2010 Fourth International Symposium on Theoretical Aspects of Software Engineering

Stack Bound Inference for Abstract Java Bytecode

Shengyi Wang and Zongyan Qiu
LMAM and Department of Informatics
School of Math., Peking University
Beijing, China
{*wangshengyi,zyqiu*}*@pku.edu.cn*

Shengchao Qin
School of Computing
University of Teesside
Middlesbrough, UK
S.Qin@tees.ac.uk

Wei-Ngan Chin
Department of Computer Science
National University of Singapore
Singapore
chinwn@comp.nus.edu.sg

***Abstract*—Ubiquitous embedded systems are often resource-constrained. Developing software for these systems should take into account resources such as memory space. In this paper, we develop and implement an analysis framework to infer statically stack usage bounds for assembly-level programs in abstract Java Bytecode. Our stack bound inference process, extended from a theoretical framework proposed earlier by some of the authors, is composed of deductive inference rules in multiple passes. Based on these rules, a usable tool has been developed for processing programs to capture the stack memory needs of each procedure in terms of the symbolic values of its parameters. The final result contains path-sensitive information to achieve better precision. The tool invokes a Presburger solver to perform fixed point analysis for loops and recursive procedures. Our initial experiments have confirmed the viability and power of the approach.**

***Keywords*-Memory Inference, Program Analysis, Java Bytecode, Stack Boundary, Fixpoint Analysis, Tool**

I. Introduction

In the software verification community, the consideration of non-functional issues like resource adequacy and utilization is gradually gaining more attention. This trend has been driven by the proliferation of resource-constrained cyber-physical (embedded) systems, coupled with the high expectations on reliability and usability from consumers. Previous work in this area (amongst the real-time and embedded systems community) have mostly focused on real-time aspects, with major inroads made in WCET (worst-case execution time) domain. In this paper, we focus on stack memory as a constrained resource and analyze the stack space usage for low-level Java Bytecode-like programs.

Memory models for embedded systems are typically organized into two main parts: stack and heap. Stack is efficient for allocating and recovering memory spaces, and is particularly important for method invocations and transient data structures. Each method invocation typically reserves a frame of memory on the stack for holding local variables, etc. Heap is used for dynamically allocated data structures. For applications running in environments with very limited memory, such as smart cards or mobile devices, heap allocation is not allowed; therefore, stack space becomes the main concern. For such memory constrained applications, it is important to be fully aware of the stack space needed by each computational unit, and further, the whole program.

There are few previous studies for predicting symbolic memory usage of programs, especially for imperative programs in the Bytecode level. Works [1], [2] are mostly aiming to analyze functional programs where the immutable data structures make such analysis easier to formalize. The works [3], [4] target at Java-based Bytecode programs, but their frameworks again assume that Bytecode programs are compiled from functional programs without mutability and assignments. Other works, e.g. [5], [6], [7], merely provide frameworks for checking that memory usage of OO programs conform to user-supplied memory specifications either through static verification or runtime checking. However, user-annotations may be hard to provide and are likely to be impractical for assembly-level programs.

The previous work [8] by some co-authors of current paper has proposed a theoretical framework for analyzing memory usage bounds for low-level programs. It presents a multi-pass analysis framework to infer stack and heap bounds for a core assembly-like language. The power of the framework is that it does not require any annotation in programs, and further, working on a lower level may produce a more precise estimation. Our current work is based on this theoretical framework, but extends it or distinguishes from it in the following ways:

- We focus on the stack bound analysis and extend the previous framework for an enriched Java Bytecode-like language. For completeness, we will present both inference rules from previous work and newly proposed inference rules here.
- We focus more on the practicality of the approach and build a tool for the stack bound inference. The previous work focuses more on theoretical framework and does not have a fully workable system (apart from a preliminary prototype).
- We report challenges encountered during the implementation and solutions to tackle them. Initial experiments have further confirmed the viability of the approach.
- We have carried out experiments for many programs for the real estimations, and exercised our framework and the tool program.

978-0-7695-4148-8/10 $26.00 © 2010 IEEE
DOI 10.1109/TASE.2010.24

The rest of the paper is organized as follows. Section II introduces the abstract Java Bytecode language used in our work. Section III presents the analysis framework with three groups of rules which are used in three stages of the analysis. In Section IV we discuss some important implementation issues which are crucial for the tool. Two experiments by the tool are then illustrated in Section V, Additional related work is discussed in Section VI, followed by concluding remarks and future work.

II. The Abstract Java Bytecode Language

Our aim is to tackle the theoretical and practical challenges in memory bound inference for embedded programs, and to develop a practical tool for real use. In the work, we focus on an abstract (i.e., structured) version of Java Bytecode language, where all basic instructions are performed by a stack machine with effects on the stack. There is no concept of registers, and no explicit variables.

To facilitate the analysis and presentation, we include in the language conditional (with two branches) and while loop structures. In reality, low-level programs are generally organized as blocks of instructions and use (conditional) jumps moving between blocks. This block-level view does not cause major technical difficulty but may obscure our exposition. Moreover, it is helpful to recover higher-level language constructs when analyzing assembly-level codes, as advocated in [9].

Our language is close to Java Bytecode. In fact, all programs we have used to test our tool are bytecodes compiled from Java source but with manually recovered high-level control structures. In this work, we focus on the analysis framework and practical tool. A preprocessor for recovery of control structures from Java Bytecode is under development.

The syntax of the abstract Java Bytecode language is given in Figure 1. Compared with the language used in our previous work [8], the language has now included a set of common binary operators for arithmetic and comparison operations, which are indispensable for writing practical and useful programs with loops and recursive methods.

As shown in Figure 1, a program is composed of a sequence of method declarations. Each method declaration (M) comprises a return type (t), a method name (m), a list of parameter types($t_1, .., t_n$), and a method boy (E). An integer l between the signature and the method body denotes the number of words for storing both parameters and local values of the method. If l is larger than the space for storing the parameters, it indicates that the method requires space for its local values.

Values in the language are typed. For each type, we have a set of instructions. As said before, there is no explicit concept of *variable*. All values are stored in the stack, and copied explicitly by instructions `load` and `store` with a specification of offsets of the operands from the frame

$$
\begin{array}{rll}
P & ::= & M_1, M_2, \ldots, M_n & \text{(Program)} \\
M & ::= & t\ m(t_1, t_2, \ldots, t_n)\, l\, \{E\} & \text{(Method Decs)} \\
E & ::= & \mathit{Cmd} \mid \mathit{Binary} \mid E_1; E_2 \mid & \text{(Controls)} \\
 & & \texttt{if}\ E_1\ E_2 \mid \texttt{while}\ E & \\
\mathit{Cmd} & ::= & \texttt{load}\langle t\rangle\ i \mid \texttt{store}\langle t\rangle\ i \mid & \text{(Commands)} \\
 & & \texttt{invoke}\ m \mid \texttt{const}\langle t\rangle\ k & \\
\mathit{Binary} & ::= & \texttt{add}\langle t\rangle \mid \texttt{sub}\langle t\rangle \mid \texttt{mul}\langle t\rangle \mid & \text{(Bi-Op-Rels)} \\
 & & \texttt{div}\langle t\rangle \mid \texttt{mns}\langle t\rangle \mid & \\
 & & \texttt{gt}\langle t\rangle \mid \texttt{ge}\langle t\rangle \mid \texttt{lt}\langle t\rangle \mid \texttt{le}\langle t\rangle \mid \texttt{eq}\langle t\rangle & \\
t & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{float} \mid \texttt{ref} \mid \texttt{void} \mid \cdots & \text{(Types)}
\end{array}
$$

Figure 1. Syntax of Abstract Java Bytecode

pointer (FP) which always points to a fixed position of current frame in the stack. Instruction `load` copies a value from the designated location and push it on the top of the stack (to be determined by the stack pointer, SP), and `store` does the inverse action and pops the top. The $\texttt{const}\langle t\rangle\ k$ instruction pushes a constant k of type t onto the stack.

