A Technique for the Effective and Automatic Reuse of Classical Compiler Optimizations on Multithreaded Code

Pramod G. Joisha, Robert S. Schreiber, Prithviraj Banerjee, Hans-J. Boehm, Dhruva R. Chakrabarti

HP Laboratories
HPL-2010-81R1

Keyword(s):
compilers, parallel-program optimization, multithreading, data-flow analysis

Abstract:
A large body of data-flow analyses exists for analyzing and optimizing sequential code. Unfortunately, much of it cannot be directly applied on parallel code, for reasons of correctness. This paper presents a technique to automatically, aggressively, yet safely apply sequentially-sound data-flow transformations, *without change*, on shared-memory programs. The technique is founded on the notion of program references being "siloed" on certain control-flow paths. Intuitively, siloed references are free of interference from other threads within the confines of such paths. Data-flow transformations can, in general, be unblocked on siloed references.

The solution has been implemented in a widely used compiler. Results on benchmarks from SPLASH-2 show that performance improvements of up to 41% are possible, with an average improvement of 6% across all the tested programs over all thread counts.

External Posting Date: December 6, 2010 [Fulltext] Approved for External Publication

Internal Posting Date: December 6, 2010 [Fulltext]

Originally internally and externally posted on July 22, 2010.

An abridged version will be published at the 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2011).

A Technique for the Effective and Automatic Reuse of Classical Compiler Optimizations on Multithreaded Code

Pramod G. Joisha, Robert S. Schreiber, Prithviraj Banerjee, Hans-J. Boehm, Dhruva R. Chakrabarti

Hewlett-Packard Laboratories, Palo Alto, USA

{pramod.joisha, rob.schreiber, prith.banerjee, hans.boehm, dhruva.chakrabarti}@hp.com

Abstract

A large body of data-flow analyses exists for analyzing and optimizing sequential code. Unfortunately, much of it cannot be directly applied on parallel code, for reasons of correctness. This paper presents a technique to automatically, aggressively, yet safely apply sequentially-sound data-flow transformations, *without change*, on shared-memory programs. The technique is founded on the notion of program references being "siloed" on certain control-flow paths. Intuitively, siloed references are free of interference from other threads within the confines of such paths. Data-flow transformations can, in general, be unblocked on siloed references.

The solution has been implemented in a widely used compiler. Results on benchmarks from SPLASH-2 show that performance improvements of up to 41% are possible, with an average improvement of 6% across all the tested programs over all thread counts.

1. Introduction

It has long been known that *classical* compiler optimizations, i.e., sequentially-sound transformations based on data-flow analysis frameworks [20], cannot be directly applied on parallel code, even under conditions that would be correct for the sequential case [28, 22, 24, 32, 36, 6]. The problem stems from asynchronous updates. Classical methods were not designed to reason about them under a multiplicity of interleavings [22, 36]. Parallel-code optimization has hence been specially addressed, in mainly two ways:

- Devise analyses and optimizations from the ground up, or adapt existing sequential analyses and optimizations, often using specialized program representations [41, 22, 37, 24, 32, 36].
- Assume the program to be *well-synchronized*, i.e., free of data races, and restrict the scope of classical transformations to the *synchronization-free regions* (SFRs) of the code.

The first has been used to analyze and optimize the so-called *explicitly parallel program* (EPP). An EPP is shared-memory code in which parallelism is expressed using the `cobegin`/`coend` construct, or some equivalent. The second is how all production C/C++ compilers that we are aware of, such as the GNU C/C++ Compiler (`gcc`) and Open64, currently optimize multithreaded code.

The SFR approach promises *sequential consistency* (SC) [23, 1] to the programmer, if the code is data-race free. The EPP approach usually gives "full SC", i.e., SC even in the presence of data races.

1.1 Limitations of Past Approaches

Since the EPP approach ensures interleaving semantics, it can produce overly conservative results on well-synchronized programs. The reason is that to assure full SC, all *conflicting accesses* that *could* be performed by two threads, without them performing intervening synchronizations, have to be modeled [42].[1] Such accesses do not exist in well-synchronized code, since they would be data races. Thus, it would be easier to deduce in well-synchronized code, for example, whether the expression `x+x` is even.

Second, neither Pthreads [18] nor OpenMP [33] currently define semantics under data races. Neither do the current drafts of the C and C++ standards.[2] Implementations already take advantage of this fact, for example, by reordering memory operations on possibly shared locations. Therefore, it appears wasteful to artificially restrict an analysis not to do the same.

Third, since the EPP approach typically relies on IRs (intermediate representations) that go beyond traditional sequential IRs, it can incur high infrastructure costs. For instance, an EPP IR may include special nodes like ψ- and π-functions [41, 24]. It may include edges to reflect properties peculiar to a parallel setting, such as conflicts [24] and synchronizations [37, 32]. To exploit the information borne by these new nodes and edges, existing transformations will have to be reworked. Hence, incorporating the approach into a compiler either means a from-scratch enterprise, or extending a serial compiler's phase or recasting it to a parallel IR. Whichever way, the undertaking is expensive. In contrast, this paper's approach allows for the *direct* reuse of existing data-flow transformations.

In the SFR approach, *data statements* (i.e., synchronization-free statements [2]) are modeled without concurrency considerations. For example, may-definition may_{def} and may-use may_{use} sets, which form the basis of data-flow analyses, do *not* account for concurrent accesses at data statements. These accesses are considered only at synchronizations. In compilers like `gcc`, this presently happens automatically, albeit exceedingly conservatively, because synchronizations are viewed as fully opaque. It should be emphasized that data-race freedom is essential for this approach.

1.1.1 The Pitfall of Simply Extending Optimization Scopes

Consider the multithreaded program in Figure 1, which uses the Pthreads library [18]. The snippets in the columns, along with initialization and termination code, fully constitute the program. The left and right snippets are only executed by threads h_1 and h_2 respectively. The only shared variables `X` and `Y` are initialized to 0. Despite accesses of `X` not being protected by a lock, the program is data-race free. (The two threads coordinate on `Y`, which is always accessed with a lock held.) This program always outputs 0.

In the left snippet, the only access of `X` is the read on Line 0. A compiler, on examining it in isolation, might conclude that an opportunity to propagate through critical sections exists, from Line 0 to Line 8. Interestingly, this would preserve data-race freedom, un-

[1] Accesses of the same location, all of which are not reads, conflict [40].

[2] Java and .NET presently do define limited semantics for data races, though it remains unclear whether this can be done usefully and fully correctly [39]. The approach presented here would require adjustment for those languages.

```
                         X == Y == 0
          Thread h1                 |          Thread h2
                                    |
0 t1 := X                           | 0' do { pthread_mutex_lock(l)
1 pthread_mutex_lock(l)             | 1'    t4 := Y
2 Y := 1                            | 2'    pthread_mutex_unlock(l)
3 pthread_mutex_unlock(l)           | 3' } while (t4 == 0)
4 do { pthread_mutex_lock(l)        | 4' X := 11
5    t2 := Y                        | 5' pthread_mutex_lock(l)
6    pthread_mutex_unlock(l)        | 6' Y := 2
7 } while (t2 == 1)                 | 7' pthread_mutex_unlock(l)
8 t3 := t1                          |
9 print(t3)                         |
```

Figure 1. The two snippets comprise the program, which always prints 0. If X were copy propagated from Line 0 to Line 8, the result would be data-race free, but 11 would always be printed instead.

like many past examples of invalid transformations, such as bitfield manipulations and register promotion [6]. Nevertheless, it is incorrect since the transformed program would always output 11.[3]

The root of the problem is that the right snippet updates X when running concurrently with the left snippet. Because detecting this requires more analysis, and potentially of a whole-program nature, current compilers follow the SFR approach and safely avoid the problem because a synchronization like pthread_mutex_lock, as well as any procedure that may transitively call it, is regarded as having a possible side-effect on all globally visible data.

1.1.2 Benefits of Judiciously Extending Optimization Scopes

The opaque treatment of synchronizations has the adverse consequence that legally exploitable opportunities could, at the same time, get blocked. For instance, consider another hypothetical program consisting of the left snippet in Figure 1 but a different right snippet, one that does not update X. As before, assume that only h_1 executes the left snippet and h_2 the right snippet. In this situation, it would be correct to copy propagate X from Line 0 to Line 8.

We now discuss an example, derived from real code, where synchronization opacity hinders the recovery of useful knowledge that is obfuscated by IR lowering. Figure 2 displays a critical section originally from SPLASH-2's FMM benchmark [47]. The only difference from the original are the calls to the LockedPrint procedure, which FMM itself provides. LockedPrint writes to stdout, under the protection of a lock. The calls have been randomly inserted to illustrate the additional complexities posed by an arbitrary number of nested critical sections to the recovery process.

Modern compilers often work with a variant of the Static Single Assignment (SSA) Form [12] that takes into account aliasing effects and indirect memory operations, such as the HSSA Form by Chow et al [11]. gcc's Memory SSA Form is akin to HSSA [31]. Therefore, for pedagogical reasons, we base our discussions on HSSA.

Figure 2 also shows the critical section's lowered HSSA Form, in which special constructs, called "χ assignments" and "μ operations", denote ambiguous definitions and uses [11]. The χ assignments on Lines 3, 6 and 11 are examples—they signify that a possible side-effect of the preceding calls, including *concurrent effects*, is the changed global variable G_Memory.[4] Since these calls can have other side-effects, such as changed heap objects, there can be more χ assignments between Lines 3 and 4, 6 and 7, and 11 and 12.

```
LOCK(G_Memory->count_lock);
my_id = G_Memory->id;
LockedPrint("%d\n", my_id);
G_Memory->id++;
LockedPrint(...);
UNLOCK(G_Memory->count_lock);
```

```
            ⇓
0 t1 := G_Memory
1 t2 := &(t1->count_lock)
2 pthread_mutex_lock(t2)
3 t' := χ(G_Memory)
  ⋮
```

```
4  t3 := t'->id
5  LockedPrint("%d\n", t3)
6  t'' := χ(G_Memory)
   ⋮
7  t4 := t''->id
8  t5 := t4+1
9  t''->id := t5
10 LockedPrint(...)
11 t''' := χ(G_Memory)
   ⋮
12 t6 := &(t'''->count_lock)
13 pthread_mutex_unlock(t6)
```

Figure 2. A critical section originally from SPLASH-2's FMM program, and its lowered HSSA Form. Lowering hides the equivalence of the pointers t_2 and t_6. A challenge is to recover such knowledge under multithreading, by safely applying classical methods.

All lines except 2, 5, 10 and 13 are purely data statements. Under data-race freedom, concurrent effects need not be accounted for against them, because the χ assignments against synchronizations account for these effects. This, however, stymies a classical optimizer from establishing the equivalence of t_2 and t_6. A standard application of a pointer analysis will not uncover this equivalence. If t_2 and t_6 had nonsingleton may-points-to sets, nothing can be said about their equivalence.[5] Likewise, no equivalence conclusions can be drawn if t_2 and t_6 have empty must-points-to sets. We shall later see, in Section 4.7.1, how a classical optimizer can be transparently empowered to safely regain such knowledge.

1.2 A New Approach: The Siloed-References (SR) Technique

This paper presents a two-step technique for extending the scope of arbitrary classical optimizations *across* synchronizations. First, program references to objects that can be statically proved to be free of cross-thread interference are determined. The specific attribute our analysis identifies is the siloed property. Informally, an object reference is "siloed" on a procedure f if no other thread writes (reads or writes) the object whenever a thread is executing a path in f in which it reads (writes) the object. Second, the IR's may_{def} and may_{use} sets are narrowed using these siloed references—this unveils previously blocked classical transformation opportunities. Since these abstractions are only narrowed, the outcome of the technique is not the same as synchronization removal [10, 5, 7, 35].

Our methods use two existing compiler algorithms, designed for sequential programs, as building blocks. They are a flow-insensitive interprocedural pointer analysis, and a transformation to an HSSA-like form. The former is used to derive aliasing information. Its usage is valid because it is known that flow-insensitive schemes retain correctness in a multithreaded context [36, Page 71]. The usage of the latter is also valid because synchronizations are, by default, treated as opaque procedure calls. They can hence affect all globally visible memory. By generating new symbols at these calls, the HSSA conversion phase conservatively models the asynchronous updates that could occur in a data-race-free program.

1.2.1 Advantages

Because the SR Technique sharpens key data-flow abstractions, it permits arbitrary *bidirectional* classical data-flow analyses across synchronizations, in well-synchronized programs. As far as we know, this is a first. Previous work on program transformations for

[3] The example assumes a fair scheduler, so the program is guaranteed to make progress. It would have to be modified for the case of a nonpreemptive uniprocessor scheduler, by suitably inserting thread-yielding operations.

[4] For brevity, μ operations have not been shown.

[5] A *sound* may-points-to set is never ∅ because a pointer will always point to something. ("Special" pointer values, such as NULL, are modeled by separate targets.) So nonsingleton here means two or more elements.

the data-race-free model only considered what was effectively *forward* data-flow through a lock synchronization, and *backward* data-flow through an unlock synchronization [39]. This is insufficient, for instance, to expose the equivalence of t_2 and t_6 in Figure 2. And in previous work for the full SC model, synchronization knowledge was only used to eliminate conflict edges in the EPP IR [32, 42].

The second important advantage is that the technique needs no new IR constructs, no modifications to the semantics of existing constructs, and no changes to existing phases. In fact, existing phases are treated as black boxes. Because the technique only prunes may_{def} and may_{use} sets, dropping a phase that implements it into a serial compiler's phase-pipeline will automatically "energize" downstream data-flow phases to better optimize parallel code.

