Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs

Rajkishore Barik
IBM India Research Lab, rajbarik@in.ibm.com

Abstract. Modeling of runtime threads in static analysis of concurrent programs plays an important role in both reducing the complexity and improving the precision of the analysis. Modeling based on type based techniques merges all runtime instances of a particular type and thereby introduces inaccuracy in the analysis. Other approaches model individual runtime threads explicitly in the analysis and are of high complexity. In this paper we introduce a thread model that is both context and flow sensitive. Individual thread abstractions are identified based on the context and multiplicity of the creation site. The interaction among these abstract threads are depicted in a tree structure known as Thread Creation Tree (TCT). The TCT structure is subsequently exploited to efficiently compute May-Happen-in-Parallel (MHP) information for the analysis of multi-threaded programs. For concurrent Java programs, our MHP computation algorithm runs 1.77x (on an average) faster than previously reported MHP computation algorithm.

1 Introduction

As concurrent programming is embraced by more and more users, there are several ongoing research activities for the last few years in the area of static analysis of concurrent programs. To name a few of these activities: computation of May Happen in Parallel (MHP) information, detection of synchronization anomalies like data races and deadlock, hiding the effect of weak memory models at the programming level, improving the accuracy of data flow analysis, and optimization of concurrent programs.

May Happen in Parallel (MHP) analysis computes pairs of statements that may be executed concurrently in a multi-threaded program. This information can be used in program optimization [9], debugging, program understanding tools, improving the accuracy of data flow approaches, and detecting synchronization anomalies like data races.

Several approaches for computing MHP information for programs have been suggested in the past: B4 analysis by Callahan et al. [3], inter-procedural B4 analysis by Dueterwald et al. [6], non-concurrency analysis by Masciole et al. [14], and data flow analysis based MHP computation for programs with a rendezvous model of concurrency by Naumovich et al. [16]. Most recently [15] developed an efficient algorithm for computing MHP information for concurrent Java programs. Their algorithm uses a data flow framework to compute a conservative estimate of MHP information and is shown to be more efficient than reachability analysis based algorithms that determines 'ideal' static MHP information. However, the underlying thread model used in the data flow framework explicitly enumerates all runtime threads during compilation time leading to the complexity of the algorithm bounded by number of runtime threads, i.e., \( \Theta((pN)^3) \) complexity, where \( p \) is the number of runtime threads and \( N \) is the maximum number of statements per runtime thread. Such an explicit enumeration of threads makes the algorithm time consuming, and it is inapplicable to programs with unbounded or large number of runtime threads.

Subsequently, there has been work [13] on aiding a feasible implementation of the MHP algorithm presented by Naumovich et al. [16]. Their main focus is to reduce the size of the program execution graph (PEG) which is the core of MHP algorithm.

1.1 Our Contribution

The main contribution of this paper are:
- We introduce a static model of threads that is flow sensitive and context sensitive; this model is more precise than type based thread disambiguation used in previous approaches [20,18]; yet our model is capable of handling an indefinite number of runtime threads.
- We introduce a thread structure analysis and the concept of the thread creation tree (TCT), which captures the start and join interactions among threads.
- We present an efficient algorithm that computes the MHP information at two levels: first at the thread level, then at the node level. The complexity of our algorithm is $\Theta((kN)^2)$ where $k$ is the number of thread abstractions and $N$ is the maximum number of inter-procedural control flow graph nodes per thread abstraction.

Our results show that our MHP algorithm runs 1.77x faster than MHP algorithm presented by naumovich et al. [16] using our context and flow sensitive thread model.

1.2 Example

Figure 1 shows a sample program that updates a shared object of class Shared concurrently. Main thread creates two Task1 threads. These Task1 threads in turn create various Task2 threads. Note that modifications of the shared object in Task2 threads are synchronized. In addition, Task2 threads join back to Task1 threads without causing any exception.

For this example, the thread model presented by [15] considers only 3 thread abstractions during the analysis: initial thread starting at main method, 2 Task1 threads, and each Task1 thread creating 20 Task2 threads. Management of such a huge number of runtime threads in the static analysis requires a lot of space and is computationally expensive.

However, the type based thread disambiguation model described in [20,18] considers only 3 thread abstractions during the analysis: initial thread starting at main method, one for Task1 thread and one for Task2 thread. This kind of modeling seems very efficient but does not produce precise results. To elaborate this: Let us consider the MHP information computation problem. The type based thread modeling concludes that the shared object access in Line 9 of Main thread may execute in parallel with the access in Line 24 of Task1. This is not always true as the same access in Line 24 for t2 instance of Task1 never executes in parallel with Line 9 of Main thread (as t2 is started after Line 9 has finished execution). Additional machinery has to be built into these type based techniques to obtain such precise results.

2 Flow and Context Sensitive Thread Model

2.1 Abstract thread

An abstract thread is a compile time entity that corresponds to a call of the thread::start method in a certain context. Contexts are determined along a symbolic execution of the whole program [18]. In this paper, we use the terms thread and abstract thread interchangeably; if we refer to runtime threads, we note that explicitly.

