Concurrency Analysis for Java

Cristian Ungureanu and Suresh Jagannathan

NEC Research Institute
4 Independence Way
Princeton NJ 08540, USA
{cristian,suresh}@research.nj.nec.com

Abstract. Concurrency is an integral feature of Java. While there has been recent research [CGS+99,BH99,WR99,Bla99] on devising analyses to eliminate the overhead imposed by synchronization, these analyses do not explicitly track multiple threads of control, nor do they appear particularly well-suited to facilitate other concurrency-related optimizations that may be applicable in a parallel or distributed environment.
In this paper, we develop a novel program analysis for Java, which explicitly incorporates an abstract (semantic) notion of threads. Our analysis framework is distinguished from related efforts in three important respects:

1. It employs a whole-program flow analysis adapted from a simple sequential analysis framework that formally defines a notion of an *abstract* thread of control. Our initial approximation defines for each thread the set of objects potentially locked or accessed by that thread.
2. The approximation imposes no restriction on the lifetime of objects involved in a synchronization event. Thus, objects may freely escape from the context in which they are created without necessarily being regarded as shared by multiple threads.
3. To provide added precision, the analysis is subsequently refined to use a per program-point abstract heap. The use of such a heap allows thread-specific flow-sensitive optimizations to be selectively applied in different program contexts, but requires a more sophisticated notion of reachability. One immediate consequence of this framework is its ability to support strong-updates [CWZ90] of global shared data.

Our analysis has been implemented as part of a native-code optimizing compiler for Java currently under development. Benchmark results indicate the analyses have relatively small computation cost, but can lead to significant improvements in the quality of generated code.

[0] **Appears in the Proceedings of the 2000 International Static Analysis Symposium.**

1 Introduction

An important reason for Java's growing popularity is its integral support for concurrency. Java's simple thread semantics coupled with various extensions for distributed and parallel computing [Jav98,Jav99] are likely to make it increasingly attractive as a vehicle for expressing parallel and distributed applications.

In this paper, we define a novel flow analysis framework that deals explicitly with Java's concurrency features. Our starting point is a monovariant analysis for Java's sequential core extended with support for threads. The analysis defines an approximate notion of an exact thread, and uses this approximation to associate with each object o, the set of threads that lock or access o during execution of the program. The analysis also imposes constraints to allow it to identify potentially multiple (active) exact instances of the same abstract thread.

To improve precision, we subsequently develop a flow-sensitive refinement of this analysis which introduces a per program-point abstract heap. For many concurrent programs, an object that is not initially shared by multiple threads, may *eventually* become shared. By using a per program-point heap, the analysis can effectively capture this intuition; the structure of the heap at a given program point reveals the sharing properties of an object with respect to the threads which reference it at that point. By incorporating a notion of data reachability at method call and return points, unreachable paths can be filtered, further improving the precision. One immediate consequence of this analysis is its ability to support strong-update [CWZ90] of globally shared data.

We envision a number of optimizations facilitated by our analysis framework. Synchronization elimination is one obvious candidate. Researchers have recently focussed on eliminating synchronization overhead in Java programs because of its negative impact on sequential programs [BKMS98]. Because Java's thread semantics implicitly associates a lock with every object, many instances whose state is not actually accessed simultaneously by multiple threads will nonetheless incur synchronization penalties. Some implementations, including ours, reserve space in each object for a lock. By knowing which objects will never be involved in a synchronization event, this space overhead can be eliminated. Moreover, synchronization operations on objects that cannot be locked by other threads can be eliminated. Instances of the same type but allocated at different program points, and calls to the same synchronized method made at different call points, may have different synchronization requirements. The flow analysis framework proposed here allows optimizations to *selectively* eliminate both kinds of synchronization overhead.

For distributed programs, our analysis can also help improve locality. For example, an object known to be local to a given thread can be allocated on the same processor as that thread. An object shared among multiple threads can be allocated on a heap accessible to just these threads. By explicitly tracking the relation between objects and the threads which reference them, locality-specific optimizations that improve memory layout are more easily expressed.

We have implemented the analysis as part of an optimizing native-code Java compiler under development. Experimental results for both sequential and multi-threaded Java programs are encouraging.

The remainder of the paper is organized as follows. Sections 2 and 3 introduce a core language and its semantics. Section 4 presents the control-flow analysis for this language. Section 5 extends this analysis with a notion of heap reachability, leading to an analysis that yields a collection of heaps, indexed by program points. Experimental results are described in Section 6, and related work is discussed in Section 7.

2 Syntax

We describe our analysis using a small kernel typed intermediate language (*IL*) shown in Figure 1. A program in this language consists of a collection of type, class, and static definitions. The expression $\mu f.fn$ defines a local recursive function; $x.i$ selects field i from instance x; $x \Leftarrow m$ extracts m from x and binds *self* in m to x; $x \Leftarrow c.m$ extracts method m from class c and binds *self* to x; $r \cdot \underline{i}$ and $r \cdot \underline{m}$ extract static fields and methods respectively; $\texttt{spawn}\,(c, m)$ allocates a new thread object and creates a new thread of control to run method m defined in class c; $\texttt{synch}\,(x)$ blocks execution if x is already locked by a different thread; $y \rightarrow [\iota_i \Rightarrow exp_i]^* \;\texttt{else}\; exp_n$ defines a case statement that dispatches on y; `store` and `fetch` have the obvious meaning. To make the presentation tractable, we have omitted many important features found in Java that must be expressed within a realistic intermediate language; exceptions, mutable local variables, type-casting, and arrays are some notable examples. Adapting these features into the analysis is straightforward.

P	$::= T^* \; C^* \; S^* \; exp$	program
T	$::= \texttt{type}\; ot \; \left[\; <: \; ot_j \right] \texttt{fields}\; \{(i : \; t)^*\}\; \texttt{methods}\; \{(m : mt)^*\}$	types
C	$::= \texttt{class}\; c \;\texttt{implements}\; ot \; \left[\texttt{inherits}\; \{c'\}\right] \;\texttt{with}\; \{(m = \lambda\texttt{self}.fn)^*\}$	classes
S	$::= \texttt{record}\; r \;\; \{(\underline{i} : \; t)^*\} \;\; \{(\underline{m} : t = fn)^*\}$	statics
fn	$::= \lambda x.exp$	functions
exp	$::= x \mid \textsf{let}\; x = ntExp \;\textsf{in}\; exp \;\textsf{end}$	expressions
$ntExp$	$::= \iota \mid \mu f.fn \mid x.i \mid x \Leftarrow m \mid x \Leftarrow c.m \mid r \cdot \underline{i} \mid r \cdot \underline{m} \mid$ $x(x^*) \mid \texttt{new}\; c \mid \texttt{spawn}\,(c, m) \mid \texttt{synch}\,(x) \mid$ $y \rightarrow [\iota_i \Rightarrow exp_i]^* \;\texttt{else}\; exp_n \mid \texttt{store}\,(x, y) \mid \texttt{fetch}\,(y)$	

Fig. 1. The Language.