1.3 Paper Overview

The rest of this article is organized as follows. Section 2 lays out the requisites for our work. Section 3 shows how to build an abstraction called the Procedural Concurrency Graph (PCG) that enables the calculation of concurrency and interference information. Siloed references are computed from the PCG, using the algorithm presented in Section 4. This section also explains how a data-flow analysis can take advantage of the computed information. An important tradeoff involving interference is described in Section 5. Experiments on an implementation in `gcc` are reported in Section 6. Finally, Section 7 discusses related work, and Section 8 concludes.

2. Preliminaries

The term "object" in this paper has the sense used in the C Standard—it means any named piece of storage in the execution environment [19]. To simplify the presentation, our usage of this term will encompass procedures. Two objects are distinct if either their lifetimes or address ranges are not identical. Thus, if `x` is a `struct` variable with field `f`, then `x` and `x.f` are distinct objects (although the first contains the second). As in the C Standard, we use the term *lvalues* for syntactic expressions that refer to objects, such as `x` and `x.f`. Specific and arbitrary lvalues will be displayed, respectively, with typewriter and italicized fonts.

2.1 The Program's Call Universe

A program comprises parts for which an IR is available, and parts for which it is not—we call the former *user code*. A procedure invoked in user code is either *defined*, *abstracted* or *inscrutable*. It is defined if its IR is known, abstracted if undefined but with a summary capable of describing its effects at call sites, and inscrutable if undefined and not abstracted. These classes form the sets $\mathscr{F}_d$, $\mathscr{F}_a$ and $\mathscr{F}_{inscr}$—together, they constitute the program's call universe.

If $STMTS(f)$ is the set of statements in a defined procedure f, then $\mathscr{U} = \cup_{f \in \mathscr{F}_d} STMTS(f)$ is the set of all user-code statements. An $s \in \mathscr{U}$ is either non-call code or a call.[6] The function *proc* maps a call to the callee in $\mathscr{F}_d \cup \mathscr{F}_a \cup \mathscr{F}_{inscr}$, and non-call code to $\top$.

2.2 Synchronizations

It is posited that a subset of $\mathscr{F}_a \cup \mathscr{F}_{inscr}$ are *synchronizing procedures*, and that the program utilizes them to attain *data-race freedom* [2]. A "synchronization" is a synchronizing-procedure call.

Let $\mathscr{U}_{sync}$ be the set of all synchronizations in $\mathscr{U}$. Statements in a fixed subset $\mathring{\mathscr{U}}_{sync}$ of $\mathscr{U}_{sync}$ are postulated to have changeable may_{def} and may_{use} sets. All other user-code statements, i.e., $\mathscr{U} - \mathring{\mathscr{U}}_{sync}$, have fixed may_{def} and may_{use} sets—these will be referred to as the program's "unaffected statements".

There is full latitude in choosing $\mathring{\mathscr{U}}_{sync}$. As will be shown in Section 5, the choice influences the precision of the PCG and depends on the PCG's intended use. For instance, if unblocking optimization opportunities is the goal, $\mathring{\mathscr{U}}_{sync}$ consists of synchronizations whose may_{def} and may_{use} sets are to be sharpened. $\mathring{\mathscr{U}}_{sync}$ are then the "interesting synchronizations", as far as this goal is concerned.

2.3 On Lvalues

An lvalue can refer to several objects. For example, if `r` is a pointer variable that is only assigned the returned value of a `malloc` call, then the lvalue `*r` denotes all objects created at that allocation site.

2.3.1 Aliasing

Lvalues *alias* if they name overlapping objects. For instance, if `p` and `q` are pointer variables, `*p` and `*q` alias if `p` and `q` target overlapping objects. By this definition, `p` and `q` themselves do not alias because they name nonoverlapping objects in C. Hence, aliasing in this paper is not a points-to relation. The predicate $x \sim y$ is true if the lvalues x and y may alias. We use the *set-aliasing operator* $\approx$ to find the may-aliases in the lvalue sets X and Y:

$$X \approx Y = \{z \mid \exists x \in X, \exists y \in Y. \big(x \sim y \wedge (z \equiv x \vee z \equiv y)\big)\}, \qquad (1)$$

where $u \equiv v$ is true iff u and v are identical lvalues. As an example, if $*\mathtt{p} \sim \mathtt{y}$, $*\mathtt{p} \not\sim *\mathtt{q}$, $*\mathtt{q} \not\sim \mathtt{x}$, then $\{*\mathtt{p}, \mathtt{x}\} \approx \{*\mathtt{q}, \mathtt{y}\} = \{*\mathtt{p}, \mathtt{y}\}$.

2.3.2 Renaming Assumptions

Without loss of generality, local variables in user code are assumed to be appropriately renamed so that no two defined procedures declare local variables with the same name. Also assumed is an SSA-based IR for defined procedures that takes into consideration aliasing effects and indirect memory operations, such as HSSA [11]. For instance, HSSA renames two occurrences of `*r` to different versions if `r` is redefined between them, even if `*r` is not redefined. These two assumptions ensure that identical lvalues, irrespective of where they occur in user code, always name the same set of objects.

2.3.3 Important Lvalue Sets

Suppose $\mathscr{L}$ is the set of all lvalues that may be accessed in user code. An object is either *user-visible* or *user-invisible*, depending on whether there is an lvalue in $\mathscr{L}$ that refers to it. Not all objects existent in a run are user-visible. For example, external library state that cannot be referred to by any lvalue in $\mathscr{L}$ is user-invisible.

Our analyses are underpinned by a bunch of lvalue sets: $\mathscr{L}_h$, $\mathscr{L}_{inscr}$, R_i, W_i, $\overline{R}_i$, $\overline{W}_i$, *SYNC* and $\overline{SYNC}$. Most of them, as well as sets derived from them, contain only lvalues for user-visible objects. This is because the SR Technique prunes may_{def} and may_{use} sets, which normally are subsets of $\mathscr{L}$.[7] Exceptions are the *SYNC* and $\overline{SYNC}$ sets, which may include *virtual lvalues* for covering sync objects strictly internal to undefined procedures.

$\mathscr{L}_h$: User-visible objects allocated on the heap. We represent by $\mathscr{L}_h$ a subset of $\mathscr{L}$ that covers all user-visible objects that may be allocated on the heap. To ascertain $\mathscr{L}_h$, only undefined-procedure calls need to be considered. It is assumed we are given a set $\mathscr{H} \subseteq \mathscr{L}$ covering all user-visible heap objects that may be allocated by abstracted-procedure calls. For instance, if `malloc` is an abstracted procedure, and if `r` is assigned its result, then `*r` is in $\mathscr{H}$. When inscrutable-procedure calls are absent, $\mathscr{H}$ is a safe choice for $\mathscr{L}_h$. But not when they are present, since they could allocate heap objects and later expose them to user code in myriad ways. Since exposure implies being accessible through some pointer-type lvalue

[6] Indirect calls are modeled as sets of direct calls to possible targets. This is to simplify the presentation; the implementation retains indirect calls as is.

[7] It may sometimes be more convenient to model all user-invisible objects with a single virtual lvalue *ui*, and to define $\mathscr{L}$ as including *ui*.

in $\mathscr{L}$, the following is a conservative formulation for $\mathscr{L}_h$:

$$\mathscr{L}_h = \begin{cases} \mathscr{H} & \text{if } proc(s) \notin \mathscr{F}_{inscr} \forall s \in \mathscr{U}, \\ \{*x \mid x \in \mathscr{L} \wedge (\text{type of } x \text{ is } \texttt{T*} \\ \qquad \text{where } \texttt{T} \text{ is not } \texttt{void})\} & \text{otherwise.} \end{cases} \tag{2}$$

It should be stressed that the second case in Equation (2) is very conservative. For example, if none of the invoked inscrutable procedures have pointer parameters or return pointers, and if none of the pointer variables in $\mathscr{L}$ have *external linkage* [19], then choosing just $\mathscr{H}$ for the second case is also safe.

$\mathscr{L}_{inscr}$: User-visible objects immediately accessible in inscrutable procedures. Let $\mathscr{L}_{addr} \subseteq \mathscr{L}$ be the set of all variables and procedures whose addresses are taken in user code.[8] Let $\mathscr{L}_e \subseteq \mathscr{L}$ be the set of all variables and procedures that have external linkage. An lvalue is *immediately accessed* in a procedure f if a memory operation m in f reads or writes it. m is then an immediate access. The following lvalue set conservatively covers user-visible objects that may be immediately accessed in any inscrutable procedure:

$$\mathscr{L}_{inscr} = \mathscr{L}_{addr} \cup \mathscr{L}_e \cup \mathscr{L}_h. \tag{3}$$

There is potential for greater accuracy by calculating $\mathscr{L}_{inscr}$ differently for different categories of inscrutable procedures. For instance, inscrutable procedures that belong to external libraries predating the user code can never *explicitly* access lvalues in $\mathscr{L}_e$. Therefore, $\mathscr{L}_{inscr}$ for that category can be set to just $\mathscr{L}_{addr} \cup \mathscr{L}_h$.

The static call graph models both defined and undefined procedures. For an inscrutable-procedure call-graph node, procedure lvalues in $\mathscr{L}_{inscr}$ will cover all of its immediate successors.

R_i, W_i, $\overline{R}_i$, $\overline{W}_i$: User-visible objects that may be immediately accessed. We denote the sets of lvalues that may be immediately read and immediately written at a statement s as $R_i(s)$ and $W_i(s)$. Determining these sets is simple when $proc(s) \in \{\top\} \cup \mathscr{F}_d$.

But when an undefined procedure g is called at s, there is the issue of accounting for the accesses that occur in it and procedures that it reaches in the call graph. Our approach is to include in $R_i(s)$ and $W_i(s)$ the reads and writes immediate to procedures that lie in a call-graph path of undefined procedures starting at g. In other words, the treatment is as if g, along with any undefined procedure that it may reach in the call graph without going through a defined procedure, were inlined at the call site s. Thus, the effects of undefined-procedure calls are accounted for on the caller's side. This is basically a context-sensitization—it avoids spuriously conflating the undefined procedure's effects over several call sites.

An example is the call `pthread_mutex_lock(`l`)`. Its $R_i(s) = \{l, *l\}$ and $W_i(s) = \{*l\}$. (A synchronization is a read, and conservatively, also a write since the state of the operated-on sync object can change.) Another example is `r := scanf(p, q)`. Its $R_i(s) = \{\texttt{p}, \texttt{q}, *\texttt{p}, \texttt{stdin}, *\texttt{stdin}\}$ and $W_i(s) = \{\texttt{r}, *\texttt{q}, *\texttt{stdin}\}$.[9]

We postulate the existence of the partial functions $\check{R}_i$ and $\check{W}_i$ that give the immediate read and immediate write sets when s is either an abstracted-procedure call (such as the previous `scanf` and `pthread_mutex_lock` invocations) or not a call. These functions can be used to compute $R_i(s)$ and $W_i(s)$ for any s, as shown below:

$$R_i(s) = \begin{cases} largs(s) & \text{if } proc(s) \in \mathscr{F}_d, \\ largs(s) \cup \mathscr{L}_{inscr} & \text{else if } proc(s) \in \mathscr{F}_{inscr}, \\ \check{R}_i(s) & \text{otherwise,} \end{cases} \tag{4}$$

$$W_i(s) = \begin{cases} \{ret(s)\} & \text{if } proc(s) \in \mathscr{F}_d, \\ \{ret(s)\} \cup \mathscr{L}_{inscr} & \text{else if } proc(s) \in \mathscr{F}_{inscr}, \\ \check{W}_i(s) & \text{otherwise.} \end{cases} \tag{5}$$

The first two cases in Equations (4) and (5) are when s invokes a defined or inscrutable procedure. They use the partial functions *largs* and *ret*, which give the set of argument lvalues and the lvalue assigned the returned result for an invocation s. That is, if s is z `:=` f`(`x_1`,` x_2`, ...,` x_k`)`, then $largs(s) = \{x_1, x_2, \ldots, x_k\}$ and $ret(s) = z$. Because the second case in both equations coincides with an inscrutable-procedure invocation, $\mathscr{L}_{inscr}$ is included in $R_i(s)$ and $W_i(s)$. The third case is when $proc(s) \notin \mathscr{F}_d \cup \mathscr{F}_{inscr}$. Since this corresponds to either non-call code or an abstracted-procedure call, $R_i(s)$ and $W_i(s)$ can be simply expressed as $\check{R}_i(s)$ and $\check{W}_i(s)$.

Lvalues that are immediately accessed at the granularity of a defined procedure f can be computed as follows:

$$\overline{R}_i(f) = \bigcup_{s \in STMTS(f)} R_i(s), \quad \overline{W}_i(f) = \bigcup_{s \in STMTS(f)} W_i(s). \tag{6}$$

SYNC, $\overline{SYNC}$: Sync objects. Synchronizations communicate using sync objects. Locks, barriers and condition variables are examples of sync objects. A partial function *SYNC* is posited that gives an lvalue set covering the user-visible and user-invisible sync objects that may be immediately or *transitively* accessed at a synchronization.[10] *SYNC* can be used to find $\overline{SYNC}(f)$, the set of lvalues for all sync objects that may be accessed by a thread executing f:

$$\overline{SYNC}(f) = \bigcup_{s \in \mathscr{U}_{sync} \cap STMTS(f)} SYNC(s). \tag{7}$$

Although user-invisible sync objects are not exposed to user code, they may be needed for analysis, such as when building the PCG. They will then be named by the virtual lvalue ui_{sync}.

2.4 Notational Conventions

Names of program-wide sets use a calligraphic font for the main lettering—e.g., $\mathscr{F}_d$, $\mathscr{F}_{FOLLOW}$ and $\mathscr{L}_h$. Functions with a procedure domain may sometimes have names that share common letters with other names—e.g., $\overline{R}_i$ and R_i, $\overline{W}_i$ and W_i, and $\overline{I}_i$ and I_i. A bar on top is then used to distinguish them. A subscript i is used when a set only consists of immediately accessed lvalues. An index that maps a notation to its pertinent equation and/or section is in the appendix.