There is a rich set of binary operations (including relation operations) for each numeric type, where these operations take (and pop) two values from the stack and pushes the result back. Here $\texttt{mns}\langle t\rangle$ is unary minus instruction for numeric type t which modifies the top element of the stack (pops the top entry and then pushes the result).

For control structures, there are sequential and conditional, and `while` loop. The `invoke` m instruction invokes the method m. This instruction will create a new frame for method m, save and set the FP and SP pointers.

The frame of a method invocation is illustrated in Fig 2: The first two slots (-1, 0) in each frame contain pointers to the caller's frame and return address, above them ($1..\ell$) are the space for parameters and local values, and above them are the space for other temporary values (operands).

Taking a low level language as the target for analysis has at least two benefits. Firstly, the memory estimation can be more precise at this level, thus is more useful in practice. On the contrary, resource usage may be affected by optimizing compilers which might render memory analysis done at the source level unsafe to use. Secondly, programs written in a variety of higher-level languages are usually translated into to low-level forms before execution. Thus, this study can be used to serve a wider range of different languages.

A sample program is listed in Figure 3, with necessary comments as the explanation. We declare here a method `abc`, which has one integer parameter, where $l = 1$ means no local "variable" except the parameter. The main part of the method body is an `if` statement. In the second branch of the `if`, there is a recursive invocation to the method itself. We see many other instructions in the method, including

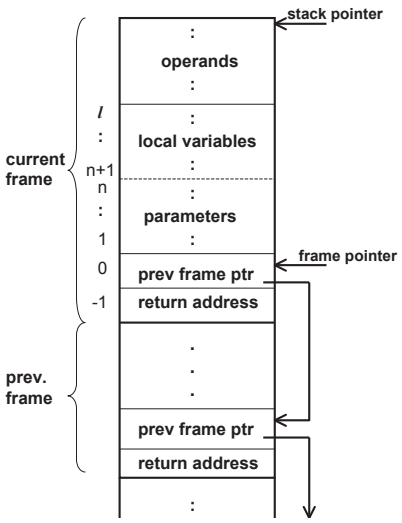

Figure 2. Stack Frames

```
int abc(int) 1 {
  load<int> 1; // push parameter to stack
  const<int> 0; // push 0 to stack
  ge<int>; // pop and compare top 2 values,
           // save the result
  if {
    load<int> 1; // if parameter <= 0,
                 // return the parameter
  } {
    load<int> 1; // push parameter to stack
    const<int> 1; // push 1 to stack
    sub<int>; // calculate (parameter - 1)
    invoke abc; // call abc(parameter - 1)
  };
}
```

Figure 3. A Sample Program

`load`, `store`, `const`, and some binary operations.

Our stack bound inference process aims to compute a safe and yet precise bound on the stack space required by the safe execution of programs such as `abc`. In what follows, we shall present the theoretical framework and discuss practical issues when implementing the framework.

III. Theoretical Foundation

Now we present the theoretical framework, which is an adapted and extended version of an earlier framework reported in [8].

Our stack boundary analysis is based on formal inference rules designed for predicting symbolic memory usages. In these rules, we use *Presburger arithmetic formulae* to describe states of programs. For method calls and loops, we may build up recursive predicates to specify memory usage in the form of constraint abstractions [10]. We resort to an external tool [11] to calculate fixed points for such constraint abstractions.

The inference process is divided into three key stages: (i) frame bound inference, (ii) abstract state inference, and (iii) stack bound inference. Each stage applies some inference rules according to program text. A later stage depends on results from the previous one(s). The first two stages make intraprocedural analyses while the last one conducts inter-procedural analysis. The final target of the inference process is to generate a set of annotations for each method declaration. Given a method:

$$t\ m(t_1, \ldots, t_n)\, \ell\, \{\ldots\}$$

Here $\{\ldots\}$ denotes method body. The inference will produce an extended declaration for it, with the form:

$$t\ m(t_1, \ldots, t_n)\, \ell; \phi_{pr}; \mathcal{F}; \phi_{po}; \mathcal{S}\, \{\ldots\}$$

where $\mathcal{F}$ is method m's frame bound, ϕ_{pr} its pre-condition, ϕ_{po} its post-condition, and $\mathcal{S}$ its stack bound.

Each method invocation reserves on stack a frame holding its parameters and local values until it returns. The stack may grow or shrink due to pushes/pops by instructions. For memory usage, there are usually two key metrics for each code unit: one is net usage between the start and end of the execution of the unit, the other is usage bound — the high watermark of memory usage at all points during the execution. As far as the stack is concerned, we only need to compute the latter for each method, since net stack usage at method boundary is always zero due to perfect space recovery for each call. To compute the stack bound, we keep the trace of stack usage at every possible program point so as to choose the maximum one as its high watermark.

As mentioned before, each stage of the analysis is carried out using a set of inference rules. In the rest part of this section, we present and discuss each set of the rules, according to the order of the work performed in analysis.

A. Frame Bound Inference

In first stage, we infer frame bound for each method in the program. Rules for this are listed in Figure 4. We have one rule for each basic instruction and control form. The rule (FBBiOp) describes how to deal with binary operations, and the rule (FBBiRel) covers all binary relations. Note that there is no rule for `mns` since it does not affect the frame size.

