Efficient Computation of Communicator Variables for Programs with Unstructured Parallelism

Christoph von Praun*

IBM T. J. Watson Research Center,
Yorktown Heights, NY

Abstract. We present an algorithm to determine communicator variables in parallel programs. If communicator variables are accessed in program order and accesses to other shared variables are not reordered with respect to communicators, then program executions are sequentially consistent. Computing communicators is an efficient and effective alternative to delay set computation. The algorithm does not require a thread and whole-program control-flow model and tolerates the typical approximations that static program analyses make for threads and data. These properties make the algorithm suitable to handle multi-threaded object-oriented programs with unstructured parallelism. We demonstrate on several multi-threaded Java programs that the algorithm is effective in reducing the number of fences at memory access statements compared to a naive fence insertion algorithm (the reduction is on average 28%) and report the runtime overhead caused by the fences (between 0% and 231%, average 81%).

1 Introduction

The reordering of seemingly independent memory access in individual threads of a parallel program can lead to violations of sequential consistency (SC). This phenomenon can be prevented by the selective insertion of memory fences into a parallel program. Algorithms that insert memory fences to maintain SC basically proceed along three steps:

1. *May-happen-in-parallel analysis* identifies statements that are executed concurrently by different threads [8]. *Static data race analysis* goes a step further and identifies those concurrent statements that operate on the same shared data [2,14].
2. Given statements that execute concurrently, *delay-set analysis* [10,5,6,1] determines the set of pairs of object access statements that should occur in program order. If all such ordering constraints are met at runtime, the execution SC [10].

* This work was done, in part, while the author was working at the Laboratory for Software Technology, ETH Zürich, 8092 Zürich, Switzerland.

R. Eigenmann et al. (Eds.): LCPC 2004, LNCS 3602, pp. 209–223, 2005.

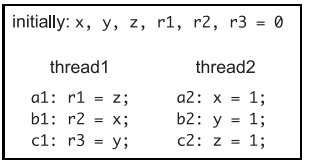

Fig. 1. Scenario with a communicator variable

3. *Fence-insertion* implements the ordering constraints (delays) through the placement of memory fences. Specific properties of memory ordering of a computer architecture and its fence model can be exploited to combine and to reduce the number of memory fences [3].

The three steps build on one another, i.e., the results of one are the input to the next. The first and second step are machine-independent and account only for programming language properties, e.g., the ordering semantics of synchronization features. The third step is machine-specific. The overall goal of a the fence insertion is to place a sufficient number of fences to ensure correctness with respect to the memory model (SC) and at the same time to minimize the number of executed memory fences.

This paper presents a novel procedure for the second phase of the algorithm. Instead of a *delay set*, we compute a set of so called *communicator variables*. Intuitively, a variable acts as a communicator if it allows one thread to determine the progress made by another thread. In the program in Figure 1, variable `z`, e.g., acts as communicator: if thread1 observes a value of 1 at statement $a1$, then thread2 must have made progress at least till statement $c2$, and hence the effects of $a2$ and $b2$ must also observable by thread1, i.e., $r2$ and $r3$ must receive a value of 1. If communicator variables are accessed in program order and accesses to other shared variables occur in program order with respect to communicators, then program executions are SC. The novel aspect of our approach is that it is effective for the most general, i.e., MIMD-style, object-oriented programs; unlike previous algorithms for delay set computation that have polynomial complexity (for SPMD programs), the runtime of our algorithm is mostly linear in the size of the program.

1.1 Background

The fundamental principles of computing the delay set for a given parallel program execution has been first described in [10]. Since then, several algorithms have been designed that approximate the delay set for parallel programs. The general procedure followed by these algorithms is to compute an abstract model for possible program executions from the program and then apply a variant of the delay set analysis proposed in [10] to this abstract execution model. For realistic programs, however, there are several aspects that complicate this procedure; we describe these difficulties in the following paragraphs.

The algorithm in [10] assumes a precise model of threads and shared variables. In practice, a static analysis is faced with the ambiguity in the distinction of

variables through aliasing and a relatively coarse model of threads that may leave the multiplicity of threads unspecified.

Krishnamurthy and Yelick [5] showed that the computation of delays according to the original algorithm of Shasha and Snir [10] is NP hard for general MIMD programs. The same authors present an algorithm with polynomial worst case complexity for a restricted class of programs, namely SPMD programs [5] ($O(n^3)$, n is program length). The efficiency of this algorithm is further improved by recent work in [6,1], yet these algorithms have polynomial complexity ($O(n^2)$, n is program length).