3. Building the Procedural Concurrency Graph

The PCG indicates whether a pair of procedures may concurrently execute, and if so, objects accessed in one that may "interfere" with objects accessed in the other. Our approach to building the PCG is to begin with a solution that is possibly imprecise but assuredly correct, and to then subject it to a series of transformations, called refinements, that progressively improve its precision.

3.1 PCG Definition

A PCG is the labeled undirected graph $G_p = (\mathscr{F}_d, E, I_i)$, where $I_i : E \mapsto 2^{\mathscr{L}}$ is the *immediate interference function*. Nodes in $\mathscr{F}_d$ correspond to the defined procedures in user code. An edge $(a, b) \in E$ means that the execution of a by one thread may overlap with

[8] Use of the `&` operator is not a requisite for an "address-taken" qualification. In C, procedure and array names can be directly assigned to pointers [19].

[9] `*q` may not be written, for instance, if assignment suppression is used [19]. But including `*q` in $W_i(s)$ is safe since these sets encode "may information".

[10] An object is transitively accessed at a call site s if it is immediately accessed in a procedure that lies in a call-graph path from the callee at s.

an execution of b by a different thread—this is the standard MHP (may-happen-in-parallel) relation [29]. Then, $I_i\big((a,b)\big)$ is the set of lvalues on which a and b may immediately interfere. Since a could run in parallel with itself, G_p can have self-loops.

3.1.1 Immediate Interference

The *overlapping regions* of two control-flow graph (CFG) paths P and P', executed by different threads, are subpaths $p_1 p_2$ in P and $p_3 p_4$ in P'—where p_1 through p_4 are points—such that p_1 abuts p_3 and p_2 abuts p_4 in some interleaving. Given this, two procedures f and f' are said to *immediately interfere* on an lvalue x if there exist two immediate accesses, m of x in f and m' of an lvalue x' in f', for which the following conditions simultaneously hold:

C1. m and m' conflict—i.e., at least one is a write, and x aliases x'.

C2. m and m' either lie in the overlapping regions of paths in f and f' that are executed by different threads, or there are no *unaffected statements* whose may_{def} and may_{use} sets prevent them from ending up in such regions due to a sequentially-sound transformation based on *data-flow analysis frameworks* [20].

From the above definition, f and f' also immediately interfere on x'. Therefore, immediate interference is a special kind of conflict [40]. We say that f and f' "may immediately interfere" on x (and x') if each of the above conditions only *may* hold.

Condition C2 is a disjunction of two clauses. The first covers the situation of an interleaving in which m and m' abut, and are executed by different threads. The second conservatively anticipates the emergence of such situations after a class of sequentially-sound transformations. Recall from Section 2.2 that for an unaffected statement, the may_{def} and may_{use} sets are fixed. So an unaffected statement *always* prevents the movement of an access m'' across it if the lvalue involved in m'' belongs to its may_{def} set, or if m'' is a write and the lvalue involved belongs to its may_{use} set.

For the may-immediately-interfere case, Condition C2 is assumed true unless there is evidence to the contrary. Refinement 3 exemplifies how evidence to the contrary falsifies the condition.

3.2 The Initial PCG

The initial PCG $G_p^0 = (\mathscr{F}_d, E^0, I_i^0)$ is a complete graph with self-loops, and has an I_i^0 that is set so that

$$I_i^0\big((a,b)\big) = \big(\overline{R}_i(a) \approx \overline{W}_i(b)\big) \cup \big(\overline{W}_i(a) \approx (\overline{R}_i(b) \cup \overline{W}_i(b))\big) \quad (8)$$

for all $(a,b) \in E^0$, where $\approx$ is as defined by Equation (1).

Equation (8) finds all lvalues that may satisfy Condition C1; these approximate the may-immediate interference between a and b because Condition C2 can always be regarded to hold for them.

Observe that accesses in defined procedures called from a or b do not affect $I_i^0\big((a,b)\big)$. As an example, if b invokes $c \in \mathscr{F}_d$, then the immediate accesses in c do not affect $I_i^0\big((a,b)\big)$. These accesses interfere with those in a only if c and a can execute in parallel.[11] But then, $I_i^0\big((a,c)\big)$ would capture this interference.

3.3 Iteratively Improving a PCG's Precision

A *refinement* maps a PCG $G_p^j = (\mathscr{F}_d, E^j, I_i^j)$ to G_p^{j+1}. Thus, given an initial G_p^0, a sequence $G_p^1, G_p^2, \ldots$ can be generated by successively applying refinements. All refinements, by definition, possess the following two properties: $E^{j+1} \subseteq E^j$, and $I_i^{j+1}(e) \subseteq I_i^j(e)$ for all $e \in E^{j+1}$. Hence, the PCG sequence converges.

A refinement is a *concurrency type* if $I_i^{j+1} = I_i^j / E^{j+1}$, where '$F/A$' denotes the restriction of a function F to a subset A of its domain. It is a *purely interference type* if $E^{j+1} = E^j$. While the two types are not mutually exclusive, the identity refinement is the only one that is both a concurrency type and a purely interference type. This paper only explores refinements that are one of these two types. But clearly, there could be other types of refinements depending on how E^{j+1} and I_i^{j+1} are related to E^j and I_i^j.

The purpose of a refinement is to improve the precision of a PCG. As more refinements get invented (each capitalizing on a different class of opportunities), the likelihood of an accurate PCG for the general multithreaded program increases. And the more precise a PCG, the less conservative is the immediate interference between a pair of procedures. This translates into a potentially larger set of siloed lvalues, as Section 4.3 will show.

3.3.1 A Thread-Based Classification of Procedures

Several of the refinements in this paper are formulated using a special classification of a program's invoked procedures. This classification, which is specific to a POSIX-like threading model, categorizes every invoked procedure, whether defined or undefined, into one or more of the following groups: "start routines", spawners, spawnees and "follow routines". A procedure is a *start routine* if it may be the entry point of a spawned thread. It is a *spawnee* if it may be executed by a spawned thread. It is a *spawner* if it may create a thread and return with the created thread still running. It is a *follow routine* if a thread may invoke it after that thread returns from a spawner. These categories are respectively represented by the sets $\mathscr{F}_{START}$, $\mathscr{F}_{SPAWNEE}$, $\mathscr{F}_{SPAWNER}$ and $\mathscr{F}_{FOLLOW}$.

Start routines are often easily recognizable. For instance, they are targeted by the third argument of a `pthread_create` call.

Start routines, and every procedure they *call-reach* (i.e., reach in the static call graph), are conservatively marked spawnees.

Procedures that call-reach a spawner are conservatively also spawners. This is a recursive definition—its base case is every undefined procedure that is a spawner, such as `pthread_create`. A procedure does not become a spawner by the mere act of spawning a thread. It is not a spawner if the created threads are not existent on its return. Thus, procedures that create threads and wait for them to exit before returning are not spawners.

A conservative set of follow routines can be obtained thus: (1) a procedure whose call site lies in a control-flow path that starts just after a spawner's call site is a follow routine;[12] and (2) procedures call-reachable from follow routines are also follow routines.

There is a subtlety about follow routines. Consider two successive calls to `pthread_create`. The second one is a follow routine.[13] But this will not cause its start routine f (and procedures call-reachable from f) to become a follow routine. This is because start routines are not traditional callbacks. So `pthread_create` call-graph nodes will not have outgoing arcs. When a thread calls `pthread_create`, execution of the start routine occurs in a *different* thread. Since the call graph only models the call path in the invoking thread, the `pthread_create` node will just be a sink.

3.3.2 Concurrency-Type Refinements

Refinements 1 and 2 below are of the concurrency type. Because their only effect on the immediate interference function is to restrict it to the new edge set E^{j+1}, they can be formally stated by just describing their effects on E^j. All of the refinements in this paper are provably sound—i.e., the new PCG never omits an immediate interference or happens-in-parallel event that occurs at run time.

[11] It is also possible for c and a to not execute in parallel—e.g., a and b begin running together, a finishes before b, and b invokes c after a finishes.

[12] Because the path starts just after the spawner's call site, spawners themselves are not follow routines unless they are called in the path.

[13] The discussion here is assuming a context-sensitization, so the attributes for the second call are not merged with those for the first.

```
0 GetArguments();           4 CREATE(ParallelExecute,...);
1 InitGlobalMemory();       5 WAIT_FOR_END(...);
2 InitExpTables();          6 printf(...);
3 CreateDistribution(...);  7 PrintTimes();
```

Figure 3. An excerpt from the `main` procedure of SPLASH-2's FMM benchmark. Calls preceding `CREATE` perform initialization tasks. Calls after `WAIT_FOR_END` are to output procedures.

Refinement 1.

$$E^{j+1} = E^j - \{(a,b) \mid a \notin \mathscr{F}_{SPAWNEE} \wedge b \notin \mathscr{F}_{SPAWNEE}\}.$$

Soundness Proof. Since $a, b \notin \mathscr{F}_{SPAWNEE}$, the main thread is the only thread that can execute a and b. Hence the claim. □

Refinement 2.

$$E^{j+1} = E^j - \{(a,b) \mid a \notin (\mathscr{F}_{SPAWNEE} \cup \mathscr{F}_{SPAWNER} \cup \mathscr{F}_{FOLLOW})\}.$$

Soundness Proof. Let $a \notin (\mathscr{F}_{SPAWNEE} \cup \mathscr{F}_{SPAWNER} \cup \mathscr{F}_{FOLLOW})$. Since $a \notin \mathscr{F}_{SPAWNEE}$, only the main thread can execute a. A thread spawned by the main thread may still be running when a is entered. Or the main thread may call a spawner when an invocation of a is active. The other two possibilities are that the main thread spawns a thread after a returns, or that it never spawns a thread. The first two cases cannot happen because $a \notin \mathscr{F}_{FOLLOW} \cup \mathscr{F}_{SPAWNER}$. And under the last two, a cannot run in parallel with any procedure. □

Note that (a,b) signifies an unordered pair, so the subtrahend in Refinement 2 is the set of all unordered pairs for which at least one of the components in the pair satisfies the stated predicate.

The above refinements go after different concurrency-paring opportunities. Refinement 1 removes an edge if the involved procedures can only be executed by the main thread. Refinement 2 addresses opportunities in which a spawned thread may execute at most one of the procedures in an edge. The refinements do not depend on knowledge pertaining to synchronization orders or thread termination. More concurrency-type refinements can be devised if calls to other thread-related procedures, such as `pthread_join` and `pthread_cond_wait`, are not handled opaquely.[14]

3.3.3 Demonstrations of Refinements 1 and 2 on Real Code

Figure 3 shows a code fragment from the `main` procedure of SPLASH-2's FMM benchmark [47]. The macros `CREATE` and `WAIT_FOR_END` expand to calls to `pthread_create` and `pthread_join`. Only the main thread executes the procedures invoked on Lines 0 to 3, and on Line 7. Therefore, these procedures cannot execute concurrently with each other. Refinement 1 detects this since none of them belong to $\mathscr{F}_{SPAWNEE}$. Indeed, Refinement 1 also determines that none of them can happen in parallel with themselves.

The first argument to `CREATE` is a start routine. Since the procedures invoked on Lines 0 to 3 are neither spawnees, spawners nor belong to $\mathscr{F}_{FOLLOW}$, Refinement 2 discovers that none of them can execute in parallel with `ParallelExecute` or any procedure call-reachable from `ParallelExecute`.

This example also shows the complementarity of Refinements 1 and 2. That is, Refinement 1 will not detect that `ParallelExecute` and its call-descendants cannot run in parallel with the procedures invoked on Lines 0 to 3. And because `PrintTimes` is in $\mathscr{F}_{FOLLOW}$, Refinement 2 will not uncover that it is never concurrent with itself.

3.3.4 Purely Interference-Type Refinements

As remarked in Section 3.3, purely interference-type refinements only affect the immediate interference function I_i. Our first purely interference-type refinement is based on the observation that lvalues that are only accessed before spawner call sites, in procedures that are only executed by the main thread, can never interfere with accesses that occur in concurrent procedures *if* the spawner call sites are unaffected statements. Our second purely interference-type refinement is based on the observation that under a certain condition, two procedures that do not synchronize on common sync objects cannot interfere in a data-race-free program.

A flow-sensitive refinement. A statement s_2 "follows" a statement s_1 if s_2 occurs in a control-flow path that starts just after s_1. Given a set of statements S', we use $follow(S')$ to designate the set of all statements each of which follows some statement in S'. Let $spawner(f)$ be the set of all statements that invoke spawners in a defined procedure f. If the spawner call sites all belong to $\mathscr{U} - \mathring{\mathscr{U}}_{sync}$, their MOD-REF sets (which correspond to may_{def} and may_{use} sets) will ensure that any *unsafe* movement of shared-object accesses across them by a data-flow analysis is blocked. This presents a pruning opportunity, formalized in Refinement 3, because lvalues that are not accessed after spawner call sites can never satisfy Condition C2. Notice that no edge-set effects are shown because for purely interference-type refinements, $E^{j+1} = E^j$.

Refinement 3. If spawner call sites are unaffected statements, then

$$I_i^{j+1}(e) = I_i^j(e) - \{x \mid x \notin R_i(s) \cup W_i(s) \forall s \in follow(spawner(a))\},$$

where $e = (a,b) \in E^j$, and $a \notin \mathscr{F}_{SPAWNEE} \cup \mathscr{F}_{FOLLOW}$.

