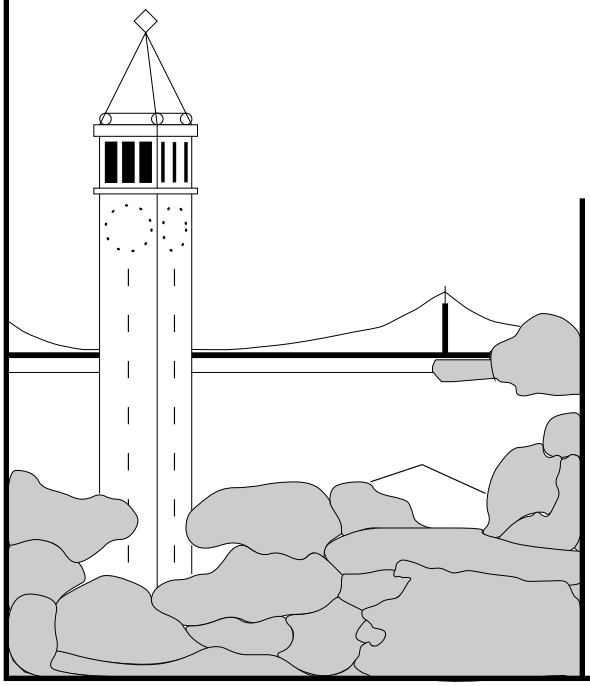
Polynomial-time Algorithms for Enforcing Sequential Consistency in SPMD Programs with Arrays

Wei-Yu Chen *Arvind Krishnamurthy* *Katherine Yelick*



Report No. UCB/CSD-3-1272

September 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Polynomial-time Algorithms for Enforcing Sequential Consistency in SPMD Programs with Arrays

Wei-Yu Chen Arvind Krishnamurthy Katherine Yelick

Abstract

The simplest semantics for parallel shared memory programs is sequential consistency in which memory operations appear to take place in the order specified by the program. But many compiler optimizations and hardware features explicitly reorder memory operations or make use of overlapping memory operations which may violate this constraint. To ensure sequential consistency while allowing for these optimizations, traditional data dependence analysis is augmented with a parallel analysis called cycle detection. In this paper, we present new algorithms to enforce sequential consistency for the special case of the Single Program Multiple Data (SPMD) model of parallelism. First, we present an algorithm for the basic cycle detection problem, which lowers the running time from $O(n^3)$ to $O(n^2)$. Next, we present three polynomial-time methods that more accurately support programs with array accesses. These results are a step toward making sequentially consistent shared memory programming a practical model across a wide range of languages and hardware platforms.

1 Introduction

In a uniprocessor environment, programs have a simple execution model to follow: all instructions appear to execute serially in the order specified by the code. Compilers and hardware are free to reorder memory accesses for optimization purposes, provided that the resulting execution is indistinguishable from one that strictly follows the program order. To satisfy this notion of correctness, compiler and hardware transformations must adhere to a data dependency constraint: the orders of all pairs of conflicting accesses (accesses to the same memory location, with at least one a write) must be preserved.

The execution model for parallel programs is considerably more complicated, since each thread executes its own portion of the program asynchronously, and there is no predetermined ordering among accesses issued by different threads to shared memory locations. A memory consistency model defines the memory semantics and restricts the possible execution orders of memory operations. Of the various memory models that have been proposed, the most intuitive is *sequential consistency*, which states that a parallel execution must behave as if it is an interleaving of the serial executions by individual threads, with each execution sequence preserving the program order [13]. Sequential consistency is a natural extension of the uniprocessor execution model and is violated when the reordering operations performed by one thread can be observed by another thread, and thus potentially visible to the user. Figure 1 shows a violation of sequential consistency due to reordering of memory operations. Although there are no dependencies between the two write operations in one thread or the two read operations in the other, if either pair is reordered, a surprising behavior may result, which does not satisfy sequential consistency.

Despite its advantage in making parallel programs easier to understand, sequential consistency can be expensive to enforce. A naive implementation would forbid any reordering of shared memory operations by both restricting compile-time optimizations and inserting a memory fence between every consecutive pair of shared memory accesses from a given thread. The fence instructions are often expensive, and the optimization restrictions may prevent code motion, prefetching, and pipelining [17]. Rather than restricting reordering between all pairs of accesses, a more practical approach computes a subset that is sufficient to ensure sequential consistency. This set is called a *delay set*, because the second access will be delayed until the first has completed. Several researchers have proposed algorithms for finding a *minimal delay set*, which is the set of pairs of memory accesses whose order must be preserved in order to guarantee sequential consistency [21, 11, 15].

The problem of computing delay sets is relevant to any programming model that is explicitly parallel and allows processors to access shared variables, including serial languages extended with a thread library and languages like Java with a built-in notion of threads. It is especially relevant to *global address space languages* like UPC [4], Titanium [5], and Co-Array Fortran [19], which are designed to run on machines with physically distributed memory, but

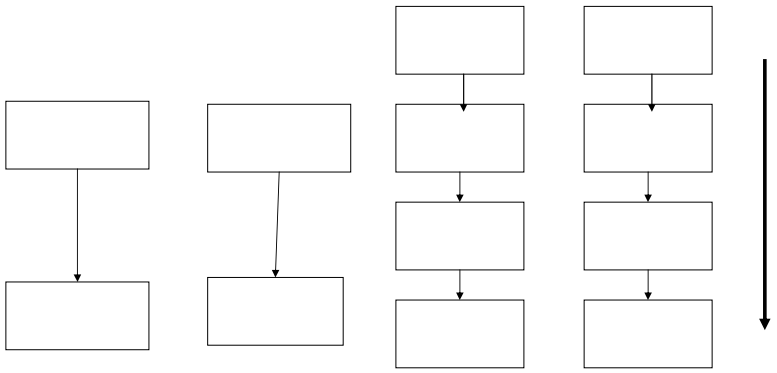


Figure 1. Violation of Sequential Consistency: The actual execution may produce results that would not happen if execution follows program order

allow one processor to read and write the remote memory on another processor.