In the invocation, all real parameters, local values and temporaries for a method are placed in the method's frame. During the execution, the frame size may change frequently. To capture the upper bound of the size, we embed a top pointer (denoted by an offset) of current frame at each program point. In this stage we compute recursively all the frame top pointers and save them in company with the program code in a structure. This information will be used by analyses in later stages. In the implementation, we traverse

$$\frac{k :: t \quad \Gamma_1 = t : \Gamma}{l, \Gamma \vdash_F \texttt{const}\langle t\rangle\, k \rightsquigarrow (|\Gamma|, \texttt{const}\langle t\rangle k), \Gamma_1, |\Gamma_1|} \quad \text{(FBCon)}$$

$$\frac{i \le l \quad \Gamma[i] = t \quad \Gamma_1 = t : \Gamma}{l, \Gamma \vdash_F \texttt{load}\langle t\rangle\, i \rightsquigarrow (|\Gamma|, \texttt{load}\langle t\rangle i), \Gamma_1, |\Gamma_1|}$$

$$\frac{i \le l \le |\Gamma| \quad \Gamma_1 = \Gamma \oplus (i \mapsto t) \quad r = |\Gamma| + 1}{l, t : \Gamma \vdash_F \texttt{store}\langle t\rangle\, i \rightsquigarrow (r, \texttt{store}\langle t\rangle i), \Gamma_1, r}$$

$$\frac{l, \Gamma \vdash_F E_1 \rightsquigarrow A_1, \Gamma_1, \mathcal{F}_1 \quad l, \Gamma_1 \vdash_F E_2 \rightsquigarrow A_2, \Gamma_2, \mathcal{F}_2}{l, \Gamma \vdash_F E_1; E_2 \rightsquigarrow (|\Gamma|, A_1; A_2), \Gamma_2, \max(\mathcal{F}_1, \mathcal{F}_2)}$$

$$\frac{\begin{array}{c} l, \Gamma \vdash_F E_1 \rightsquigarrow A_1, \Gamma_1, \mathcal{F}_1 \quad l \le |\Gamma| \quad |\Gamma_1| = |\Gamma_2| \\ l, \Gamma \vdash_F E_2 \rightsquigarrow A_2, \Gamma_2, \mathcal{F}_2 \\ \mathcal{F}_3 = \max(\mathcal{F}_1, \mathcal{F}_2) \quad \Gamma_3 = \Gamma_1 \sqcup \Gamma_2 \end{array}}{l, \texttt{bool} : \Gamma \vdash_F \texttt{if}\, E_1\, E_2 \rightsquigarrow (|\Gamma| + 1, \texttt{if}\, A_1\, A_2), \Gamma_3, \mathcal{F}_3}$$

$$\frac{l \le |\Gamma| \quad l, \Gamma \vdash_F E \rightsquigarrow A, \texttt{bool} : \Gamma, \mathcal{F}}{l, \texttt{bool} : \Gamma \vdash_F \texttt{while}\, E \rightsquigarrow (|\Gamma| + 1, \texttt{while}\, A), \Gamma, \mathcal{F}}$$

$$\frac{\begin{array}{c} t\ m(t_1, \ldots, t_n) \cdots \{\cdots\} \in P \\ \Gamma = [t_n, \ldots, t_1] + \Gamma_1 \quad l \le |\Gamma_1| \\ \Gamma_2 = t : \Gamma_1 \quad \mathcal{F} = \max(|\Gamma|, |\Gamma_2|) \end{array}}{l, \Gamma \vdash_F \texttt{invoke}\, m \rightsquigarrow (|\Gamma|, \texttt{invoke}\, m), \Gamma_2, \mathcal{F}} \quad \text{(FBInv)}$$

$$\frac{l, [\top]_{i=n+1}^{l} + [t_n, \cdots, t_1] \vdash_F E \rightsquigarrow A, t : \Gamma, \mathcal{F} \quad |\Gamma| = l}{\vdash_F t\, m(t_1, \ldots, t_n) l\{E\} \rightsquigarrow t\, m(t_1, \ldots, t_n)\, l; \mathcal{F} + 2\ \{A\}} \quad \text{(FBMthd)}$$

$$\frac{\Gamma_1 = t : \Gamma \quad r = |\Gamma| + 2}{l, t : t : \Gamma \vdash_F BiOp\langle t\rangle \rightsquigarrow (r, BiOp\langle t\rangle), \Gamma_1, r} \quad \text{(FBBiOp)}$$

$$\frac{\Gamma_1 = \texttt{bool} : \Gamma \quad r = |\Gamma| + 2}{l, t : t : \Gamma \vdash_F BiRel\langle t\rangle \rightsquigarrow (r, BiRel\langle t\rangle), \Gamma_1, r} \quad \text{(FBBiRel)}$$

Figure 4. Rules for Frame Bound Inference

the abstract syntax tree (AST) and store all the information as decorations on the AST nodes.

All rules in this group have the form:

$$l, \Gamma \vdash_F E \rightsquigarrow A, \Gamma_1, \mathcal{F}$$

where l indicates size of the local values area in the frame, and Γ (respectively, Γ_1) captures the types of elements in current frame before (after) the execution of code fragment E. In addition, $\mathcal{F}$ denotes the high watermark of stack frame size inferred so far for E.

Γ is a list of types, where $\Gamma = [t_n, \ldots, t_1]$ means there are n elements in the frame, and the element at the stack top is of type t_n, the one at frame bottom is of type t_1. For each E with stack frame Γ, we embed its current top pointer $p = |\Gamma|$ into an intermediate form as $A = (p, E_A)$. If E is not a compound statement, E_A is the same as E. Otherwise we do this for components of E recursively.

The form of A is defined inductively as follows:

$$\begin{aligned} A &::= (p, E_A) \\ E_A &::= Cmd \mid A; A \mid \texttt{if}\, A\, A \mid \texttt{while}\, A \end{aligned} \quad (1)$$

In rules in Figure 4, we use $E :: t$ to state that E is of type t. Given $\Gamma = [t_n, \ldots, t_1]$, notation $t : \Gamma$ cons a type t to the head of Γ, thus $t : \Gamma = [t, t_n, \ldots, t_1]$. Here $+$ denotes sequence concatenation, $|\Gamma|$ represents the number of elements in Γ, and $\Gamma[i]$ retrieves the ith element of Γ. Expression $\Gamma \oplus (i \mapsto t)$ returns a sequence similar to Γ but with its ith element replaced by t. Function $\max(n_1, n_2)$ returns the maximum of n_1 and n_2, while function $\Gamma_1 \sqcup \Gamma_2$ computes the least upper bound of types over two sequences Γ_1 and Γ_2 which are of the same length.

We elaborate some rules as follows:

Rule (FBCon): The rule says, when k is of type t, after the execution of $\texttt{const}\langle t\rangle\, k$, the current frame becomes Γ_1 with the frame bound $|\Gamma_1|$. The effect of $\texttt{const}\langle t\rangle\, k$ is that it loads a constant of type t on top of the frame.

Rule (FBInv): From the premise, we know the first n slots of current frame Γ is $[t_n, \ldots, t_1]$. These slots just match the parameters of method m. After the execution of $\texttt{invoke}\, m$, all these n slots are popped out. The returning type t is added on top of frame Γ_1. The high water mark after this instruction is the maximum of $|\Gamma|$ and $|\Gamma_2|$

Rule (FBMthd): The rule is the main inference rule which may invoke all other rules. The premise tells the initial status of the frame and the conclusion tells it after the execution of the body of method m. The final frame bound is $\mathcal{F} + 2$, where 2 denotes the presence of the return address and a pointer to the previous frame.

Rule (FBBiOp): $BiOp$ represents binary operators like `add`, `sub`, etc. Before the execution, on top of the frame are two values of type t. The $BiOp$ pops out these two values and saves the result on top. This rule reflects the effect of such instructions on the frame.