The problem of computing delay sets for general MIMD parallel programs has been addressed by Midkiff, Sura, Lee, and Padua in [7,12]. Their model of multi-threaded programs [7] is based on `cobegin/coend` (MIMD) and `parallel do` (SPMD) constructs that precisely capture the extent of parallelism with respect to the program scope – this is however uncommon for many multi-threaded object-oriented programs such as the Java programs that we use in Section 3. In [12] the authors present the architecture of an optimizing compiler with fence-insertion for general Java programs. The approach is based on a comprehensive set of techniques for the analysis of multi-threaded object-oriented programs such as thread-based escape analysis, MHP analysis, and synchronization analysis. Some of the implementation is elaborated in more detail in [3]. The proposed delay set analysis, however, is still performed on a 'traditional' control-flow model based on `cobegin/coend` and `parallel do`.

1.2 Approach

Our work differs from previous approaches in that it does not try to capture the control flow and thread structure of the overall program. This makes the approach suited for programs with unstructured parallelism. Instead of the program structure, we build a model of those shared variables that may not behave sequentially consistent. A static analysis approximates the order in which individual methods access these variables and determines from that ordering those variables that act as 'communicators' between threads. The algorithm is efficient because each method is treated once and individually.

1.3 Contributions

Algorithm. We present the concept of communicator variables and specify an algorithm for the computation of communicator variables in unstructured parallel programs. This algorithm provides information for the sparse insertion of memory fences (for SC) and is more efficient than previous algorithms that compute delay sets.

Evaluation. We demonstrate the effectiveness of communicator information for fence insertion on a number of common multi-threaded Java programs and compare the results to a naive fence insertion procedure. Moreover, we combine the computation of communicators with a powerful conflict analysis [14] and quantify the total runtime overhead of the resulting programs after fence insertion.

2 Algorithm

2.1 Preliminaries

SC is violated through the reordering of memory accesses inside individual threads. According to Shasha and Snir [10], explicitly enforcing an order among *some* memory accesses can provide SC. Naturally, only accesses to shared memory need to be considered; more precisely, only those accesses to shared memory that participate in a data race. A *data race* occurs, if several threads access the same variable without order, i.e., ordering is not established through explicit means of synchronization such as locks, and if at least one access is a write [9].

Ultimately, a compiler has to establish access order among statements and hence we need to convey the notion of a data race to a concept that is available to a static analysis: For short we say that a statement is *conflicting* if it may issue an access that participates in a data race.

2.2 Overview

Naive Algorithm. A naive delay set computation requires that all memory accesses that may participate in data races (i.e., conflicting accesses) occur in program order at each processor. The delays required in the naive scheme can be enforced by consistently inserting a memory fence either before, or after each conflicting statement.

Ideal Algorithm. Shasha and Snir describe an algorithm that determines a minimal set of delays based on a program execution trace; the algorithm is NP complete and does not easily map to general parallel programs with loops, and procedures. Some papers report the results of an algorithm with ideal precision and effectiveness as 'manual', because the escape information, conflicting statements, and fence instructions are determined through manual program inspection and not through an automated algorithm [8,3]. We do not attempt to determine such an 'ideal' setting as a lower boundary. Our recent work on data race detection [13,14] has shown that there are typically no or very few actual data races in correct multi-threaded programs and hence an ideal tool would, for most programs, not need to insert any fences at all.

Our Approach. Our approach to selectively establish access ordering is different from previous delay set analyses that determine the necessity of a delay pairwise for conflicting statements. We determine communicator variables and ensure that access to communicators is associated with a memory fence. The property of a communicator is a whole program property and hence unnecessary delays may be enforced if a variable is used as a communicator in some context and in other contexts as 'ordinary' shared variable. The fence insertion based on communicators should however be comparable to approaches that are based on a delay set analysis and also result in fewer fences than the naive algorithm.