In this paper, we focus on efficient algorithms to compute the delay sets for various types of Single Program Multiple Data (SPMD) programs. For example, given the sample code in Figure 1, the analysis would determine that neither pair of accesses can be reordered without violating sequential consistency. Our analysis framework is based on the cycle detection problem first described by Shasha and Snir [21]; previous work showed [10] that such analysis for SPMD programs can be performed in polynomial time. In this paper we substantially improve both the speed and the accuracy of the SPMD cycle detection algorithm described in [10]. By utilizing the concept of strongly connected components, we improve the running time of the analysis asymptotically from $O(n^3)$ to $O(n^2)$, where n is the number of shared memory accesses in the program. We then present three methods that extend SPMD cycle detection to handle program with array accesses, by incorporating into our analysis data dependence information from array indices. All three methods significantly improve the accuracy of the analysis for program with loops; each differs in their relative precision and offers varying degrees of applicability and speed, so developers can efficiently exploit their tradeoffs. Finally, we present techniques that exploit language features to further enhance the analysis.

The rest of the paper is organized as follows. We formally define the problem in Section 2 and summarize the earlier work on it in Section 3. Section 4 describes our improvements to the analysis' running time, while Section 5 present extensions to the cycle detection analysis that significantly improve the quality of the results for programs with array accesses. In Section 6, we show how to incorporate information from language features to enhance the accuracy of the analysis. Section 7 concludes the paper.

2 Problem Formulation

Our analysis is designed for shared memory (or global address space) programs with an SPMD model of parallelism. An SPMD program is specified by a single program text, which defines an individual program order for each thread. Threads communicate by explicitly issuing reads and writes to shared static or heap variables. For simplicity, we consider the program to be represented by its control flow graph, P. An *execution* of an SPMD program for n threads is a set of n sequences of operations, each of which is consistent with P. An execution defines a partial order, $\prec$, which is the union of those n sequences.

Definition 1 (Sequential consistency) *An execution is sequentially consistent if there exists a total order consistent with the execution's partial order, $\prec$, such that the total order is a correct serial execution.*

We are interested only in the behavior of the shared memory operations, and therefore restrict our attention to

the subgraphs containing only such operations. In general, parallel hardware and conventional compilers will allow memory operations to execute out of order as long as they preserve the program dependences. We model this by relaxing the program orders for each thread, and instead use a subset of P called the *delay set*, D.

Definition 2 (Sufficient Delay Set) *Given a program graph P and a subgraph D, D is a* sufficient delay set *if all executions of D are equivalent to a sequentially consistent execution of P.*

All executions must now observe only the program dependencies within each thread and the orderings given in D. Intuitively, the delay set contains pairs of memory operations that execute in order. They are implemented by preventing program transformations that would lead to reordering and by inserting memory fences during code generation to ensure that the hardware preserves the order. A naive algorithm will take D to be the entire program ordering P, forcing compilers and hardware to strictly follow program order. A delay set is considered *minimal* if no strict subset is sufficient. We are now ready to state the problem in its most general form:

Given a program graph P for an SPMD parallel program, find the sufficient delay set D for P.

---

3 Background

3.1 Related Work

Shasha and Snir [21] pioneered the study of correct execution of explicitly parallel programs and characterized the minimal set of delays required to preserve sequential consistency. Their results are for an arbitrary set of parallel threads (not necessarily an SPMD program), but does not address programs with branches, aliases or array accesses. Midkiff and Padua [17] further demonstrated that the delay set computation is necessary for performing a large number of standard compile-time optimizations. They also extended Shasha and Snir's characterization to work for programs with array accesses, but did not provide a polynomial-time algorithm for performing the analysis.

Krishnamurthy and Yelick [11, 10] later showed that Shasha and Snir's framework for computing the delay set results in an intractable NP hard problem for MIMD programs and proposed a polynomial-time algorithm for analyzing SPMD programs. They also improved the accuracy of the analysis by treating synchronization operations as special accesses whose semantics is known to the compiler. They also demonstrated that the analysis enables a number of techniques for optimizing communication, such as message pipelining and prefetching.

Once the delay set has been computed, sequential consistency can be enforced by inserting memory barriers into the program to satisfy the delays. Lee and Padua [14] presented a compiler technique that reduces the number of fence instructions for a given delay set, by exploiting the properties of fence and synchronization operations. Their work is complementary to ours, as it assumes the delay set is already available, while we focus on earlier problem of computing the minimal set itself.

Recent studies have focused on data structures for correct and efficient application of standard compile-time optimizations for explicitly parallel programs. Lee *et al.* [15] introduced a concurrent CFG representation for summarizing control flow of parallel code, and a concurrent SSA form that encodes sequential data flow information as well as information about possibly conflicting accesses from concurrent threads. They also showed how several classical analyses and optimizations can be extended to work on the CSSA form to optimize parallel code without violating sequential consistency. Knoop and Steffen [8] showed that unidirectional *bitvector analyses* can be performed for parallel programs to enable optimizations such as code motion and dead code elimination without violating sequential consistency.

3.2 Cycle Detection

Analyses in this paper are based on Shasha and Snir's [21] cycle detection algorithm, which we briefly describe here. All violations of sequential consistency can be attributed to conflicting accesses:

Definition 3 (Conflicting Accesses) *Two shared memory operations u, v from different threads are said to conflict if they access the same memory location, and at least one of them is a write.*

Conflicting accesses are the mechanism by which parallel threads communicate, and also the means by which one thread can observe reordered memory operations by another. The program order P defines a partial order on individual threads' memory accesses, but does not impose any restrictions on how operations from different threads should be interleaved, so there is not a single program behavior against which we can define correct reorderings. Instead, a happens-before relation for shared memory accesses originating from different threads is defined at runtime based on the time of their occurrences to fully capture the essence of a parallel execution. Due to its nondeterministic nature, each instance of parallel execution defines a different happens-before relation, which may affect execution results depending on how it orders conflicting accesses.