Soundness Proof. Consider an $a \notin \mathscr{F}_{SPAWNEE} \cup \mathscr{F}_{FOLLOW}$. It can only be executed by the main thread. Because $a \notin \mathscr{F}_{FOLLOW}$, it can run concurrently with another procedure only if it is a spawner. Then, all statements executed in it until a spawner call site is reached will not happen in parallel with any statement. Let Z be the set of all lvalues immediately accessed only in statements preceding spawner call sites, and not elsewhere in a. If no spawned thread accesses an object covered by Z, the refinement trivially holds. So suppose an object covered by a $z \in Z$ is accessed by a spawned thread h. Then the may_{use} and may_{def} sets of the call site s'' that spawns h will factor in the read and write of z concurrently performed by h. Now, $may_{use}(s'')$ and $may_{def}(s'')$ are fixed since $s'' \in \mathscr{U} - \mathring{\mathscr{U}}_{sync}$. So they will *always* obstruct the unsafe movement of z across s'' by a data-flow analysis. Either way, Conditions C1 and C2 can never both hold. Hence, the immediate interference between a and its PCG neighbors can never include Z. □

The expression $follow(spawner(a))$ above can be ascertained by finding basic blocks reachable in the CFG from spawner invocation sites. It can also be used to calculate the program's $\mathscr{F}_{FOLLOW}$. That is, suppose $RTC(f)$ is the set of all immediate successors of a procedure f in the reflexive transitive closure of the static call graph. Then, $f \in \mathscr{F}_{FOLLOW}$ if there exists a procedure f' such that $f \in RTC(f')$ and a call site of f' belongs to some $follow(spawner(a))$.

A refinement based on data-race freedom. Consider two procedures a and b that may execute simultaneously. Equation (7) gives their $\overline{SYNC}$ sets. If none of the lvalues in $\overline{SYNC}(a)$ and $\overline{SYNC}(b)$ can alias each other, then a and b cannot immediately interfere with each other, provided the no-chain condition specified in the statement of Refinement 4 holds. a and b can make conflicting accesses of a data object (i.e., non-sync object) when they do not overlap during execution.[15] If the conflicting accesses were to oc-

[14] If some thread-related procedure is not modeled, it should be treated opaquely, and not ignored, in order to ensure correctness.

[15] For instance, by acquiring a common lock before calling a and b.

cur when their executions overlap, an interleaving exists in which the accesses are adjacent. But then, the program has a data race.

Refinement 4. If the program is data-race free and $(a,b) \in E^j$, then

$$I_i^{j+1}\big((a,b)\big) = \varnothing \text{ if } \overline{SYNC}(a) \approx \overline{SYNC}(b) = \varnothing,$$

and the "no-chain" condition holds, i.e., there are no procedures $f_1 f_2 \dots f_n$ ($n > 2$) that satisfy three clauses: (1) $f_1 = a$ (or b), $f_n = b$ (or a) *can* happen in parallel, (2) for at least one k, $\overline{SYNC}_i(f_k) \approx \overline{SYNC}_i(f_{k+1}) \neq \varnothing$ and f_k, f_{k+1} can happen in parallel, and (3) for all other k, a call of f_k precedes a call of f_{k+1} in program order. ($\overline{SYNC}_i(f)$ is the set of immediately accessed sync objects in f.)

Soundness Proof. We show by contradiction that $I_i^{j+1}\big((a,b)\big)$ can be set to $\varnothing$. Assume a and b immediately interfere on x—thus, by Conditions C1 and C2, there can be an execution instance in which accesses of x in a and b conflict and lie in overlapping regions. Then, there must be a happens-before relation $\overset{hb}{\rightarrow}$ between an $s_a \in STMTS(a)$ and an $s_b \in STMTS(b)$—otherwise, the program has a data race on x [3]. Happens-before is the irreflexive transitive closure of the program order $\overset{po}{\rightarrow}$ and synchronization order $\overset{so}{\rightarrow}$ relations [3]. Without loss of generality, let $s_a \overset{hb}{\rightarrow} s_b$. Then, there is a chain of statements $s_1 s_2 \dots s_m$ such that $s_1 = s_a$, $s_m = s_b$, and either $s_k \overset{so}{\rightarrow} s_{k+1}$ or $s_k \overset{po}{\rightarrow} s_{k+1}$. If $s_k \overset{so}{\rightarrow} s_{k+1}$, then there exist procedures f_k and f_{k+1} such that $\overline{SYNC}_i(f_k) \approx \overline{SYNC}_i(f_{k+1}) \neq \varnothing$ and f_k and f_{k+1} can happen in parallel. If $s_k \overset{po}{\rightarrow} s_{k+1}$, then s_k and s_{k+1} belong to procedures f_k and f_{k+1} that either are the same, or f_k is called before f_{k+1} in program order. Now, $s_k \overset{so}{\rightarrow} s_{k+1}$ for at least one k—otherwise, an interleaving can be constructed that has a data race on x. Hence, there is a set of procedures that violates the no-chain condition. Thus, a and b cannot immediately interfere on x. □

The no-chain condition holds if any of the three clauses is false. It is not checked by our current implementation, though for all our tested SPLASH-2 benchmarks, it holds whenever Refinement 4 is applied. That is, some clause is false whenever $\overline{SYNC}(a) \approx \overline{SYNC}(b) = \varnothing$ by the time Refinement 4 is applied.

The no-chain condition, as presented, is conservative. It can be tightened if other procedure-level information is used—e.g., the absence of spinning synchronizations inside procedures.

3.3.5 Demonstrations of Refinements 3 and 4 on Real Code

The `InitExpTables` call on Line 2 in Figure 3 initializes `Zero` and `One`, which are two global `struct` variables. Various procedures call-reachable from `ParallelExecute` use these variables. `InitExpTables` is sufficiently small that `gcc` inlines it when the `-O3` switch is turned on. Because `ParallelExecute` and all of its call-descendants may run concurrently with `main`, Equation (8) includes `Zero` and `One` in the initial immediate interference sets between `main` and procedures in RTC(`ParallelExecute`) that access these variables. Nevertheless, `main` is neither a spawnee nor a follow routine, and statements in *follow*(*spawner*(`main`)) do not access these variables. Hence, Refinement 3 removes `Zero` and `One` from all of the above immediate interference sets.

We illustrate Refinement 4 by considering `InitBox` and `CreateBoxes`, two procedures call-reachable from `ParallelExecute`. Since `CREATE` may spawn multiple threads, the two may run in parallel. Both write into a global array called `Local`, so the initial immediate interference set for the pair is nonempty.

Now, `pthread_mutex_lock` and `pthread_mutex_unlock` are the only synchronizations performed when `InitBox` and `CreateBoxes` are active. `InitBox` performs them through the callee `LockedPrint`. The locks held are all different, however, and `InitBox` and `CreateBoxes` satisfy the no-chain condition. (FMM invokes a barrier between `CreateBoxes` and `InitBox`, ensuring that the two cannot happen in parallel.) So Refinement 4 reduces the immediate interference between them to the empty set.

3.4 Dealing with Inscrutable Procedures

The description thus far of the PCG construction algorithm is adequate for handling programs in which only defined and abstracted procedures are called. We now show that the algorithm works even when there are inscrutable-procedure calls, e.g., into arbitrary third-party libraries distributed as pure binaries. The issue boils down to understanding the effects of inscrutable-procedure calls on the construction of the initial PCG and on Refinements 1 to 4.

3.4.1 Effect on Building the Initial PCG

Inscrutable procedures do not affect the PCG's node set $\mathscr{F}_d$. From Equations (4) and (5), R_i and W_i are well defined in the presence of undefined-procedure calls. Therefore, from Equation (6), $\overline{R}_i(f)$ and $\overline{W}_i(f)$ are well defined for all $f \in \mathscr{F}_d$. Since Equation (8) remains operable, inscrutable procedures pose no problems to building G_p^0.

3.4.2 Effect on Refinements 1 to 3

But inscrutable-procedure calls may affect sets such as $\mathscr{F}_{SPAWNER}$, $\mathscr{F}_{SPAWNEE}$ and $\mathscr{F}_{FOLLOW}$; these, in turn, affect Refinements 1 to 3:

- $\mathscr{F}_{SPAWNER}$: In the absence of information to the contrary, inscrutable procedures must be regarded as spawners. Then, every inscrutable procedure g, and every procedure that call-reaches g, must be added to $\mathscr{F}_{SPAWNER}$.
- $\mathscr{F}_{START}$: Suppose $\mathscr{F}_{START}$ is initially the set of start routines, obtained by ignoring all inscrutable-procedure calls. If there is even one such call, every procedure in $\mathscr{L}_{inscr}$ whose function type is indicative of a start routine must be added to $\mathscr{F}_{START}$.
- $\mathscr{F}_{SPAWNEE}$: $\mathscr{F}_{SPAWNEE}$ is just $\cup_{f \in \mathscr{F}_{START}} RTC(f)$.
- $\mathscr{F}_{FOLLOW}$: Let $\mathscr{F}'_{FOLLOW}$ be initially the set of follow routines, ignoring all inscrutable-procedure calls. If there is a call to an inscrutable procedure g, every procedure in $\mathscr{L}_{inscr}$ would have to be included in $\mathscr{F}'_{FOLLOW}$ because g could invoke all of them after spawning a thread. $\mathscr{F}_{FOLLOW}$ can then be obtained by applying the algorithm in Section 3.3.1 using $\mathscr{F}'_{FOLLOW}$.

3.4.3 Effect on Refinement 4

The weak-ordering model of memory consistency prescribes an algorithm for safely distinguishing synchronizations in a data-race-free program [2, Page 75]. The idea is to mark a statement as a synchronization if treating it as a data operation *could* lead to a data race. We assume that inscrutable-procedure calls have been appositely marked as synchronizations using this algorithm.

For instance, if $\mathscr{L}_{inscr}$ is the empty set, then inscrutable-procedure invocations need not be regarded as synchronizations. But in the situation that a call to an inscrutable procedure g should be treated as a synchronization, the $\overline{SYNC}$ set of every defined procedure that call-reaches g will include the virtual lvalue ui_{sync}.

Refinement 4 does not require a special handling of the above two situations. It automatically will *not* apply in the latter situation since $\overline{SYNC}(a) \approx \overline{SYNC}(b)$ will then be nonempty.

4. Enabling Optimizations on Siloed References

PCGs have several applications. One is determining a class of *references* (i.e., accesses) in a multithreaded program on which classical optimization opportunities can be safely unblocked. Members of this class have the read- and write-siloed properties on certain intraprocedural paths. More precise may_{def} and may_{use} sets can be obtained by leaving out these references. This section proves that the resulting sets *always* remain sound for a data-flow analysis.

4.1 The Read-Siloed and Write-Siloed Properties

Let P be a control-flow path between two program points. We say that an lvalue x is "read-siloed in a thread h on P" if once h enters P, no other thread writes an object named by x until h exits P. This definition is best understood by considering an execution interleaving, such as the one below:

$$\ldots\ s'_1\ s'_2\ \mathbf{s_1}\ s'_3\ s'_4\ s'_5\ \mathbf{s_2}\ s'_6\ \mathbf{s_3}\ s'_7\ s'_8\ \ldots\ \mathbf{s_n}\ s'_m\ s'_{m+1}\ \ldots$$

The boldface symbols $\mathbf{s_1}$ to $\mathbf{s_n}$ are instructions executed by h, and form the path P. $\mathbf{s_1}$ and $\mathbf{s_n}$ are also the first and last instructions in P. The lightface symbols are instructions executed by other threads. If x is read-siloed in h on P, then any write of x by another thread would have to precede s'_2 or succeed s'_m.[16] Similarly, x is said to be "write-siloed in h on P" if no other thread reads or writes x once h enters P and until it exits P. If these definitions were true for all h, we would just say that x is "read- or write-siloed on P". P would then be a *read-* or *write-siloed path* with respect to x.

There are numerous points about these definitions. First, they do not mention an occurrence of x in P. An lvalue x can be read-siloed in h even on a stretch of code free of x, so long as no other thread updates x when h is at any point in this code stretch. Second, write-siloed is a stronger property than read-siloed. If x is write-siloed in h on P, then it is also read-siloed in h on P. Third, an lvalue y that is read-siloed (write-siloed) on P in all threads has the salient quality that a write (read or write) of y by any thread outside P can only occur when no other thread is within the confines of P. This trait is stronger than read- or write-siloed references within P being just data-race free. It means accesses of y in P, including those involving proper synchronization, are free of cross-thread effects.

4.2 The Siloed-on-a-Procedure Property

The siloed concept can be extended to whole procedures. Let $stmts(P)$ be the set of statements in a path P. Then, an lvalue z is said to be "siloed on a procedure f" if two conditions are met:

S1. z is write-siloed on every path P in f in which it may be immediately written at a statement s and is not in $may_{def}(s') \cup may_{use}(s')$ of any unaffected statement s' in $stmts(P) - \{s\}$.

S2. z is read-siloed on every path P in f in which it may be immediately read at a statement s and is not in $may_{def}(s')$ of any unaffected statement s' in $stmts(P) - \{s\}$.

The set $SOP_i(f)$ consists of lvalues siloed on f. As our work is the first treatment of the siloed concept, the focus will be on this simpler procedure-level variant, although working with the concept at a finer granularity will likely yield more powerful results. In this paper, *siloed* without qualification means siloed on a procedure.

4.3 Computing Siloed Lvalues

Let $MHP(f)$ be the set of neighbors of a procedure f in the program's PCG. Then f's *overall immediate interference* is

$$\overline{I}_i(f) = \bigcup_{f' \in MHP(f)} I_i\big((f, f')\big). \tag{9}$$

From Equation (9), it is clear that for every $x \in \overline{I}_i(f)$, there is some $f' \in MHP(f)$ such that f and f' may immediately interfere on x. So lvalues in $\overline{I}_i(f)$ may not be siloed on f. But those in

$$S_i(f) = (\overline{R}_i(f) \cup \overline{W}_i(f)) - \overline{I}_i(f) \tag{10}$$

will be, as Theorem 1 shows. The subscript i, as usual, signifies that only immediately accessed lvalues comprise the *siloed-lvalue set*.

