Making It Practical and Effective: Fast and Precise May-Happen-in-Parallel Analysis

Congming Chen  
State Key Laboratory of Computer Architecture, CAS  
Graduate University of CAS  
Beijing, China  
chencongming@ict.ac.cn

Wei Huo  
State Key Laboratory of Computer Architecture, CAS  
Beijing, China  
huowei@ict.ac.cn

Xiaobing Feng  
State Key Laboratory of Computer Architecture, CAS  
Beijing, China  
fxb@ict.ac.cn

ABSTRACT

May-Happen-in-Parallel (MHP) analysis is a very important and fundamental mechanism to facilitate concurrent program analysis. But the limitation of its efficiency keep it away from being practical and effective in analyzing large scale real world concurrent programs. We proposed a novel MHP algorithm by performing a reachability analysis on a so-called parallel reachability graph of a program. The MHP algorithm mainly comprises two phases: pre-computation of initial MHP information and top-down propagation of this information along the parallel reachability graph. Our algorithm is fast as it has a low complexity $O(|N|+|E|)$, in which $N$ is the number of nodes in the parallel reachability graph and $E$ is the number of edges in this graph. Our preliminary experiment on 13 concurrent programs indicates that our approach is extremely faster than two state-of-art approaches, respectively achieving a relative geometry average speed up of 395.53× and 136.37×, while yielding the same precision with these two approaches.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors - Compilers; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages - Program Analysis

General Terms

Algorithms, Performance, Languages

Keywords

may happen in parallel, concurrent, program analysis

1. INTRODUCTION

May-Happen-in-Parallel (MHP) analysis is a very important and fundamental mechanism which can facilitate many other techniques such as concurrent control flow and data flow analysis, concurrent program optimization, and even synchronization anomalies detection. Generally speaking, the problem of precise MHP analysis for all pairs of statements in a concurrent program is undecidable. Taylor[1] has proved that this problem is NP-complete given that all control paths in all threads are executable. Thus, most MHP algorithms are computing an approximate result.

Until now, many approaches have been proposed to compute MHP information for different concurrent programming models. However, the state-of-art approaches are still facing the efficiency obstacle when applying them to analyze large scale or even medium scale concurrent programs, for instance, most programs tested in previous works are less than 10KLOC[2], which is far smaller than the scale of real world programs. In the other side, popular programming models such as Java or POSIX Threads are difficult to analyze efficiently due to their low level concurrent syntax and complex concurrent structures[2-4].

Figure 1: Simple PRG involved 3 threads

Before stating the motivation of our approach, we will introduce the primary program representation in our approach. Our main program representation is called Parallel Reachability Graph (PRG), which is a directed graph built by connecting different intra-threaded control flow graphs with kinds of edges. Each node $n$ in PRG is represented as a triple $<O, Ti, Tr>$, where $O$ is the operation performed by $n$, for example, we use operation $\text{spawn}$ to create a new thread, we use operation $\text{terminate}$ to terminate an existed thread. $Ti$ is the local thread which $n$ belongs to. $Tr$ is the target thread on which $n$ operates, and $Tr$ will be set to $\text{null}$ if $n$ has no target thread. A simple PRG involved 3 threads is shown in figure 1 as our motivating example. Suppose we need to compute the MHP analysis results for node $n_3$, then intuitively, we will find out that in threads $t_2$ and $t_3$, all nodes which are reachable from $n_7$ but not dominated by $n_3$ may be executed in parallel with $n_3$. Moreover, the MHP results of $n_4$ and $n_5$ are the same with $n_3$, and thus, they can be simply computed by just propagating the MHP information from $n_3$. In another word, the MHP results of $n_4$ and $n_5$ are determined by $n_3$. Same rules can be applied to $n_9$ and $n_{10}$ in figure 1. The inspiration we get from this example is that different kinds of nodes have different influences on MHP analysis results. MHP information of certain nodes with concurrent syntax can be pre-computed by performing an elaborately designed inter-threaded reachability analysis, while the MHP information of nodes without concurrent syntax can be determined by this pre-computation. Therefore, there’s no need to iteratively process each statement according to the predefined data flow equations as in Naumovich’s approach[2] or exhaustively judge that whether each pair of statements in different threads may happen in parallel or not according to the predefined rules as in Barik’s approach[4].

Copyright is held by the author/owner(s).  
PACT’12, September 19-23, 2012, Minneapolis, Minnesota, USA.  
ACM 978-1-4503-1182-3/12/09.
2. DESIGN and IMPLEMENTATION

For the sake of computing and preserving MHP results, each node \( n \) in PRG have a set \( \text{DEL}(n) \) to preserve part of nodes which definitely cannot happen in parallel with \( n \), and a set \( M(n) \) to preserve all nodes may happen in parallel with \( n \). A point worth emphasizing here is that \( \text{DEL}(n) \) and \( M(n) \) are not complementary.

Based on the above definitions and taking the motivation in section 1 as the fundamental principle of our approach, we present a MHP algorithm including two phases in this paper:

1. Pre-computation. In this phase, we compute the initial \( M_{\text{init}} \) set and \( \text{DEL}_{\text{init}} \) set for certain nodes by performing an elaborately designed reachability analysis and dominating analysis on PRG.