For a given parallel execution, let E be the partial order on conflicting accesses that is exhibited at runtime, which is determined by the values returned by reads from writes. The graph given by $P \cup E$ captures all information necessary to reproduce the results of a parallel execution: P orders accesses on the same thread, while E orders accesses from different threads to the same memory location. If there is a violation of sequential consistency, then for two accesses u, v, both (u, v) and (v, u) are related in $P \cup E$. Viewed as a graph, such situation occurs exactly when $P \cup E$ contains a cycle that includes E edges[1]. Since we cannot predict at compilation time which access in a conflicting pair will happen first, we approximate E by C, the *conflict relation* which is a superset of E and contains all pairs of conflicting accesses. The conflict relation is irreflexive, symmetric, and not transitive, and can be represented in a graph as bidirectional edges between two conflicting accesses.

The goal of Shasha and Snir's analysis is thus to perform cycle detection on the graph $P \cup C$ of a parallel program. Their algorithm uses the notion of *critical cycle* to find the minimal delay set necessary for sequential consistency:

Definition 4 (Critical Cycle) *A critical cycle in $P \cup C$ is a simple cycle with the property that for any two non-adjacent nodes u, v in the cycle, $(u, v) \notin P$.*

In other words, when detecting cycles we always attempt to find the shortest one available, and a critical cycle can have at most two (successive) nodes in any thread. Shasha and Snir proved that the P edges in the set of critical cycles form a delay set that guarantees sequential consistency:

Theorem 1 (Existence of a Minimal Delay Set for SC) *Let D be the set of edges (u, v) in straight-line code, where $(u, v) \in P$ is part of a critical cycle. Then any execution order that preserves delay D is sequentially consistent; furthermore, the set D is minimal.*

Proof: See [21]. ■

Figure 2 shows how a critical cycle can be used to compute the minimal delay set for sequential consistency, for the sample code from Figure 1.

3.3 Cycle Detection for SPMD Programs

Detecting critical cycles for an arbitrary program order P, unfortunately, is NP-hard as the running time is exponential in the number of threads. Krishnamurthy and Yelick [10] proposed a polynomial time algorithm for common special case of SPMD programs, taking advantage of the fact that all threads execute identical code. Their algorithm, explained in detail in [9], works as the follows:

[1] intrinsic cycles in P due to loops are ruled out

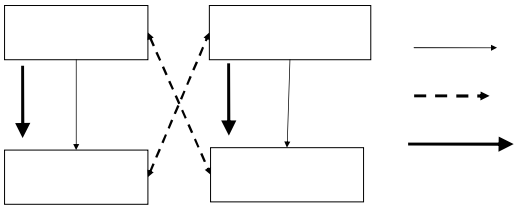


Figure 2. Computing the Delay Set

Definition 5 (Conflict Graphs for SPMD Programs) *Consider P_l, P_r to be two copies of the original program P, so that $u_l \in P_l$ and $u_r \in P_r$ if $u \in P$. Define C to be the set of conflicting accesses, and*

$$T_1 = \{(u_l, v_r), (v_l, u_r) | (u, v) \in C\} \quad (1)$$

$$T_2 = \{(u_r, v_r) | (u, v) \in C\} \quad (2)$$

$$T_3 = \{(u_r, v_r) | (u, v) \in P\} \quad (3)$$

$$CG = T_1 \cup T_2 \cup T_3 \quad (4)$$

The graph CG, named the *conflict graph*, will also be used in later analyses in this paper. Figure 3 demonstrates how the conflict graph can be constructed according to Definition 5. The right side of the conflict graph P_r is identical to $P \cup C$, while the left side P_l has no internal edges and connects to the right side via the conflict edges. Krishnamurthy and Yelick described an algorithm that computes the delay set by detecting a back-path in the transformed graph for each P edge (u_l, v_l):

Theorem 2 (Cycle Detection for SPMD Programs) *for an edge $(u, v) \in P$, if there exists a path from v_l to u_l in CG, then (u, v) belongs to the minimal delay set. Furthermore, the delay set computed is the same as the one defined in Theorem 1.*

Proof: See [10]. ■

Based on the above theorem, they claimed that cycle detection for SPMD programs can be performed in $O(n^3)$ time (Algorithm 1), where n is the number of shared memory accesses in P. While no algorithm is specified, it is easy to see that Algorithm 1 will compute the delay set that guarantees sequential consistency.

4 A Faster Algorithm for SPMD Cycle Detection

In this section, we show a slight modification of Krishnamurthy and Yelick's algorithm that can compute the iden-

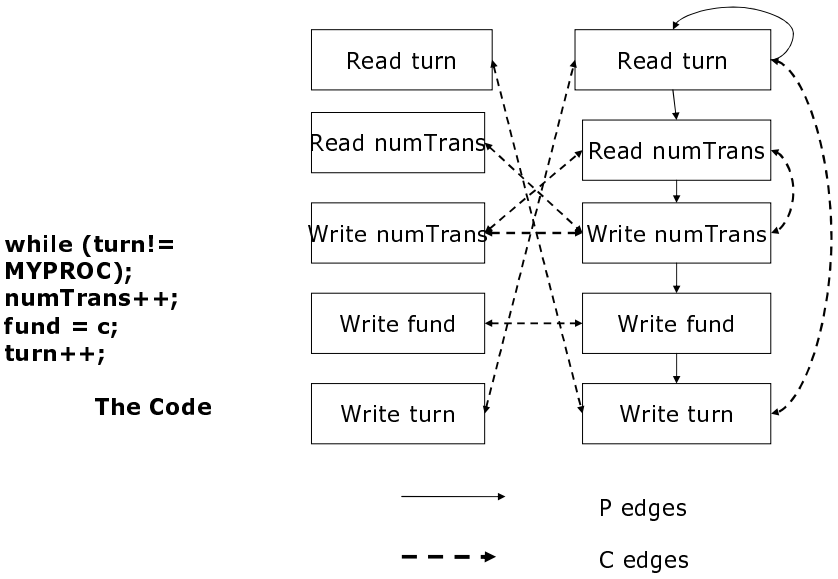


Figure 3. Cycle Detection for SPMD Programs

Input: P and C of a SPMD program
Output: delay set for P
1. Construct CG following the descriptions in Definition 5;
2. For every $u_l \in P$, perform a breadth-first search with the vertex as root;
3. Check for every $(u, v) \in P$ whether u_l is reachable starting from v_l in CG, using results from step 2. If yes, then (u, v) belongs to the delay set.

