Thread-Sensitive Points-to Analysis for Multithreaded Java Programs

Byeong-Mo Chang[1⋆] and Jong-Deok Choi[2]

[1] Dept. of Computer Science,
Sookmyung Women's University, Seoul 140-742, Korea
chang@sookmyung.ac.kr

[2] IBM T. J. Watson Research Center
P.O. Box 704 Yorktown Heights, NY 10598 USA
jdchoi@watson.ibm.com

Abstract. Every running thread has its own thread context that consists of values of the fields of the target thread object. To consider the thread context in understanding the behaviors of concurrently running threads, we propose a thread-sensitive interprocedural analysis for multithreaded Java applications. Our thread-sensitive analysis exploits thread-context information, instead of the conventional calling-context information, for computing dataflow facts holding at a statement. The thread-sensitive analysis is highly effective in distinguishing dataflow facts for different threads, producing more precise dataflow information than non-thread-sensitive analysis. The analysis is also generally much more efficient than conventional (calling) context-sensitive analysis. It uses the target thread objects at a thread start site to distinguish different thread contexts. We give a thread-sensitive points-to analysis as an instance of thread-sensitive analysis. We have implemented it and give some experimental results. We discuss several possible applications of the analysis.

1 Introduction

Multithreading in Java has become widely used in developing concurrent and reactive software. One of the most important analyses for Java is points-to analysis, which provides information about the objects, to which references point. Potential applications of points-to analysis for multithreaded programs include synchronization elimination, alias analysis, escape analysis, static datarace detection, software engineering tools and compiler optimizations [2–4, 12, 9]. Several points-to analyses was proposed for Java [8, 13]. However, they treat threads just like methods without paying special attention to threads, and don't provide experimental results for multithreaded Java programs.

In multithreaded programs, there are multiple concurrently running threads for one thread definition. Every running thread has its own thread context that

⋆ This Research was supported by the Sookmyung Women's University Research Grants 2004

consists of values of the fields of the target thread object. Moreover, it has its own instances for locals, newly created objects, and method invocations.

In this paper, we propose a thread-sensitive interprocedural analysis for multithreaded Java programs to consider the thread context in understanding the behaviors of concurrently running threads. Our thread-sensitive analysis exploits thread-context information, instead of the conventional calling-context information, for computing dataflow facts holding at a statement. It is highly effective in distinguishing dataflow facts for different threads, producing more precise dataflow information than non-thread-sensitive analysis. It is also generally much more efficient than conventional (calling) context-sensitive analysis. It uses the target thread objects at a thread start site to distinguish different thread contexts. We give a thread-sensitive points-to analysis as an instance of it.

For thread-sensitive points-to analysis, we first identify thread objects statically by making one (abstract) thread object for each thread object creation site. Then we do separate points-to analyses for possible target thread objects at each thread start site. Conceptually, every thread is replicated for each possible target thread object. A thread-sensitive analysis for each target thread object starts from its `run()` method and analyzes all methods reachable from it. For clear presentation, we present the thread-sensitive analysis based on constraint-based analysis [5]. While context-insensitive analysis makes one set-variable for every reference variable, the thread-sensitive analysis makes as many set-variables as the number of its target thread objects for every reference variable (not static) in a thread definition to model that every thread has its own instance. We construct a separate collection of set-constraints for each target thread.

We have implemented the thread-sensitive points-to analysis based on conventional iterative algorithm by extending the inclusion-based points-to analysis [13]. We first do context-insensitive points-to analysis of the main thread to extract abstract thread objects, and then identify the `run()` method of each abstract thread object. Then we do separate points-to analysis specific to each target thread. We have evaluated the impact of the analysis over nine benchmark programs along with the work in [13]. We also observed that codes for `run()` methods have to be connected with its start site in the analysis, because threads are started by native code. Otherwise, it will be missing as in [13] when analyzing multithreaded Java programs. We have implemented the functionality to connect and analyze the `run()` methods of started threads automatically.