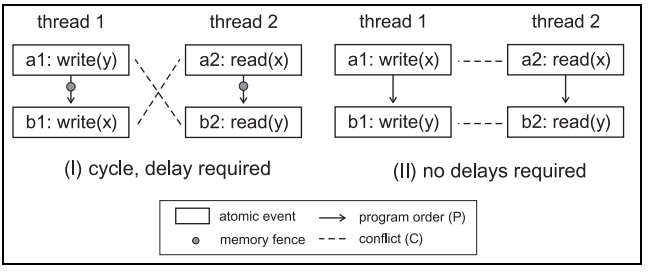

Fig. 2. Program order / conflict relation and required delays

2.3 Conflict Analysis

We use a static whole program analysis for multi-threaded Java programs to determine a conservative set of statements that are conflicting. The analysis is based on a flow-insensitive symbolic execution on an abstract thread and heap domain and tracks the thread, lock, and heap context when analyzing methods and their access to shared objects. The conflict analysis is tuned to recognize monitor-style and fork-join synchronization idioms. As it will be reported in Section 3, the effectiveness of the analysis is quite good, i.e., few or no statements are reported as conflicting for correctly synchronized programs. This is important because it narrows the scope of the communicator computation (Section 2.6). Details of the conflict analysis are discussed in [14].

2.4 Intra-thread Variable Access Ordering

A variable acts as a *communicator* if it may communicate the occurrence of updates to another thread. In Figure 2 (I), e.g., the update of `x` in statement $b1$ communicates the information that `y` has been updated (at statement $a1$) to thread2 that first reads `x` and then `y`. Hence, if access of shared variable `y` occurs in program order with respect to the communicator `x`, then executions are guaranteed to be SC; this is achieved by the memory fences in the scenario (I). The scenario (II) in Figure 2 is different, namely all threads access variables in a specific global order: first variable `x`, then `y`. Although there are conflicts, there is no cycle in the P/C relation, there are no communicator variables and hence no fences are necessary for SC.

In the following, we abstract from the read/write property of memory accesses and discuss the issue of intra-thread variable access ordering in more detail. We present a data structure that captures the intra-thread variable access ordering (Section 2.5) and finally show how that information is used to determine communicator variables (Section 2.6).

Absolute access order (AO): The execution in Figure 3 (I) exhibits critical accesses to variable `z` that occur after (in program order) the accesses to `x` and `y` in *all threads*.[1] Hence, no delay is necessary to enforce this ordering

[1] In Section 2.5, we describe a technique that allows to treat every method like a different thread. This allows to define absolute access order as "order in *all methods*".

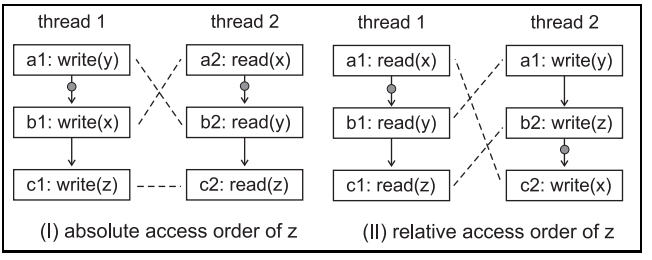

Fig. 3. Scenarios of conflicting accesses that do not require delays (for SC)

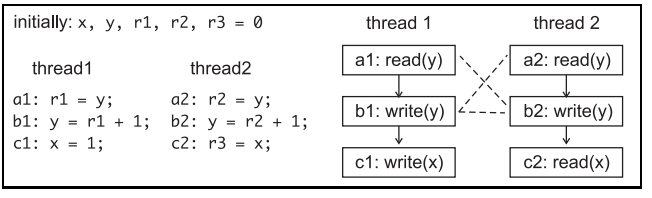

Fig. 4. Implicit ordering due to a data hazard

because the values of `z` will appear consistent to thread2 even if access to `z` is hoisted above or between the `x`, `y` access sequence. This means that for the observations of `z`, the effects of access reordering and different thread interleaving are equivalent.

Relative access order (RO): Figure 3 (II) resembles a scenario presented in [10]: The delays required in this scenario are $\{(a1, b1), (a1, c1), (a2, c2), (b2, c2)\}$. These delays are enforced by two memory fences that are specified in the figure. Intuitively, variable `x` acts as a *communicator* between threads. It is sufficient to ensure that access to shared variables occurs in program ordered with respect to the access to the communicator variable. We observe that there is a relative access order, i.e., access to `z` consistently occurs after access to `y`; hence access to `y` and `z` can be reordered among each other without changing the the memory semantics that a different thread interleaving would have also allowed.