Algorithm 1: Krishnamurthy and Yelick's Algorithm for SPMD Cycle Detection

tical delay set in $O(n^2)$ time. Algorithm 1 is easy to understand but inefficient due to the breadth-first search required for each node. Instead, we can improve its running time by using strongly connected components (SCC) to avoid the redundant computations performed for each node. Our algorithm is similar to the one proposed in [12] in that both rely on the concept of strong connnectivity; an important distinction, however, is that we do not require initialization writes for every variable. If all accesses are read-only, step 3 fails due to the absence of conflicts, and no edges will be added to the delay set. This difference is vital if we want to combine the algorithm with synchronization analysis of barriers, since it is common for SPMD variables to be read-only in some phases of the program. Before explaining Algorithm 2, we prove the claim in step 3:

Corollary 3 *For every node $u \in P$, all of its conflicting accesses belong to the same strongly connected component in P_r.*

Input: P and C of a SPMD program
Output: delay set for P
1. Create the graph P_r as appeared in Definition 5, by taking $P \cup C$;
2. Identify the strongly connected components in P_r;
3. For every node $u \in P$, find the strongly connected component SCC_u that u's conflicting accesses belong to. (We will prove that they must all be in the same SCC.);
4. For each $(u, v) \in P$, if there is a path from SCC_v to SCC_u in the direct acyclic graph of SCCs, we add (u, v) to the delay set.

Algorithm 2: A $O(n^2)$ Algorithm for Computing Delay Set

Proof: Consider a node u and any two of its conflicting accesses v_1, v_2. Since there exist bidirectional edges between u, v_1 and u, v_2 in T_2 (the C edges), it is clear that they all belong to the same strongly connected component in P_r. ■

We can now show that for a SPMD program this modified algorithm is equivalent to Algorithm 1:

Theorem 4 *Algorithm 2 returns the same delay set as Algorithm 1 for any SPMD program.*

Proof: Suppose $(u, v) \in P$ is in the delay set computed by Algorithm 2. In terms of Definition 5, this means we can obtain an edge from v_l to some node t_1 in SCC_v that conflicts with v. From t_1, we can find a path that reaches some

node t_2 in SCC_u (since step 4 of Algorithm 2 states there is a path from SCC_v to SCC_u), such that t_2 is a conflicting access with u. $< v_l, t_1, ..., t_2, u_l >$ is thus a valid path in CG, and (u, v) is in the delay set computed by Algorithm 1.

Now we prove the reverse, and assume (u, v) is in the delay set of Algorithm 1, so that there exists a path $< v_l, t_1, ..., t_2, u_l >$, with t_1, t_2, and every node between them in P_r[2]. The two edges (v_l, t_1) and (t_2, u_l) imply that SCC_v contains t_1 and SCC_u contains t_2; furthermore, the path from t_1 to t_2 means that SCC_u is reachable from SCC_v, so that the conditions in step 4 of Algorithm 2 are satisfied. Therefore, (u, v) will be part of the delay set computed by Algorithm 2. ■

Finally, we calculate the running time of Algorithm 2.

Theorem 5 *Algorithm 2 runs in $O(n^2)$ time, where n is the number of shared accesses in P.*

Proof: Step 1 takes time proportional to number of edges in the graph, or $O(n^2)$. Step 2 of the algorithm can be computed in at most $O(n^2)$ time for our problem. Step 3 can be done in linear time with respect to n, if we cache the results from step 2 about which SCC that each node belongs to. Finally since the strongly connected component graph in step 4 is acyclic, we can calculate the reachability for every pair of nodes in $O(n^2)$ time by performing a topological sort. So the algorithm takes at most $O(n^2)$ time. ■

5 Extending SPMD Cycle Detection for Array Accesses

Another area in which the SPMD cycle detection algorithm can be improved is the quality of the delay set for array accesses. Although Theorem 2 states that the delay set computed by the algorithm is "minimal", the claim holds only for straight-line code with perfect alias information. The algorithm is therefore overly conservative when analyzing array accesses in loops; every P edge inside a loop can be included in the delay set, as a back-path can be constructed using the loop's back edge. This has an undesirable effect on performance, as the false delays can thwart useful loop optimizations such as loop-invariant code motion and software pipelining.

In this section, we present an analysis framework that extends SPMD cycle detection to handle array accesses. After describing an existing approach that requires exponential time in Section 5.1, we present three polynomial-time algorithms that could significantly reduce the number of delays inside loops. While all three techniques collect information from array subscripts to make the analysis more precise, they differ in their approaches for representing and propagating information: classical graph theory algorithms, data-flow analysis, and integer programming methods. The choice of the three methods largely depends on the amount of information that can be statically extracted from the array subscripts; for instance, the data-flow analysis approach sacrifices some precision for a more efficient algorithm, and the integer programming techniques supports complex affine array expressions at the cost of increased complexity.

For simplicity, we consider nested well-behaved loops in C, where each dimension of the loop is of the form $for(i = init; cond(i); i+ = k)\{loop\_body\}$, with the following provisions: both k and $loop\_body$ may be different for each thread, the loop index i is not modified in the loop body, and array subscripts are affine expressions of loop indices. While the definition may seem restrictive, in practice loops in scientific applications with regular access patterns frequently exhibit this characteristic. We further assume that the base address of the array access is a constant, and that different array variables do not overlap (i.e., arrays but not pointers in C).