The rest of the paper is organized as follow. We first give a motivation in Section 2. We present the thread-sensitive analysis in Section 3 and its implementation and experiments in Section 4. Section 5 discusses some applications of it such as synchronization removal, datarace detection, and software engineering tools. Section 6 discusses related works and Section 7 concludes this paper.

2 Motivation

We consider a multithreaded Java program in Figure 1 from the book [7]. This program divides a picture to be rendered into two parts, `leftHalf` and `rightHalf`, and then creates two threads, `leftThread` and `rightThread`, to render left and right parts respectively. After rendering, it finally combines the two images rendered by the two threads.

```
class RenderWaiter extends Thread {
  private PictureRenderer ren;   // service object
  private byte [] arg;           // arguments to its method
  private Picture rslt = null;   // results from its method
  RenderWaiter(PictureRenderer r, byte[] raw) {
    ren = r; arg = raw;
  }
  synchronized Picture result() { return rslt; }
  public void run() {
    rslt = ren.render(arg);
  }
}

class DumbPictureRenderer implements PictureRenderer {
  public Picture render(byte[] raw) {
    return new Picture(new String(raw, raw.length));
  }
}

public class SplitRenderer implements PictureRenderer {
  PictureRenderer renderer1;       //  group member 1
  PictureRenderer renderer2;       //  group member 2
  public SplitRenderer() {
    renderer1 = new DumbPictureRenderer();
    renderer2 = new DumbPictureRenderer();
  }
  public Picture render(byte[] rawPicture) {
    byte[] rawLeft = leftHalf(rawPicture);   // split
    byte[] rawRight = rightHalf(rawPicture);
    RenderWaiter leftThread = new RenderWaiter(renderer1, rawLeft);
    RenderWaiter rightThread = new RenderWaiter(renderer2, rawRight);
    leftThread.start();      // start threads
    rightThread.start();
          ...    // join both of them
    Picture leftImg = leftThread.result();   // use results
    Picture rightImg = rightThread.result();
    return combinePictures(leftImg, rightImg);
  }
  byte[] leftHalf(byte[] arr) { ... }
  byte[] rightHalf(byte[] arr) { ... }
  Picture combinePictures(Picture a, Picture b) {
    return new Picture(a.image() + b.image());
  }
}
```

Fig. 1 Example program

Conventional context-insensitive points-to analyses collect all the objects passed to a parameter, into one set-variable [8, 13]. Because they also collect target thread objects in the same way when analyzing `run` method, they cannot distinguish different thread contexts at a thread start site. However, each thread can have different thread contexts and can behave differently depending on its target thread object. For example, the method `run` in `RenderWaiter` accepts the two thread objects, `leftThread` and `rightThread`, passed to `this` parameter. So, in the context-insensitive analysis, the field `this.ren` points to the objects pointed by `renderer1` and `renderer2`, and the field `this.arg` points to the objects pointed by `rawLeft` and `rawRight`. So, it cannot provide separate points-to analysis information specific to each target thread, because it does neither pay special attention to threads and nor distinguish different thread contexts.

In this paper, our thread-sensitive analysis will analyze `RenderWaiter` twice with `leftThread` and `rightThread` as its target thread object, respectively. The analysis now can provide separate analysis information specific to each thread. For example, we can determine from the analysis information that `this.ren` in each thread actually points to *only one* object. This information can be useful for such thread-related applications as synchronization removal or static datarace detection.

In this paper, the set of all reference variables in a program is denoted by Ref. The set of all abstract objects in a program is denoted by $AbsObj$. Each abstract object $O \in AbsObj$ is a mapping $O : Field \rightarrow \wp(AbsObj)$ where $Field$ is the set of fields of the object O. We denoted by O_{c_ℓ} an abstract object of a class c created at a label l. For simple presentation, we only discuss the statements: (1)Direct assignment: `p = q` (2) Instance field write: `p.f = q` (3) Static field write: `c.f = q` (4) Instance field read: `p = q.f` (5) Static field read: `p = c.f` (6) Object creation: `p = new c` (7)Virtual invocation: $\mathtt{p} = \mathtt{a_0.m(a_1, ..., a_k)}$.