B. Abstract State Inference

In second stage, we infer an abstract program state (via strongest post-condition reasoning in an abstract domain) for each program point. The abstract states used for fixed point analysis are expressed as Presburger formulae over values in the stack. The syntax of Presburger formulae is:

$$\begin{aligned} \phi &::= b \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \\ &\quad \exists n \cdot \phi \mid \forall n \cdot \phi && \text{(Presburger Formula)} \\ b &::= \texttt{true} \mid \texttt{false} \mid s = s \mid s < s \mid \\ &\quad s \le s \mid s > s \mid s \ge s && \text{(Boolean Expression)} \\ s &::= k \mid \pi_i \mid \pi_i' \mid k * s \mid s + s \mid -s \mid \\ &\quad \max(s, s) \mid \min(s, s) && \text{(Arithmetic Expression)} \end{aligned}$$

Here k denotes an integer and n denotes logical variable(s).

For recursive methods and loops, we need to build a constraint abstraction before applying fixed point analysis. The frame's top pointers generated in first stage are used here. Rules for this stage are given in Figure 5.

All rules in this group have the form:

$$\Delta \vdash_A A \rightsquigarrow B, \Delta_1$$

$$\frac{\Delta_1 = \Delta \circ_{\{\pi_i\}} \pi_i' = \pi_p'}{\Delta \vdash_A (p, \texttt{store}\langle t\rangle\, i) \rightsquigarrow (p, \Delta, \texttt{store}\langle t\rangle\, i), \exists \pi_p' \cdot \Delta_1}$$

$$\frac{\Delta_1 = \Delta \wedge \pi_{p+1}' = \pi_i'}{\Delta \vdash_A (p, \texttt{load}\langle t\rangle\, i), \Delta_1} \quad \text{(ASLod)}$$

$$\frac{\Delta \vdash_A A_1 \rightsquigarrow B_1, \Delta_1 \quad \Delta_1 \vdash_A A_2 \rightsquigarrow B_2, \Delta_2}{\Delta \vdash_A (p, A_1; A_2) \rightsquigarrow (p, \Delta, B_1; B_2), \Delta_2}$$

$$\frac{\begin{array}{c}\exists \pi_p' \cdot (\Delta \wedge \pi_p' = 1) \vdash_A A_1 \rightsquigarrow B_1, \Delta_1 \\ \exists \pi_p' \cdot (\Delta \wedge \pi_p' = 0) \vdash_A A_2 \rightsquigarrow B_2, \Delta_2\end{array}}{\Delta \vdash_A (p, \texttt{if}\, A_1\, A_2) \rightsquigarrow (p, \Delta, \texttt{if}\, B_1\, B_2), \Delta_1 \vee \Delta_2}$$

$$\frac{\begin{array}{c}\bigwedge_{i=1}^{p-1} \pi_i' = \pi_i \vdash_A A \rightsquigarrow B, \Delta_1 \quad \textit{fresh}\ r_1, \ldots, r_{p-1} \\ \Delta_a = \pi_p' = 0 \wedge \bigwedge_{i=1}^{p-1} r_i = \pi_i' \vee \pi_p' = 1 \wedge \alpha(\pi_1' \ldots \pi_{p-1}', r_1 \ldots r_{p-1}) \\ \phi_{rec} = \{\alpha(\pi_1, \ldots, \pi_{p-1}, r_1, \ldots, r_{p-1}) = \Delta_1 \wedge \Delta_a\} \\ \rho = [r_i \mapsto \pi_i'] \quad \Delta_{post} = \textit{fixpt}(\phi_{rec}) \\ \Delta_2 = \quad (\exists \pi_p' \cdot \Delta \wedge \pi_p' = 0) \vee \\ \qquad ((\exists \pi_p' \cdot \Delta \wedge \pi_p' = 1) \circ_{\{\pi_1 .. \pi_{p-1}\}} \rho \Delta_{post})\end{array}}{\Delta \vdash_A (p, \texttt{while}\, A) \rightsquigarrow (p, \Delta, \texttt{while}\, B\, \Delta_1), \Delta_2}$$

$$\frac{\begin{array}{c}t\, m(t_{1..n})l; \phi_{pr}; \mathcal{F}; \phi_{po}; \{\cdots\} \in P \quad \textit{fresh}\ r \\ \rho = [\pi_i \mapsto \pi'_{p-n+i}]_{i=1}^{n} \cup [\pi'_{l+1} \mapsto r] \quad \Delta \Rightarrow \rho \phi_{pr} \\ \Delta_1 = (\exists \pi'_{p-n+1} .. \pi_p' \cdot \Delta \wedge \rho \phi_{po}) \wedge (\pi'_{p-n+1} = r)\end{array}}{\Delta \vdash_A (p, \texttt{invoke}\, m), \exists r \cdot \Delta_1}$$

$$\frac{\begin{array}{c}\Delta = \bigwedge_{i=1}^{n} \pi_i' = \pi_i \quad \Delta \vdash_A A \rightsquigarrow B, \Delta_1 \\ \phi_{rec} = \{m(\pi_1, \ldots, \pi_n, \pi'_{l+1}) = \Delta_1\} \\ \phi_{pr} = \textit{prefixpt}(\phi_{rec}) \quad \phi_{po} = \textit{fixpt}(\phi_{rec})\end{array}}{\vdash_A t\, m(t_{1..n})l; \mathcal{F}\{A\} \rightsquigarrow t\, m(t_{1..n})l; \phi_{pr}; \mathcal{F}; \phi_{po}\{B\}}$$

$$\frac{\Delta_1 = \Delta \circ_{\{\pi_{p-1}\}} \pi'_{p-1} = \pi_{p-1} + \pi'_p}{\Delta \vdash_A (p, \texttt{add}\langle t\rangle) \rightsquigarrow (p, \Delta, \texttt{add}\langle t\rangle), \exists \pi'_p \Delta_1} \quad \text{(ASAdd)}$$

$$\frac{\Delta_1 = \Delta \circ_{\{\pi_{p-1}\}} \pi'_{p-1} = \pi_{p-1} - \pi'_p}{\Delta \vdash_A (p, \texttt{sub}\langle t\rangle) \rightsquigarrow (p, \Delta, \texttt{sub}\langle t\rangle), \exists \pi'_p \Delta_1}$$

$$\frac{\Delta_1 = \Delta \circ_{\{\pi_p\}} \pi'_p = -\pi_p}{\Delta \vdash_A (p, \texttt{mns}\langle t\rangle) \rightsquigarrow (p, \Delta, \texttt{mns}\langle t\rangle), \exists \pi'_p \Delta_1}$$

$$\frac{\Delta_1 = \Delta \circ_{\{\pi_{p-1}\}} \quad ((\pi'_{p-1} = 1 \wedge \pi'_p > \pi_{p-1}) \vee (\pi'_{p-1} = 0 \wedge \pi'_p \leq \pi_{p-1}))}{\Delta \vdash_A (p, \texttt{gt}\langle t\rangle) \rightsquigarrow (p, \Delta, \texttt{gt}\langle t\rangle), \exists \pi'_p \Delta_1}$$

$$\frac{\Delta_1 = \Delta \circ_{\{\pi_{p-1}\}} \quad ((\pi'_{p-1} = 1 \wedge \pi'_p \geq \pi_{p-1}) \vee (\pi'_{p-1} = 0 \wedge \pi'_p < \pi_{p-1}))}{\Delta \vdash_A (p, \texttt{ge}\langle t\rangle) \rightsquigarrow (p, \Delta, \texttt{ge}\langle t\rangle), \exists \pi'_p \Delta_1}$$