3 *IL* Exact Semantics

We present an exact semantics for our IL as a small-step operational semantics. The semantic domain of values is given in Figure 2. Methods evaluate to closures, which are pairs consisting of expressions and environments; local recursive functions are defined in the obvious way [MTHM97]. A class value is a record, containing the definition of method bodies; the static fields and static methods of a class are collected in a separate record. The elaboration of declarations produces an initial environment, $\rho_\perp$, containing bindings for these records. Every instance is represented as a triple consisting of the instance's class, the variable to which the instance is bound, and an integer used to distinguish multiple incarnations of the instance from the same program point. A location is either an instance, a static field, or an instance field. An executing thread consists of an identifier, the expression currently being evaluated by the thread, an environment, and a continuation stack which is a list of continuation frames. There are only three kinds of frames in the semantics: *return* frames are used to pass control from return points at non-tail calls; *unlock* frames are used to handle synchronization; and *halt* frames are used to mark terminated threads.

$$
\begin{array}{rrcl}
v \in & \mathit{Value} & = & \mathit{Loc} + \mathit{Clos} + \mathit{Record} + \mathit{Const} \\
\mathtt{Closure}\langle \mathit{exp}, \rho \rangle \in & \mathit{Clos} & = & \mathit{LambdaExp} \times \mathit{Env} \\
\mathtt{Rec}\,\langle f, \mathit{exp}, \rho \rangle \in & \mathit{Rec} & = & \mathit{Var} \times \mathit{LambdaExp} \times \mathit{Env} \\
\rho \in & \mathit{Env} & = & \mathit{Name} \xrightarrow{fin} \mathit{Value} \\
\sigma \in & \mathit{Store} & = & \mathit{Loc} \xrightarrow{fin} \mathit{Value} \\
l \in & \mathit{Loc} & = & \mathit{Inst} + (\mathit{ClassName} + \mathit{Inst}) \times \mathit{FieldName} \\
r \in & \mathit{Record} & = & \mathit{Name} \xrightarrow{fin} \mathit{Value} \\
o \in & \mathit{Inst} & = & \mathit{ClassName} \times \mathit{Var} \times \mathit{Integer} \\
t \in & \mathit{Thread} & = & \mathit{ThreadID} \times \mathit{Exp} \times \mathit{Env} \times \mathit{Kont}^* \\
\mathit{tid} \in & \mathit{ThreadID} & = & \{Main\} + \mathit{Inst} \\
TM \in & \mathit{ThreadMap} & = & \mathit{ThreadID} \xrightarrow{fin} \mathit{Thread} \\
LM \in & \mathit{LockMap} & = & \mathit{Inst} \xrightarrow{fin} \mathcal{P}(\mathit{ThreadID} \times N) \\
k \in & \mathit{Kont} & = & \mathsf{Return}\langle y, \rho, e \rangle + \mathsf{Unlock}\langle y \rangle + \mathsf{Halt}
\end{array}
$$

Fig. 2. Semantic Domains.

A program *configuration* consists of a *thread map*, a *store*, and a *lock map*. A thread map binds thread identifiers to thread states. A lock map binds instances to pairs $\langle \mathit{tid}, n \rangle$ consisting of a thread identifier *tid* for the thread that has locked it and the number n of times it has been locked by that thread. The transition rules are shown in Figure 3; we omit the rules for declarations. In presenting the rules, we also omit unneeded components of a configuration.

$$\boxed{\mathit{Thread} \times \mathit{Store} \times \mathit{LockMap} \;\rightarrow\; \mathcal{P}(\mathit{Thread}) \times \mathit{Store} \times \mathit{LockMap}}$$

$\langle tid, x, \rho, \mathsf{Halt} : k^* \rangle \;\rightarrow\; \{\}$
$\langle tid, x, \rho, \mathsf{Return}\langle y, \rho', e\rangle : k^* \rangle \;\rightarrow\; \{\langle tid, e, \rho'[y \mapsto \rho(x)], k^* \rangle\}$
$\langle tid, x, \rho, \mathsf{Unlock}\langle y\rangle : k^* \rangle, \sigma, LM \;\rightarrow\; \{\langle tid, x, \rho, k^* \rangle\}, \sigma, LM'$
  if $\mathcal{C}heck\mathcal{U}nlock(\rho(y), tid, LM)$ and $LM' = \mathcal{U}nlock(\rho(y), tid, LM)$
$\langle tid, \mathsf{let}\ x = \iota\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle \;\rightarrow\; \{\langle tid, e, \rho[x \mapsto \iota], k^* \rangle\}$
$\langle tid, \mathsf{let}\ x = r \cdot \underline{i}\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle \;\rightarrow\; \{\langle tid, e, \rho[x \mapsto \rho(r).\underline{i}], k^* \rangle\}$
$\langle tid, \mathsf{let}\ x = r \cdot \underline{m}\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle \;\rightarrow\; \{\langle tid, e, \rho[x \mapsto \rho(r).\underline{m}], k^* \rangle\}$
$\langle tid, \mathsf{let}\ x = \mathtt{new}(c)\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle, \sigma \;\rightarrow\; \{\langle tid, e, \rho[x \mapsto \langle c, x, n\rangle], k^* \rangle\}, \sigma'$
  where $\sigma' = \sigma[\langle c, x, n\rangle \mapsto \mathcal{N}ew\mathcal{O}bject(c)]$ and n is fresh
$\langle tid, \mathsf{let}\ x = y.i\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle \;\rightarrow\; \{\langle tid, e, \rho[x \mapsto \sigma(\rho(y)).i], k^* \rangle\}$
$\langle tid, \mathsf{let}\ x = y \Leftarrow\ m\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle \;\rightarrow\; \{\langle tid, e, \rho', k^* \rangle\},$
  where $\sigma(\rho(y)).m = \mathtt{Closure}\langle \lambda self.\lambda z.e', \rho_\perp\rangle$
  and $\rho' = \rho[x \mapsto \mathtt{Closure}\langle \lambda z.e', \rho_\perp[self \mapsto \rho(y)]\rangle]$
$\langle tid, \mathsf{let}\ x = y \Leftarrow c.m\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle \;\rightarrow\; \{\langle tid, e, \rho', k^* \rangle\}$
  where $\rho(c).m = \mathtt{Closure}\langle \lambda self.\lambda z.e', \rho_\perp\rangle$
  and $\rho' = \rho[x \mapsto \mathtt{Closure}\langle \lambda z.e', \rho_\perp[self \mapsto \rho(y)]\rangle]$
$\langle tid, \mathsf{let}\ x = y(z)\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle \;\rightarrow\; \{\langle tid, e', \rho'[w \mapsto \rho(z)], \mathsf{Return}\langle x, \rho, e\rangle : k^* \rangle$
  where $\rho(y) = \mathtt{Closure}\langle \lambda w.e', \rho'\rangle$
$\langle tid, \mathsf{let}\ x = \mu f.fn\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle \;\rightarrow\; \{\langle tid, e, \rho[x \mapsto \mathtt{Rec}\ \langle f, fn, \rho\rangle], k^* \rangle\}$
$\langle tid, \mathsf{let}\ x = y(z)\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle \;\rightarrow\; \{\langle tid, e', \rho'', \mathsf{Return}\langle x, \rho, e\rangle : k^* \rangle\}$
  where $\rho(y) = \mathtt{Rec}\ \langle f, \lambda w.e', \rho'\rangle, \rho'' = \rho'[w \mapsto \rho(z), f \mapsto \mathtt{Rec}\ \langle f, \lambda w\,.\,e', \rho'\rangle]$
$\langle tid, \mathsf{let}\ x = \mathtt{store}\ (y, z)\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle, \sigma \;\rightarrow\; \{\langle tid, e, \rho, k^* \rangle, \sigma[\rho(y) \mapsto \rho(z)]\}$
$\langle tid, \mathsf{let}\ x = \mathtt{fetch}\ (y)\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle, \sigma \;\rightarrow\; \{\langle tid, e, \rho[x \mapsto \sigma(\rho(y))], k^* \rangle\}$
$\langle tid, \mathsf{let}\ x = \mathtt{spawn}\ (c, m)\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle, \sigma \;\rightarrow$
  $\{\langle tid, e, \rho[x \mapsto \langle c, x, n\rangle], k^* \rangle, \langle tid', e', \rho_0[self_i \mapsto \langle c, x, n\rangle], \mathsf{Halt}\rangle\}, \sigma'$
  where tid' is fresh and $\sigma' = \sigma[\langle c, x, n\rangle \mapsto \mathcal{N}ew\mathcal{O}bject(c)]$ and $m = \lambda self_i.\lambda().e'$
$\langle tid, \mathsf{let}\ x = \mathtt{synch}\ (y)\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle, \sigma, LM \;\rightarrow\; \{\langle tid, e, \rho', \mathsf{Unlock}\langle y\rangle : k^* \rangle\}, \sigma, LM'$
  if $\mathcal{C}heck\mathcal{L}ock(\rho(y), tid, LM) = \mathtt{true}, \rho' = \rho[x \mapsto \{\}], LM' = \mathcal{L}ock(\rho(y), tid, LM)$
$\langle tid, \mathsf{let}\ x = y \rightarrow [\iota_i \Rightarrow e_i]^*\ \mathtt{else}\ e_n\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle \;\rightarrow\; \{\langle tid, \mathsf{let}\ x = e'\ \mathsf{in}\ e\ \mathsf{end}, \rho, k^* \rangle\}$
  where $e' = e_i$ if $\rho(y) = \iota_i,$ and $e' = e_n$ otherwise