Context-insensitive points-to analysis makes one set-variable $\mathcal{X}_v$ for every reference variable $v \in Ref$. A set variable $\mathcal{X}_v$ is for objects, which the reference v points-to. A set expression se is an expression to denote a set of objects, which is of this form:

$$se ::= O_{c_\ell}(\text{new object})|\ \mathcal{X}_v(\text{set variable})|\ se \cup se(\text{set union})|\ se \cdot f(\text{object field})$$

where a set-expression $se \cdot f$ represents the set of objects pointed by the field f of the objects represented by se.

A set-constraint is of this form: $\mathcal{X}_v \supseteq se$ or $\mathcal{X}_v \supseteq_f se$. The meaning of a set constraint $\mathcal{X} \supseteq se$ is intuitive: the set $\mathcal{X}_v$ contains the set of objects represented by the set expression se. A set constraint $\mathcal{X}_v \supseteq_f se$ means that the field f of the objects in the set $\mathcal{X}_v$ contains the set of objects represented by se. Multiple constraints are conjunctions. We write $\mathcal{C}$ for such conjunctive set of constraints. The semantics of the context-insensitive points-to analysis can be found in [8].

3 Thread-Sensitive Points-to Analysis

We describe a thread-sensitive interprocedural analysis for multithreaded Java applications in terms of points-to analysis. Our thread-sensitive analysis exploits thread-context information, instead of the conventional calling-context information, to distinguish dataflow facts for different threads. It uses the target thread objects at a thread start site to distinguish different thread contexts.

We define a thread-sensitive points-to analysis for multithreaded Java programs based on constraint-based analysis framework [5]. To distinguish different thread contexts, we analyze each thread definition separately for each target thread object, on which it may be started. Conceptually, every thread can be thought to be replicated for each possible target thread object.

We denote a thread definition (i.e. thread defining class) by T. We make its abstract thread objects for every thread creation site for T. We denote them by $O_{T_1}, ..., O_{T_n}$. An abstract thread represented by $T_\imath$ is a replica of the thread definition with a possible thread object $O_{T_\imath}$ as its target object. It actually includes the `run` method started with $O_{T_\imath}$ as its target object and all methods that can be reachable from it.

Our thread-sensitive points-to analysis consists of three steps:
(1) *Identification of abstract thread objects*: We first identify abstract thread objects by examining thread creation sites. We can identify a number of abstract thread objects $O_{T_1}, ..., O_{T_n}$ for a thread defining class T.
(2) *Set-constraint construction*: We construct separate set-constraints for each abstract thread $T_\imath$ with its target thread object O_{T_i}.
(3) *Solving the set-constraints*: Solve all the set-constraints constructed in the second step.

While conventional context-insensitive analysis makes one set-variable for every reference variable, our thread-sensitive analysis makes as many set-variables as the number of its abstract threads for every reference variable in a thread definition, which is not static. This can model that every thread has its own instances for every reference variable if it is not static.

$$\begin{array}{ll}
\langle p = new\ c_\ell \rangle & \Rightarrow \{\mathcal{X}^\imath_p \supseteq O^\imath_{c_\ell}\} \\
\langle p = q \rangle & \Rightarrow \{\mathcal{X}^\imath_p \supseteq \mathcal{X}^\imath_q\} \\
\langle p = c.f \rangle & \Rightarrow \{\mathcal{X}^\imath_p \supseteq \mathcal{X}_{c.f}\} \\
\langle c.f = q \rangle & \Rightarrow \{\mathcal{X}_{c.f} \supseteq \mathcal{X}^\imath_q\} \\
\langle p.f = q \rangle & \Rightarrow \{\mathcal{X}^\imath_p \supseteq_f \mathcal{X}^\imath_q\} \\
\langle p = q.f \rangle & \Rightarrow \{\mathcal{X}^\imath_p \supseteq \mathcal{X}^\imath_q \cdot f\} \\
\langle p = a_0.m(a_1, ..., a_k) \rangle & \Rightarrow \{\mathcal{X}^\imath_{f_0} \supseteq \mathcal{X}^\imath_{a_0}, ..., \mathcal{X}^\imath_{f_k} \supseteq \mathcal{X}^\imath_{a_k}, \mathcal{X}^\imath_p \supseteq \mathcal{X}^\imath_{ret} | \\
 & \quad m(f_0, ..., f_k, ret) \in targets(\mathcal{X}^\imath_{a_0}, mc)\}
\end{array}$$