$$\frac{\Delta_1 = \Delta \circ_{\{\pi_{p-1}\}} \quad ((\pi'_{p-1} = 1 \wedge \pi'_p < \pi_{p-1}) \vee (\pi'_{p-1} = 0 \wedge \pi'_p \geq \pi_{p-1}))}{\Delta \vdash_A (p, \texttt{lt}\langle t\rangle) \rightsquigarrow (p, \Delta, \texttt{lt}\langle t\rangle), \exists \pi'_p \Delta_1}$$

$$\frac{\Delta_1 = \Delta \circ_{\{\pi_{p-1}\}} \quad ((\pi'_{p-1} = 1 \wedge \pi'_p \leq \pi_{p-1}) \vee (\pi'_{p-1} = 0 \wedge \pi'_p > \pi_{p-1}))}{\Delta \vdash_A (p, \texttt{le}\langle t\rangle) \rightsquigarrow (p, \Delta, \texttt{le}\langle t\rangle), \exists \pi'_p \Delta_1}$$

$$\frac{\Delta_1 = \Delta \circ_{\{\pi_{p-1}\}} \quad ((\pi'_{p-1} = 1 \wedge \pi'_p = \pi_{p-1}) \vee (\pi'_{p-1} = 0 \wedge (\pi'_p < \pi_{p-1} \vee \pi'_p > \pi_{p-1})))}{\Delta \vdash_A (p, \texttt{eq}\langle t\rangle) \rightsquigarrow (p, \Delta, \texttt{eq}\langle t\rangle), \exists \pi'_p \Delta_1}$$

Figure 5. Rules for Abstract State Inference

Here Δ is a Presburger formula over values in current frame $[\pi_p, \ldots, \pi_1]$ of the stack where each π_i is a value at location i in the frame, and A has the form as defined in (1).

In the rules, we use π_i to denote original value in current frame at location i, and π_i' the latest value at the same location. We use Δ (resp., Δ_1) to represent the abstract state before (after) the evaluation of A. Note that input A is an expression annotated with top frame pointers. We obtain B expressions from the As by inserting the corresponding abstract state into each program point.

The form of B expressions is inductively defined as:

$$\begin{aligned} B &::= (p, \Delta, E_B) \\ E_B &::= Cmd \mid B; B \mid \texttt{if}\, B\, B \mid \texttt{while}\, B\, \Delta_1 \end{aligned} \quad (2)$$

The *fixpt*(ϕ) and *prefixpt*(ϕ) in the rules are the fixed point and pre-fixed point of logic formula ϕ, respectively. For example, given a recursive foumula:

$$rec(n, r) = n \leq 0 \wedge r = 1 \vee rec(n - 1, r - 2)$$

Its fixed point is:

$$rec(n, r) = (n \leq 0 \wedge r = 1) \vee (n > 0 \wedge r = 2n + 1)$$

Given an existing state Δ (represented by a Presburger formula) and another Presburger formula ϕ representing changes to the state, whereby $X = \{x_1, \ldots, x_n\}$ is the set of variables to be updated, the composition operator $\circ_X$ is defined as follows:

$$\Delta \circ_X \phi =_{df} \exists\, r_1 \ldots r_n \cdot \rho_2 \Delta \wedge \rho_1 \phi$$

where $r_1, \ldots, r_n$ are fresh variables, and

$$\rho_1 = [x_i \mapsto r_i]_{i=1}^{n}, \text{ and } \rho_2 = [x_i' \mapsto r_i]_{i=1}^{n}$$

Here ρ_1 and ρ_2 are substitutions, and $\rho_2\Delta$ and $\rho_1\phi$ represent the applications of them to Δ and ϕ, respectively.

For example:

$$\begin{array}{ll} & (\pi_1' = \pi_1 \wedge \pi_5' = \pi_1 + 2) \circ_{\{\pi_1\}} (\pi_1' = \pi_5') \\ \equiv & \exists r \cdot r = \pi_1 \wedge \pi_5' = \pi_1 + 2 \wedge \pi_1' = \pi_5' \end{array}$$

Assuming that Δ is $\pi_1' = \pi_1 \wedge \pi_5' = \pi_1 + 2$ and the top frame pointer $p = 5$, after the execution of $\texttt{store}\langle int\rangle\, 1$, from the rule in Figure 5, we have

$$\exists \pi_5', r \cdot r = \pi_1 \wedge \pi_5' = \pi_1 + 2 \wedge \pi_1' = \pi_5'$$

Logically, it is equivalent to $\pi_1' = \pi_1 + 2$. The rule captures the effect of `store` instruction: the value (π_1+2) on top of the stack frame (slot 5) is stored into the slot 1.

We elaborate some rules as follows:

Rule (ASLod): Because the stack pointer before $\texttt{load}\langle t\rangle\, i$ is at position p (with respect to the FP), after execution of

$$\frac{I = Cmd \mid BiOpRels}{a \vdash_S (p, \Delta, I) \rightsquigarrow \{\}} \quad \text{(SBBin)}$$

$$\frac{a \vdash_S B_1 \rightsquigarrow \mathcal{S}_1 \quad a \vdash_S B_2 \rightsquigarrow \mathcal{S}_2}{a \vdash_S (p, \Delta, B_1; B_2) \rightsquigarrow \mathcal{S}_1 \cup \mathcal{S}_2}$$

$$\frac{a \vdash_S B_1 \rightsquigarrow \mathcal{S}_1 \quad a \vdash_S B_2 \rightsquigarrow \mathcal{S}_2}{a \vdash_S (p, \Delta, \texttt{if}\ B_1\ B_2) \rightsquigarrow \mathcal{S}_1 \cup \mathcal{S}_2}$$

$$\frac{\begin{array}{l} a \vdash_S B \rightsquigarrow \mathcal{S} \\ \mathcal{S}_{rec} = \{\alpha(\pi_1, \ldots, \pi_{p-1}) = \mathcal{S} \cup \\ \qquad enrich(p-1, \Delta_1 \wedge \pi'_p = 1, \alpha(\pi'_1, \ldots, \pi'_{p-1}))\} \end{array}}{\begin{array}{l} a \vdash_S (p, \Delta, \texttt{while}\ B\ \Delta_1) \rightsquigarrow \\ \qquad enrich(a, \Delta \wedge \pi'_p = 1, \mathit{fixpt}(\mathcal{S}_{rec})) \end{array}}$$

$$\frac{\begin{array}{l} t\, m_1(t_1, \ldots, t_n) l; \phi_{pr}; \mathcal{F}; \phi_{po}; \mathcal{S}\{B\} \quad r = p - n + 2 \\ \rho = [\pi_i \mapsto \pi'_{p-n+i}]_{i=1}^{n} \quad \mathcal{S}_1 = enrich(a, \Delta, \rho\mathcal{S}) + r \end{array}}{a \vdash_S (p, \Delta, \texttt{invoke}\, m_1) \rightsquigarrow \mathcal{S}_1}$$

$$\frac{\begin{array}{c} n \vdash_S B \rightsquigarrow \mathcal{S} \\ \mathcal{S}_{rec} = \{m(\pi_1, \ldots, \pi_n) = \mathcal{S} \cup \{\phi_{pr} \rightarrow \mathcal{F}\}\} \\ \mathcal{S}_\mu = \mathit{fixpt}(\mathcal{S}_{rec}) \end{array}}{\begin{array}{l} \vdash_S t\, m(t_{1..n}) l; \phi_{pr}; \mathcal{F}; \phi_{po}\{B\} \rightsquigarrow \\ \qquad t\, m(t_{1..n}) l; \phi_{pr}; \mathcal{F}; \phi_{po}; \mathcal{S}_\mu\{B\} \end{array}}$$

Figure 6. Rules for Stack Bound Inference

`load`, the stack pointer moves one slot up, and value in slot $p+1$ (offset from the FP) is equal to the value in slot i after.