Fig. 3. Exact semantics

The rules make use of auxiliary functions ($\mathcal{C}heck\mathcal{L}ock$, $\mathcal{C}heck\mathcal{U}nlock$, $\mathcal{L}ock$ and $\mathcal{U}nlock$) that track how locks are acquired and released. Their definitions are obvious and omitted here.

We define the semantics via an evaluation relation, $Eval \subseteq Prog \times Value$:

$$Eval(T^*\ C^*\ exp) = \rho_0(last(exp)),\ \text{if}\ \{\langle Main, exp, \rho_0, \mathsf{Halt}\rangle\}, \phi, \phi \Longrightarrow^* \quad TM, \sigma, LM$$

$$\text{where} \begin{cases} \text{for all } tid \in Dom(TM), TM(tid) = \langle tid, x, \rho, \mathsf{Halt} : k\rangle. \\ Main \text{ is the main thread's id.} \end{cases}$$

The transition relation, written as $\rightarrow_\top$, is a relation on configurations,

$$\rightarrow_\top \subseteq (TM \times Store \times LM) \times (TM \times Store \times LM).$$

Thus,

$$TM, \sigma, LM \rightarrow_{\top} TM - \{t_i\} \bullet S, \sigma', LM'$$

if $t_i, \sigma, LM \rightarrow S, \sigma', LM'$ where t_i is some thread in TM, and S is the set of new threads created during the one-step transition of t_i. We write $F \bullet S$ to mean $F[tid \mapsto \langle tid, e, \rho, k\rangle]$ for all $\langle tid, e, \rho, k\rangle \in S$.

4 Abstract Semantics

Our initial specification is a monovariant analysis similar in spirit to other flow systems developed for object-based languages [PC94,DGC98,PS91]. We differ primarily in our support for concurrency. We subsequently refine the analysis to derive a more precise heap reachability map. The analysis is given as a triple $\langle F_{env}, F_{cell}, F_{lock}\rangle$ consisting of an abstract environment, an abstract store, and an abstract lock map. We call this triple an *abstract configuration*; unlike the exact semantics, the analysis computes a single abstract configuration for a given program. An abstract environment maps variables to abstract values, *i.e.*, a finite approximation of the set of values to which a variable may be bound during program execution. An abstract store maps abstract locations to abstract values. An abstract location approximates a set of locations. An abstract lock map approximates the set of threads that synchronize on a particular instance. Unlike the exact semantics, all abstract threads share a single abstract environment, F_{env}. Since the analysis models the collection of exact states in a program's evaluation by a single abstract configuration, the only relevant information about a thread is its identity, making an abstract thread map unnecessary. Thread identifiers are represented by abstract thread instances.

$$\begin{array}{lll}
F_{env} & \in \mathit{Flow} & = \mathit{Var} \rightarrow \mathit{Aval} \\
F_{cell} & \in \mathit{FStore} & = \mathit{Aloc} \rightarrow \mathit{Aval} \\
F_{lock} & \in \mathit{FLock} & = \mathit{Ainst} \rightarrow \mathcal{P}(\mathit{AThreadId}) \\
\hat{v} & \in \mathit{Aval} & = \mathcal{P}(\mathit{Aloc} + \mathit{ARecord} + \mathit{LambdaExp} + \{\mathbf{ground}\}) \\
\hat{l} & \in \mathit{Aloc} & = \mathit{Ainst} + (\mathit{ClassName} \times \mathit{FieldName}) + (\mathit{Ainst} \times \mathit{FieldName}) \\
\hat{o} & \in \mathit{Ainst} & = \mathit{ClassName} \times \mathit{Var} \\
\hat{r} & \in \mathit{ARecord} & = \mathit{Name} \rightarrow \mathit{Aval} \\
\hat{tid} & \in \mathit{AThreadId} & = \{\mathit{Main}\} + \mathit{Ainst}
\end{array}$$

Fig. 4. Domains for the flow analysis.

One useful approximation is to regard the abstract value of an instance as a set of class names. If o is an object instantiated from a class named c and bound to variable x, and if the flow analysis associates abstract value $\hat{v}$ with o, it must be the case that $\langle c, x\rangle \in \hat{v}$. Similarly, the abstract value of a method selection of the form "$o \Leftarrow m$" is the set of λ-expressions named m occurring in the classes denoted by o's abstract value. Ground values are used to approximate values of primitive type. Abstract cells approximate exact locations. Abstract values and cells are partially ordered by ordinary set inclusion.

To formally specify the meaning of an analysis, we define a relation between exact and abstract configurations:

Definition 1. *An exact configuration* $\langle TM, \sigma, LM \rangle$ agrees *with an abstract configuration,* $\langle F_{env}, F_{cell}, F_{lock} \rangle$ *written* $\langle TM, \sigma, LM \rangle \sqsubseteq \langle F_{env}, F_{cell}, F_{lock} \rangle$ *if* $TM \sqsubseteq F_{env}$*,* $\sigma \sqsubseteq F_{cell}$*, and* $LM \sqsubseteq F_{lock}$*.*

The agreement relation is defined inductively over terms and values in the exact semantics. Its definition is given in Figure 5.