Theorem 1. $S_i(f) \subseteq SOP_i(f)$.

Proof. If $x \in S_i(f)$, then $x \notin \overline{I}_i(f)$ by Equation (10). We prove that if P is a path in f with a possible immediate write-access m of x at a statement s, and $x \notin may_{def}(s') \cup may_{use}(s')$ for all unaffected statements s' in $stmts(P) - \{s\}$, then x must be write-siloed on P. If not, there exist threads h and h' such that when h is in P in some execution instance, h' performs an access m' of an alias x' of x. Now, m' is immediate to some $f' \in \mathscr{F}_d$ and is in a path P' in f'. Since none of the unaffected statements in $stmts(P) - \{s\}$ define or use x, a data-flow transformation *could* move m to any point in P. Then, because m and m' also conflict, Conditions C1 and C2 can both hold for x. Thus, $x \in I_i\big((f, f')\big)$. So by Equation (9), $x \in \overline{I}_i(f)$, a contradiction. Condition S2 can be similarly proved for paths that may immediately read x. Therefore, $x \in SOP_i(f)$. □

There is an important case for Equations (9) and (10) that we highlight. If lvalues in $\overline{SYNC}(f)$ do not alias with lvalues in $\overline{SYNC}(f')$ for all $f' \in MHP(f)$, then $\overline{I}_i(f)$ will be $\varnothing$ if the program is given to be data-race free and if f and f' satisfy the no-chain condition in Refinement 4. This is because Refinement 4 will force all $I_i\big((f, f')\big)$ to $\varnothing$; $S_i(f)$ will then equal $\overline{R}_i(f) \cup \overline{W}_i(f)$. Since $SOP_i(f) \subseteq \overline{R}_i(f) \cup \overline{W}_i(f)$, we would then have $S_i(f) = SOP_i(f)$.

4.4 A More Accurate May-Definition Set

Let $DU(s)$ be the set of all user-code lvalues z for which there is an intraprocedural path from a potential access of z to the statement s, or from s to a potential access of z.[17] From a data-flow analysis standpoint, it is enough if $may_{def}(s)$ includes two groups of lvalues when s is a synchronization: (1) those in $\mathscr{L}$ that could be written at s by a thread h executing s; and (2) those in $DU(s)$ that could be concurrently written when h is executing s. The first group is drawn from $\mathscr{L}$, and not the possibly smaller $DU(s)$, because *isolated* definitions may be used, or may kill definitions, in concurrent threads. But only those concurrent definitions that may kill definitions in the current thread, or that may be later killed or used in the current thread, need to be factored into $may_{def}(s)$—hence, $DU(s)$ is the superset for the second group. This suggests that

$$may^t_{def}(s) = W(s) \cup (DU(s) \cap CW(s)) \subseteq may_{def}(s) \tag{11}$$

is a more precise may-definition set for synchronizations, where

$$W(s') = \{x \mid x \in \mathscr{L} \wedge (\text{the thread executing } s' \text{ may immediately or transitively write } x \text{ at } s')\}, \tag{12}$$

$$CW(s') = \{x \mid x \in \mathscr{L} \wedge (\text{an execution exists wherein when a thread is at } s', \text{ another thread writes } x)\} \tag{13}$$

for *any* statement s'.

It is easy to calculate $W(s')$ using the immediate-write sets W_i and $\overline{W}_i$, which were defined in Equations (5) and (6):

$$W(s') = W_i(s') \cup \begin{cases} \varnothing & \text{if } proc(s') = \top, \\ \bigcup_{f' \in RTC(proc(s'))} \overline{W}_i(f') & \text{otherwise.} \end{cases} \tag{14}$$

The first case in the above coincides with non-call code. The second case coincides with a call. Besides $W_i(s')$, it includes the $\overline{W}_i$ set of every defined or undefined procedure that is call-reachable from s'.

$CW(s')$ models the parallel writes at s'. Lemma 1 states that for any synchronization s, $DU_i(s) \cap CW(s)$ and $SOP_i(f)$ are disjoint. $DU_i(s)$, which is a subset of $DU(s)$, comprises lvalues $z \in \mathscr{L}$ for which a potential immediate access of z at a statement s' reaches just before s, or is reached from just after s, by a path P in f in which the unaffected statements among $stmts(P) - \{s'\}$ do not

[16] To preclude data races, s'_2 and s'_m should not be writes of x.

[17] Interprocedural definition-use flow can also be modeled, by expediently adding artificial assignments and uses to the CFG's entry and exit nodes.

access z. Theorem 2 uses Lemma 1 to prove an “upper bound” on $may^{t}_{def}(s)$ that is better than $may_{def}(s)$.

Lemma 1. *For all synchronizations s in f,*

$$DU_i(s) \cap CW(s) \cap SOP_i(f) = \varnothing.$$

Proof. Consider an $x \in DU_i(s) \cap CW(s)$. Since $x \in DU_i(s)$, there is a path P in f from (to) a statement s' that may immediately access x, to just before (from just after) s, such that besides s', none of the unaffected statements in P access x. By Equation (13), an execution exists in which when a thread is at s, another updates x. Because x is not read-siloed on P, $x \notin SOP_i(f)$. □

Theorem 2. *For all synchronizations s in f,*

$$\big(may_{def}(s) - (DU_i(s) \cap SOP_i(f))\big) \cup W(s) \supseteq may^{t}_{def}(s).$$

Proof. Let $x \in may^{t}_{def}(s)$. Then $x \in may_{def}(s)$. By Equation (11), $x \in W(s) \cup (DU(s) \cap CW(s))$. If $x \notin DU(s) \cap CW(s)$, then $x \in W(s)$. If $x \in DU(s) \cap CW(s)$, then x is either in or not in $DU_i(s) \cap CW(s)$. If in, then $x \notin SOP_i(f)$ by Lemma 1. If not in, then $x \notin DU_i(s)$, since $DU_i(s) \subseteq DU(s)$. Either way, $x \notin DU_i(s) \cap SOP_i(f)$. So $x \in \big(may_{def}(s) - (DU_i(s) \cap SOP_i(f))\big) \cup W(s)$ always holds. □

Theorem 2 can only be used to tighten the may_{def} sets of interesting synchronizations, since the siloed-on-a-procedure notion is stipulated on the other synchronizations (i.e., unaffected statements) having fixed may_{def} and may_{use} sets. Thus, by using Theorems 1 and 2, $may_{def}(s)$ of an interesting synchronization s can be replaced by $\big(may_{def}(s) - (DU_i(s) \cap S_i(f))\big) \cup W(s)$.

The tightening of may_{def} sets may unfetter optimization opportunities. It may also lead to more accurate MOD sets for procedures that call-reach interesting synchronizations.

4.5 A More Accurate May-Use Set

There is a similar result for may_{use} sets. That is,

$$\big(may_{use}(s) - (DD_i(s) \cap SOP_i(f))\big) \cup R(s) \supseteq may^{t}_{use}(s),$$

where $R(s) \subseteq \mathscr{L}$ is the set of all lvalues that could be immediately or transitively read at s. $DD_i(s) \subseteq DU(s)$ involves immediate writes, and is similar to $DU_i(s)$—i.e., it is a set of lvalues $z \in \mathscr{L}$ for which a potential immediate write of z at a statement s' reaches just before s, or is reached from just after s, by an intraprocedural path P in which the unaffected statements in $stmts(P) - \{s'\}$ do not access z.

4.6 Using Siloed Information for Incremental SSA Updating

The HSSA Form can be rebuilt once the may_{def} and may_{use} sets are sharpened. This is a from-scratch computation. Instead, an incremental approach is to examine the χ assignments and μ operations of an interesting synchronization s. Consider one such statement, of the form t' `:=` $\chi(t)$ (μ operations are similarly handled). Clearly, $t \in DU_i(s)$. So if t is siloed on the procedure in question, and $t \notin W(s)$, the SSA updater can propagate t into all uses of t'. The original χ assignment can then be removed as dead code.

4.7 Examples

We now show how identifying siloed lvalues in some of the previous code fragments can expose optimization opportunities in them.

4.7.1 A Value-Numbering Opportunity

The critical section in Figure 2 is from FMM’s `ParallelExecute` procedure. Now, `InitGlobalMemory` is the only FMM procedure that writes `G_Memory`. As discussed in Section 3.3.3, the PCG reveals that `InitGlobalMemory` cannot happen in parallel with `ParallelExecute`. So `ParallelExecute`’s overall immediate interference will not contain `G_Memory`. Equation (10) then ascertains that `G_Memory` is siloed on `ParallelExecute`. Hence, by Theorem 2, the χ assignments on Lines 3, 6 and 11 can be removed, and t', t'' and t''' can be replaced by `G_Memory`. A value-numbering pass will now be able to catch the equivalence of t_2 and t_6.

```
     fa:               a:               fb:               b:
1.0 LOCK(Lx)   2.0 ta  := x     3.0 LOCK(Ly)   4.0 tb  := y
1.1 LOCK(Ly)   2.1 y  := ...    3.1 LOCK(Lz)   4.1 z  := ...
1.2 ...y...    2.2 UNLOCK(Ly)   3.2 ...z...    4.2 UNLOCK(Lz)
1.3 a(...)     2.3 ...ta...     3.3 b(...)     4.3 ...tb...
1.4 UNLOCK(Lx) 2.4 return       3.4 UNLOCK(Ly) 4.4 return
```

Figure 4. A program in which different optimization opportunities exist, depending on whether the `UNLOCK` on Line 2.2 in procedure `a` is an interesting synchronization or an unaffected statement.

4.7.2 A Copy Propagation Opportunity

Assume that the left and right snippets in Figure 1 are from procedures a and b respectively (a and b could be the same). Then, there will be an edge between a and b in the PCG. By Equation (9), `X` and `Y` will be in the overall immediate interference of both a and b. Hence, by Equation (10), both `X` and `Y` will not be in either $S_i(a)$ or $S_i(b)$. So no optimization opportunities on `X` or `Y` get unblocked.

For the second hypothetical program, from Section 1.1.2 (same left snippet, but a different right snippet, one in which `X` is not updated), `X` will be in both $S_i(a)$ and $S_i(b)$. So a copy propagation opportunity involving `X` will get unblocked in the left snippet.

5. A Tradeoff Involving Interference

The reason for Condition C2’s second clause is that data-flow information conservatively killed at an interesting synchronization s, due to $may_{def}(s)$ and $may_{use}(s)$, could later flow through s, since $may_{def}(s)$ and $may_{use}(s)$ are alterable. This could allow the movement of one or both of the accesses mentioned in Section 3.1.1, into an overlapping region of execution. Thus, the fewer the interesting synchronizations among $\mathscr{U}_{sync}$, the more the unaffected statements in user code, and so the less the chance of two procedures immediately interfering as a result of a data-flow transformation.

On the other hand, not designating a synchronization s as interesting could result in s unnecessarily killing useful data-flow information. Therefore, there is a tradeoff between immediate interference and the marking of synchronizations as interesting.

As an example, spawner call sites can be marked as interesting synchronizations, since spawning imposes a synchronization order [2]. But then, an unconditional application of Refinement 3 would not be guaranteed sound, because subsequent changes to a spawner call site’s may_{def} and may_{use} sets could lift a killing effect, which may allow the movement of an access across the site, which in turn could change a noninterfering access into an interfering one. Therefore, spawner call sites must belong to $\mathscr{U} - \mathring{\mathscr{U}}_{sync}$ for Refinement 3 to be applicable.

Another example is in Figure 4. The only shared data objects are `x`, `y` and `z`, which are always accessed holding the locks $L_{\mathtt{x}}$, $L_{\mathtt{y}}$ and $L_{\mathtt{z}}$ respectively. Procedures `a` and `b` are always invoked with the respective lock pairs $L_{\mathtt{x}}$, $L_{\mathtt{y}}$ and $L_{\mathtt{y}}$, $L_{\mathtt{z}}$ held. Specimen invocations are shown on Lines 1.3 and 3.3. Now, irrespective of whether the `UNLOCK` on Line 2.2 is an unaffected statement, the read of `x` on Line 2.0 cannot satisfy Condition C2. Hence, $\mathtt{x} \notin \overline{I}_i(\mathtt{a})$ for the perfect $\overline{I}_i(\mathtt{a})$. So if Line 2.2 were an interesting synchronization, this would permit the SR Technique to drop `x` from $may_{def}(s_{2.2})$, thus enabling the copy propagation of `x` from Line 2.0 to Line 2.3.

On the other hand, marking Line 2.2 as an interesting synchronization means a and b may immediately interfere on y. This is because there would then be no unaffected statements that block the movement of y across Line 2.2. If Line 2.2 were instead an unaffected statement, the accesses of y on Lines 2.1 and 4.0 cannot satisfy Condition C2. Then, $y \notin \bar{I}_i(b)$ for the perfect $\bar{I}_i(b)$. This would allow the SR Technique to drop y from $may_{def}(s_{4.2})$, if Line 4.2 were an interesting synchronization. This, in turn, would permit the copy propagation of y from Line 4.0 to Line 4.3.

6. Experimental Results

We have implemented the SR Technique in revision 148810 of gcc, a pre-release of version series 4.5 of the compiler. This section reports measurements demonstrating how our implementation fared on benchmarks from the SPLASH-2 suite [47]. Our test bed was a four-socket 64-bit server, in which each socket was a 2.40GHz quad-core Intel Xeon E7330 processor having a 1066MHz front-side bus. A socket has two dies, with two cores per die. The per-core L1 instruction and data cache sizes were 32KB each. The per-die L2 cache was 3MB. The system ran Redhat Enterprise Linux 5 (kernel release 2.6.18), and had 32GB of available memory.