5.1 Existing Approach

Midkiff et al. [18] proposed a technique that extends Shasha and Snir's analysis to support array accesses. Under their approach, every edge of the conflict graph (named a *s-level graph* in their work) is associated with a linear constraint that relates the array subscripts on its two nodes. A conflict edge generates an equality constraint, since the existence of a conflict implies the subscripts must be equal. Also, the constraint of each conflict edge will use a fresh variable for the loop index, as in general the conflicts can happen in different iterations. The constraint for a P edge is only slightly more complicated. Consider $(A[f(i)], B[g(i')]) \in P$, where i and i' represent possibly different loop index values, and f and g are affine functions. From the definition of P we could immediately derive $i' = i + k_1$ where k_1 is a multiple of the loop increment, since $A[f(i)]$ happens first by program order. The inequality constraint for k_1 depends on the context of the P edge; we specify $k_1 \geq 1$ if it is the back edge, and $k_1 \geq 0$ otherwise. Figure 4 shows the constraints generated by each edge in the sample graph.

Once the constraints are specified, the next step is to generate all cycles in the graph that may correspond to critical cycles (Definition 4). Midkiff et al. showed that the problem can be reduced to finding solutions to the linear system formed by the constraints of every edge in the cycle; a delay is necessary only for edges on a cycle with a consistent linear system. If a cycle satisfies the criteria, a final filtering step is applied to see if it can be discarded because it shares common solutions with another smaller cycle.

While their technique successfully incorporates depen-

[2]We can always replace a node in P_l with its counterpart in P_r

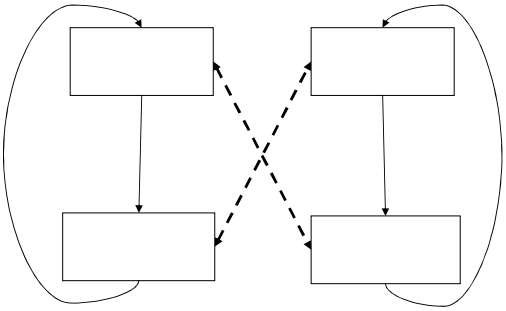


Figure 4. Conflict Graph with Corresponding Constraints

dence information to improve accuracy of the analysis, its applicability appears limited due to two factors. First, it does not specify how to generate the cycles in the conflict graph; the total number of (simple and non-simple) cycles is exponential in the number of nodes, so a brute force method that examines all is clearly not practical. Another limitation is the cost of solving each linear system, which is equivalent to integer linear programming, a well-known NP-complete problem. Since a cycle can contain $O(n)$ edges and thus constraints, solving the system again requires exponential time. As a result, in the next section we will present several polynomial-time algorithms that make cycle detection practical for programs with loops.

5.2 Polynomial-time Cycle Detection for Array Accesses

Our analysis framework combines array dependence information with the conflict graph, except that we assign each P edge with an integer weight equal to the difference between the array subscripts of its two nodes. Scalars can be considered as array references with a zero subscript. Also, an optional preprocessing step can apply affine memory disambiguation techniques [16] to eliminate conflict edges between independent array accesses. Figure 5 illustrates this construction[3], where the two edges in the loop body receive weights of 1 and -1, and the back edges are assigned the value of 0 and 2 to reflect both the difference between the array subscripts and the increment on the loop index variable after each iteration. Conflict edges always have zero weight, as the presence of a conflict implies the two array subscripts must be equal. For an edge $(u_l, v_l) \in P$, the goal of the analysis is not only to detect a path from v_l to u_l in the conflict graph, but also to verify that the back-path together with the edge forms a (not necessarily simple) cycle with zero weight:

[3]We showed only the right part of the conflict graph, as the left part remains unchanged

Theorem 6 (Cycle Detection with Weighted Edges) *With the above construction, an edge $(u_l, v_l) \in P$ is in the delay set if it satisfies the conditions in Theorem 2, and $W(u_l, v_l) + W(backpath(v_l, u_l)) = 0$, where $W(e)$ is the weight of edge e.*

Proof: Consider an edge $(A[a], B[b]) \in P$, and assume there exists a path $(B[b], B[c], ..., A[d], A[a])$ in the conflict graph. Since $(B[b], B[c])$ and $(A[d], A[a])$ are conflict edges, we have $a = d$ and $b = c$. So the path $(B[c], ..., A[d])$ must have weight $a - b$. ■

5.2.1 Zero Cycle Detection

If all edge weights are compile-time constants, the problem can be reduced to one of finding zero cycles in the conflict graph. On the surface the reduced problem still seems difficult to solve, as finding a simple cycle with zero total weight is known to be NP-complete. For our purposes, however, we are interested in finding zero cycles that need not be simple, as a zero cycle that visit a node multiple times conveys a delay due to conflicts among array accesses in different iterations. Several studies [7, 6] have presented recurrence equations and linear programming techniques to solve the general form of the ZERO-CYCLE problem, which determines for a graph G with k-dimensional vector edge weights if it contains a cycle whose weight sums to the zero vector. In particular, Cohen and Megiddo [3] proved that zero cycle detection for a graph with fixed k can be performed in polynomial time; they further showed that the special case of $k = 1$ can be answered in $O(n^3)$ time using a modified all pairs shortest path algorithm, where n is the number of

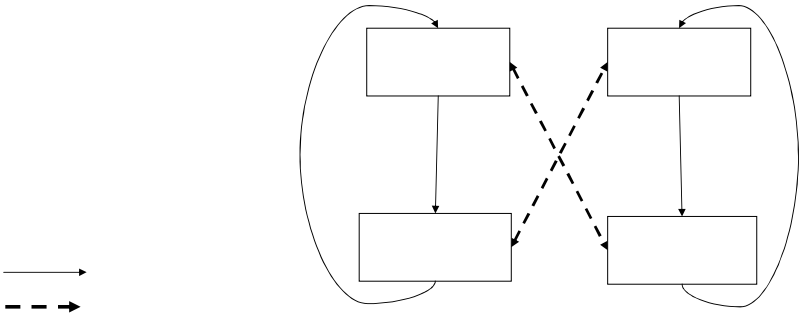


Figure 5. Adding Edge Weights for Cycle Detection

nodes. Algorithm 3 computes the delay set based on this result.