1. $TM \sqsubseteq F_{env}$ if for all $tid \in Dom(TM)$ such that $TM(tid) = \langle tid, exp, \rho, k^* \rangle$, $\rho \sqsubseteq F_{env}$ and $k^* \sqsubseteq F_{env}$.
2. $\rho \sqsubseteq F_{env}$ if for all $x \in Dom(\rho)$, $\rho(x) \sqsubseteq F_{env}(x)$.
3. $\sigma \sqsubseteq F_{cell}$ if for all $l \in Dom(\sigma)$ there exists $\hat{l} \in Dom(F_{cell})$ such that $l \sqsubseteq \hat{l}$.
4. $LM \sqsubseteq F_{lock}$ if for every $o \in Dom(LM)$, there exists $\hat{o} \in Dom(F_{lock})$ such that for all $tid \in LM(o)$ there exists $\hat{tid} \in F_{lock}(\hat{o})$ such that $tid \sqsubseteq \hat{tid}$.
5. $l \sqsubseteq \hat{l}$ if $\sigma(l) \sqsubseteq F_{cell}(\hat{l})$.
6. $k^* \sqsubseteq F_{env}$ if for every $\mathsf{Return}\langle y, \rho, e \rangle \in k^*$, $\rho \sqsubseteq F_{env}$.
7. $\iota \sqsubseteq \hat{v}$ if **ground** $\in \hat{v}$.
8. $\langle c, x, n \rangle \sqsubseteq \langle c, x \rangle$.
9. $\langle o, f \rangle \sqsubseteq \langle \hat{o}, f \rangle$ if $o \sqsubseteq \hat{o}$.
10. $\texttt{Closure}\langle \lambda x \,.\, e, \rho \rangle \sqsubseteq \hat{v}$ if $\lambda x \,.\, e \in \hat{v}$ and $\rho \sqsubseteq F_{env}$.
11. $\texttt{Rec}\ \langle f, fn, \rho \rangle \sqsubseteq \hat{v}$ if $fn \in \hat{v}$ and $\rho \sqsubseteq F_{env}$.
12. $r \sqsubseteq \hat{r}$ if for every $x \in Dom(r), r(x) \sqsubseteq \hat{r}(x)$.

Fig. 5. Agreement relation.

For a specific program, there may be many flow functions which satisfy the constraints shown in Figure 6. The following theorem states that any flow satisfying the safety constraints conservatively approximates the exact semantics.

Theorem 1 (Safety $\Rightarrow$ Soundness). *If* $\langle TM, \sigma, LM \rangle \sqsubseteq \langle F_{env}, F_{cell}, F_{lock} \rangle$, $\langle F_{env}, F_{cell}, F_{lock} \rangle$ *is* safe *and* $\langle TM, \sigma, LM \rangle \rightarrow_{\top} \langle TM', \sigma', LM' \rangle$ *then* $\langle TM', \sigma', LM' \rangle \sqsubseteq \langle F_{env}, F_{cell}, F_{lock} \rangle$

Proof. By induction on exact transition sequences.

4.1 Multiplicity

To improve precision, the constraints incorporate a simple liveness property among methods and threads. Recall that the flow analysis computes for every instance an approximation to the set of exact threads that synchronize on

Definition 2. Let R be an indexed set, whose indices are drawn from the set of abstract threads. Each element in the set is itself a set of expressions *reachable* from the representative thread. Let $P = T^* \ C^* \ S^* \ exp$ where $exp \in R_{Main}$. Define for each $x \in P$, $\top \geq \mathcal{M}(x)$. Then, a flow analysis $\langle F_{env}, F_{cell}, F_{lock} \rangle$ is safe if the following conditions hold:

For all $exp \in R_{\hat{tid}}$
if $exp =$ `let` $(x = b)$ `in` exp' `end`, then $exp' \in R_{\hat{tid}}$ and
(a) if $b = \iota$, then $\{\mathbf{ground}\} \leq F_{env}(x)$ if $PrimType(c)$;
(b) if $b = \mu f.fn$ then $\{fn\} \leq F_{env}(f)$ and $\{fn\} \leq F_{env}(x)$;
(c) if $b = y.i$, then $\forall \hat{o} \in F_{env}(y), F_{cell}(\hat{o}).i \leq F_{env}(x)$;
(d) if $b = r.\underline{i}$ then $\forall \hat{r} \in F_{env}(r)$, $\hat{r}(i) \leq F_{env}(x)$
(e) if $b = r.\underline{m}$ then $\forall \hat{r} \in F_{env}(r)$, $\hat{r}(m) \leq F_{env}(x)$
(f) if $b = y \Leftarrow \ m$, then $\forall \hat{o} = \langle c, z \rangle \in F_{env}(y)$, $GetMethod(c, m) = \lambda self_i \,.\, f$,
1. $\{f\} \leq F_{env}(x)$
2. $F_{env}(y) \leq F_{env}(self_i)$;

(g) if $b = y \Leftarrow \ c.m$ and $GetMethod(c, m) = \lambda self_i \,.\, f$, then
1. $\{f\} \leq F_{env}(x)$
2. $F_{env}(y) \leq F_{env}(self_i)$;

(h) if $b = f(y_1, \ldots, y_n)$ then $\forall \lambda_c = \lambda w_1, \ldots, w_n \,.\, exp'' \in F_{env}(f)$,
1. for $i = 1, \ldots, n$, $F_{env}(y_i) \leq F_{env}(w_i)$
2. $F_{env}(last(exp'')) \leq F_{env}(x)$
3. $\mathcal{M}(\lambda_c) = \mathcal{M}(\lambda_c) \sqcap (\langle \lambda_c, x \rangle \bowtie \langle \lambda_r, \mathcal{M}(\lambda_r) \rangle)$ where λ_r encloses b.
4. $exp'' \in R_{\hat{tid}}$;

(i) if $b =$ `store` (y, z) then $\forall \hat{l} \in F_{env}(y), F_{env}(z) \leq F_{cell}(\hat{l})$;
(j) if $b =$ `fetch` (y) then $\forall \hat{l} \in F_{env}(y), F_{cell}(\hat{l}) \leq F_{env}(x)$
(k) if $b = y \rightarrow [k_i \Rightarrow exp_i]^*$ `else` exp_n then
1. for $i = 1, \ldots, n$, $exp_i \in R_{\hat{tid}}$
2. for $i = 1, \ldots, n$, $F_{env}(last(exp_i)) \leq F_{env}(x)$;

(l) if $b =$ `new` (c) then $\{\langle c, x \rangle\} \leq F_{env}(x)$;
(m) if $b =$ `spawn` (c, m) and $GetMethod(c, m) = \lambda self_i \,.\, f$, then
1. $\{\langle c, x \rangle\} \leq F_{env}(x)$
2. $\{\langle c, x \rangle\} \leq F_{env}(self_i)$
3. $exp'' \in R_{\langle c, x \rangle}$, where $f = \lambda z.exp''$
4. $\mathcal{M}(f) = \mathcal{M}(f) \sqcap (\langle f, x \rangle \bowtie \langle \lambda_r, \mathcal{M}(\lambda_r) \rangle)$ where λ_r encloses b;

(n) if $b =$ `synch` (y) then $\forall \hat{o} \in F_{env}(y)$, $\hat{tid} \leq F_{lock}(\hat{o})$.