Implicit access order due to hazard (IO): Figure 4 shows an execution with a P/C relation that is cyclic due to accesses to shared variable `y`. The procedure in [10] breaks the cycle through a delay between accesses $a1$,$b1$ and $a2$,$b2$. In this case, the WAR (write after read) hazard does, however, prevent reordering anyway and hence no explicit delay (fence) is necessary. Note that hazards are a program property and affect ordering independently of the compiler and machine architecture. SC executions allow all combinations of values in $(r1, r2, r3)$ except $(1, 1, *)$. The accesses to `x` can be reordered arbitrarily without changing this semantics (there is an absolute access order between `y` and `x`).

2.5 Variable-Order Graph

The principal idea of the algorithm is to compute a relation between shared conflicting variables that expresses ordering (absolute (AO), relative (RO), and implicit ordering (IO)) constraints that *should* be met by the program when accessing theses variables (program order). The relation is recorded in the so called *variable-order graph* that is presented in this section.

Nodes in the variable-order graph stand for shared variables. Edges in the graph represent *possible reordering* that may let actual variable access order deviate from the program order at runtime. An edge between nodes corresponding to field f_s and f_t means: in some thread, access to f_s occurs immediately before f_t in program order; however, these accesses might be reordered at runtime. To facilitate the computation of the graph at compile-time, we make the following abstractions:

- First, instead of individual variables, we only distinguish different fields; the analysis makes the conservative assumption that two variables are the same if they are the same field although the fields may belong to different object instances at runtime. While a compiler could use points-to information to distinguish object instances in certain cases, we chose to distinguish just fields to facilitate the description and also the implementation of the analysis.
- We assume that each method behaves as if it would run in its own thread. To establish any ordering that might be necessary, we require a fence at the beginning and the end of a method if necessary (Section 2.7).
- The edges in the graph correspond to a partial order of object access program order in each method. Assume that s, t are object access statements in the same method to fields f_s, f_t; both access statements are conflicting (not necessarily with statements in the same method). Then, s, t will lead to an edge $f_s \longrightarrow f_t$ in the variable-order graph if s may immediately precede t in the program order specified by the method. A loop is treated like a series of method invocations that call the loop body. Hence, a special node `meth/loop` precedes the first critical object access(es) in a loop body (or method). During the analysis of the graph, this special node expresses uncertainty about previously accessed variables and leads to a conservative placement of fences (Section 2.7).
  If s and t access the same variable and s is a read and t is a write, then no edge is created in the graph due to the WAR hazard that guarantees ordering (recall that an edge expresses a potential deviation from the program order).

Examples. Figure 5 shows two methods, their control flow, and the variable-order graph; the program corresponds to the execution in Figure 3 (II); all statements/variables in method1 and method2 are conflicting. The variable-order graph in Figure 5 (III) reflects the immediate successor relation of the basic blocks of the two methods. The nodes for `x` and `y` have the special node `meth/loop` as predecessor because those variables are accessed at the beginning of the methods.

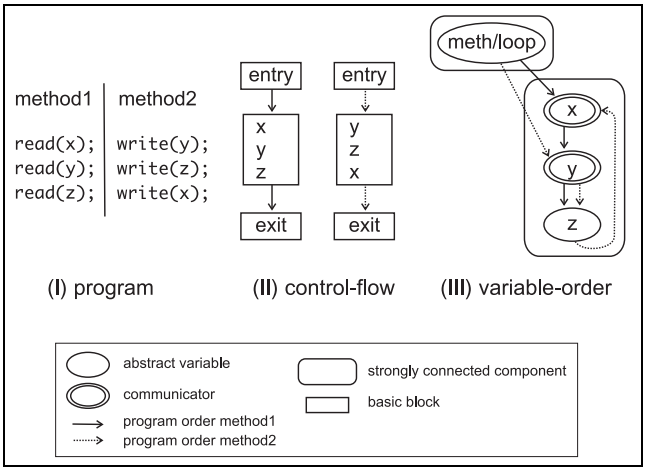

Fig. 5. Two methods, their control flow and variable-order graph

Figure 6 illustrates the variable-order graph for a complex control flow with branches and loops. The graph records the immediate program order relation among object accesses, in this case across basic block boundaries. Note that the control-flow back-edge from block 5 to block 1 is not considered when creating the variable-order graph. Hence, edge $d \longrightarrow b$ is not due to the loop, but due to the access sequence in block 6 and 7.