An abstract thread $t_i$ might correspond to one or multiple runtime threads. In cases where the static analysis can determine that an abstract thread $t_i$ is not started in a loop or recursion (and the creator thread is itself unique), $t_i$ has a unique runtime correspondence, and the predicate $isUnique[t_i]$ holds.

In the example of Figure 1, our thread model computes 7 different abstract threads: thread corresponding to the main method denoted as $t_0$, Task1 thread in Line 8 denoted as $t_1$, Task2 thread started in Line 28 denoted as $t_3$, Task2 thread started in Line 37 of $t_1$ denoted as $t_4$, Task1 thread started in Line 11 denoted as $t_2$, Task2 thread started in Line 28 of $t_2$ denoted as $t_5$, and Task2 thread started in Line 37 of $t_2$ denoted as $t_6$. The abstract thread $t_1$ started in line 8 is unique because the creator thread (main) is unique, and the start site is not executed in a loop/recursion. The abstract thread $t_3$ created in line 28, in contrast is not unique, because it is started inside a loop.

3 Program Representation

In this section, we describe other data structures that are necessary for performing MHP analysis on concurrent programs. The thread creation graph (TCG) data structure depicts various start-join interactions among abstract threads and is used to develop an efficient algorithm for MHP.
```java
1 class Shared { int field=0; }
2 class Main {
3     public static void main(String[] args) {
4         static Shared s;
5         new Shared();
6     }
7     s.field++; // t1
8     t1.start();
9     s.field++;
10    Thread t2 = new Task1();
11    t2.start(); // t2
12    s.field++;
13 }
14 synchronized(Main.s){
15     Main.s.field++;
16     public void run() {
17         t1[i].start(); // t3, t5
18         for(int i=0; i<10; i++) {
19             ta[i] = new Task2();
20         }
21     }
22     class Task1 extends Thread {
23         public void run() {
24             Main.s.field++;
25             Thread ta = new Thread[10];
26             for(int i=0; i<10; i++) {
27                 ta[i] = new Task2();
28             }
29         }
30         Thread tb = new Thread[10];
31         for(int i=0; i<10; i++) {
32             tb[i].start(); // t4, t6
33         }
34         for(int i=0; i<10; i++) {
35             tb[i].join();
36         }
37     }
38     class Task2 extends Thread {
39         public void run() {
40             ta[i].start(); // t3, t5
41         }
42     }
43 }
```

![Fig. 1. Example program.](image)

### 3.1 Intra-thread control flow graph

The control-flow structure of an abstract thread $t_i$ is represented in an intra-thread control flow graph (ICFG), i.e., $ICFG(t_i) = (V(t_i), E(t_i))$ where $E(t_i)$ denotes the intra-procedural and inter-procedural control flow edges of abstract thread $t_i$, and $V(t_i)$ comprises of the following types of nodes:

- **USE**($t_i$) refers to the set of shared read access (get/load of shared reference/field/array) nodes in $t_i$.
- **ASS**($t_i$) refers to the set of shared write access (put/store of shared reference/field/array) nodes in $t_i$.
- **NEW**($t_i$) refers to the set of allocation nodes in $t_i$.
- **BEGIN**($t_i$) refers to the set of method entry nodes in $t_i$.
- **END**($t_i$) refers to the set of method exit nodes in $t_i$.
- **ENTRY**($t_i$) refers to the unique thread entry node for $t_i$.
- **EXIT**($t_i$) refers to the unique thread exit node for $t_i$.
- **CSTART**($t_i$) refers to the set of abstract thread start nodes in $t_i$.
- **CJOIN**($t_i$) refers to the set of abstract thread join nodes in $t_i$.
- **CALL**($t_i$) refers to the set of method call nodes in $t_i$.
- **ACQUIRE**($t_i$) refers to the set of monitor enter nodes in $t_i$.
- **RELEASE**($t_i$) refers to the set of monitor exit nodes in $t_i$.

$V(t_i)$ contains two special nodes: **ENTRY**($t_i$) and **EXIT**($t_i$). There is an edge from **ENTRY**($t_i$) to any node at which the thread can be entered, and there is an edge to **EXIT**($t_i$) from any node that can exit the thread.

**$E(t_i)$** contains intra-procedural and inter-procedural control flow edges in $t_i$. The inter-procedural control flow edges do not comprise of subsequent thread creation edges from $t_i$.

Certain statements need not be represented in the ICFG, e.g., statements that only have a thread-local effect. This includes access nodes (USE, ASS) that operate on thread local objects (the underlying object model and analysis for determining thread locality is presented in [5, 18]
Figure 2 shows the inter-procedural control flow graph for the main abstract thread of the example program. Each node in the figure is annotated with the object/field it accesses. CSTART[t1] and CSTART[t2] nodes represent the invocation of abstract threads t1 and t2 respectively. Note that there is no inter-procedural control flow edge connecting the node CSTART[t1] to ICFG(t1).

Let the creation node of an abstract thread tj in ti be denoted as CSTART(ti, tj), i.e., CSTART(ti, tj) ∈ CSTART(ti). There is no inter-procedural control flow edge from ti to tj in ICFG(ti). Similarly, the join node of an abstract thread tj in ti is denoted as CJOIN(ti, tj).