Input: P and C of a SPMD program
Output: delay set for P
Construct CG following the descriptions in Definition 5;
Annotate each P edge in CG with its corresponding weight;
foreach $(u, v) \in P$ **do**
  Add (u_l, v_l) to P_l, with its edge weight;
  Run the zero cycle detection algorithm from [3] on CG;
  If (u_l, v_l) is part of a zero cycle, add it to the delay set;
end

Algorithm 3: Handling Array Accesses Through Zero Cycle Detection

As each invocation of the zero cycle detection algorithm takes $O(n^3)$ time, this algorithm unfortunately has a running time of $O(n^5)$; we are currently exploring more efficient methods. The loss in efficiency is compensated, however, by obtaining a much more accurate delay set. Figure 6 demonstrates the analysis's benefit: while plain SPMD cycle detection (Algorithm 1) will incorrectly include every P edge in the delay set due to spurious cycles created by the loop back edge, Algorithm 3 can accurately eliminate these unnecessary delays. Another benefit of this algorithm is that it can be easily extended to support multidimensional arrays. For a k-dimensional iteration space, we simply construct CG using k-dimensional vectors as its edge weights, with each element corresponding to a loop index variable. As the level of loop nests in real programs rarely exceed 4 or 5, this more complex scenario can still be solved in the same asymptotic time as the scalar-weight case.

5.2.2 Data-flow Analysis Approximation

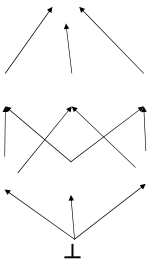


Figure 7. Lattice and Flow Equations for Algorithm 4

The major limitation of Algorithm 3 is that edge weights in general may not be compile-time constants; for example, it is common in scientific code to have a loop performing strided array accesses with an either dynamic or run-time constant stride value. The signs of the weights, however, are usually statically known, and using abstract interpretation techniques [1] we can deduce the sign of a cycle's weight sum. If every edge of the cycle has the same sign, it can never be a zero cycle; otherwise we conservatively assume that it may satisfy the conditions in Theorem 6. Algorithm 4 generalizes this notion by applying data-flow analysis with the lattice and flow equations in Figure 7 to estimate the weight sum of each potential cycle. For each P edge (u, v), $sgn(w)$ represents the possible sign of any paths from u to w; therefore, if u_l is reachable from v_l (indicating a back-

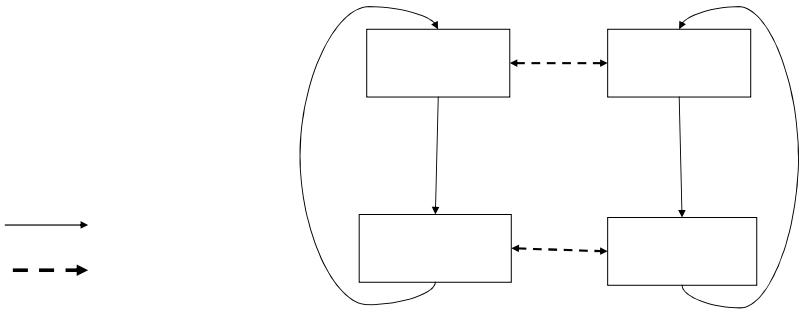


Figure 6. SPMD Code for which Algorithm 3 Is More Accurate Than Algorithm 1

path) and $sgn(u_l)$ is either $+$ or $-$, by definition (u, v) will not be part of any zero cycle.

Input: P and C of a SPMD program, with weighted edges
Output: delay set for P
Construct CG following the descriptions in Definition 5;
foreach $(u, v) \in P$ **do**
  Initialize $sgn(v_l)$ to be the sign of $W(u, v)$ (one of $0, +, -$), and $sgn(w)$ to be $\perp$ for all other nodes w;
  Apply data-flow analysis starting from v_l until no nodes have their signs changed;
  If u_l is reachable from v_l and $sgn(u_l) \in \{0, +0, -0, +-, \top\}$, then add (u, v) to the delay set.
end

Algorithm 4: Handling Array Accesses Using Data-flow Analysis

This approach is a sound but conservative approximation of the zero cycle detection problem, and thus may compute some false positive delays. While it gives the same result as Section 5.2.1 for Figure 5 (delays) and 6 (no delays), a more complicated example in Figure 4 illustrates their differences. Although the analysis from Section 5.2.1 correctly concludes that sequential consistency could never be violated there due to the absence of zero cycles, Algorithm 4, affected by the negative edge from $S3$ to $S4$, will conservatively place every P edge in the delay set. For the common cases of loops with monotonic array subscripts, however, this analysis is as accurate as the one in the previous section.

Since the lattice has a height of three, the data-flow analysis step will finish in at most $O(n^2)$ time. As the analysis step needs to be done for each P edge, it appears that we have a $O(n^4)$ algorithm. The insight here, however, is that when initializing the data-flow analysis for an edge (u, v), v_l can take only one of the three different values; it thus suffices to run the data-flow analysis three times for each node in the graph to cover all possible initial conditions of the analysis. So Algorithm 4 has a worst-case $O(n^3)$ time bound. Extensions of this approach to support nested loops is straightforward; we can run the analysis separately for each dimension, and add an edge to the delay set only when all dimensions return a sign that is not $\perp$, $+$, or $-$.

5.2.3 Integer Programming Based Method

In the most general case, array subscripts will be affine expressions with arbitrary constant coefficients and symbolic terms, so the previous methods are no longer applicable as neither the value nor sign of edge weights are statically known. In this case, we can still attempt to perform cycle detection by adopting the technique from Section 5.1 to convert it into an integer programming problem.