Algorithm. The computation of the variable-order graph starts with the creation of nodes: there is one node for every field that may be subject to a data race and additionally the special node `meth/loop`. Then, edges are established through the analysis of every method with conflicting accesses: If critical access s may be immediately followed by access t, i.e., there is no other critical access in between, then an edge is established between field f_s and f_t. In this context, the notion of "followed" considers only forward edges in the control flow.[2] No edge is created if s and t access the same variable (same field and same object) and s is a read and t is a write (reordering not permissible, see case IO in Section 2.4).

2.6 Determining Communicator Variables

A communicator variable is characterized by the fact that the variable is accessed in contexts that do not consistently provide guarantees about absolute ordering (AO) or relative ordering (RO) with respect to other shared variable accesses.

Examples. In Figure 5 (III), variable `x` is marked as a communicator because `z` occurs as an immediate predecessor of `x` (method2), there is however no absolute

[2] If s and t occur inside the same loop, then reordering along the back-edge of the loop is generally prevented by a fence at loop boundaries (Section 2.7).

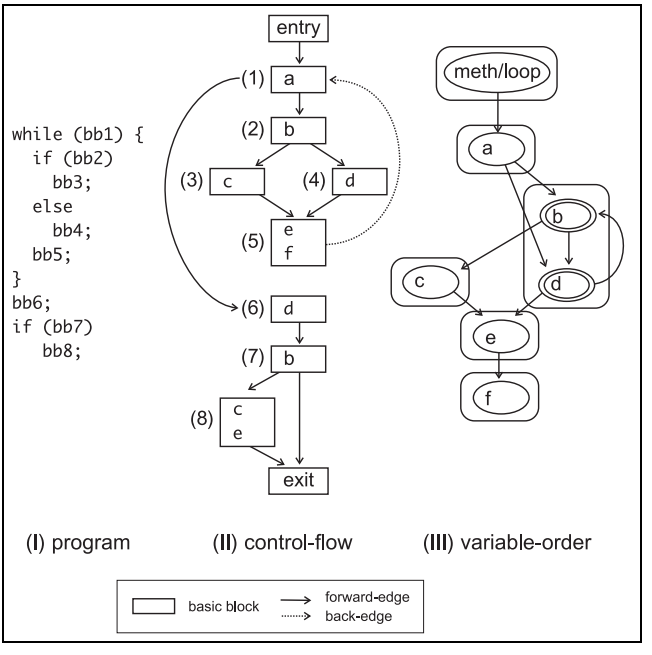

Fig. 6. Variable-order graph for control flow with branches and loops

ordering between access `z` and `x` because `z` is also reachable as a (transitive) successor of `x` (method1). There is no relative order between `z` and `x` either, because `x` has node `meth/loop` as predecessor and hence any shared variable access could immediately precede (in program order) and be reordered with access of `x`; the access that is reordered with `x` could occur in a different method or loop iteration. Similarly, variable `y` is marked as a communicator.

Algorithm. The algorithm starts with the computation of the acyclic components (ACG) of the variable-order graph. All nodes (except the special node `meth/loop`) are potential candidates for communicators; some of the nodes can be excluded from being communicators by determining the absolute and relative ordering of the accesses to the variables that these nodes stand for:

Absolute ordering (AO): Accesses to variable f are absolutely ordered in all methods with respect to other accesses if all of the following hold: (1) the node corresponding to field f is the sole node in its SCC of the variable-order graph; (2) there is no self-edge $f \longrightarrow f$.[3]

Relative ordering (RO): Accesses to variable f are relatively ordered with respect to accesses to some other variable g if all of the following hold: (1) the

[3] A self-edge means that accesses to the same variable or to the same field on different object instances might be reordered.

node corresponding to field f falls into an SCC with several other nodes; (2) the node corresponding to f has only one predecessor, namely the one representing field g, which should be inside the same SCC.

Variables that do not enjoy absolute or relative ordering in the variable-order graph are designated as communicators.

Further Examples. In Figure 6 (III), `a`, `c`, `e`, and `f` are not communicators because they enjoy *absolute ordering*, i.e., they occur strictly after (`c`, `e`, `f`) or before (`a`) accesses to `b` and `d`. In Figure 5 (III), field `z` is not a communicator because all accesses to this field enjoy *relative ordering*, i.e., they occur always after an access to `y`.

Note that it is always correct to insert additional edges in the variable-order graph, as those edges might lead to larger SCCs and hence additional – not fewer – variables are designated as communicators.