Fig. 6. Safety constraints on flows.

that instance. Because an abstract thread is represented by a birthplace, this information is insufficient to distinguish between one or multiple invocations of a thread created at a given `spawn` point. Consequently, the effectiveness of optimizations such as synchronization elimination, that rely crucially on knowing if there exists a *unique* thread accessing an instance, would be weakened.

To alleviate this problem, we fold a simple liveness calculation into the flow analysis to reveal multiplicity properties of threads and methods. Given a method m, we compute a conservative approximation of the number of times

m is invoked. To do so, we first define a *multiplicity* map that associates with every method an element in a flat lattice:

$$\mathcal{M} : \mathit{LambdaExp} \rightarrow M + \mathit{CallSite} + \top$$

where $M \leq \mathit{CallSite} \leq \top$, and $\mathit{CallSite}$ ranges over variables bound to call expressions in the program. The bottom element of the lattice is M, indicating the method is invoked multiple times. Initially, $\mathcal{M}$ maps all methods to $\top$, indicating that there are no calls to any method. The other elements of the lattice are call-sites; if a method has at least two different call sites, then it is considered multiple: $p \sqcap q = M$ if $p \neq q$.

However, a method can be multiple even if it has a single call site, if that site is located in a method which is itself *multiple*. Thus, in stating the safety rules for our flow analysis, we must take into account the multiplicity of the caller as well as that of the callee. For this, we define a join operation, ($\bowtie$), on tuples of methods and multiplicities:

$$(\langle m_1, p\rangle \bowtie \langle m_2, q\rangle) = \begin{cases} p & \text{if } m_1 \neq m_2 \text{ and } q \neq M \\ M & \text{otherwise} \end{cases}$$

Then, the multiplicity of a method m_1 called at site p in method m_2 with multiplicity q, is obtained from joining $\langle m_1, p\rangle$ with $\langle m_2, q\rangle$, and then doing a meet operation with the previous value of m_1's multiplicity.

4.2 Constraints

We specify the analysis as a collection of constraints (see Figure 6). We regard these constraints as *safety* rules that specify the structure of abstract value sets. These constraints must ensure that the abstract values computed by an implementation of the analysis can be interpreted as a conservative approximation of a program's exact behavior. The rules specify a monovariant flow analysis akin to 0CFA [Shi91] for expressions.

The rules refer to several auxiliary functions. The function *getMethod*, when given a class name and a method name, returns the definition of the method found in the class; and, *last* returns the variable bound to the last sub-term of an expression. The rules are sensitive to *reachable* expressions. Informally, an expression is reachable if its abstract value set is not empty. To express reachability in the presence of threads, we define an index set of reachable expressions, with each element in the set corresponding to the set of expressions reachable from a particular thread. We express reachability in the rules in several places. Obviously, the main expression of a program is reachable. Second, if a `let` expression is reachable, then its body is also reachable. Third, when a new thread is spawned, the body of the thread is reachable, *i.e.*, as in `spawn` (c, m) the body of m is reachable. Finally, the body of a method is reachable if there is an application of the method that is reachable (i.e., the λ-expression corresponding to the method occurs as an abstract value element in the flow).

```
class Container {
  static Container container;
  Vector leftV;
  Vector rightV;
  Container() {
    leftV = new Vector();  // p3
    rightV = new Vector(); // p4
  }

  static void main() {
    container = new Container(); // p0
    CThread leftT =  new CThread( container.leftV); // p1
    CThread rightT  = new CThread( container.rightV); // p2
    leftT.start();
    rightT.start();
  }
}
```

```
class CThread extends Thread {
  Vector vec;
  CThread( Vector v) { vec = v;  }
  public void run() {
    vec.addElement( new Integer(1));
  }
}
```

F_{env}

container $\mapsto \langle$ Container $, p0\rangle$
leftV $\mapsto \langle$ Vector $, p3\rangle$
rightV $\mapsto \langle$ Vector $, p4\rangle$
leftT $\mapsto \langle$ CThread $, p1\rangle$
rightT $\mapsto \langle$ CThread $, p2\rangle$

F_{cell}

Container . container $\mapsto \langle$ Container $, p0\rangle$
$\langle$ Container $, p0\rangle$. leftV $\mapsto \langle$ Vector $, p3\rangle$
$\langle$ Container $, p0\rangle$. rightV $\mapsto \langle$ Vector $, p4\rangle$
$\langle$ CThread $, p1\rangle$. vec $\mapsto \langle$ Vector $, p3\rangle$
$\langle$ CThread $, p2\rangle$. vec $\mapsto \langle$ Vector $, p4\rangle$

F_{lock}

$\langle$ Vector $, p3\rangle \mapsto \langle$ CThread $, p1\rangle$
$\langle$ Vector $, p4\rangle \mapsto \langle$ CThread $, p2\rangle$
$\langle$ CThread $, p1\rangle \mapsto Main$
$\langle$ CThread $, p2\rangle \mapsto Main$

$\mathcal{M}$

$\langle$ CThread $, p1\rangle \mapsto p1$ (single)
$\langle$ CThread $, p2\rangle \mapsto p2$ (single)
$Main$ (single)

Fig. 7. A simple multithreaded program and the result of its flow analysis.

Of particular interest are the rules for application and thread creation. Condition (3) in the rule for function application reflects the multiplicity constraints described in the previous section. The multiplicity of a function at a call point is computed as $\mathcal{M}(\lambda_c) \sqcap (\langle \lambda_c, x\rangle \bowtie \langle \lambda_r, \mathcal{M}(\lambda_r)\rangle)$.

The $\bowtie$ operator returns a multiplicity based on the multiplicities of the caller (λ_r) and of the callee (λ_c). The callee is marked M if it has been applied at some other call-site, or if the caller is itself M. The rule for thread creation also uses the multiplicity map to record the multiplicity of the function being spawned.

4.3 Example

Figure 7 presents an example to illustrate the analysis. Objects of type `Container` have two fields, `left` and `right`, each of type `Vector`. The main method allocates a container and stores it in a static field, and then starts two threads each of which operates on only one of the fields. Even though both vectors are reachable from a (global) static field, and also from the thread instances themselves, neither vector is accessed by multiple threads. The bottom of Figure 7 shows the relevant maps produced by the analysis. The program points $p0$ to $p4$ are the birthplaces of the respective instances. Note that because of the `start` method, both thread objects must be `synchronized` in the main method. The results obtained by the analysis enable the removal of *all* synchronization operations on `rightV` and `leftV` in the program, since no object is locked by two different threads.

5 Flow-Sensitive Extensions

The analysis presented in the previous section tells us the objects for which contention among various threads in the program is possible. However, even objects manipulated by multiple threads are not always shared during their entire lifetime. An analysis that tracks sharing properties of threads and instances at different program points can lead to further optimization opportunities.

Conceptually, the refined analysis builds for each program point an approximation of the heap *that is relevant at that program point*. For efficiency and precision, it is important that we do not build an approximation of the entire heap at every program point. For example, the interesting portion of the heap to a leaf procedure that does not have any `synchronized` operations, and does not refer to any static field, is limited to its arguments and what is reachable from them. Since only a few operations affect the heap, abstract heaps can often be shared among many program points.