![Diagram](image)

Fig. 2. Inter-procedural control flow graph (ICFG)

### 3.2 Must-join

A common pattern in parallel programs is that some threads create subsidiary threads and later join those. We capture this information using the concept of a must-join abstract thread. Let CSTART(ti, tj) be the node where abstract thread tj is created in ti. Let CJOIN(ti, tj) ∈ V(tk) be the node where abstract thread tj is joined. tj is then termed as a must-join abstract thread if ti = tk and CJOIN(ti, tj) postdom CSTART(ti, tj).
3.3 Thread Creation Tree (TCT)

Threads can be structured according to their start-relationships. The thread creation tree (TCT) encodes this information: Abstract threads are represented as nodes, edges encode the start relation. The main thread constitutes the root, threads started by the main thread are found at the first hierarchy level etc.

The must-join information for each node in the TCT is encoded using a predicate $\text{mjoin}$, i.e., $\text{mjoin}[t_i] = \text{true}$ if $t_i$ is a must-join abstract thread.

![Fig. 3. Thread Creation Tree.](image)

The TCT for the program in Figure 1 is given in Figure 3. $t_1$ and $t_2$ are colored black as they do not join the main abstract thread $t_0$, i.e., $\text{mjoin}[t_1] = \text{mjoin}[t_2] = \text{false}$.

The specific case of a mutual thread creation inside a recursion, might lead to an unbounded TCT. We detect this case and resolve it by combining the involved abstract threads. For example, abstract thread $t_3$ creates $t_j$: Both $t_i$ and $t_j$ have the same static thread type. Since there is recursion involved in the static types of $t_i$ and $t_j$, the TCT will be unbounded. To handle this, we add only one node to the TCT with ICFG as the ICFG of $t_i$. The number of runtime instances of the added node in the TCT is not unique. The must-join information of the added node is set based on must-join information of $t_i$ or $t_j$. In general, if a set of static types are involved in mutual recursion, we create a single node for the same in TCT. The ICFG of this node is created by combining ICFG of all the involved static types (details described in Appendix A).

4 MHP computation

Given all abstract threads of a program, their ICFGs and the TCT, we compute nodes which may potentially execute in parallel, i.e., MHP information. This computation is performed at two levels: first at the abstract thread level and then at node level. At the abstract thread level, MHP computes pairs of abstract threads that may potentially execute in parallel. This is coarse-grained MHP information. Node level MHP refines this information by considering the individual statements and control-flow structure of threads that are identified as MHP at the thread-level. Since we are doing a compile time approximation of MHP (considering every control flow path), the MHP information we compute is a conservative superset of what actually happens at runtime.

Apart from ordering criteria among threads due to thread start and join, locks are also commonly used to order the execution among threads. We conservatively compute the locks statically using the following manner: In Java, locks are used in a scoped manner. Locks held during an access statement are recorded during the creation of the ICFG and associated with the corresponding node. We define $\text{locks}[v_i] \subseteq \text{ specified set of objects that are locked while executing any node } v_i \in V(t_i)$. Nodes that execute in the context of a common unique lock cannot execute concurrently.

Our MHP analysis is based on graph algorithms like reachability and dominance. We write $x \rightarrow y$ to indicate a directed path from start node $x$ to end node $y$. A null path is a path whose start node and end node are the same, i.e., a single node. A non-null path from $x$ to $y$ is written as $x \rightarrow y$. This path definition applies to both ICFG and TCT.

A directed path $t_1 \rightarrow t_n$ in the TCT is called a must-join path if all the nodes that lie on the path from $t_1$ to $t_n$ are must-join abstract threads, i.e., $\forall i = 1, \cdots, n$. For example, the path $t_0 \rightarrow t_1 \rightarrow t_3$ in Figure 3 is not a must-join path as $\text{mjoin}[t_1] = \text{false}$. 
The dominance relation between two nodes in the ICFG is represented by \textit{dom}. Further, we denote node dominance as \( \text{dom}_v^{m} [v_i^n] \) that consists of all nodes that lie on all possible directed paths from \( v_i^m \in V(t_i) \) to \( v_i^n \in V(t_i) \) in \textit{ICFG}(t_i).

### 4.1 Thread level MHP

![Thread level MHP](image)

**Fig. 4.** Thread level MHP.

Thread level MHP computes pairs of abstract threads that may execute in parallel. It exploits the rooted tree structure of the TCT to determine such information.