2.7 Fence Insertion

Memory fences implement the ordering of variable accesses that is required for (1) access to communicator variables, (2) method boundaries, and (3) loop boundaries. We use a simple model for a memory fence: the occurrence of a fence ensures that all preceding memory accesses complete and that no subsequent memory access starts before the fence completes. In our model, a fence affects all memory accesses independently of the type of access (read/write) and the target location.

Algorithm. The algorithm processes every method that accesses conflicting variables in three steps:

Fence due to communicator: A fence is inserted before every statement that accesses a communicator variable. In case of a read immediately followed by a write to the same variable, a fence is only inserted before the read; the following write is implicitly ordered (IO) with respect to the read.

Fence at loop boundary: A fence is inserted at the beginning of the loop header if there are accesses to conflicting variables in the loop that are not communicators. The fence is repeated at every loop iteration.

Fence at method boundary: A fence is inserted at the beginning and the end of every method that accesses conflicting variables that are not communicators. The insertion at method begin is omitted if an already inserted fence postdominates the method begin.

Handling of Methods. Figure 7 shows two programs that access variables `a` and `b`. In scenario (I), the accesses occur in the `run` method and the corresponding variable-order graph designates `a` as a communicator variable; the resulting fence instructions occur before the read access statements of `a` in method `run`.

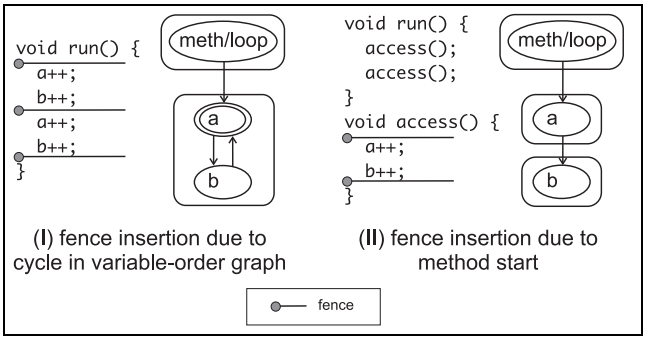

Fig. 7. Handling of methods

Scenario (II) is similar, however the access sequence `a,b` is factored out into method `access`. Although the resulting variable-order graph is different, i.e., there is no communicator because there is an absolute order in the accessed variables in each method, a fence is inserted at the beginning of method `access`, resulting in a runtime delay structure that is equivalent to the delays created in scenario (I).

2.8 Complexity

The construction of the variable-order graph treats each method once and individually. The worst case complexity *per method* is quadratic because ordering may need to be determined for all pairs of conflicting statements inside a method; for typical control flows, however, the complexity is almost linear. Note that the worst case quadratic complexity here applies only to the statements in the scope of a method. The delay set analysis in, e.g., [6,1], has polynomial, more precisely quadratic worst case complexity that applies to all statements of the program.

The analysis of the variable-order graph requires the computation of the ACG, which is linear in the size of nodes and edges. Overall, we observe the the cost of the graph construction and the computation of communicators is linear in the program size (number of conflicting statements in the program); the execution time of the overall analysis is in the order of hundreds of milliseconds on a Pentium IV 1.4 MHz system for the programs we use in Section 3.

3 Experience

We applied the communicator analysis to the following multi-threaded programs: philo is a simple dining philosopher application. elevator is a real-time discrete event simulator that is used as an example in a course on concurrent programming. jvm98_mtrt is a multi-thread raytracer from the JVM98 benchmark suite [11]. sor and tsp are data- and task-parallel applications. The jgf_xxx programs are scientific application kernels from the multi-threaded Java Grande benchmark suite [4].

We have implemented the communicator analysis in a Java-X86 way-ahead compilation environment that we used in earlier work to study static and dynamic race detection for object-oriented programs [14]. The runtime system is based on GNU libgcj version 2.96. On the Intel Pentium III architecture that we used for the study, the bidirectional memory fence described in Section 2.7 is implemented by an `add` operation with memory operand and `lock` prefix.

Table 1 specifies the number of nodes in the variable-order graph and its ACG. Due to the special node `meth/loop`, this number amounts to the total number of conflicting fields + 1. The conflicting fields are partitioned in rows *fields*, into communicators (*comm*), absolutely (*AO*), and relatively (*RO*) ordered.

The conflict analysis is quite precise for most programs and hence few field variables are specified as conflicting; for some programs that operate on shared arrays however, e.g., sor and jgf_lufact, most accesses target a single conceptual field that represents all slots of the shared arrays. Thus, for these programs a large number of access statements are classified as conflicting (they all target the same conceptual field). For jgf_raytracer, the conflict analysis is not able to determine a significant part of the application data as thread-local and hence, also for this benchmark, the imprecision of the conflict analysis is significant.