Rule (ASAdd): Current frame top is at p, after the execution, two values at the top are popped out and the result is put on the top. Thus π'_{p-1} is the final result of the addition where π_{p-1} and π'_p denotes two values at the top of the stack before execution.

We treat `while` loops as a special form of methods. In the constraint abstraction $\alpha(\pi_1, \ldots, \pi_{p-1}, r_1, \ldots, r_{p-1})$, $r_1, \ldots, r_{p-1}$ are used to denote the outputs for input parameters $\pi_1, \ldots, \pi_{p-1}$. This is captured by the abstraction ϕ_{rec} built from Δ_1 (which captures the post-state of A) and Δ_a (which captures the effect of conditional prior to termination or loop).

C. Stack Bound Inference

The frame bound inference limits all effects of primitive operations to caller's local frame. We must also analyze how method invocations affect the stack space. As discussed earlier, because the space on stack is reclaimed perfectly when the method returns, the net stack usage is always zero for each method invocation, thus, we only need to infer stack bound. Rules for this inference are given in Figure 6.

In the stack bound inference, we use guarded expression of the form $\{g_i \rightarrow \mathcal{B}_i\}_{i=1}^{n}$ in performing path-sensitive analysis for conditional branches, where each g_i is a guard and the corresponding $\mathcal{B}_i$ is the stack size under this guard (under this condition). A guard g_i is expressed in terms of the parameters of the method, thus the result can give precise estimations according to actual application scenarios.

All the stack inference rules have the following form:

$$a \vdash_S B \rightsquigarrow \mathcal{S}$$

where a is the number of current method's parameters, and B is the result produced by prior analysis stages (with the form as defined in previous subsection), which has two additional components: a top frame pointer, and an abstract state inserted. The inferred result $\mathcal{S}$, produced by this group of rules, denotes the high watermark of stack usage encountered during the execution of B. Function $enrich(a, \Delta, \mathcal{S})$ incorporates path-sensitive guarded formula $\mathcal{S}$ into current abstract state Δ, as follows:

$$enrich(a, \Delta, \mathcal{S}) =_{df} \{\exists \pi_{a+1} \ldots \cdot \Delta \wedge g \rightarrow s \mid (g \rightarrow s) \in \mathcal{S}\}$$

The guarded formulae used in the rules are built from two operators, namely $\cup$ (for upper bound) and + (for summation). Both operators are associative and commutative, with + distributing over $\cup$. The guarded formulae can be simplified by the following normalization rules:

$$\begin{array}{rl} \{\texttt{false} \rightarrow s\} & \Rrightarrow \{\} \\ \{p_1 \rightarrow s_1\} \cup \{p_2 \rightarrow s_2\} & \Rrightarrow \{p_1 \wedge p_2 \rightarrow \max(s_1, s_2)\} \cup \\ & \qquad \{p_1 \wedge \neg p_2 \rightarrow s_1\} \cup \\ & \qquad \{\neg p_1 \wedge p_2 \rightarrow s_2\} \\ \{p_1 \rightarrow s_1\} + \{p_2 \rightarrow s_2\} & \Rrightarrow \{p_1 \wedge p_2 \rightarrow s_1 + s_2\} \end{array}$$

So the `if` rule in stack bound inference just means the final bound of `if` is the upper bound of its two branches.

In each stage of the inferences, the algorithm adds annotations to each program point recursively so as to exploit the rules. Finally we can obtain the stack bound estimation for all the methods declared in the program.

IV. Challenges for Practical Implementation

There is often a huge gap between theory and practice. As we work on an implementation for the estimation framework described in Section III, we have to make many design decisions which would affect the final result of the work. We show some of the challenging issues and the way we have chosen to tackle them.

A. Recursion and Loop

The analysis of recursive methods and loops is always a challenge because we cannot know in general how many times some code segment will execute before running. This is why fixed point analysis matters. The aforementioned rules do not cover all aspects of our programming language. Especially, there is no explicit rule for recursive function calls, not to mention mutual recursive function calls.

To make the problem clear, we define first a concept "simple program", which is a program satisfying the two conditions: Firstly, each group of mutual recursive methods in the program contains only one method, either it is recursive itself or not. Secondly, in any method, there is

no invocation to the method itself in a loop. As we can see before, in the last two inference stages, the rule for loops is only a special form of the rule for method invocation. In fact, the two conditions for simple program is actually one: there are not two or more methods which call one another, i.e., no real mutual-recursion presents in the program.

For "simple programs", rules for recursion are natural to define. The rule for abstract state inference is:

$$\frac{\begin{array}{c} t\ m(t_{1..n})l; \phi_{pr}; \mathcal{F}; \phi_{po}; \{\cdots\} \in P \quad \textit{fresh}\ r \\ \rho = [\pi_i \mapsto \pi'_{p-n+i}]_{i=1}^{n} \cup [\pi'_{l+1} \mapsto r] \quad \Delta \Rightarrow \rho\phi_{pr} \\ \Delta_1 = \ (\exists \pi'_{p-n+1}..\pi'_p \cdot \Delta \wedge \rho m(\pi_1, \ldots, \pi_n, r)) \\ \qquad \wedge\ (\pi'_{p-n+1} = r) \end{array}}{\Delta \vdash_A (p, \texttt{invoke}\, m), \exists r \cdot \Delta_1}$$

We have another rule for self-recursive methods in stack bound inference. Our tool implemented these two rules, so it can deal with self-recursive methods correctly.

For mutual recursive cases, these rules are not applicable. However, this difficulty can still be solved if we sacrifice some precision. For two methods which call each other, we can combine them into one new method which contains both methods' bodies in different branches of an `if` statement. Then we extend the parameter list to accommodate two original methods, add another parameter indicating which branch is used, and modify method invocations accordingly. This new method is self recursive, thus can be handled by our rules. Because the parameter list is extended, the final estimation on stack usage would be larger than actual one. This rule has not been implemented yet in our current system.

B. Formulae Construction and Simplification

Before starting the work, we do not think that the simplification is crucial to the success of the analysis tool, because the fixed point calculator can handle any complex boolean formula, and what we need is just to care about the formation of the formulae, rather than manipulating them. However, several problems present that make the formulae manipulation and simplification indispensable.