Let \( ||_i \) denote the MHP relation between two abstract threads. The ancestors of an abstract thread \( t_i \) in the TCT are represented in a set \( \text{anc}(t_i) \). \textit{child} and \textit{parent} represent the child and parent relationship in the TCT. Let \( \text{yca}(t_i, t_j) \) denote the youngest common ancestor of \( t_i \) and \( t_j \) in TCT. Let \( \text{canc}(t_i, t_j) \) be the child of the abstract thread \( t_i \) that is either \( t_i \) itself or an ancestor of \( t_j \). Mathematically,

\[
\text{yca}(t_i, t_j) = \begin{cases} 
  t_k & \text{if } t_k \text{ is the youngest common ancestor of } t_i \text{ and } t_j \\
  \text{null} & \text{otherwise}
\end{cases}
\]

\[
\text{canc}(t_i, t_j) = \begin{cases} 
  t_j, & \text{if } t_j = \text{child}(t_i) \\
  \text{child}(t_i), & \text{if } \text{child}(t_i) \in \text{anc}(t_j) \\
  \text{null} & \text{otherwise}
\end{cases}
\]

Computation of thread level MHP is conservative. If an abstract thread \( t_i \) is an ancestor of another abstract thread \( t_j \), then we conservatively assume that \( t_i \) and \( t_j \) run in parallel with each other, i.e., \( t_i || t_j \). Further refinement to this MHP information is done in node level MHP in which we consider fine-grained statement level parallelism.

\[
t_i || t_j = \text{true} \quad \text{if } t_i \in \text{anc}(t_j) \text{ or } t_j \in \text{anc}(t_i)
\]

Apart from the above conservative case, all other possible cases to determine if any two TCT nodes \( t_i \) and \( t_j \) may execute in parallel are presented below. For compact representation of the cases we denote the youngest common ancestor of \( t_i \) and \( t_j \) as \( t_{yca} \), i.e., \( t_{yca} = \text{yca}(t_i, t_j) \).
- **Case 1:** Let us consider the case where neither the TCT path \( \text{canc}(t_{yca}, t_i) \rightarrow t_i \) nor the TCT path \( \text{canc}(t_{yca}, t_j) \rightarrow t_j \) is a must-join path. The TCT for this case is shown in Figure 4(a). \( t_i \) and \( t_j \) may execute in parallel, if at least one of the following conditions holds: (1) their common parent \( t_{yca} \) is not unique, or (2) both threads \( \text{canc}(t_{yca}, t_i) \) and \( \text{canc}(t_{yca}, t_j) \) may be started in some control-flow in \( ICFG(t_{yca}) \). This case is mathematically presented in Table 1.

\[
| t_i \| t_j | = \begin{cases} 
\text{true,} & \text{if } \text{isUnique}[t_{yca}] = \text{false} \\
\left( \text{CSTART}(t_{yca}, \text{canc}(t_{yca}, t_i)) \rightarrow \text{CSTART}(t_{yca}, \text{canc}(t_{yca}, t_j)) \right) \quad \lor \quad \\
\left( \text{CSTART}(t_{yca}, \text{canc}(t_{yca}, t_j)) \rightarrow \text{CSTART}(t_{yca}, \text{canc}(t_{yca}, t_i)) \right) \\
\text{otherwise} 
\end{cases}
\]

**Table 1.** Thread Level MHP: Case 1

- **Case 2:** Let us consider the case where the TCT path \( \text{canc}(t_{yca}, t_i) \rightarrow t_i \) is a must-join path and the TCT path \( \text{canc}(t_{yca}, t_j) \rightarrow t_j \) is not a must-join path. This case is shown in Figure 4(b). \( t_i \) may execute in parallel with \( t_j \) if at least one of the following conditions holds: (1) \( t_{yca} \) has multiple runtime instances, (2) there is a control-flow path from \( \text{CSTART}(t_{yca}, \text{canc}(t_{yca}, t_j)) \) to \( \text{CSTART}(t_{yca}, \text{canc}(t_{yca}, t_i)) \) in \( ICFG(t_{yca}) \), or (3) there is a control-flow path from \( \text{CJOIN}(t_{yca}, \text{canc}(t_{yca}, t_j)) \) to \( \text{CJOIN}(t_{yca}, \text{canc}(t_{yca}, t_i)) \) without \( \text{CJOIN}(t_{yca}, \text{canc}(t_{yca}, t_i)) \) in \( ICFG(t_{yca}) \). This case is mathematically presented in Table 2.

\[
| t_i \| t_j | = \begin{cases} 
\text{true,} & \text{if } \text{isUnique}[t_{yca}] = \text{false} \\
\left( \text{CSTART}(t_{yca}, \text{canc}(t_{yca}, t_i)) \rightarrow \text{CSTART}(t_{yca}, \text{canc}(t_{yca}, t_j)) \right) \quad \lor \quad \\
\left( \text{CSTART}(t_{yca}, \text{canc}(t_{yca}, t_j)) \rightarrow \text{CSTART}(t_{yca}, \text{canc}(t_{yca}, t_i)) \right) \\
\text{otherwise} 
\end{cases}
\]

**Table 2.** Thread Level MHP: Case 2

- **Case 3:** Let us the consider the case where the TCT paths \( \text{canc}(t_{yca}, t_i) \rightarrow t_i \) and \( \text{canc}(t_{yca}, t_j) \rightarrow t_j \) are must-join paths. This case is shown in Figure 4(c). \( t_i \) may execute in parallel with \( t_j \) if at least one of the following conditions holds: (1) \( t_{yca} \) has multiple runtime instances, (2) there is a control-flow path from \( \text{CSTART}(t_{yca}, \text{canc}(t_{yca}, t_i)) \) to \( \text{CJOIN}(t_{yca}, \text{canc}(t_{yca}, t_j)) \) without \( \text{CJOIN}(t_{yca}, \text{canc}(t_{yca}, t_i)) \) in \( ICFG(t_{yca}) \), or (3) there is a control-flow path from \( \text{CSTART}(t_{yca}, \text{canc}(t_{yca}, t_i)) \) to \( \text{CJOIN}(t_{yca}, \text{canc}(t_{yca}, t_j)) \) without \( \text{CJOIN}(t_{yca}, \text{canc}(t_{yca}, t_i)) \) in \( ICFG(t_{yca}) \). This case is mathematically presented in Table 3.