6.1 Compilation Details

In all our experiments, the -O3 switch was turned on. Our prototype operates in gcc's whole-program compilation mode. This is turned on by the -combine and -fwhole-program flags, which require all source files to be on a single command-line. Until recently, it was the only way to do a whole-program interprocedural analysis (IPA) in gcc. New LTO (Link-Time Optimization) support in the current release series of gcc (4.5) should remove this deficiency.

6.1.1 Design of the SR Phase

An IPA-based SR phase was created to implement the SR Technique. To evaluate, we focused on opportunities enabled by the technique in some of gcc's existent phases that implement fundamental and commonly used optimizations. In particular, we quantified exposed opportunities in the following five *arbitrarily* selected intraprocedural phases: pass_ccp (conditional constant propagator), pass_fre (full-redundancy eliminator), pass_copy_prop (copy propagator), pass_merge_phi (phase that merges directly linked ϕ-nodes), and pass_dce (dead-code eliminator). When invoked from within the SR phase, these were executed in the given order, as part of an opts_on_srefs pass list.

The SR phase is structured as a loop. In each iteration, the SR Technique is applied once, followed by an application of opts_on_srefs. This "SR loop" is repeated until the IR no longer changes. Applying the technique once means building the PCG using the algorithm of Section 3, and sharpening data-flow abstractions using the algorithm of Section 4. Thus, the SR phase aims to unfetter a *maximal* set of opportunities in opts_on_srefs.[18]

6.1.2 'Enabled' and 'Baseline' Executables

Just prior to the SR phase, opts_on_srefs is repeatedly applied until the IR reaches quiescence. Hence, executables produced with and without the SR phase differ in the optimizations enabled by the SR phase. We will refer to these executables as *enabled* and *baseline* respectively. It should be noted that optimization effects unblocked in phases downstream from the SR phase are included in the enabled executables. These are due to a one-time use of siloed information, unlike those unblocked in the opts_on_srefs phases.

Program	Average $\|S_i\|$ after Refinements 1 to k				Reductions	
	$k=1$	$k=2$	$k=3$	$k=4$	Edges	Intf.
m-fmm	0.53	1.28	1.37	4.96	813	362
ocean-c	8.05	11.95	28.41	42.67	186	113
barnes	2.51	4.81	5.05	8.89	664	71
wr-spl	1.03	2.05	3.53	4.29	112	27
wr-nsq	0.84	1.81	3.11	3.70	109	19
lu-c	1.10	1.10	1.10	5.39	15	12
radix	0.54	0.54	0.54	2.33	3	2
fft	0.90	0.90	0.90	2.79	10	6

Table 2. Metrics showing how the S_i set precision accrues with additional refinements. 'Edges' are edges removed by Refinements 1 and 2. 'Intf.' are the $I_i(e)$ sets pared by Refinements 3 and 4.

6.2 Benchmark Details

Table 1 shows the eight SPLASH-2 benchmarks used in our experiments: m-fmm (Fast Multipole Method), ocean-c (Contiguous Ocean), barnes (Barnes-Hut), wr-spl (Water-Spatial), wr-nsq (Water-Nsquared), lu-c (Contiguous LU), radix (Radix), and fft (FFT). The SPLASH-2 suite has a total of 12 "application" and "kernel" benchmarks. The remaining four did not successfully compile with the -combine/-fwhole-program combination.[19]

m-fmm is a slightly modified version of SPLASH-2's implementation of the Fast Multipole Method. The modification was to outline two adjacent loops into their own procedure. This was done to overcome a limitation in gcc's IRA (Integrated Register Allocator) phase, and is explained further in Section 6.4.2.

The 'Problem Size' column displays the inputs to our benchmarks. These were always above the original defaults [47], and were chosen, to the extent possible, to make the 16-thread execution times measurably significant; 16 was the maximum number of threads used. The 'Program Size' section states benchmark sizes in terms of total lines of source code, number of source files, and the following at the start of the SR phase: total number of basic blocks, and total number of defined procedures. Therefore, the 'Files' column is indicative of the length of a compilation command-line with the -combine/-fwhole-program combination.

The 'Static Synchronization Statistics' section reflects the static usage intensity of undefined synchronizing procedures. For SPLASH-2, these are Pthreads library calls. For instance, 37% of m-fmm's 86 defined procedures are 'Transitive Callers'—these procedures eventually reach the Pthreads library in the static call graph. (By that measure, m-fmm is among SPLASH-2's most Pthreads-intensive programs.) Therefore, their data-flow abstractions, such as MOD-REF sets, may be more conservative than necessary.

Among the synchronizations, pthread_mutex_lock and pthread_mutex_unlock were the only ones treated as interesting.

There is built-in support in gcc for certain procedures, such as puts and strtol. These are defined procedures, and are counted in $|\mathscr{F}_d|$. All other external library procedures used by SPLASH-2 can be abstracted, since they have specifications—e.g., Pthreads.

6.3 Static Metrics of Improvement

Table 2 is a quantification of the effectiveness of our refinement-based approach to constructing the PCG. We measured the size of the S_i set, averaged over $|\mathscr{F}_d|$. From Equations (9) and (10), we observe that as the PCG is refined, the overall immediate interference tends to decrease, which tends to increase the size of the

[18] "Maximal" because for a different opts_on_srefs pass order, a different set of opportunities may be unfettered by the time the IR stops changing.

[19] The failures seem to be related to cross-file IPA not being a routinely used feature in gcc. This will likely change with LTO coming online.

Program	Problem Size	Program Size				Static Synchronization Statistics				$\lvert\overline{R}_i\rvert_{\max}$	$\lvert\overline{W}_i\rvert_{\max}$
		LOC	Files	BBS	$\lvert\mathscr{F}_d\rvert$	Transitive Callers	Direct Callers	Sync Sites	Critical Sections		
`m-fmm`	1048576 particles	4381	17	969	86	32	19	52	17	43	36
`ocean-c`	4098 × 4098 grid	4774	10	1670	35	4	4	37	4	129	185
`barnes`	1048576 particles	2887	15	465	50	7	7	21	5	50	50
`wr-spl`	1331 molecules	2670	23	447	35	7	7	37	8	64	51
`wr-nsq`	4096 molecules	2063	23	324	34	6	6	41	8	59	51
`lu-c`	6000 × 6000 matrix, 16 × 16 blocks	911	1	301	26	4	4	11	1	33	28
`radix`	335544320 integers	833	1	212	18	2	2	31	6	34	29
`fft`	67108864 data points	899	1	250	24	3	3	13	1	40	38

Table 1. The SPLASH-2 benchmarks used in our experiments. 'LOC' means total lines of code. 'BBS' is the total number of basic blocks on entry to the SR phase. 'Transitive Callers' are defined procedures that transitively reach a Pthreads procedure. 'Direct Callers' are the direct invokers among them. 'Sync Sites' are the counts of the Pthreads call sites. $|\overline{R}_i|_{\max}$ and $|\overline{W}_i|_{\max}$ are the largest $\overline{R}_i$ and $\overline{W}_i$ set sizes.

Program	Propagations		ϕ-Node Merges	Eliminations	Lock Pointer Equivalences
	Constant	Copy			
`m-fmm`	48	190	7	449	17
`ocean-c`	146	1334	2	2040	4
`barnes`	4	145	2	246	3
`wr-spl`	228	265	0	621	6
`wr-nsq`	317	311	93	677	8
`lu-c`	0	200	0	273	1
`radix`	0	187	0	277	6
`fft`	1	152	2	299	1

Table 3. Static metrics on a maximal set of optimization opportunities that were enabled by the SR phase in `opts_on_srefs`.

siloed-lvalue set. Thus, the average $|S_i|$ and the precision of S_i are correlated. To check how the refinements affect this precision, we measured the average $|S_i|$ over multiple compilations, successively turning on the four refinements in this paper. The column '$k = 1$' shows these measurements with just Refinement 1 turned on, the column '$k = 2$' shows them with *both* Refinements 1 and 2 turned on, and so on. From these numbers, we observe that the average increase in $|S_i|$ across all tested programs is 62%, 37% and 170% on moving from $k = 1$ to $k = 4$. Refinement 4 is the most powerful by this measure, at least for the benchmarks studied.

The last two columns show absolute reductions. The reductions can also be expressed as percentages. For example, since the number of edges in G_p^o is $|\mathscr{F}_d|(|\mathscr{F}_d|+1)/2$, the percentage edge reductions follow from Table 1: 22%, 30%, 52%, 18%, 4%, 2%, 3%.

The measurements in Table 2 were made on the last iteration of the SR loop. They are therefore a conservative representation of the improvements, which tend to be higher in the initial iterations.

Table 3 shows the total number of opportunities that were enabled in `opts_on_srefs`, at the end of the SR phase. An "opportunity" is an instance of an optimizing transformation. For example, 'Constant' and 'Copy' are unblocked propagations of constants and copies. 'ϕ-Node Merges' are unblocked opportunities in `pass_merge_phi`. 'Eliminations' is the sum of unblocked opportunities in `pass_fre` and `pass_dce`. Since the numbers do not include opportunities unblocked in downstream phases, Table 3 is a conservative reflection of all the enabled opportunities.

The 'Lock Pointer Equivalences' column displays the number of uncovered lock-pointer equivalences, of the kind discussed in Section 4.7.1. Every such equivalence is indicative of a critical section. We were therefore able to detect all of the critical sections in the source code, except for two each in `barnes` and `wr-spl`. Although the missed ones in `wr-spl` were because of an imprecision in our may-alias information, the undetected ones in `barnes` were due to lock-pointer accesses done through `volatile` variables.

6.4 Execution-Time Improvements

Figure 5 shows the execution times T_B and T_E of the baseline and enabled executables, and the relative performance T_B/T_E. Each reported time is the average of the last five of an eight-run experiment. The graphs show that improvements can sometimes be substantial, as in the `m-fmm` case, where it ranged from 41% to 19%.

In all of the tested benchmarks except two, relative improvements of 5% or more were seen at one or more thread counts. As an example, `ocean-c` exhibited improvements of 22% and 15% at two and four threads. It uses a red-black Gauss-Seidel multigrid solver; we suspect this benefits more from the enabled optimizations at those thread counts. `wr-spl` showed 14% and 9% improvements at two and 16 threads. Another example is `barnes`, which registered 15% and 11% improvements at four and eight threads.

The gains on `fft` and `radix` were at most 4% and 3%. Although `radix` has six critical sections, they are all in one procedure.

The average improvements across all the tested programs at thread counts 1, 2, 4, 8, 16 were 5%, 9%, 8%, 6%, 5% respectively. Thus, the average improvement over all threads was 6%.

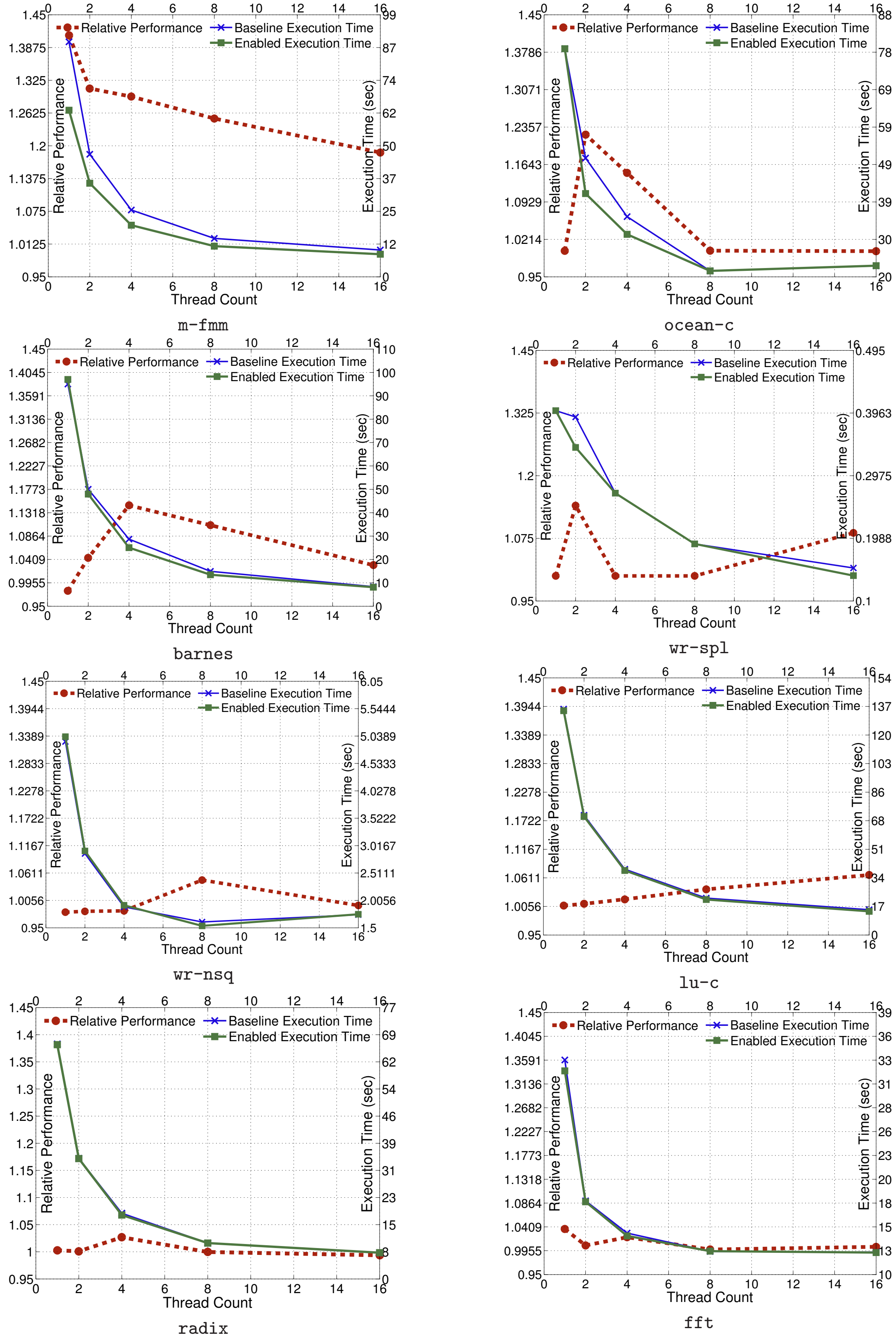
Since 37% of `m-fmm`'s defined procedures ultimately touch a synchronization, it is perhaps unsurprising that the SR Technique benefits it the most. A serial compiler would normally be unduly conservative on opportunities associated with those procedures.