To avoid the exponential cost of exhaustively searching for cycles and solving linear systems, we can take advantage of the properties of our conflict graph representation. A P edge (u, v) can be a delay only if it has a back-path $(v_l, t_1, ...t_2, u_l)$ such that the generated cycle has a consistent system. While the number of back-paths may be exponential, they all share the structure that both (v_l, t_1) and (t_2, u_l) are C edges crossing between the left and right part of the graph. If the internal path $(t_1, ..., t_2)$ contains no C edges, it can be viewed as a single P edge and represented as one constraint on the subscripts of t_1 and t_2. We have thus significantly reduced the number of cycles that need to be considered for each P edge; since a node can participate in at most $O(n)$ conflicts, the number of such cycles never

exceeds $O(n^2)$. Furthermore, since each cycle examined can now have only four edges, the cost of solving a linear system is constant independent of problem size. This technique is in a sense a conservative approximation of Section 5.1, as it ignores the C edges in the internal path, which results in additional constraints that may cause the system to become inconsistent. Such an approximation, however, is necessary for soundness anyway, since it may be possible to construct a back-path without using internal conflict edges for loops of pure SPMD programs.

Input: P and C of a SPMD program
Output: delay set for P
Construct CG following the descriptions in Definition 5;
foreach $(u, v) \in P$ **do**
  Let $C(u)$ and $C(v)$ be the set of conflict edges for u and v;
  foreach $c_1 \in C(u)$ *and* $c_2 \in C(v)$ **do**
    Construct the linear system associated with the two conflict edges as mentioned above;
    If the system has an integer solution, add (u, v) to the delay set;
  end
end

Algorithm 5: Handling Array Accesses Using Integer Programming

Algorithm 5 describes how to compute the delay set by solving the set of linear constraints. For each P edge we need to verify if there exists a back-path whose corresponding linear system has a solution; the solution of the system gives the iterations that are involved in the conflict. As an example, for the edge $(S1, S2)$ in Figure 4 we can identify its only pair of conflict edges $(S2, S3)$ and $(S4, S1)$, which generates the following constraints:

$$i = j - 2 \qquad i' + 1 = j' \qquad i' = i + 3k_1 \quad (5)$$
$$j = j' + 2k_2 \qquad k_1 \geq 0 \qquad k_2 \geq 0 \quad (6)$$

Simple arithmetic reveals that the system has no integer solution, and we therefore conclude that the edge is not part of the delay set. In the worst case this algorithm will take a running time of $O(n^4)$, as the Cartesian product of C edges may have a $O(n^2)$ cardinality. Like the previous methods, this algorithm can also be adopted to support multidimensional arrays; each dimension of the access is handled independently, and an edge is added to the delay set only if all of its dimensions have identified a linear system that is consistent.

5.3 Algorithm Evaluation

We have presented three polynomial-time algorithms in this section that extend SPMD cycle detection analysis to support array accesses. Here we compare the three techniques using the following criteria: applicability, accuracy, as well as running time and implementation difficulty. In terms of applicability, the data-flow analysis method is the clear winner as it can be applied even to subscripts with non-affine terms, provided that the sign of the edge weights are still computable. Integer programming technique is also general enough to handle any affine array accesses, while zero cycl detection can only apply to simple subscript expressions. What the zero cycle algorithm lacks in generality, however, it compensates with greater accuracy by computing the correct and smallest delay set. Integer programming also offers good accuracy, especially when the loop bounds can be calculated statically so that the linear system can incorporate inequality constraints between loop index variables and the loop bounds. The data-flow analysis method is, as expected, the least accurate of the three and not compatible for loops with non-monotonic access patterns; its accuracy, however, can easily be improved by introducing more constant values into the lattice, at the cost of increased analysis time.

With regards of the running time, Algorithm 3 and 4 have the same asymptotic bound, but the latter can be more easily incorporated into a compiler's optimization framework as it is based on data-flow analysis. The integer programming method, on the other hand, is the most difficult to implement due to the construction and solving of linear systems. This suggests the following implementation strategy. In normal cases data-flow analysis will be the method of choice, while the more accurate zero cycle algorithm is applied to hot-spots in the program where aggressive optimization is desired; the integer programming technique is used for complex affine terms where neither Algorithm 3 and 4 is applicable.

5.4 Coping with Pointer Aliasing

So far we have assumed that no shared array objects will overlap, so that accesses to distinct array objects will never reference the same memory location. This precondition allows us to assign the weight of conflict edges to be zero, as the two accesses on each end must have the same subscript value. For languages such as C that allows pointer arithmetic to be used in place of array indexing, however, pointer aliases violate this assumption and force conflict edges to be added between two array accesses any time the points-to analysis fails to disambiguate them. As a result, the edge weights in our conflict graphs can no longer be solely based on the subscript expressions, and must include the base ad-

dress of the accesses. For example, if A and B in Figure 6 are pointer variables that may alias, the $(S1, S2)$ edges will need to have a weight of $B - A$, while the $(S2, S1)$ edges take a weight of $A - B + 1$.

Since the addresses of A and B generally are not statically known, we use symbolic analysis techniques [20] to calculate their lower and upper bound. If the bounds of the two variables do not overlap, we can determine the sign for $A - B$ and thus apply the data-flow analysis method from Section 5.2.2. Otherwise, we can still use the technique from Section 5.2.3 by introducing the symbolic bounds of the pointers as additional constraints that need to be satisfied by the integer programming problem.

6 Incorporating Language Features to Improve the Analysis

[10] shows that the size of the delay set can be significantly reduced if synchronization information can be incorporated into cycle detection. For example, since barriers define a precedence relation between its predecessors and successors, we can correctly assume that no conflict edges will cross any barrier statements. Similarly, specific features in the targeted parallel language can help prune the delay set further; using Unified Parallel C (UPC) [4] as an example, we show how some of its features can be useful in improving both the accuracy and speed of our analysis.

Like many parallel languages, UPC includes a builtin `upc_forall` loop to simplify the task of parallelization. Every iteration of the loop is executed by exactly one thread, based on the results of a runtime test. The UPC language specification [4] forbids any loop-carried dependences between iterations executed by different threads; for our analysis, this means that we can eliminate conflict edges between any pair of accesses in the same `forall` loop body. Although this condition is not as strong as we would like (a P edge in a `forall` statement may still be a delay due to conflicts with accesses outside the loop), it still permits us to eliminate many conflict edges without performing complex data dependence analysis.