Consider our example program and its corresponding TCT in Figure 3. \( t_5 \) cannot execute in parallel with \( t_4 \) because abstract thread \( t_5 \) joins \( t_1 \) before abstract thread \( t_4 \) is started. Similarly, \( t_5 \) can never run in parallel with \( t_6 \). However, all other pairs of abstract threads may run in parallel with each other.

### 4.2 Node level MHP

Thread level MHP \( |_{l} \) is a coarse grained approximation of MHP information, because all statements of a thread are subsumed and given the same MHP information. MHP information among statements from threads \( t_i \) and \( t_j \) can be refined further at the node level in the case where either \( t_i \) is an ancestor of \( t_j \) or \( t_j \) is ancestor of \( t_i \) in TCT.

Consider our example program and its corresponding TCT in Figure 3. Thread level MHP computation computes that \( |_{l} t_3 | t_1 \). This suggests that all statements of threads \( t_1 \) occur in parallel with
statements in thread $t_3$, i.e., $t_1 \parallel t_3$. However, the ICFG nodes corresponding to statement 33 in $t_1$ will never run in parallel with ICFG nodes corresponding to statement 18 of $t_3$. This is because the abstract thread $t_3$ terminates before thread $t_1$ executes statement 33.

We use the symbol $\parallel_n$ to denote node level MHP information between two ICFG nodes. Let $t_i$ and $t_j$ be two abstract threads such that $t_i \in \text{anc}[t_j]$. All possible cases to determine if any two ICFG nodes $v^m_i$ and $v^m_j$ may execute in parallel are presented below:

- **Case 1**: Let us consider the case where the TCT path $\text{canc}(t_i, t_j) \rightarrow t_j$ is not a must-join path. This case is shown in Figure 5(a). $v^m_i$ may execute in parallel with $v^m_j$ if at least one of the following conditions holds: (1) $t_i$ has multiple runtime instances, or (2) there is a control-flow path from $\text{CSTART}(t_i, \text{canc}(t_i, t_j))$ to $v^m_i$ in $\text{ICFG}(t_i)$. This case is mathematically presented in Table 4.

$$v^m_i \parallel_n v^m_j = \begin{cases} \text{true,} & \text{if } \text{isUnique}[t_i] = \text{false} \\ \text{CSTART}(t_i, \text{canc}(t_i, t_j)) \rightarrow v^m_i & \text{otherwise} \end{cases}$$

| Table 4. Node Level MHP: Case 1

- **Case 2**: Let us consider the case where the TCT path $\text{canc}(t_i, t_j) \rightarrow t_j$ is a must-join path. This case is shown in Figure 5(b). $v^m_i$ may execute in parallel with $v^m_j$ if at least one of the following conditions holds: (1) $t_i$ has multiple runtime instances, or (2) there is a control-flow path from $\text{CSTART}(t_i, \text{canc}(t_i, t_j))$ to $v^m_i$ without the $\text{CJOIN}(t_i, \text{canc}(t_i, t_j))$ in $\text{ICFG}(t_i)$. This case is mathematically presented in Table 5.

To summarize the MHP information based on thread level and node level, let $\parallel$ denote the generic MHP information between any two nodes $v^m_i \in V(t_i)$ and $v^m_j \in V(t_j)$. Then the condition under which $v^m_i$ may execute in parallel with $v^m_j$ is given in Table 6. Besides the thread and node level MHP relations, the condition also accounts for ordering through common lock protection and concurrency among nodes of abstract threads that are not unique.

![Figure 5. Node level MHP.](image)

| Table 3. Thread Level MHP: Case 3

- **Case 2**: Let us consider the case where the TCT path $\text{canc}(t_i, t_j) \rightarrow t_j$ is a must-join path. This case is shown in Figure 5(b). $v^m_i$ may execute in parallel with $v^m_j$ if at least one of the following conditions holds: (1) $t_i$ has multiple runtime instances, or (2) there is a control-flow path from $\text{CSTART}(t_i, \text{canc}(t_i, t_j))$ to $v^m_i$ without the $\text{CJOIN}(t_i, \text{canc}(t_i, t_j))$ in $\text{ICFG}(t_i)$. This case is mathematically presented in Table 5.