For every heap at every program point, there is a set of roots that is considered *shared*; any other node that is reachable in the heap from these roots is potentially *shared* as well. The root set must include: (1) any thread object (because it is accessible to both the new thread of control and the thread that created it); and (2) static fields whose contents consists of objects touched by more than one thread.

5.1 Computing a Reachability Map

Prior to computing the structure of the per program-point heap, we must first compute a transitive closure of the static fields accessed by a method. To compute the closure we need:

1. some approximation of the control flow graph. In our case, this approximation is given by the earlier analysis;
2. the set of static fields accessed by each method

From (1) and (2) we compute the transitive closure of the static field accesses for a method and all the methods it calls.

$$\begin{array}{ll}
FreeVarMap: & Method \rightarrow \mathcal{P}(Aval) \\
FreeVarMap(\lambda self.\lambda x.exp) & = FreeVarMap(exp)
\end{array}$$

$$\begin{array}{ll}
FreeVarMap: & Exp \rightarrow \mathcal{P}(Aval) \\
FreeVarMap(x) & = \{\} \\
FreeVarMap(\texttt{let}\ (x = b)\ \texttt{in}\ exp'\ \texttt{end}) & = FreeVarMap(exp') \bigcup
\end{array}$$

$$\begin{cases} \langle r, \underline{i} \rangle & \text{if } b \equiv r.\underline{i} \\ \bigcup_j FreeVarMap(\lambda_j) & \text{if } b \equiv f(y_1, \ldots, y_n), \text{ where } \lambda_j \in F_{env}(f) \\ \{\ \} & \text{otherwise} \end{cases}$$

5.2 Abstract Domains

The refined analysis computes an *abstract heap* at every program point. A heap is a map from nodes to sets of nodes. For our purposes, a node is represented as an abstract value. Given node n, $AH(n)$ denotes the set of nodes immediately reachable from n.

$$\begin{aligned}
AH \in AHeap &= Node \rightarrow \mathcal{P}(Node) \\
n \in Node &= Aval \\
e \in Edge &= Node \times Node
\end{aligned}$$

The function *Reach* computes the part of the abstract heap that is reachable starting from the roots (a set of abstract values) given as arguments:

$$Reach : AHeap \times \mathcal{P}(Aval) \rightarrow AHeap$$

It is used to propagate heaps from caller to callee at method call points, and from callee to caller at method return points.

Single is a function from abstract cells to Booleans: $Single : \mathcal{P}(Aloc) \rightarrow Boolean$. *Single* returns true if a *strong update* is possible on the argument. Strong update is a property of an analysis that allows the contents of an abstract cell to be overwritten if it is known that the cell corresponds to one exact cell. Thus, the function returns true in the following cases:

- the cell corresponds to a local variable of a function called only once as defined by the function's multiplicity.
- the cell corresponds to a *clean* static field (*i.e.*, a static field which contains objects touched by only one thread).
- the cell corresponds to an instance field, and the instance is clean and single (*i.e.*, the instance is allocated in a function called only once)

Figure 8 contains the rules to compute the abstract heaps. Since most of the expressions do not change the heap, the heap at the program point succeeding the expression is the same as that of the preceding program point. The notable exceptions are instance allocation, cell update, control flow join points and function calls. To build new heaps, the analysis uses two functions, *Node* and *Edge* to inject nodes (respectively edges) into heaps, and a join operation, $\oplus$ to merge two heaps. Their definitions are obvious, and are omitted here.

Definition 3.
For all $exp \in R_{\hat{tid}}$

1. if $exp =$ `let` $(x = b)$ `in` exp' `end`, then $exp' \in R_{\hat{tid}}$ and
   (a) if $b = f(y_1, \ldots, y_n)$ then $\forall \lambda_j = \lambda w_1, \ldots, w_n \,.\, exp_j \in F_{env}(f)$, let
   - $F = FreeVarMap(\lambda_j)$
   - $S = \{\hat{o} \in F_{env}(y_i) | \hat{o} \in AH_{exp}\}$.

   Then,
   i. $AH_{exp_j} = Reach(AH_{exp}, F \cup S)$
   ii. $AH_{exp'} = \oplus_j \; Reach(AH_{last(exp_j)}, F \cup S \cup \{F_{env}(last(exp_j))\})$
   iii. $exp_j \in R_{\hat{tid}}$

   (b) if $b =$ `new` (c), then $AH_{exp'} = AH_{exp} \oplus Node(\langle c, x\rangle)$;
   (c) if $b =$ `store` (y, z) then $\forall \hat{l} \in F_{env}(y)|_{AH_{exp}}$,
   let $S = \{Edge(\hat{l}, \hat{o}) \mid \hat{o} \in F_{env}(z)|_{AH_{exp}}\}$; then
   i. if $Single(\hat{l})$, $AH_{exp'} = (AH_{exp} - D) \oplus S$,
   where $D = \{e = Edge(\hat{l}, \hat{v}) | e \in AH_{exp}\}$
   ii. otherwise, $AH_{exp'} = AH_{exp} \oplus S$.

   (d) if $b = y \rightarrow [k_i \Rightarrow exp_i]^*$ `else` exp_n then
   i. for $i = 1, \ldots, n$, $AH_{exp_i} = AH_{exp}$
   ii. $AH_{exp'} = \oplus_i \; AH_{last(exp_i)}$
   iii. $exp_i \in R_{\hat{tid}}$

   (e) if $b =$ `spawn` (c, m), and $GetMethod(\langle c, m\rangle) = \lambda self \,.\, \lambda() \,.\, exp''$
   i. $AH_{exp'} = AH_{exp''} = AH_{exp} \oplus Node(\langle c, x\rangle)$,
   ii. $exp'' \in R_{\langle c, x\rangle}$

   (f) $AH_{exp'} = AH_{exp}$ otherwise

Fig. 8. Safety constraints on abstract heaps.

5.3 Filtering Environments

This analysis uses the results of monovariant analysis presented in the previous section which provides safe flow and cell environments. Because the results of the flow analysis are valid at any program point (due to the safety rules), there are cases when values are added to the environment because of merging induced by the constraint satisfaction algorithm [JW95]. Since the second analysis computes a solution per program point, we could compute a tighter approximation as follows. Initially, the starting heap contains only the (empty) cells corresponding to the static fields. Whenever an allocation expression is encountered, the abstract values corresponding to the new object is added as a new node in the heap at that program point. Now, when we use the flow environment of some variable, we can restrict the set of values to those that actually exist in the heap at that program point. The result is an analysis whose implementation has slower convergence, but improved precision.

The filtering of the environment explains why the rules for allocation of instances (`new` and `spawn`) introduce new nodes. Edges are added by the `store` expression which augments the contents of a node with a new abstract value. As mentioned above, some of these updates can be *strong* if the cell that is updated is *single*.

5.4 Thread Local Objects

The results of the flow analysis can be used to eliminate runtime overhead incurred by unnecessary synchronization in two ways. First, we can eliminate any lock operations performed on objects that are locked by a *single* abstract thread. Second, we can eliminate synchronization on *local objects*. A local object is any non-thread object that is accessible only from stack or from other local objects. Thread instances and objects that are stored in static fields are *non-local* (or *global*). Because an abstract thread that is multiple might correspond to many exact threads, these two optimizations are complementary.