Fig. 2 Thread-sensitive set constraints for statements

$$\begin{array}{ll}
\langle p = new\ c_\ell \rangle & \Rightarrow \{\overrightarrow{\mathcal{X}_p} \supseteq \overrightarrow{O_{c_\ell}}\} \\
\langle p = q \rangle & \Rightarrow \{\overrightarrow{\mathcal{X}_p} \supseteq \overrightarrow{\mathcal{X}_q}\} \\
\langle p = c.f \rangle & \Rightarrow \{\overrightarrow{\mathcal{X}_p} \supseteq \mathcal{X}_{c.f}\} \\
\langle c.f = q \rangle & \Rightarrow \{\mathcal{X}_{c.f} \supseteq \bigcup_\imath \mathcal{X}^\imath_q | \mathcal{X}^\imath_q \in \overrightarrow{\mathcal{X}_q}\} \\
\langle p.f = q \rangle & \Rightarrow \{\overrightarrow{\mathcal{X}_p} \supseteq_f \overrightarrow{\mathcal{X}_q}\} \\
\langle p = q.f \rangle & \Rightarrow \{\overrightarrow{\mathcal{X}_p} \supseteq \overrightarrow{\mathcal{X}_q} \cdot f\} \\
\langle p = a_0.m(a_1, ..., a_k) \rangle & \Rightarrow \{\overrightarrow{\mathcal{X}_{f_0}} \supseteq \overrightarrow{\mathcal{X}_{a_0}}, ..., \overrightarrow{\mathcal{X}_{f_k}} \supseteq \overrightarrow{\mathcal{X}_{a_k}}, \overrightarrow{\mathcal{X}_p} \supseteq \overrightarrow{\mathcal{X}_{ret}} | \\
 & \quad m(f_0, ..., f_k, ret) \in targets(\mathcal{X}^\imath_{a_0}, mc)\}
\end{array}$$

Fig. 3 A system of thread-sensitive set constraints

Specifically, for each abstract thread $T_\imath$ of a thread class T, we make one set variable $X^\imath_p$ for a reference variable p, if it is not static. As in Figure 2, we construct set-constraints for all statements in all instance methods of every abstract thread $T_\imath$. In case of static variables or static methods we make set variables or set-constraints as conventional context-insensitive analysis.

We first consider an object creation statement `p = new c`. Since each thread $T_\imath$ has its own instance $p^\imath$ for the local reference variable p and its own instance $O^\imath_{c_\ell}$ for the new object, we construct a set constraint $\mathcal{X}^\imath_p \supseteq O^\imath_{c_\ell}$. For a direct assignment `p = q`, we simply construct a set constraint $\mathcal{X}^\imath_p \supseteq \mathcal{X}^\imath_q$ to model that the variable instance $p^\imath$ gets the objects of the variable instance $q^\imath$ in each thread $T_\imath$. Consider an instance field read `p = q.f`. Each thread $T_\imath$ has its own instances $p^\imath$ and $q^\imath$ and the instance $p^\imath$ gets the field `f` of the instance $q^\imath$. So we construct a set constraint $\mathcal{X}^\imath_p \supseteq \mathcal{X}^\imath_q \cdot f$. In case of a static field write `c.f = q`, because every thread can share a static field $c.f$, we have only one instance for it and so construct a set-constraint $\mathcal{X}_{c.f} \supseteq \mathcal{X}^\imath_q$.