To summarize the MHP information based on thread level and node level, let $\parallel$ denote the generic MHP information between any two nodes $v^m_i \in V(t_i)$ and $v^m_j \in V(t_j)$. Then the condition under which $v^m_i$ may execute in parallel with $v^m_j$ is given in Table 6. Besides the thread and node level MHP relations, the condition also accounts for ordering through common lock protection and concurrency among nodes of abstract threads that are not unique.
The skeleton of the MHP algorithm is provided in Algorithm 1. Step 1 computes the abstract threads and their ICFGs along a symbolic program execution [18]. Step 3 computes postdom relation which is necessary to determine if the abstract thread is a must-join abstract thread or not. Step 4 finds out all possible execution paths in the ICFG. Step 5-7 compute node dominance with respect to various CSTART nodes in the abstract thread. Step 8 adds a TCT node along with its must-join information. Step 10 computes all possible must-join chains and also computes youngest common ancestor information for each pair of nodes in TCT. This can be obtained by performing a bottom-up traversal of the TCT. Steps 11-20 compute MHP information between every pair of nodes across all abstract threads using the equation given in Table 6. Since MHP information between a pair of nodes is symmetric, we carefully choose $t_j$ in step 12 so as to reduce the number of comparisons.

**Algorithm 1** MHP computation.

1: Perform a symbolic execution over the whole program to identify various abstract threads and their ICFGs.
2: for every abstract thread $t_i$ in the program do
3:     Compute postdom($v^m_i$) for each $v^m_i \in V_i$.
4:     Compute reachability information ($\rightarrow$) for every pair of nodes in $V_i$.
5:     for every child abstract thread $t_j$ created by $t_i$ do
6:         Compute dom$_{CSTART}(t_i, t_j)[v^m_i]$ for each $v^m_i \in V_i$.
7:     end for
8:     Add appropriate node to TCT.
9: end for
10: Compute must-join chains and gather youngest common ancestor information for every pair of nodes in TCT.
11: for all abstract thread $t_i$ do
12:     for all abstract thread $t_j$ do
13:         for all $v^m_i \in V_i$ do
14:             for all $v^m_j \in V_j$ do
15:                 Determine $v^m_i \parallel v^m_j$ using Table 6.
16:             end for
17:         end for
18:     end for
19: end for

<table>
<thead>
<tr>
<th>$v^m_i \parallel v^m_j$ =</th>
</tr>
</thead>
</table>
| $\left\{ \begin{array}{ll}
\text{true,} & \text{if $isUnique[t_i] = \text{false}$} \\
CSTART(t_i, canc(t_i, t_j)) \rightarrow v^m_i \\
\wedge
CJOIN(t_i, canc(t_i, t_j)) \notin \text{dom}_{CSTART}(t_i, canc(t_i, t_j)}[v^m_i] \\
\text{otherwise}
\end{array} \right.$ |

**Table 5.** Node Level MHP: Case 2.

$\begin{align*}
v^m_i \parallel v^m_j & = \begin{cases}
(\text{locks}[v^m_i] \cap \text{locks}[v^m_j]) = \emptyset & \text{if } t_i = t_j \text{ and } isUnique(t_i) = \text{false} \\
(\text{locks}[v^m_i] \cap \text{locks}[v^m_j]) = \emptyset & \wedge \\
(t_i \parallel t_j) & \text{otherwise}
\end{cases} \\
\end{align*}$

**Table 6.** Final MHP computation formula.

4.3 Complexity analysis

Let $k$ be the total number of abstract threads. Let $N$ be the total number of ICFG nodes per abstract thread. Step 3 can be computed in $\Theta(N^2)$ time using the algorithm suggested by Alstrup et al. [2]. Reachability information in Step 4 can be computed in $\Theta(N^2)$ time using standard depth first search
algorithm. Since dominance with respect to a single node is computed in $\Theta(N^2)$ time, steps 2-9 can be executed in a worst case complexity of $\Theta((kN)^2)$. Computation of must-join chain and common parent information in step 10 can be obtained in $\Theta(k^2)$ complexity using a bottom up traversal of TCT. Careful selection of $t_i$ will yield a time complexity of $\Theta((k + \binom{k}{2})N^2)$ for steps 11-21. Hence, the overall worst case time complexity of the algorithm is $\Theta((kN)^2)$. Note that the complexity analysis does not include the cost of computation of abstract threads and their ICFGs.

5 Implementation details

The abstract threads and their ICFGs are computed by performing a symbolic execution over the whole program. The focus of the description here is on the MHP analysis and details of the symbolic execution are discussed in [18].

5.1 Intra-procedural analysis

During intra-procedural analysis, we obtain a flow-sensitive control flow graph for a method. Each node in this graph corresponds to instructions in the original program/byte-code sequence: BEGIN and END nodes to indicate begin and end of methods, USE and ASS nodes for accessing and modifying shared data, CSTART and CJOIN nodes to indicate child abstract thread start and joins, ACQUIRE and RELEASE nodes to represent monitor regions, NEW nodes to indicate object/array allocations, CALL nodes to denote method invocations, and ENTRY and EXIT nodes to indicate thread entry and exit points (these two nodes can be maintained separately or merged with BEGIN and END nodes of the run method of the thread). While creating CSTART nodes, we create new abstract threads. For the main thread in Java, we create a special abstract thread.