1) Introducing Fresh Variables: As mentioned above, our rules may produce formulae such as

$$\exists \pi'_2, r \cdot r = \pi_1 \wedge \pi'_2 = \pi_1 + 2 \wedge \pi'_1 = \pi'_2$$

un-simplified. For a single formula, there is no problem. However, when we compose a formula from some small ones, we need to gather all bound variables together. When a bound variable appears in different sub-formulae, these formulae can not be put together directly. To handle it correctly, when we need to add a new sub-formula, we determine all the bound variables in the new formula, then replace every occurrence of these variables in the existing formula to fresh ones, then add the new sub-formula with the original variable unchanged.

2) Removing Negation, Simplifying Formulae: Another problem is more challenging. As mentioned above, in the stack inference, we use guarded expression of the form $\{g_i \rightarrow \mathcal{B}_i\}_{i=1}^{n}$ for path-sensitive analysis of conditional branches. The inference rules need to find the fixed point of these guarded forms. However, current external fixed point calculator can only handle Presburger formulae, thus we must convert the guarded assertions to Presburger formulae.

Intuitively, we may think that $\{g_i \rightarrow \mathcal{B}_i\}_{i=1}^{n}$ can be transformed directly to $\bigvee_{i=1}^{n}(g_i \wedge \mathcal{B}_i)$. Unfortunately, this direct transformation is wrong, because in fact, $\{p_1 \rightarrow s_1\} \cup \{p_2 \rightarrow s_2\}$ is equivalent to $(p_1 \wedge p_2 \rightarrow \max(s_1, s_2)) \vee (p_1 \wedge \neg p_2 \rightarrow s_1) \vee (\neg p_1 \wedge p_2 \rightarrow s_2)$. This transformation introduces negation into the formulae, but the fixed point calculator can not handle negation directly.

To overcome this problem, we need to know the exact meaning of the guard condition g_i so as to present $\neg g_i$ in an explicit form without negation. Our implementation uses an "ad hoc" boolean expression simplifier. We do a "two-step" simplification as follows.

In the first step we simplify the "disjunction" formula. When an `if` statement appears in the program, there must be a comparison instruction before it. According to the rule, the resulting formula will have the form:

$$(\pi'_p = 0 \wedge \Delta) \vee (\pi'_p = 1 \wedge \neg\Delta)$$

All the "$\vee$"s in formulae are introduced by this construction step. In each branch of `if` statement, there is a $\pi'_p = 1$ or $\pi'_p = 0$ according to the inference rule. So the formula in each branch may look like this:

$$((\pi'_p = 0 \wedge \Delta) \vee (\pi'_p = 1 \wedge \neg\Delta)) \wedge \cdots \wedge (\pi'_p = 1) \wedge \ldots$$

To simplify formulae of this form, we look for the closest equation related to π'_p outside the "disjunction" in the formula. According to the value of π'_p, the formula can be either Δ or $\neg\Delta$. For above formula, the final result is $\neg\Delta$.

After careful observation, we determine that after the first step of simplification, the resulting Presburger formula will be always in the conjunctive normal form, where each atomic formula is an equation. So in the second step, we try to solve the equations, in order to remove all unnecessary variables, and shorten the formula as much as possible. We reduce variables under each $\exists$ one by one. For example, after this simplification, formula $\exists r \cdot \pi_1 = r \wedge \pi_2 = 3 + r$ becomes $\pi_1 = \pi_2 - 3$.

3) Positions of Function Abstraction: The simplifier described above is not sufficient for the transformation from guarded form to Presburger formula. When the function is recursive, the function abstraction appears in some $\mathcal{B}_i$ part of the guarded form. In this situation, the simplifier cannot work correctly.

For example, simple treatment may put a function abstraction $\alpha(\pi'_1, \pi'_2 + 3)$ with symbols π'_1, π'_2 in a improper

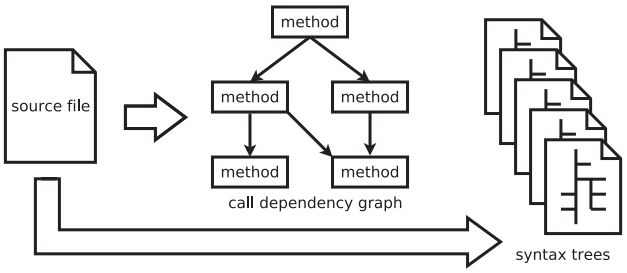

Figure 7. Preprocessing

position, and delivers a wrong formula, e.g.,

$$(\exists \pi_1', \pi_2' \cdot \pi_1' = \pi_1 - 1 \wedge \pi_2' = \pi_1' - 1) \wedge \alpha(\pi_1', \pi_2' + 3)$$

where bound variables π_1' and π_2' in α is not in the scope of $\exists$. To fix this problem, we added another procedure to seek the right position for function abstractions.

In fact, a function abstraction is in the right place at the stage of abstract state inference. Our improved algorithm marks its place at first and puts the function abstraction in $\mathcal{B}_i$ in stack bound inference. When we convert a guarded form to Presburger formula, we insert the function abstraction to its original place with necessary modifications. For example, if the original formula to handle is:

$$\exists \pi_1', \pi_2' \cdot (\pi_1' = \pi_1 - 1 \wedge \pi_2' = \pi_1' - 1 \wedge \alpha(\pi_1', \pi_2')).$$

According to the rules in Figure 6, the guarded form for this formula will be

$$(\exists \pi_1', \pi_2' \cdot \pi_1' = \pi_1 - 1 \wedge \pi_2' = \pi_1' - 1) \rightarrow \alpha(\pi_1', \pi_2) - 3.$$

After our modified transformation, the right result is

$$\exists \pi_1', \pi_2' \cdot (\pi_1' = \pi_1 - 1 \wedge \pi_2' = \pi_1' - 1 \wedge \alpha(\pi_1', \pi_2', \pi_3 + 3)).$$

Here, π_3 denotes the return result. The last two α in the formulae have different meaning. In guarded form, it represents an arithmetic result, but in the final formula, it recursively represents the whole abstraction of the formula. The arithmetic result now is represented by π_3.

V. Tool and Experiment

Our tool program is written in C++. We use the *Spirit Framework* in *boost library* for two parsers, one for the structured Java Bytecode language, and one for logic formulae coming from the fixed point calculator invoked by our analyzer. The main part of the tool makes the stack estimation by traversing the AST of programs for several times, and making the inference for each program point. The algorithm is divided into two main phases. One is the preprocessing, the other is the concrete inference for each method. Of course, we assume programs to analyze are well-typed, that is guaranteed by the compilers.

A. Preprocessing and Estimation

Any source file written in the structured Java bytecode language needs to be preprocessed before it is sent to the analysis algorithm. During parsing, we build also a call dependency graph for the program, where each strongly connected component represents a set of mutual recursive methods for analysis simultaneously. Of course, for "simple program", each node contains only one method.

By abstracting these connected components as vertices, we get a directed acyclic graph (DAG). A topological order of the DAG determines an inference order which ensures that inter-procedural analysis can be carried with prior known properties of called methods. Each of the DAG node is then parsed into a syntactic tree for subsequent analyses. The whole process is shown in Figure 7.

