 2004. LNCS, vol. 3349, pp. 83–97. Springer, Heidelberg (2005)
14. Hoeflinger, J., de Supinski, B.: The OpenMP Memory Model. In: Proceedings of the first International Workshop on OpenMP (IWOMP 2005) (2005)
15. Bronevetsky, G., de Supinski, B.: Complete Formal Specification of the OpenMP Memory Model. In: Proceedings of the second International Workshop on OpenMP (IWOMP 2006) (2006)