5.2 Inter-procedural analysis

The CALL nodes of various methods are linked to their polymorphic callee's BEGIN nodes. The END nodes of the callees are connected back to the successors of the caller's CALL node. In case a method is involved in recursion, we reuse the already computed intra-thread control flow graph nodes and hence do not descend into its call again. This approach can lead to artifact paths in the ICFG that cannot execute in real program execution. However, this does not affect the conservative results of the analysis. In case the target of a CALL node is not involved in any shared data access (leads to side effect free calls), we do not descend into it.

The nodes in ICFG are properly annotated with current set of locks. The lock sets are propagated as a stack in a flow sensitive manner along with the symbolic execution. Since the symbolic execution in every method is performed in a depth first order, the lock set of a successor depends both on the lock set of one of the predecessors and on the current node. Lock sets are modified appropriately for ACQUIRE and RELEASE nodes.

Along with the symbolic execution we gradually update the TCT. Initially TCT contains one node for the abstract thread corresponding to the main thread. Then as and when we encounter new CSTART nodes at various contexts, we create new abstract threads and add them to TCT.

5.3 Barriers

A barrier synchronization point has the effect of causing all threads to wait at the barrier until every thread has reached it. Barriers can be implemented in various ways in Java [12]. Since it is hard to detect barrier synchronization points using program analysis, we annotate programs at barrier synchronization points. This annotation helps us reduce the MHP pairs as the following way: statements above a barrier point never execute concurrently with the ones below the barrier.

5.4 Limitation

The 2-level MHP algorithm computes MHP information for programs with no synchronization constructs like wait, notify and notifyAll. The presence of such constructs may require the MHP algorithm to enumerate every runtime threads explicitly in the compilation time and thereby making the analysis expensive and inapplicable to unbounded number of threads.
6 Experience

In this section, we report our experience in a Java-IA32 way-ahead compilation environment on a Pentium IV CPU at 2.66GHz running Redhat Linux. Our runtime system is based on GNU libgcj version 2.96 [7]. The numbers we present refer to the overall program including library classes, and excluding native code. The effect of native code for aliasing and object access has been modeled explicitly in the compiler.

We use several multi-threaded benchmark programs [10, 24] to evaluate the precision of our analysis. JGFCrypt, JGFSeries, JGFSort, JGFLUFact, JGFSparsematmult, JGFMoldyn, JGFRaytracer, and JGFRaytracer are multi-threaded benchmarks from Java Grande Forum [10]. Other benchmarks philo, elevator, sor and tsp are described in [18].

We compare the running time of our analysis with that of [16] et al. We modified their MHP algorithm to use our context and flow sensitive thread model. We also use the interprocedural control flow graph structure (ICFG) described in Section 3.1 instead of the Program Execution Graph (PEG) that they proposed. To model PEG interactions at thread start and join in ICFG, we keep additional information in ICFG nodes regarding threads started and joined at that node; this helps us propagate the OUT and M information in their MHP algorithm. Abstract threads which do not represent multiple instances of the runtime threads are handled easily by their MHP algorithm. For a non-unique abstract thread, we add additional explicit MHP computation among the nodes of the abstract thread (similar to the way our MHP algorithm computes MHP information for non-unique abstract threads).

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Naumovich et al. MHP [16] in milliseconds</th>
<th>Our 2-level MHP in milliseconds</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGFSor</td>
<td>51</td>
<td>27</td>
<td>1.89</td>
</tr>
<tr>
<td>JGFSparsematmult</td>
<td>34</td>
<td>9</td>
<td>3.78</td>
</tr>
<tr>
<td>JGFSeries</td>
<td>33</td>
<td>11</td>
<td>3.00</td>
</tr>
<tr>
<td>JGFLUFact</td>
<td>50</td>
<td>29</td>
<td>1.72</td>
</tr>
<tr>
<td>JGFCrypt</td>
<td>163</td>
<td>83</td>
<td>1.96</td>
</tr>
<tr>
<td>JGFMoldyn</td>
<td>13415</td>
<td>13119</td>
<td>1.02</td>
</tr>
<tr>
<td>JGFMontecarlo</td>
<td>3242</td>
<td>3193</td>
<td>1.02</td>
</tr>
<tr>
<td>JGFRaytracer</td>
<td>2176</td>
<td>2034</td>
<td>1.07</td>
</tr>
<tr>
<td>philo</td>
<td>34</td>
<td>15</td>
<td>2.43</td>
</tr>
<tr>
<td>elevator</td>
<td>248</td>
<td>183</td>
<td>1.36</td>
</tr>
<tr>
<td>sor</td>
<td>338</td>
<td>210</td>
<td>1.61</td>
</tr>
<tr>
<td>tsp</td>
<td>696</td>
<td>696</td>
<td>1.00</td>
</tr>
<tr>
<td>mtrt</td>
<td>4217</td>
<td>3823</td>
<td>1.10</td>
</tr>
</tbody>
</table>

Table 7. Running time of our MHP algorithm vs Naumovich et al.

Table 8 reports number of abstract threads and their corresponding number of ICFG nodes. In all the benchmarks, except the main thread which is unique, other abstract threads have multiple instances. Table 7 compares the running time of our MHP algorithm as opposed to Naumovich et al. On an average, we show 1.77x speedup on the running time of MHP algorithm.