6.4.1 Analyzing the Relative Performance Trends

While `m-fmm`'s relative performance decreases with increasing thread count, `lu-c`'s increases, from 1% to 7%. For the others, there is no specific trend. Using simple arguments based on serial fractions, we show below that even when the enabled performance is always better than the baseline performance, the relative performance trend can be either increasing, decreasing, or flat.

Assuming a program has a fixed serial part and the rest of it is perfectly parallelizable, its execution time $T(n)$ with n threads is

$$T(n) = T(1)\mathfrak{f} + \frac{T(1)(1-\mathfrak{f})}{n}, \tag{15}$$



m-fmm

ocean-c

barnes

wr-spl

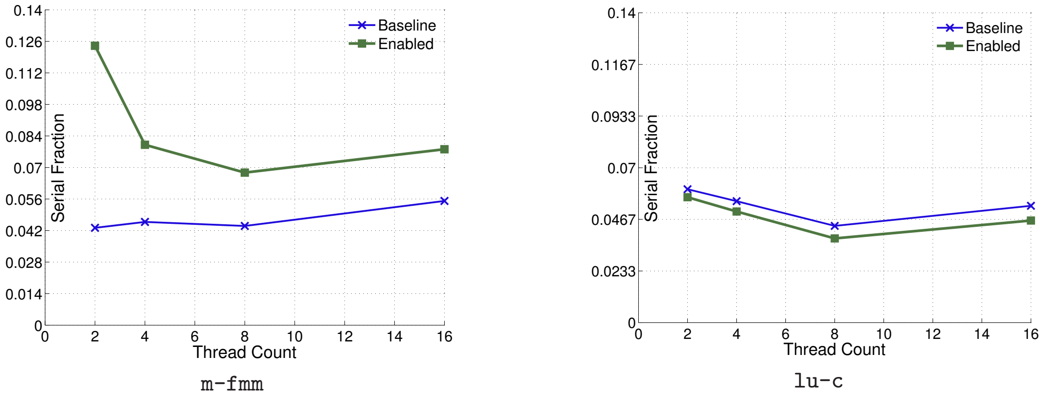
wr-nsq

lu-c

radix

fft

Figure 5. Relative performances of the enabled and baseline executables. Each graph's right axis measures T_B and T_E, the execution times in seconds of the baseline and enabled executables of a benchmark at thread counts 1, 2, 4, 8 and 16. The left axis measures T_B/T_E.



m-fmm

lu-c

Figure 6. Serial fractions, computed from execution-time measurements. For m-fmm, $\mathfrak{f}_E > \mathfrak{f}_B$ always holds. For lu-c, $\mathfrak{f}_B > \mathfrak{f}_E$ always holds.

where $\mathfrak{f}$ is the so-called serial fraction [21]. Therefore, the program's relative performance, as a function of n, is

$$RP(n) = \frac{T_B(n)}{T_E(n)} = \frac{T_B(1)}{T_E(1)} \frac{(n-1)\mathfrak{f}_B + 1}{(n-1)\mathfrak{f}_E + 1}, \qquad (16)$$

where $\mathfrak{f}_B$ and $\mathfrak{f}_E$ are the serial fractions of the baseline and enabled executables. These serial fractions can be determined from observed run times [21]; Figure 6 shows them for m-fmm and lu-c.

Since $RP(n+1) - RP(n) = \kappa(n)(\mathfrak{f}_B - \mathfrak{f}_E)$, where $\kappa(n)$ is always positive, $RP(n+1) \geq RP(n)$ iff $\mathfrak{f}_B \geq \mathfrak{f}_E$. Now, if the enabled optimizations have a bigger impact on the parallel parts of the code, we will have $\mathfrak{f}_B < \mathfrak{f}_E$. If they have a bigger impact on the serial parts, we will have $\mathfrak{f}_B > \mathfrak{f}_E$. From Figure 6, the former appears to be the case for m-fmm and the latter for lu-c. Hence the downward and upward relative performance trends for m-fmm and lu-c.

6.4.2 Impact of Enabled Optimizations on Register Pressure

A secondary effect of enabling optimizations is that live ranges generally become longer. Hence, register pressure can increase. Register allocators typically cope with register pressure by spilling to memory. Because spilled code has costs, the enabling of optimizations can negatively influence the quality of the generated code. Register allocation in gcc happens in the IRA phase, which is a fairly recent addition—it replaced the old allocator in release series 4.4 [14], the series preceding our revision. In the original FMM program, the SR phase exposed a number of common-subexpression opportunities in VListInteraction, its hottest procedure. Several of these were between expressions involving field-based array-element accesses, in which one was deep inside a loop and the other was outside of it, with another loop in between. Unblocking them resulted in the extension of live ranges across entire loops; this sufficiently strained the IRA phase that improvements from the enabled optimizations were masked. We suppressed this register-allocation artifact by outlining the two loops into their own procedure. In the absence of the SR phase, the run time of this modified FMM program, i.e., m-fmm, does not perceptibly change. Its statistical execution profile also remains essentially the same.

6.5 Compile-Time Measurements

Table 4 shows the times for producing the baseline versions. The 'SR Technique Statistics' section displays data on compiling into the enabled versions. Shown is the number of times the SR loop iterated before the IR reached quiescence. This is exactly the number of times the SR Technique was applied for a benchmark. 'Total Time' is the aggregate time over all these applications. An average per-application time for the SR Technique can therefore be obtained—this is shown in column 'A'. We thus see that the SR Technique increases baseline compilation times by 5% on average.

Program	Baseline Time (B)	SR Technique Statistics			$\frac{A}{B}$
		Total Time	No. of Applications	Average Time (A)	
m-fmm	9.36	0.88	3	0.29	3%
ocean-c	26.88	9.95	4	2.49	9%
barnes	6.68	0.58	3	0.19	3%
wr-spl	5.93	0.87	3	0.29	5%
wr-nsq	5.22	2.97	11	0.27	5%
lu-c	2.47	0.36	3	0.12	5%
radix	1.93	0.56	4	0.14	7%
fft	1.75	0.33	3	0.11	6%

Table 4. Measurements of compilation times, in seconds. 'B' is the time to produce the baseline executables. 'A' is the average time per application of the SR Technique within the SR phase.

7. Related Work

There have been numerous works in the broad area of multithreaded-program optimization. Some were discussed in Section 1. This section covers other works in the area that are relevant to our effort.

7.1 Concurrency Analyses

Past analyses have utilized a variety of techniques for discovering whether code fragments may execute in parallel. An early one, by Bristow et al., built the Interprocess Precedence Graph, an abstraction for denoting the synchronization-imposed execution ordering among processes [8]. They modeled synchronization using event variables, with set/post, clear/reset and wait being the supported operations. Taylor proposed a state-based technique for generating, via simulation, an Ada program's "concurrency history" [43]. The resulting state space, however, can be exponential in the number of "tasks", i.e., groups of computations that may concurrently execute.

A few projects calculated the MHP relation for Java [30, 25]. These were based on an abstraction called the Parallel Execution Graph (PEG). A drawback with PEGs is that they combine the CFGs of individual threads. Therefore, not only do they require a bound on the number of coincident threads modeled, but they also potentially grow in size with this bound.

Several projects tackled the concurrency-determination problem by deducing complementary knowledge, such as partial execution orders and the Cannot-Happen-Together (CHT) relation [9, 13, 27]. Of these, the "nonconcurrency" analysis by Masticola and Ryder is perhaps closest to our refinement-based method of constructing PCGs [27]. Their work computes the CHT relation by progressively improving an approximation through a series of refinements. But they diverge in several crucial ways from our work:

- ***No notion of interference.*** Ascertaining only the CHT relation means that interference is at best either "none" or "anything".
- ***The assumption of single-instance tasks.*** Their initial CHT solution is $Task(s) - \{s\}$ for each statement s, where $Task(s)$ is the set of statements in the task containing s. This is premised on there being at most one instance of a task at any moment. In a POSIX-like threading model, however, statements reachable from a start routine may belong to multiple coincident tasks.
- ***Refinements specific to alternative parallel-programming models.*** Their refinements were primarily designed for Ada's rendezvous synchronization mechanism and binary semaphores.

7.2 Interference on Shared Data

A past abstraction probably closest to the PCG is the Concurrency Graph (CG), by Zhang et al [48]. Like the PCG, it is a labeled undirected graph in which edges represent the MHP relation. Nodes, however, stand for critical sections. But more importantly, the CG has a coarse notion of interference. An edge is labeled I if conflicting accesses exist between the corresponding critical sections, and labeled N otherwise. Zhang et al. used CGs for a purpose different from in our work—to assign locks to critical sections. They assumed CGs to be given, and manually constructed them [48].

Rodríguez et al. presented extensions to the Java Modeling Language for specifying the noninterference of methods [34]. They informally described two kinds of interference: "internal" and "external". Internal interference is when a concurrent thread alters program invariants between a method's entry and exit. External interference is when pre- and post-conditions are violated due to changes by a concurrent thread between a call and method entry, and between a method exit and caller resumption. Their work did not address the issue of automatically inferring interference.

In Hendren and Nicolau's analysis of recursive data structures, the concept of interference is synonymous with conflict [16]. Other authors have considered interference as conflict combined with the MHP relation [22, 36]. In our work, this would be analogous to an edge in the PCG plus Condition C1.

There is a family of static analyses for determining whether an object may *thread-escape* [10, 5, 7]. An object o thread-escapes if it could be accessed by more than one thread [10]. This definition lacks temporality. That is, even if o is accessed by threads in disjoint time intervals, it still escapes. Although thread-escaping is not the same as interference, the idea is nonetheless complementary because threads can never interfere on objects that do not thread-escape.

Praun and Gross devised a static analysis for Java to determine conflicting object accesses [45]. This was based on an abstraction called the Object Use Graph (OUG), which is built per abstract object using symbolic execution. Conflict in their terminology has the sense of interference in our work. There are, however, important differences. Some of them are: (1) there is no counterpart to Condition C2's second clause; (2) to establish pairs of conflicting accesses, OUGs need to be processed for pairs of conflicting events.

7.3 IR-Related Issues in the EPP Approach

A notable aspect of the EPP approach is that it usually relies on specialized IRs. Examples of EPP IRs include the Parallel Control Flow Graph [41, 17], the Parallel Program Graph (PPG) [37], the CSSA Form [24], and the CSSAME Form [32].

```
 0  a = 0;
 1  b = 0;
 2  cobegin
 3    begin
 4      LOCK(L);
 5      a = 5;
 6      b = a+3;
 7      if (b > 4) {
 8        a = a+b;
 9      }
10      x = a;
11      UNLOCK(L);
12    end
13    begin
14      LOCK(L);
15      a = b+6;
16      y = a;
17      UNLOCK(L);
18    end
19  coend
20  print(x);
21  print(y);
```

```
 0′  a0 = 0;
 1′  b0 = 0;
 2′  cobegin
 3′    begin
 4′      LOCK(L);
 5′      a1 = 5;
 6′      ta1 = π(a1, a4);
 7′      b1 = ta1+3;
 8′      if (b1 > 4) {
 9′        ta11 = π(a1, a4);
10′        a2 = ta11+b1;
11′      }
12′      a3 = ϕ(a1, a2);
13′      ta12 = π(a3, a4);
14′      x0 = ta12;
15′      UNLOCK(L);
16′    end
17′    begin
18′      LOCK(L);
19′      tb0 = π(b0, b1);
20′      a4 = tb0+6;
21′      ta4 = π(a4, a1, a2);
22′      y0 = ta4;
23′      UNLOCK(L);
24′    end
25′  coend
26′  a5 = ϕ(a3, a4);
27′  print(x0);
28′  print(y0);
```

Figure 7. A program and its CSSA Form, by Novillo et al [32]. Since the code is well-synchronized, the π-function assignments can be elided *if* the concurrent access of shared variables is accounted for in the may_{def} and may_{use} sets of the synchronizations.

Most EPP efforts, excluding some around the PPG and the CSSAME Form [37, 32], have either ignored synchronizations [22, 36] or only modeled a simple event-variable-based version of it, in which `post` and `wait` are the only supported operations, with no options for resetting [28, 41, 24].

The PPG has special edges for depicting arbitrary synchronization orders. But as far as we know, algorithmic methods for deducing them for a POSIX-like threading model have not been given.

The CSSAME Form is the CSSA Form after simplifications using certain kinds of mutual exclusion information [32]. Similar simplifications directly follow with the SR approach, since it assumes data-race freedom (unlike the EPP approach).

Specifically, the π-function is used by the CSSA and CSSAME Forms to signify conflicting accesses. Its structure for a shared variable v is $\pi(v_1, v_2, \ldots, v_n)$ [24]. Its value is one of the v_is. Also, one of the v_is indicates v's reaching definition in the current thread. The other v_is are v's reaching definitions from concurrent threads. In a memory model where data races are allowed, π-functions are needed before uses that may conflict with potentially concurrent definitions [24]. Under data-race freedom, however, these π-functions can be dropped if, between the may_{def} and may_{use} sets of the synchronizations, all the concurrent conflicting accesses are modeled. The informal rationale for this is that in any interleaving, a pair of conflicting accesses m_1 and m_2 by threads h_1 and h_2 must be separated by a synchronization s' performed by one of them. Otherwise, using arguments similar to that in the proof for Refinement 4, a sequentially consistent schedule can be constructed in which m_1 and m_2 are adjacent, implying the contradiction of a data race. Since m_1 and s', or s' and m_2, are performed by the same thread, a control-flow path always exists between a synchronization and a conflicting access. Hence, for well-synchronized programs, it suffices to only model concurrent accesses at synchronization points.