The number of nodes in the ACG show that, perhaps surprisingly, not all fields are lumped together into a single SCC without ordering; for jgf_moldyn, where this effect is especially pronounced, relative ordering can be determined for a number of field accesses. For other benchmarks, e.g., elevator, several fields enjoy absolute access ordering.

Table 2 compares the results of the naive fence insertion with the fence insertion that is guided through communicator information. The number of fences inserted by the naive strategy corresponds to the number of conflicting statements in the program. For some programs, the communicator analysis reduces this number considerably, e.g., for jgf_moldyn. Column *[% naive]* specifies the number of fences inserted due to access to communicator variables relative to

Table 1. Number of nodes in the variable-order graph and classification of fields

program	*nodes*		*fields*		
		acg	*comm*	*AO*	*RO*
philo	2	2	1	0	0
elevator	11	9	3	6	1
jvm98_mtrt	3	3	1	1	0
sor	2	2	1	0	0
tsp	5	4	3	1	0
jgf_crypt	2	2	1	0	0
jgf_lufact	4	3	3	0	0
jgf_series	4	2	2	0	1
jgf_sor	3	2	2	0	0
jgf_sparsematmult	2	2	0	1	0
jgf_montecarlo	2	2	0	1	0
jgf_moldyn	68	3	52	0	15
jgf_raytracer	12	4	10	1	0

Table 2. Number of fences inserted by the *naive* algorithm and according to our *communicator analysis*

program	*naive*	*communicator analysis*				
		stmt	*[% naive]*	*loop*	*begin*	*end*
philo	9	9	100.0	0	1	3
elevator	97	75	77.3	1	12	23
jvm98_mtrt	10	5	50.0	0	2	5
sor	42	42	100.0	0	3	3
tsp	62	47	75.8	0	7	11
jgf_crypt	16	16	100.0	0	0	1
jgf_lufact	43	39	90.7	0	9	11
jgf_series	18	11	61.1	0	4	6
jgf_sor	32	27	84.4	0	3	3
jgf_sparsematmult	2	0	50.0	1	1	1
jgf_montecarlo	2	0	0.0	0	0	0
jgf_moldyn	744	397	53.4	0	1	11
jgf_raytracer	103	88	86.4	1	5	20
average			**71.5**			

the naive insertion strategy. The static reduction of inserted fences relative to the naive strategy is on average 28.5%. Besides the naive strategy, there are more precise algorithms to determine the delay set, e.g., [1,6,12]. A comparison of our communicator analysis to these analyses (some of which are restricted to SPMD style programs) is not done here and is left for future work.

The number of fences inserted at at loop boundaries (column *loop*), at method begin (column *begin*), and at method end (column *end*) is moderate for most benchmarks; these numbers are not included in the relative comparison with the naive fence insertion strategy (column *[% naive]*). Note that the insertion of fences at method boundaries is not due to an inherent property of the computation of the program (e.g. conflicting access to shared variables) but merely due to the structure of the implementation.

Table 3 reports the runtime of the *orig*inal and instrumented benchmarks, both with moderate optimization (copy-propagation, partial redundancy elimination, no inlining). The numbers specify the average duration of three program runs. The effective cost of the memory fences is very high, especially for programs where the conflict analysis is unnecessarily conservative, i.e., for sor, tsp, jgf_lufact, jgf_moldyn, and jgf_raytracer. The static difference in the number of inserted fences between the naive and the communicator approach affects the runtime situation only in part: there is a significant benefit for jgf_sparsematmult and jgf_lufact, not for jgf_moldyn and jgf_raytracer however. In some situations, the naive approach of fence insertion yields better performance than the communicator analysis, e.g., for tsp, jgf_moldyn, and jgf_raytracer. Such a situation can occur, if the naive approach places fences in the body of a conditional branch that is rarely taken at runtime, while the communicator-based approach requires a fence at method start and end.