Local objects are obtained from a data dependency graph built from the flow and cell environment as follows; any distinct abstract value used during the analysis is represented by a node in the graph. Edges are introduced between: (i) an abstract instance and its abstract instance fields, (ii) an abstract cell `cell` and the abstract values in $F_{cell}($ `cell` $)$. Starting from static fields and thread instances, we compute the transitive closure of global objects. Any remain nodes represent *local objects*. Note that we can have data-paths of arbitrary length composed entirely of local objects. This gives us a first-order approximation of the set of local objects, an approximation improved upon by the second analysis. By refining the construction of abstract heaps on a per program point basis, we obtain heaps with fewer edges, thus decreasing the number of nodes that become *global*.

5.5 Abstract Heaps Example

Figure 9 contains an example for our second analysis: a tree whose nodes are implemented by vectors of successor nodes. The class `Node` defines the data type used for implementing the nodes of the tree; the class `Main` contains code to build the tree, store its root in a static field, and then start a thread to read the values stored in the nodes. Since the nodes of the tree are eventually accessed by more than one thread (in this case the main thread and an instance of `TreeThread`), our earlier analysis would be unable to reveal any opportunities for optimization. However, note that during the construction of the tree, its nodes are only accessible from one thread. By using a per program-point heap, this information is revealed. Figure 10 presents the results of the analysis at the program points shown inside comments. For the purposes of synchronization elimination, synchronization overhead can be eliminated at program point `p4`, (just before the call to `addElement`), and for the call to `addChild` at program point `p5`.

```
class Node {
    Vector successors ;
    Node() { successors = new Vector(); }
    synchronized void addChild( Node child) {
        /* p4 */  successors.addElement( child);
    }
    synchronized Enumeration getChildren( Node n) { ... }
}
class Main {
    static Node root;
    Node buildTree( int max) {
        Node n = new Node();
        if( max > 0 )
            while( randomBool()) {
                          Node t = buildTree( max - 1 );
                /* p5 */  n.addChild( t);  }
        return n;
    }
    static void main() {
        /* p1 */  root = buildTree( 5);
        /* p2 */  TreeThread thread = new TreeThread();
                  thread.start();
        /* p3 */  // ... more code here ...
    }
}
```

Fig. 9. Tree example

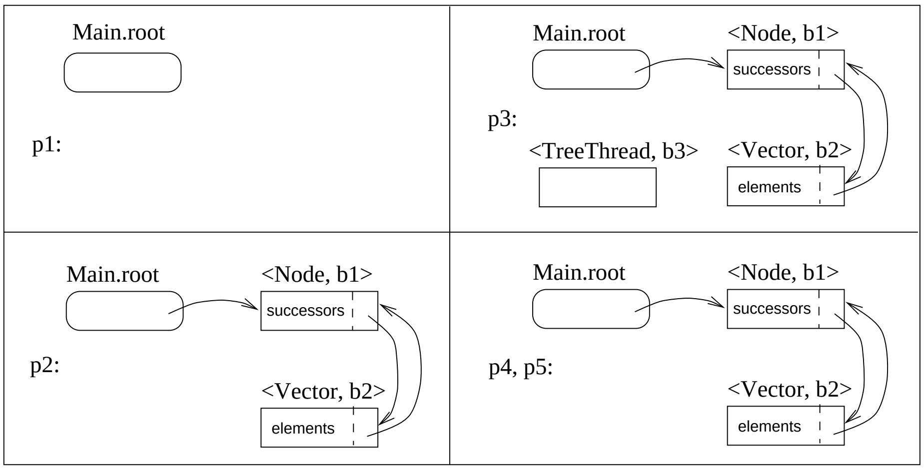

Fig. 10. Abstract heaps

6 Experiments

We have applied our analysis on the applications presented in Table 11. `JLex` and `JCup` are a lexer and parser generator; `Zothello` is a program playing the game Othello; `Dhry` is a numeric benchmark. The other two programs are multithreaded: `_224_richards` creates a number of threads running multiple versions of OS simulator, and `_233_tmix` is a mix of applications (fibonacci, sort, producer-consumer, etc.) each run in its own thread.

Since synchronization costs depend greatly on machine hardware, lock allocation strategy used, etc., measuring interesting dynamic counts presents a more objective view of the analyses' utility. Table 12 shows the dynamic counts for the number of instances allocated in both the program and libraries, the number of locks acquired, and the total number of calls made. To quantify the utility of our analyses, we present results focused on synchronization overhead; these results are easily extrapolated to other kinds of concurrency optimizations discussed in the earlier sections. Table 13 gives percentages based on the baseline numbers for the total number of allocations that were marked local by our analyses, the total number of locks acquired on objects deemed local by our analyses, and the total number of lock operations on objects (either local or global) that were synchronized by only one thread. Synchronization overhead on local objects or synchronization on an object accessed by only a single thread can be eliminated.

Note that our thread analysis reveals that no synchronization overhead must be incurred by any of the sequential programs; however, as an ancillary benefit, flow-sensitive heap analysis can also be used to reveal useful information about object locality and lifetime. For example, an overwhelming majority of locks can be removed from `jcup`, `zothello`, and `dhry` simply because the objects on which these operations are performed have been identified as local. However, using just reachability information leads to less convincing results for `jlex` because many instances are referenced via static fields. For this program, it was critical that we analyzed the structure of the global lock map computed by the thread analysis. This map indicates that no object is shared by multiple threads, even though many of these objects are accessible via static fields.

Whereas in the first four applications our initial monovariant thread analysis gave perfect information, it alone wasn't precise enough on the multi-threaded benchmarks. In `_224_richards` it found that only 5% of the lock operations are

Program	Description	Number of methods	size of classes
`jlex`	Java lexer generator	136	101K
`jcup`	Java parser generator	362	142K
`zothello`	Othello game program	132	36K
`dhry`	Dhrystone benchmark	22	6.5K
`_224_richards`	threads running OS simulator	394	143K
`_233_tmix`	thread mix	140	53K

Fig. 11. Benchmark description

Program	Allocations		Number of locks		Total number of calls
	application	libraries	application	libraries	
`jlex`	23231	20094	1379642	1141876	8263281
`jcup`	26225	80414	34013	156831	1142196
`zothello`	83782	5179	14462	3350	32389270
`dhry`	600141	227	0	145	5405595
`_224_richards`	2317	5077	66	4156	63604540
`_233_tmix`	94825	245585	708917	841336	8699788

Fig. 12. Dynamic counts

Program	Local Allocations		Synchronizations locks on local objects		Synchronizations on single threaded objects	% Locks unoptimized
	app.	app. + lib.	app.	app + lib.		
`jlex`	47%	36%	13%	39%	100%	0%
`jcup`	55%	67%	78%	93%	100%	0%
`zothello`	89%	86%	100%	99%	100%	0%
`dhry`	99%	99%	–	80%	100%	0%
`_224_richards`	20%	26%	53%	83%	5%	17%
`_233_tmix`	45%	84%	85%	93%	40%	7%

Fig. 13. Percentages of local objects and locks removed

performed on objects locked by only one thread. This was caused by the fact that the program spawns a number of identical threads from the same program point; consequently, all thread instances with the exception of $Main$ were considered *multiple*. On the other hand, in `_233_tmix` all threads are *single*, and 40% of all synchronizations are on objects locked by only a single thread. A large percentage of the remaining synchronizations are on `StringBuffer` objects used in string concatenation and on the queue of objects used in the producer-consumer component of the benchmark. In the former case, because the `concat` method is used in more than one thread, the string buffer allocated in it is considered locked on by all these threads. In the latter case, the queue of objects is indeed shared by two threads; for correctness, synchronization overhead cannot be removed here. For both these programs, the precision of the analysis was greatly enhanced by employing our flow-sensitive heap analysis, which discovered that many of the objects synchronized on were local, thus enabling the optimization of 83% (respectively 93%) of lock operations.