As an example, Figure 7 shows a program and its CSSA Form, by Novillo et al [32, Pages 358, 360]. They present two theorems that sanction the omission of a π-function argument in cases where its reaching definition from another thread resides in a critical section that holds the same lock. While the theorems are true even for programs with data races, the SR approach subsumes their implications for well-synchronized programs. For instance, constant propagation opportunities within the first critical section in Figure 7 would be exposed by their theorems. These would also be exposed to a classical constant propagator by the SR approach.

7.4 Preserving SC on Weakly Consistent Hardware

There have been efforts on mapping code written for a memory model that is presumably easy to reason about, e.g., SC, to more relaxed memory models offered by the hardware [40, 28, 42]. Their solution has been to insert special instructions, called "memory barriers" (e.g., IBM POWER3's `sync` and Intel x86's `fence`), so as to suppress the compiler and hardware from reordering code. These efforts differ from the SR approach in a number of ways:

- ***Problem solved is different.*** Their goal is to provide the appearance of SC, even in the presence of data races. The SR approach is about effectively reusing classical optimizations on data-race-free SC code, preserving both SC and data-race freedom.
- ***No reordering across synchronizations.*** Synchronization knowledge is only used to remove IR conflict edges, since doing so improves the results of the barrier-insertion analysis [40].
- ***A solution may not always be possible.*** There are programs, such as the IRIW example [6], for which inserting barriers is insufficient to realize SC on certain platforms.

7.5 Rule-Based Transformations for Data-Race-Free Models

There is work on *syntactic* elimination and reordering rules that are safe on *sequences* (i.e., traces) of memory-related operations in data-race-free programs [39]. Because of its phrasing as transformation rules on sequences, it is unclear how to incorporate the work into the existing phases of a serial compiler without nontrivial engineering. A more significant point is that the rules only consider opportunities that do not require information on cross-thread interactions. In particular, the lock/unlock-related rules only allow for the movement of code *into* critical sections, not out of them [39].

7.6 OpenMP Program Optimization

Optimizations for OpenMP programs, in general, either have used the EPP approach [38, 17] or have been restricted to within parallel constructs [44]. Satoh et al. used an IR called the Parallel Flow Graph to model the intra- and cross-thread flow of information [38]. They developed data-flow analyses for reaching definitions, memory synchronizations and cross-loop data dependences. Huang et al. observed that it is easier to optimize a high-level version of an OpenMP program than a lowered threaded version; they used an IR called the Parallel Control Flow Graph for this purpose [17].

7.7 Extending Sequential Optimizations to Parallel Code

Praun et al. showed how classical SSAPRE can be changed (manually) to consume conflict information derived from OUGs [46]. Heffner et al. described modifications to three sequential object-oriented optimizations in order to improve their effectiveness in the presence of concurrency [15]. The modified versions were based on a field-access analysis that maps every field *fld* to the duple $(\mathit{locks}, \mathit{threads})$, in which *locks* is the set of locks held on every access of *fld*, and in which *threads* is essentially a Boolean that indicates whether a single thread or multiple threads access *fld*. No distinction was made between read and write accesses. Moreover, the results of the field-access analysis were consumed by the modifications in ways that were specific to each extended optimization.

For example, one of the modifications by Heffner et al. was to limit the null safety-check optimizer to only those fields for which *threads* indicated single-thread access. Another modification, to the redundant-load elimination (RLE) optimizer, was to exclude from the kill sets of monitor-enter statements those fields that were protected by outer monitors or for which *threads* indicated single-thread access [15]. The filtering effect was achieved by subclassing the original RLE optimizer.

8. Conclusions

We presented an interprocedural static analysis that allows classical optimizations to be applied on data-race-free multithreaded programs in more cases than when synchronizations are viewed as opaque operations. We have shown that the additional precision is useful for optimization, and produces superior performance.

It may also be possible to use a version of this analysis in a static concurrency-bug detection tool, where viewing synchronizations as opaque could obscure essential information.

Much of the precision of our analysis comes from Refinement 4, which exploits the fact that C and C++ do not define semantics for data races. The coming standards for these languages will likely forbid them. As stated, Refinement 4 does not apply in all cases in which it could. In particular, it does not yet account for differences in behavior between synchronization primitives. For example, two procedures that both acquire but do not release the same lock, and that perform no other synchronization on common sync objects, must have empty interference sets, since any real interference would reflect a data race. This probably has little bearing on most existing code, but a C++ analog is likely to be important for two procedures both reading the same C++0x `atomic<T>` variable.

Acknowledgments

An abridged version of this paper will appear in the proceedings of the 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2011). We thank the anonymous POPL referees for their keen and valuable feedback on drafts of the abridged version, which helped improve this work.

References

[1] ADVE, S. V., AND BOEHM, H.-J. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Communications of the ACM 53*, 8 (Aug. 2010), 90–101.

[2] ADVE, S. V., AND GHARACHORLOO, K. Shared Memory Consistency Models: A Tutorial. *IEEE Computer 29*, 12 (Dec. 1996), 66–76.

[3] ADVE, S. V., AND HILL, M. D. Weak Ordering—A New Definition. In *Proc. International Symposium on Computer Architecture* (May 1990), pp. 2–14.

[4] BERMAN, A. M., Ed. *Conference on Object-Oriented Programming, Systems, Languages and Applications* (Nov. 1999).

[5] BLANCHET, B. Escape Analysis for Object-Oriented Languages: Application to Java. In Berman [4], pp. 20–34.

[6] BOEHM, H.-J., AND ADVE, S. V. Foundations of the C++ Concurrency Memory Model. In *Proc. Conference on Programming Language Design and Implementation* (June 2008), pp. 68–78.

[7] BOGDA, J., AND HÖLZLE, U. Removing Unnecessary Synchronization in Java. In Berman [4], pp. 35–46.

[8] BRISTOW, G., DREY, C., EDWARDS, B., AND RIDDLE, W. Anomaly Detection in Concurrent Programs. In *Proc. International Conference on Software Engineering* (Sept. 1979), pp. 265–273.

[9] Callahan, D., and Subhlok, J. Static Analysis of Low-level Synchronization. In *Proc. ACM Workshop on Parallel and Distributed Debugging* (May 1988), pp. 100–111.

[10] Choi, J.-D., Gupta, M., Sreedhar, V. C., and Midkiff, S. P. Escape Analysis for Java. In Berman [4], pp. 1–19.

[11] Chow, F., Chan, S., Liu, S.-M., Lo, R., and Streich, M. Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In *Proc. International Conference on Compiler Construction* (Apr. 1996), vol. 1060 of *Lecture Notes in Computer Science*, Springer, pp. 253–267.

[12] Cytron, R., Ferrante, J., Rosen, B. K., and Wegman, M. N. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (Oct. 1991), 451–490.

[13] Duesterwald, E., and Soffa, M. L. Concurrency Analysis in the Presence of Procedures Using a Data-Flow Framework. In *Proc. Symposium on Testing, Analysis and Verification* (Oct. 1991), pp. 36–48.

[14] GCC 4.4 Release Series—Changes, New Features, and Fixes. At `http://gcc.gnu.org/gcc-4.4/changes.html`.

[15] Heffner, K., Tarditi, D., and Smith, M. D. Extending Object-Oriented Optimizations for Concurrent Programs. In *Proc. International Conference on Parallel Architectures and Compilation Techniques* (Sept. 2007), pp. 119–129.

[16] Hendren, L. J., and Nicolau, A. Parallelizing Programs with Recursive Data Structures. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (Jan. 1990), 35–47.

[17] Huang, L., Sethuraman, G., and Chapman, B. Parallel Data Flow Analysis for OpenMP Programs. In *Proc. International Workshop on OpenMP* (June 2007), vol. 4935 of *Lecture Notes in Computer Science*, Springer, pp. 138–142.

[18] The IEEE and The Open Group. *IEEE Standard 1003.1*, 2004.

[19] C Standard ISO/IEC 9899. At `http://www.open-std.org/JTC1/`.

[20] Kam, J. B., and Ullman, J. D. Monotone Data Flow Analysis Frameworks. *Acta Informatica 7*, 3 (Sept. 1977), 305–317.

[21] Karp, A. H., and Flatt, H. P. Measuring Parallel Processor Performance. *Communications of the ACM 33*, 5 (May 1990), 539–543.

[22] Knoop, J., and Steffen, B. Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. *ACM Transactions on Programming Languages and Systems 18*, 3 (May 1996), 268–299.

[23] Lamport, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers C-28*, 9 (Sept. 1979), 690–691.

[24] Lee, J., Midkiff, S. P., and Padua, D. A. Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In Li et al. [26], pp. 114–130.

[25] Li, L., and Verbrugge, C. A Practical MHP Information Analysis for Concurrent Java Programs. In *Proc. International Workshop on Languages and Compilers for Parallel Computing* (Sept. 2004), vol. 3602 of *Lecture Notes in Computer Science*, Springer, pp. 194–208.

[26] Li, Z., Yew, P.-C., Chatterjee, S., Huang, C.-H., Sadayappan, P., and Sehr, D. C., Eds. *International Workshop on Languages and Compilers for Parallel Computing* (Aug. 1997), vol. 1366 of *Lecture Notes in Computer Science*, Springer.

[27] Masticola, S. P., and Ryder, B. G. Non-concurrency Analysis. In *Proc. Symposium on Principles and Practices of Parallel Programming* (May 1993), pp. 129–138.

[28] Midkiff, S. P., and Padua, D. A. Issues in the Optimization of Parallel Programs. In *Proc. International Conference on Parallel Processing* (Aug. 1990), vol. II, The Pennsylvania State University Press, pp. 105–113.

[29] Naumovich, G., and Avrunin, G. S. A Conservative Data Flow Algorithm for Detecting All Pairs of Statements that May Happen in Parallel. In *Proc. Symposium on Foundations of Software Engineering* (Nov. 1998), pp. 24–34.

[30] Naumovich, G., Avrunin, G. S., and Clarke, L. A. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. In *Proc. Symposium on Foundations of Software Engineering* (Sept. 1999), pp. 338–354.

[31] Novillo, D. Memory SSA—A Unified Approach for Sparsely Representing Memory Operations. In *Proc. GCC Developers' Summit* (July 2007), pp. 97–110.

[32] Novillo, D., Unrau, R., and Schaeffer, J. Concurrent SSA Form in the Presence of Mutual Exclusion. In *Proc. International Conference on Parallel Processing* (Aug. 1998), IEEE Computer Society Press, pp. 356–364.

[33] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, version 3.0 ed., May 2008.

[34] Rodríguez, E., Dwyer, M., Flanagan, C., Hatcliff, J., Leavens, G. T., and Robby. Extending JML for Modular Specification and Verification of Multi-threaded Programs. In *Proc. European Conference on Object-Oriented Programming* (July 2005), vol. 3586 of *Lecture Notes in Computer Science*, Springer, pp. 551–576.

[35] Ruf, E. Effective Synchronization Removal for Java. In *Proc. Conference on Programming Language Design and Implementation* (June 2000), pp. 208–218.

[36] Rugina, R., and Rinard, M. C. Pointer Analysis for Structured Parallel Programs. *ACM Transactions on Programming Languages and Systems 25*, 1 (Jan. 2003), 70–116.

[37] Sarkar, V. Analysis and Optimization of Explicitly Parallel Programs Using the Parallel Program Graph Representation. In Li et al. [26], pp. 94–113.

[38] Satoh, S., Kusano, K., and Sato, M. Compiler Optimization Techniques for OpenMP Programs. *Scientific Programming 9*, 2/3 (Aug. 2001), 131–142.

[39] Ševčík, J. *Program Transformations in Weak Memory Models*. PhD thesis, University of Edinburgh, 2008.

[40] Shasha, D., and Snir, M. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems 10*, 2 (Apr. 1988), 282–312.

[41] Srinivasan, H., Hook, J., and Wolfe, M. Static Single Assignment for Explicitly Parallel Programs. In *Proc. Symposium on Principles of Programming Languages* (Jan. 1993), pp. 260–272.

[42] Sura, Z., Fang, X., Wong, C.-L., Midkiff, S. P., Lee, J., and Padua, D. A. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *Proc. Symposium on Principles and Practices of Parallel Programming* (June 2005), pp. 2–13.

[43] Taylor, R. N. A General-Purpose Algorithm for Analyzing Concurrent Programs. *Communications of the ACM 26*, 5 (May 1983), 362–376.

[44] Tian, X., Bik, A., Girkar, M., Grey, P., Saito, H., and Su, E. Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. *Intel Technology Journal 6*, 1 (Feb. 2002), 36–46.

[45] von Praun, C., and Gross, T. R. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *Proc. Conference on Programming Language Design and Implementation* (June 2003), pp. 338–349.

[46] von Praun, C., Schneider, F., and Gross, T. R. Load Elimination in the Presence of Side Effects, Concurrency and Precise Exceptions. In *Proc. International Workshop on Languages and Compilers for Parallel Computing* (Oct. 2003), vol. 2958 of *Lecture Notes in Computer Science*, Springer, pp. 390–405.

[47] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta,

A. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. International Symposium on Computer Architecture* (June 1995), pp. 24–36.

[48] ZHANG, Y., SREEDHAR, V. C., ZHU, W., SARKAR, V., AND GAO, G. R. Optimized Lock Assignment and Allocation: A Method for Exploiting Concurrency among Critical Sections. CAPSL Technical Memo Revised 65, University of Delaware, Mar. 2007.

Notation Index