   To compute \( \text{DEL}_{\text{init}} \) set for certain nodes, for each thread \( t \), if \( t \) is terminated by another thread, then get \( \text{terminatesucc}(t) \), and set \( \text{DEL}_{\text{init}}(\text{terminatesucc}(t)) \) to include all nodes in thread \( t \). At the same time, we also compute a node set comprises all nodes which are dominated by \( \text{terminatesucc}(t) \) in PRG, represented as \( \text{TerminateDom}(t) \). Notice that this process is recursive, which means if there is a thread \( t_i \) which satisfies that \( \text{end}(t_i) \in \text{TerminateDom}(t) \), then \( \text{TerminateDom}(t) \in \text{TerminateDom}(t_i) \). For each thread \( t \) has a creation site, get \( \text{spawn}(t) \) and \( \text{begin}(t) \), we compute a node set comprises all nodes which are not reachable from \( \text{spawn}(t) \) in thread \( t \), we define this node set as \( \text{DEL}_{\text{init}}(\text{begin}(t)) \).

   To compute \( M_{\text{init}} \) set for certain nodes, for each node labeled as \( <\text{spawn}(t), t, i> \), we compute all nodes dominated by \( \text{begin}(t) \) in PRG, and preserve them in node set \( LR(\text{begin}(t)) \) as the local reachable node set of \( \text{begin}(t) \). \( \text{TerminateDom} \) sets will be used in this process. We also compute a node set to comprise all nodes which are reachable from \( \text{begin}(t) \) in PRG, and preserve them in node set \( R(\text{begin}(t)) \) as the reachable set of \( \text{begin}(t) \). In a similar way, we get \( LR(\text{spawn}(t)) \) and \( R(\text{spawn}(t)) \) respectively.

   Finally, we compute \( M_{\text{init}}(\text{begin}(t)) \) and \( M_{\text{init}}(\text{spawn}(t)) \) as the following equations indicate:

\[
\text{M}_{\text{init}}(\text{spawn}(t)) := R(\text{begin}(t)) \setminus LR(\text{spawn}(t)) \\
\text{M}_{\text{init}}(\text{begin}(t)) := R(\text{spawn}(t)) \setminus LR(\text{begin}(t))
\]

2. Top-down propagation. In this phase, we propagate the initial \( M \) sets and \( \text{DEL} \) sets along PRG in a top-down way, and each node will be visited only once. During the propagation, we distinguish that whether a given PRG node contains the terminate operation or not. If current node \( n \) is such a node, then we compute its \( M \) set and \( \text{DEL} \) set according to the following equations:

\[
M(n) := \bigcup_{p: \text{psucc}(n)} M(p) \cup M_{\text{init}}(n) \\
\text{DEL}(n) := \bigcup_{p: \text{psucc}(n)} \text{DEL}(p) \cup \text{DEL}_{\text{init}}(n) \\
M(n) := M(n) \setminus \text{DEL}(n)
\]

The propagation of \( M \) sets is confined to the local predecessors of \( n \), since we do not merge \( M \) sets from different threads. Meanwhile, we eliminate \( \text{DEL}(n) \) from \( M(n) \) as the concurrent syntax of a terminate operation indicates. If \( n \) does not have terminate operation, then we use the following equations:

\[
M(n) := \bigcup_{p: \text{psucc}(n)} M(p) \cup M_{\text{init}}(n) \\
\text{DEL}(n) := \bigcap_{p: \text{psucc}(n)} \text{DEL}(p) \cup \text{DEL}_{\text{init}}(n)
\]

Finally, due to the concise reachability analysis and intuitive top-down propagation in above two phases, our approach achieves a low complexity \( O(|N|+|E|) \), in which, \( N \) is the number of PRG nodes, and \( E \) is the number of edges in PRG.

In our preliminary experiment, we compared our approach with two state-of-art approaches[2][4]. Since the primary program representation used in [2] is Program Execution Graph, and the program representation used in [4] is Thread Creation Tree, we represented these two approaches as PEG and TCT respectively in our experiment. We implemented both of these approaches in an open source compiler framework, and adopted the optimization techniques proposed in [3] to reduce the size of program representation. We selected 13 programs in SPLASH2[5] as our benchmark, and ran our experiment on a Linux server with a 1.8GHz CPU and 2GB memory. Experimental data of relative speed up (in logarithm) between different approaches is shown in figure 2.

![Figure 2: Relative speed up between PEG, TCT, PRG](image-url)

For all of programs in our experiment, our approach is extremely faster than both PEG approach and TCT approach, our approach gets a geometry average speed up of 395.53× compared with PEG approach and gets a geometry average speed up of 136.37× compared with TCT approach, while yielding the same precision with these two approaches. Due to this significant improvement on efficiency while holding the precision, our approach can make MHP analysis more practical and effective.

3. ACKNOWLEDGMENTS

This work is supported by several Chinese Research Grants. The grant numbers are including: 2011CB302504, 60921002, 2011ZX01028-001-002, 2012AA010902.

4. REFERENCES