The compilation overhead incurred by our analysis has two components. The first component is constraint generation, and is about 3% of the compilation time; the constraints that are generated can be reused as long as the source code of the respective class does not change (such is the case of libraries, for example). The second component is constraint solving; for the six benchmarks presented, it took between 8% and 27% of the compilation time of the application only, with an average of 19%. These percentages are likely to go down as we fine-tune the implementation, and as more optimizations are added to the compiler.

7 Related Work

There have been many algorithms proposed for analyzing object-oriented programs with varying precision and costs [PS91,PC94,Age95,DGC98]. Many of these analyses are variants or extensions of monovariant control-flow analyses [Hei94,JM79,Shi91]. Extensions of control-flow analyses to handle concurrency have been explored in [CI92,Mer91,JW94]. While these efforts have influenced our design, there are numerous technical differences in the development. Most notably, none of these efforts consider analysis of monitor-style concurrency in an object-oriented language, nor justify the practicality of their approach with a realistic implementation.

There is a large body of work on analyzing reachability and computing shapes over an abstract representation of a program's memory; none of these efforts have considered the applicability of such analyses for optimizing concurrent programs. Ruggieri and Murtagh [RM88] compute sets of objects passed to and returned from procedures for lifetime analysis to enable stack allocation of objects on frames with greater lifetimes. Serrano and Feeley [SF96] present a similar idea for a higher-order language. Deutsch [Deu90] uses abstract reachability to enable stack allocation and to determine when objects can be destructively updated. Chase et al. [CWZ90] use a per-program point "storage shape graph" to analyze a language with mutable cons cells. They introduce the notion of "strong update" in this context. Sagiv, Reps, and Wilhelm [SRW96] present an abstract interpretation of a simple imperative language with heap-allocated data structures. Like [CWZ90], they compute a per-program point static storage graph that approximates the heaps at each point. Rather than using a notion of birthplace (or allocation point) to partition exact cells into abstract cells, their analysis keeps track of cells immediately pointed to by program variables. Abstract heap nodes are represented by sets of variables that must all point to the same runtime location, for some execution reaching that point.

More recently, there has been increased interest in exploring optimization opportunities for Java, especially in the context of its concurrency features. Bogda and Holzle developed an analysis which identifies unnecessary synchronizations [BH99] based on the observation that objects that are only reachable from the stack can not possibly be accessed by two different threads. They employ a flow-insensitive analysis that separates objects which can be stack allocated from those that cannot; stack allocated objects require no synchronization to access. Choi et al. [CGS+99] describe an escape analysis for Java that can be used to facilitate synchronization elimination. They introduce an abstraction, called *connection graph*, used to establish reachability constraints among objects and object references. Their analysis takes into account thread objects, but not threads of control. If the connection graph reveals that an object does not escape, its methods require no synchronization. Like [CGS+99], Whaley and

Rinard [WR99] present a pointer and escape analysis for Java using an abstraction of a points-to graph that describes how objects refer to one another. Blanchet [Bla99] describes another escape analysis for Java using a type-based abstract interpretation rather than a dataflow graph framework. Aldrich et al. [ACSE99] describe a static analysis for eliminating unnecessary synchronization in Java programs, which discovers locking relationships between instances. For example, no synchronization is necessary on a method that is always invoked at call points where the corresponding receiver object is already locked.

Our framework is distinguished from these efforts insofar as it deals explicitly with threads. In contrast to analyses that indirectly compute sharing properties of objects via their lifetime (escape) properties, our analysis defines an approximation in terms of how threads access objects. In particular, it derives perfect information about synchronization requirements for single-threaded programs. Our second analysis provides a flow-sensitive characterization of object reachability, and, like [WR99], supports strong-updates on assignment to global data. Using a per program-point heap abstraction in conjunction with a thread-aware analysis leads to a refined approximation useful in optimizing a program's heap and stack layout. Our refined analysis may still find a number of program points where an object is local, even though that object may eventually escape into the heap. By tracking object reachability through threads, our analyses provide a more refined approximation of the concurrency properties of a program.

References

[ACSE99] Jonathan Aldrich, Craig Chambers, Emin Sirer, and Susan Eggers. Static analysis for eliminating unnecessary synchronization in Java programs. In *International Static Analysis Symposium*, pages 19–38, September 1999.

[Age95] Ole Agesen. The cartesian product algorithm: simple and precise type inference of parametric polymorphism. *ECOOP'95 Conference Proceedings*, pages 2–26, 1995.

[BH99] Jeff Bogda and Urs Holzle. Removing unnecessary synchronization in Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 35–47, November 1999.

[BKMS98] David Bacon, Ravi Konru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 258–268, June 1998.

[Bla99] Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 20–35, November 1999.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, November 1999.

[CI92] Jyh-Herng Chow and Williams Ludwell Harrison III. Compile time analysis of parallel programs that share memory. In *19th ACM Symposium on Principles of Programming Languages*, January 1992.

[CWZ90] David. R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *ACM Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[Deu90] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *17^{th} ACM Symposium on Principles of Programming Languages*, pages 157–168, January 1990.

[DGC98] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *25^{th} ACM Symposium on Principles of Programming Languages*, pages 222–236, January 1998.

[Hei94] Nevin Heintze. Set-based analysis of ML programs. In *ACM International Conference on Lisp and Functional Programming*, pages 306–317, June 1994.

[Jav98] *ACM Workshop on Java for High-Performance Network Computing*, 1998.

[Jav99] *Java Grande 99*. ACM, June 1999.

[JM79] Neil Jones and Stephen Muchnick. Flow analysis and optimization of Lisp-like structures. In *6^{th} ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.

[JW94] Suresh Jagannathan and Stephen T. Weeks. Analyzing stores and references in a parallel symbolic language. In *ACM International Conference on Lisp and Functional Programming*, pages 294–305, 1994.

[JW95] Suresh Jagannathan and Stephen T. Weeks. A unified treatment of flow analysis in higher-order languages. In *22^{nd} ACM Symposium on Principles of Programming Languages*, pages 393–407, January 1995.

[Mer91] N. Mercouroff. An algorithm for analyzing communicating processes. In *Mathematical Foundations of Programming Semantics*. Springer-Verlag, 1991.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David B. Macqueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[PC94] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. *Conference on Object-Oriented Programming Systems, Language, and Applications*, pages 324–340, October 1994.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, October 1991.

[RM88] Cristina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *ACM Symposium on Principles of Programming Languages*, pages 285–293, January 1988.

[SF96] Manuel Serrano and Marc Feeley. Storage use analysis and its applications. In *International Conference on Functional Programming*, May 1996.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie-Mellon University, 1991.

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1996.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–207, November 1999.