For larger benchmarks like JGFMoldyn, JGFMontecarlo, JGFRayTracer, and tsp, the abstract thread(s) except the main thread have higher number of ICFG nodes (Column 2 in Table 8). Since the computation of MHP information for abstract threads having multiple instances is same for both our algorithm and Naumovich et al. algorithm (Note that Naumovich et al. modeled runtime threads and hence did not have multiple instances of a thread; we added extra code to adapt to our thread model), the improvements are not significant. However, for other benchmarks like JGFSeries and JGFSparsematmult, we obtain large running time benefits.
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Num of abstract threads</th>
<th>Num of ICFG nodes in abstract threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGFSor</td>
<td>2</td>
<td>48+69</td>
</tr>
<tr>
<td>JGFSparsematmult</td>
<td>2</td>
<td>68+20</td>
</tr>
<tr>
<td>JGFseries</td>
<td>2</td>
<td>53+21</td>
</tr>
<tr>
<td>JGFUFL</td>
<td>2</td>
<td>57+57</td>
</tr>
<tr>
<td>JGFCrypt</td>
<td>3</td>
<td>52+61+61</td>
</tr>
<tr>
<td>JGFJovelin</td>
<td>2</td>
<td>280+758</td>
</tr>
<tr>
<td>JGFMontecarlo</td>
<td>2</td>
<td>520+316</td>
</tr>
<tr>
<td>JGFRaytracer</td>
<td>2</td>
<td>387+221</td>
</tr>
<tr>
<td>philo</td>
<td>2</td>
<td>17+93</td>
</tr>
<tr>
<td>elevator</td>
<td>2</td>
<td>83+142</td>
</tr>
<tr>
<td>sor</td>
<td>3</td>
<td>83+77+77</td>
</tr>
<tr>
<td>tsp</td>
<td>2</td>
<td>181+398</td>
</tr>
<tr>
<td>mtrt</td>
<td>3</td>
<td>85+1022+1022</td>
</tr>
</tbody>
</table>

Table 8. Details about benchmarks.

7 Conclusion

In this paper, we present a new thread model where individual thread abstractions are obtained in a flow and context sensitive manner from the program. The new thread abstraction models runtime threads precisely and yet efficiently during compile time. This thread model can be used in various concurrent program analysis and optimizations to improve the precision of results.

The thread model is subsequently used to compute MHP information efficiently. Splitting the MHP computation based on thread structure level (TCT) and individual thread abstraction’s control flow structure level reduces the complexity of the algorithm as opposed to data flow-based approaches proposed by Naumovich et al. [15]. The TCT structure depicts interaction among threads and can be used to perform various thread structure analysis.

As concurrent programming is embraced by more users (and finds its way into future processor architectures), there will be increased demand on the compiler to produce precise static analysis results. Context and flow sensitive thread abstractions and thread structure analysis described in this paper can provide a solid back-bone for concurrency-aware compilation systems.

8 Acknowledgments

We thank Christoph v. Praun, Vivek Sarkar and Prof. Thomas Gross for their invaluable comments during early version of the paper. We also thank Matteo Corti and Florian Schneider for their contributions to the compiler infrastructure.

References


A Appendix – Thread Creation Tree

The thread creation tree described in Section 3.3 precisely depicts the start-join ordering semantics among abstract threads in a program. Since the tree is computed in a context and flow sensitive manner,
presence of cyclic thread creation might make the TCT unbounded. Consider the code fragment given Figure 6: Thread A creates Thread B; Thread B creates Thread C; Thread C subsequently creates Thread A. Clearly there is a recursion involved in the creation of various threads. This requires special handling to avoid the recursive invocation of start methods.

```java
class A extends Thread {
    void run() {
        Thread b = new B();
        b.start();
    }
}

class C extends Thread {
    void run() {
        Thread a = new A();
        a.start();
    }
}

class B extends Thread {
    void run() {
        Thread c = new C();
        c.start();
    }
}
```

*Fig. 6. Recursive program.*

To handle the above scenario, we perform a strongly connected component search algorithm over the call graph of the whole program to detect all those start methods of static thread types that are involved in a recursion. Let \( \{ s_1, s_2, \ldots, s_n \} \) be the set of all such strongly connected components, where each \( s_i = \{ x_{i1}, x_{i2}, \ldots, x_{im} \} \). Each \( x_{ij} \) denote a static thread type. Subsequently, we compute a conservative inter-procedural control flow graph for each \( s_i \) by combining the inter-procedural control flow graph of all \( x_{ij} \). While combining the inter-procedural control flow graphs, start method invocations for static thread types in \( s_i \) are treated as normal method invocations and are connected via control flow edges.

While performing symbolic execution (described in Section 5), if we encounter a start method invocation of a static thread type which belongs to any of the above computed \( s_i \), then we create a node in the TCT corresponding to \( s_i \). isUnique and mjoin predicates for the created TCT node are conservatively set to false. ICFG of the created TCT node is set to the inter-procedural control flow graph of \( s_i \).