Consider a virtual method invocation $\mathtt{p} = \mathtt{a_0.m(a_1, ..., a_k)}$. We denote by mc the method call $a_0.m(a_1, ..., a_k)$. Each thread $T_\imath$ has its own instance for formal parameters as local variables. To model parameter passing and return, we construct set-constraints as follows:

$$\{\mathcal{X}^\imath_{f_0} \supseteq \mathcal{X}^\imath_{a_0}, ..., \mathcal{X}^\imath_{f_k} \supseteq \mathcal{X}^\imath_{a_k}, \mathcal{X}^\imath_p \supseteq \mathcal{X}^\imath_{ret} | m(f_0, ..., f_k, ret) \in targets(\mathcal{X}^\imath_{a_0}, mc)\}$$

where the function *targets* returns the set of possible methods invoked by the method call mc for the target objects in $\mathcal{X}^\imath_{a_0}$.

In addition, if $O_{T_1}, ..., O_{T_n}$ are abstract thread objects for a thread class T such that $run(f_0) \in T$, we also make a set-constraint $\mathcal{X}^\imath_{f_0} \supseteq O_{T_\imath}$ for each abstract thread object $O_{T_\imath}$ to simulate passing the target thread object to the `run` method. If there are n abstract threads for a thread class T, there are n instances of a reference variable p in abstract threads, if it is not static. So, we can make a vector of set variables for one reference variable p in T as follows: $\overrightarrow{\mathcal{X}_p} = \langle \mathcal{X}^1_p, ..., \mathcal{X}^n_p \rangle$. In the same way, we can make a vector of abstract objects created at an object creation site ℓ as follows: $\overrightarrow{O_{c_\ell}} = \langle O^1_{c_\ell}, ..., O^n_{c_\ell} \rangle$. Then, we can make a system of set constraints for all abstract threads $T_1, ..., T_n$ of a thread class T as in Figure 3.

Consider the main thread started from `main` method. We denote this abstract thread by T_{main}. Note that there is only one thread instance for the abstract thread T_{main}. We can construct a context-insensitive points-to analysis like [8] for every statement reachable from *main* by replacing the index $\imath$ by *main* in Figure 2. So, $\mathcal{X}^{main}_v$ denotes the set of abstract objects pointed by a reference variable v in the `main` thread.

We construct set-constraints for each abstract thread (object) as in Figure 2, and collect all set-constraints including those for `main` thread. We can solve the collection of set-constraints as usual in [5]. The time complexity of the constraint-solving is $k \cdot n^3$ where k is the number of abstract thread objects and n is the number of reference variables.

4 Experimental Results

We have implemented the thread-sensitive points-to analysis by extending the inclusion-based points-to analysis, which is an iterative fixpoint algorithm based on method summaries [13]. It is implemented on an experimental Java virtual machine called joeq [14, 13]. We also observed that codes for `run()` methods have to be connected with its start site during analysis, because threads are started by native code. Otherwise, it will be missing, when analyzing multithreaded Java programs [13]. They missed codes for new threads and actually analyzed codes for main thread only in the analysis.

We have also implemented the functionality to connect and analyze the `run()` methods of started threads automatically. Our implementation consists of the following steps: (1) the context-insensitive points-to analysis of the main thread (2) extraction of abstract thread objects from that analysis (3) identification of the `run()` method of each abstract thread object, and (4) one thread-specific analysis of the `run` method for each target thread object

A thread-specific analysis starts from the `run` method of a thread class and analyzes all methods reachable from it. The algorithm iterates until there are no more newly called methods. We implement the analysis by creating one points-to analysis object and starting the iteration for each target thread object.