Another UPC feature that could assist our analysis is its support for both a strict and a relaxed memory consistency model. Every shared variable in UPC can be type qualified as either "strict" or "relaxed". The strict memory model is analogous to sequential consistency in that it requires the actual execution of the accesses on each thread to be consistent with program order, while relaxed accesses are presumed to be data-race-free and thus not subject to this restriction. A data race occurs in a parallel program when there are concurrent accesses to the same memory location and at least one is a write, a condition identical to our definition of conflicting accesses. The goal of the UPC memory model is to effectively exploit the tradeoff between programmability and performance; relaxed accesses offer better performance as they can be aggressively optimized by compilers as long as local data dependency on each thread is still preserved, but programmers are left with the burden of ensuring that their code is free of race conditions. Taking advantage of the flexibility provide by the relaxed model, we can modify cycle detection to not consider any relaxed accesses as candidates for conflicts. The "relaxed" type qualifier thus serves as a programmer-supplied hint to the cycle detection analysis to reduce its workload. This relaxation is similar in spirit to the synchronization model of weak ordering [2], which forbids data races in a program; since relaxed accesses are supposed to not cause data races, they are by definition sequentially consistent as the set of conflict edges C will be empty. Since most variables in UPC programs are declared as relaxed for performance reasons, the cost of delay set analysis can be significantly reduced if only strict accesses need to be considered.

One additional complication that could arise is when a memory location is accessed using both strict and relaxed accesses. Due to pointer aliasing, a strict variable may be accessed indirectly through a relaxed pointer, and vice versa. To preserve the interprocessor data dependencies for strict accesses, the analysis must consider a pair of strict and relaxed accesses as potentially conflicting, and we need to augment the definition of conflict accesses:

Definition 6 (Conflicting Accesses for UPC) *Two shared accesses in UPC are said to conflict if they satisfy Definition 3, and at least one is strict.*

While including relaxed accesses in the analysis could potentially slow down the algorithm, in practice mixed accesses rarely occur in UPC program; strict variables are generally used to implement user-defined synchronization constructs, and it is almost always a programming error to have concurrent relaxed accesses to such variables.

7 Conclusion

In a multiprocessor environment, most standard sequential compiler optimizations could result in unexpected changes to program behavior because they may reorder shared memory operations. In this paper, we presented an efficient algorithm that computes the minimal delay set required to enforce sequential consistency for parallel SPMD programs. This analysis can be used to implement a sequentially consistent programming model on a machine that has a weaker model. In particular, implementing a global address space language on a machine with remote memory accesses can be done by issuing nonblocking memory operations by default, except when the compiler has determined that a delay between memory operations is needed. For ma-

chines with a remote latency of thousands of machine cycles, the ability to overlap in this fashion is critical.

Our algorithm is based on the concept of cycle detection, and has an asymptotic running time of $O(n^2)$, improving on a previous $O(n^3)$ algorithm. We have also described techniques for combining array analysis with the SPMD cycle detection algorithm; this further minimizes the delay set that guarantees sequential consistency without greatly slowing down the analysis. The analysis are based on classical graph algorithms (Section 5.2.1), data-flow analyses (Section 5.2.2), and integer programming (Section 5.2.3); in practice we expect the data-flow analysis method to be most applicable.

Finally, using UPC as an example, we discussed techniques for incorporating language features to significantly reduce the complexity of the analysis. The proposed algorithms have made cycle detection analysis more practical for SPMD programs, thus opening the door for optimizations of parallel programs that do not violate sequential consistency.

8 Acknowledgment

This work was supported in part by the Department of Energy, under cooperative agreement No. DE-FC03-01ER25509. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages*, pages 63 – 102. Halsted Press, 1987.

[2] S. V. Adve and M. D. Hill. Weak ordering—A new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 2–14, 1990.

[3] E. Cohen and N. Megiddo. Strongly polynomial-time and nc algorithms for detecting cycles in periodic graphs. *Journal of the ACM*, September 1993.

[4] T. El-Ghazawi, W. Carlson, and J. Draper. *UPC specification*, 2003. http://upc.gwu.edu/documentation.html.

[5] P. Hilfinger et al. Titanium language reference manual. Technical Report CSD-01-1163, University of California, Berkeley, November 2001.

[6] K. Iwano and K. Steiglitz. Testing for cycles in infinite graphs with periodic structure. In *Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 46–55. ACM Press, 1987.

[7] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM (JACM)*, 14(3):563–590, 1967.

[8] J. Knoop and B. Steffen. Code motion for explicitly parallel programs. In *the 7th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, May 1999.

[9] A. Krishnamurthy. *Compiler Analyses and System Support for Optimizing Shared Address Space Programs*. PhD thesis, U.C. Berkeley, 1998.

[10] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Jorunal of Parallel and Distributed Computing*, 1996.

[11] A. Krishnamurthy and K. A. Yelick. Optimizing parallel programs with explicit synchronization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–204, 1995.

[12] M. Kurhekar, R. Barik, and U. Kumar. An efficient algorithm for computing delay set in spmd programs. In *International Conference on High Peformance Computing (HiPC)*, 2003.

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, January 1976.

[14] J. Lee and D. Padua. Hiding relaxed memory consistency with compilers. In *proceedings of The IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[15] J. Lee, P. Padua, and S. Midkiff. Basic compiler algorithms for parallel programs. In *7th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 1999.

[16] D. E. Maydan. *Accurate Analysis of Array References*. PhD thesis, Stanford University, 1992.

[17] S. Midkiff and D. Padua. Issues in the compile-time of parallel programs. In *Proceedings of the 19th International Conference on Parallel Processing*, August 1990.

[18] S. Midkiff, D. Padua, and R. Cytron. Compiling programs with user parallelism. In *Proceedings of the 2nd Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[19] R. Numwich and J. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[20] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, 2000.

[21] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, April 1988.