Overall, the reduction of fences inserted at memory access statements (Table 2, *communicator* vs. *naive* on avg. 28%), does not necessarily imply a re-

Table 3. Execution times for different fence insertion algorithms on a Pentium III 933 MHz system (average of three runs). The programs philo and elevator are not CPU-bound and hence omitted from the reporting. The absolute times are rounded to tens of seconds, the percentage numbers are computed from precise values

program	*orig*	*naive*		*communicator analysis*		
	[s]	*total*	*[% orig]*	*[s]*	*[% orig]*	*[% naive]*
jvm98_mtrt	19.8	20.1	1.7	19.8	0.2	-1.5
sor	2.4	7.6	218.2	7.6	218.2	0.0
tsp	7.6	12.0	57.3	12.6	65.1	5.0
jgf_crypt	3.5	4.0	14.1	4.0	14.3	0.2
jgf_lufact	3.4	8.7	156.8	8.2	141.1	-6.1
jgf_series	25.8	25.8	0.1	25.8	0.0	0.0
jgf_sor	31.9	46.9	47.0	46.8	46.7	-0.2
jgf_sparsematmult	17.4	25.0	43.4	19.3	11.2	-22.5
jgf_montecarlo	23.7	23.8	0.4	23.8	0.4	0.0
jgf_moldyn	23.1	57.9	150.8	60.6	162.4	4.6
jgf_raytracer	42.0	118.7	182.7	139.2	231.4	17.2
average			79.3		**81.0**	-0.3

duction of fences that are executed at runtime. Although we have not studied the frequency of executed fences at runtime in detail, the execution times in Table 3 (on average, there is almost no difference between *communicator* vs. *naive*) reflect this observation. There are basically two reasons for this behavior: First, the communicator analysis may allow to remove fences that are "rarely" executed, not those that are critical to the execution time. Second, the overhead due to fences at method boundaries can outweigh the benefit of fences removed by the communicator analysis.

4 Conclusions

We have presented an algorithm to determine communicator variables in parallel programs. If communicator variables are accessed in program order and access to other shared variables is not reordered with respect to communicators, then program executions are sequentially consistent. Computing communicators is an efficient and effective alternative to delay set computation, especially for object-oriented multi-threaded programs with unstructured parallelism.

We have applied the analysis in combination with a powerful conflict analysis [14] to a number of multi-threaded Java programs. Using communicators as a guide for the fence insertion yield on average about 28% fewer fences at memory access statements than a naive fence insertion algorithm. The total runtime overhead due to the fences is however still considerable (81% on average). For programs with significant overhead, we observe that this overhead is mainly due to conservatism in the automated conflict analysis (especially for programs that mainly operate on shared arrays), not due to the inability of the communicator analysis to reduce the number of fences.

Acknowledgments

We thank Thomas Gross for his support and comments on the paper. We thank the anonymous referees, Rajkishore Barik, and Zehra Sura for their comments and insightful discussions. We thank Matteo Corti for his contributions to the compiler infrastructure.

References

1. W.-Y. Chen, A. Krishnamurthy, and K. Yelick. Polynomial-time algorithms for enforcing sequential consistency in SPMD programs with arrays. In *Proceedings of the International Workshop on Compilers for Parallel Computing (LCPC'03)*, Oct. 2003.
2. J.-D. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical Report RC22146, IBM Research, Aug. 2001.
3. X. Fang, J. Lee, and S. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the International Conference on Supercomputing (ICS'03)*, pages 285–294, June 2003.
4. Java Grande Forum multi-threaded benchmark suite. `http://www.epcc.ed.ac.uk/javagrande/`.
5. A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38(2):130–144, Nov. 1996.
6. M. Kurhekar, R. Barik, and U. Kumar.exit An efficient algorithms for computing delay set in SPMD programs. In *Proceedings of the International Conference on High Performance Computing (HiPC'03)*, Oct. 2003.
7. S. Midkiff, J. Lee, and D. Padua. A compiler for multiple memory models. In *Rec. Workshop Compilers for Parallel Computers (CPC'01)*, June 2001.
8. G. Naumovich, G. Avrunin, and L. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 338–354, Sept. 1999.
9. R. Netzer and B. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, Mar. 1992.
10. D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. on Programming Languages and Systems*, 10(2):282–312, Apr. 1988.
11. SPEC JVM98 Benchmarks. `http://www.spec.org/osg/jvm98`.
12. Z. Sura, C.-L. Wong, X. Fang, J. Lee, S. Midkiff, and D. Padua. Automatic implementation of programming language consistency models. In *Record of the Workshop on Compilers for Parallel Computers (CPC'03)*, Jan. 2003.
13. C. von Praun and T. Gross. Object race detection. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 70–82, Oct. 2001.
14. C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'03)*, pages 115–129, June 2003.