Programs	Classes	Methods	Calls	Size	Points-to	Time	Iter.
SplitRenderer	308	2108	7295	12K	1.67(88.4%)	12.7	48
AssemblyLine	136	456	1423	21K	1.61(87.7%)	12.7	24
ATApplet	319	2093	7274	120K	1.73(88.5%)	243.4	48
EventQueue	107	394	1220	20K	1.55(86.4%)	12.2	23
Raytrace	108	402	1232	20K	1.55 (86.6%)	12.1	24
mtrt	108	402	1233	20K	1.55 (86.6%)	12.1	24
Timer	107	388	1176	19K	1.57(86.0%)	12.7	24
TLQ	310	2109	7309	120K	1.71(88.4%)	250.0	48
ToolTip	532	4206	18601	243K	1.86(89.8%)	292	37

Tab. 1 Benchmarks and points-to analysis of main thread

All experiments were performed on a PC with 1.3 GHz Pentium 4 processor and 1 GB of memory running Redhat Linux 7.2. We have experimented over 9 multithreaded Java benchmarks programs. All programs are compiled with IBM Java 2.13 compiler, and their bytecodes are analyzed. SplitRenderer is the example program in this paper. mtrt and raytrace are two ray tracing programs from the standard SpecJVM98 benchmark suite. ToolTip is a class library to manage tool tips, which is from the `javax.swing` package. ATApplet is an applet for auto bank transfer from the book [7]. AssemblyLine is an applet for simulating assembly line from the book [7]. EventQueue is a library class from `java.awt`. It is a platform-independent class that queues events. Timer is a class library from `java.util`. It produces tasks, via its various schedule calls, and the timer thread consumes, executing timer tasks as appropriate, and removing them from the queue when they're obsolete. TLQApplet is an applet for two lock queue with TLQProducer and TLQConsumer.

Table 1 shows some characteristics of the benchmark programs. The first four columns show the numbers of classes, methods, call sites and the size of bytecode. This table also shows analysis results of the `main` thread for each benchmark program, which is actually analysis results of [13]. The next column "Points-to" shows average number of targets per a call site and a ratio of a single target among all call sites in the parenthesis. The last two columns shows the computation time and the number of iterations to complete.

Thread classes	Abs.	Classes	Methods	Calls	Size	Points-to	Time	Iter.
SplitRenderer:T1	2	239	1178	4383	71K	2.05(91.6%)	70	19
Assembly:T1	5	5	9	12	0.16K	1.09(91.6%)	0.2	6
ATApplet:T1	4	226	1050	3938	63K	2.12(92.0%)	87	15
ATApplet:T2	3	233	1064	3941	64K	2.12(92.0%)	108	21
ATApplet:T3	3	237	1172	4360	71K	2.06(91.6%)	102	18
EventApplet:T1	1	281	1940	6685	113K	1.74(87.4%)	76.3	39
Raytrace:T1	2	139	515	2429	33K	1.21(80.0%)	8.3	17
mtrt:T1	2	139	515	2429	33K	1.21(80.0%)	8.3	17
Timer:T1	1	4	9	15	0.5K	1.07(93.3%)	0.08	5
TLQApplet:T1	4	230	1050	3912	63K	2.13(92.9%)	156	23
TLQApplet:T2	4	228	1042	3897	63K	2.14(92.0%)	158	23
TLQApplet:T3	4	239	1178	4384	71K	2.05(91.6%)	141	18
ToolTip:T1	2	336	1530	5508	83K	3.06(88.8%)	197	19

Tab. 2 Thread-sensitive analysis

In Table 2, the first column shows the number of thread classes in each benchmark program, and the second column shows the number of abstract threads, which is the same as the number of target thread objects passed to the `run` method of each thread class. In case of ATApplet, there are 3 thread classes denoted ATApplet:T1, ATApplet:T2 and ATApplet:T3. There are 4 target thread objects for the first thread class, which means 4 abstract threads. There are 3 target thread objects for the second and third thread classes, respectively. The next columns show the number of classes, methods and calls, and the bytecode size of each thread. This table shows analysis results of each thread. A thread is analyzed once for each target thread object, and their average values are listed in the table. The column "Points-to" shows average number of targets per a call site and a ratio of a single target among all call sites in the parenthesis. In case of the first thread class ATApplet:T1, average number of targets per a call site is 2.12 and single target ratio is 92%.

5 Applications

Since thread-sensitive analysis can provide analysis information specific to each thread, it can be applied to applications related with threads. Its potential applications include synchronization elimination, static datarace detection, software engineering tools, and alias analysis.
(1) *Synchronization elimination*: A simple synchronization elimination is to detect thread-local object, which do not escape its creating thread. Escape analyses have been used to identify thread-local objects and remove synchronization associated with such objects [2, 3, 12]. By the thread-sensitive analysis, we can

record all threads accessing a synchronization object, and get more precise information on use of synchronization objects from the thread-sensitive analysis. This information enables more precise synchronization elimination.
(2) *Static datarace detection*: A datarace analysis requires a points-to analysis of thread objects, synchronization objects and access objects. The thread-specific analysis can provide more precise points-to information by recording threads accessing objects. With this information, we can identify which threads access each object. With this thread-specific analysis, we can not only detects static dataraces more precisely, but it also provide more specific datarace information such as which threads may be involved in a datarace.
(3) *Software engineering tools*: There are a few known researches on debugging multithreaded Java programs utilizing static analysis information [4]. The proposed thread-specific analysis can give information specific to a particular thread. Software engineering tools together with this thread-specific information helps programmers to understand the behavior of a particular thread more precisely and to debug multithreaded program more easily.

6 Related works

Steensgaard pointer and alias analysis is a context-insensitive and flow-insensitive algorithm, which is fast but imprecise. It has almost linear time complexity. Andersen's constraint-based analysis is a more precise points-to analysis for C programs, which is also a context-insensitive and flow-insensitive algorithm [1]. It has cubic worst time complexity. Rountev et al. presented a points-to analysis for Java by extending Andersen points-to analysis for C programs [8]. They implement the analysis by using a constraint-based approach which employs annotated inclusion constraints and compare with a basic RTA analysis. Whaley and Lam also presented an inclusion-based points-to analysis for Java by adapting and extending CLA algorithm [6], which allows Andersen's algorithm to be scalable. They treat threads just like methods.

There have been several escape analyses connected with points-to analysis [2, 3, 12, 10]. They usually apply escape information to synchronization removal and/or stack allocation of objects. A combined pointer and escape analysis for multithreaded Java programs was presented based on parallel interaction graphs which model the interactions between threads [9]. The analysis information was applied to efficient region-based allocation.

Our thread-sensitive analysis is unique in that it is context-sensitive for target thread objects only and can provide separate analysis information for each thread. When k is the number of abstract thread objects and n is the number of reference variables, the time complexity of our thread-sensitive analysis is $k \cdot n^3$.

7 Conclusion

We have presented the idea of thread-sensitive analysis in terms of points-to analysis. The idea of thread-sensitive analysis can be applied to other analyses, and the analysis results can be applied to thread-related applications, since they can provide analysis information specific to each thread. There can be two research directions in future works. One direction of future works is to investigate the effectiveness of the thread-sensitive points-to analysis more by developing more applications. Another direction of future works is to extend the idea of thread-sensitive analysis to other static analyses such as escape analysis and exception analysis. This kind of analyses could provide more thread-specific information on escaping objects and exceptions.

References

1. L. Andersen. *A Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, 1994.
2. B. Blanchet. Escape analysis for object-oriented languages: Applications to Java. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 20-34, 1999.
3. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar and S. Midkiff. Escape analysis for Java. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.
4. J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M.Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *Proceedings of ACM Conference on Programming Languages Design and Implementation*, pages 258-269, 2002.
5. N. Heintze. Set-based program analysis. Ph.D thesis, Carnegie Mellon University, October 1992.
6. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code. In *Proceedings of ACM Conference on Programming Languages Design and Implementation*, pages 85-96, 1998.
7. D. Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, 2000.
8. A. Rountev, A. Milanova and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.
9. A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, pages 12-23, 2001.
10. E. Ruf. Effective synchronization removal for Java. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 208-218, 2000.
11. B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 32-41, 1996.
12. J. Whaley and M. Linard. Compositional pointer and escape analysis for Java programs. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.
13. J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In*Proceedings of Static Analysis Symposium*, 2002.
14. J. Whaley. Joeq: A Virtual Machine and Compiler Infrastructure. In *Proceedings of ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators*, June 2003, also available at http://joeq.sourceforge.net.