After the preprocessing, the analysis stages are carried on to produce the stack bounds of the program.

We have conducted a number of experiments with the tool we build. We shall now present two of them to demonstrate the viability and power of our method.

B. Example 1

We take the program in Figure 3 as the first example. The program contains one method with a single parameter, and has the following behavior: when the parameter is less than one, it returns the value of the parameter; otherwise it decreases the parameter by one and calls itself again. This method may be of little practical value, but it does present a general recursive pattern, where many useful methods work in the same manner (e.g., the factorial function).

Our tool produces a guarded formula which expresses the stack usage estimation as follows (We rewrite it to a more readable form here). The parameter is denoted by π_1. The memory usage is denoted by $\text{memg}(\pi_1)$.

$$\text{memg}(\pi_1) = \{\pi_1 < 1 \rightarrow 5\} \cup \{\pi_1 \geq 1 \rightarrow \text{memg}(\pi_1 - 1) + 3\}$$

The fixed point calculator can not handle guarded formulae, so our tool converts the above formula to Presburger form. The memory usage is denoted by π_2 and the whole formula is denoted by $\text{memp}(\pi_1, \pi_2)$.

$$\text{memp}(\pi_1, \pi_2) = \quad (\pi_1 < 1 \wedge \pi_2 = 5) \vee ((\pi_1 \geq 1) \wedge \text{memp}(\pi_1 - 1, \pi_2 - 3))$$

We then get the ideal result from the fixed point calculator:

$$(\pi_1 \leq 0 \wedge \pi_2 = 5) \vee (\pi_1 \geq 1 \wedge \pi_2 = 3\pi_1 + 5)$$

This is the fixed point of the recursive Presburger formula $\text{memp}(\pi_1, \pi_2)$, which means: when the actual parameter is less than or equal to 0, the stack bound is 5; but if the actual parameter is greater than or equal to 1, the stack bound is three times of the actual parameter and plus 5.

```
int pow(int base, int index) {
  if (index <= 0) {
    return 1;
  } else {
    return base * pow(base, index - 1);
  }
}
int sum(int from, int to, int index) {
  int sum = 0;
  for (int i = from; i <= to; i++) {
    sum += pow(i, index);
  }
  return sum;
}
----------------------------------------
int pow (int, int) 2 {
    load<int> 2;
    const<int> 0;
    ge<int>;
    if {
       const<int> 1;
    } {
      load<int> 1;
      load<int> 1;
      load<int> 2;
      const<int> 1;
      sub<int>;
      invoke pow;
      mul<int>;
    };
}
int sum(int, int, int) 5 {
    const<int> 0;
    store<int> 4;
    load<int> 1;
    store<int> 5;
    load<int> 5;
    load<int> 2;
    ge<int>;
    while {
       load<int> 4;
       load<int> 5;
       load<int> 3;
       invoke pow;
       add<int>;
       store<int> 4;
       const<int> 1;
       add<int>;
       load<int> 5;
       load<int> 2;
       ge<int>;
    };
    load<int> 4;
}
```

Figure 8. A Larger Program

C. Example 2

Figure 8 shows a bigger program. The high-level code is shown in the upper part, and the bytecode version is in the lower part. The program contains two methods: `pow` calculates b^i where b is the value of parameter `base` and i is the value of parameter `index`. Method `sum` invokes `pow`, and calculates $\sum_{j=\texttt{from}}^{\texttt{to}} j^i$ where i is the value of parameter `index`. The Bytecode version is rather lengthy.

For method `pow`, the final stack space estimation by our tool is (formatted to mathematical formula):

$$(\pi_2 \leq 0 \wedge \pi_3 = 8) \vee (\pi_2 \geq 1 \wedge \pi_3 = 5\pi_2 + 8)$$

where π_3 is the stack space estimation and π_2 denotes the second parameter of `pow`. From the high-level code, we can see that parameter `index` affects the memory usage.

For method `sum`, the final stack space estimation is:

$$(\pi_3 \leq 0 \wedge \pi_4 = 10) \vee (\pi_3 \geq 1 \wedge \pi_4 = 5\pi_3 + 10)$$

where π_4 is the stack space estimation and π_3 denotes the third parameter of `sum`. From the high-level code, It is `index` which affects the memory usage.

VI. Related Work

Here we discuss some recent related work, except for those we have mentioned in the introduction section.

The work [12] presents a fully automatic static type-based analysis for inferring upper bounds on resource usage for functional programs involving general algebraic data types and full recursion. The work [13] does a similar analysis where the stack bounds are given as the max-plus expressions on the depth of data structures. This analysis is still for functional programs. The other work [14] presents a technique to compute symbolic polynomial approximations of the amount of heap memory required to safely execute a method without running out of memory, for Java-like imperative programs. The more recent work [15] can determine upper-bound functions on the use of quantitative resources for strict, higher-order, polymorphic, recursive functional programs dealing with possibly-aliased data. As a closely related work, [16] demonstrates how to use a static analysis system COSTA to obtain safe symbolic upper bounds on the resource (heap) usage of a large class of general-purpose programs written in mainstream programming languages, such as Java (bytecode). However, it does not study stack bound.

As a summary, most of the work focuses on the memory estimation of functional languages, with an exception being the work [14] which focuses on the heap usage.

VII. Conclusion

Resource estimation is very important for cyber-physical (embedded) systems, because many such systems are highly resource limited. Existing work in this area have mostly focused on real-time aspects, especially the WCET (worst-case execution time) estimation. In this paper, we focus on the stack memory as a constrained resource and analyze the stack boundary for low-level Java Bytecode-like programs.

We define, firstly, an abstract Java Bytecode language which captures the spirit of low-level languages on stack machines. All the basic instructions have their effects on a stack. For abstraction, we include in the language control structures including conditional branch and while loop. However, the language is close enough to Java Bytecode, to rebuild the structural control flow from Java Bytecode program is straightforward in most cases.

The analysis work is divided into three inference stages, each of them infers and collects some analysis results. The inference stages are: *Frame Bound Inference* which determines for each method the maximum size of its stack frame, *Abstract State Inference* which infers abstract state for each program point and represents it as a Presburger formula, *Stack Bound Inference* which gives the final results. We give three sets of rules for these inferences, and explain some of them for better understanding. For implementing the rules, our tool program traverses the AST of each method, and makes decorations on AST nodes to record the information which has been found. After running these analyses in sequence, we have all the necessary information for the stack usage bound prediction.

During the development of the tool based on our inference rules, we have encountered many challenges, and also found solutions to tackle them. Our initial experiments have further confirmed the viability of the approach. We show two examples in the paper for the readers to see how the symbolic stack boundary the tool will produce in analyzing programs, and why it is useful.

As ongoing work, we are developing a tool to recover control structures in Java Bytecode programs; after that our tool can work directly on Java Bytecode. As the Presburger formula is not expressive enough, we hope to articulate some advanced resource analysis techniques into our framework.

ACKNOWLEDGMENT

Shengyi Wang and Zongyan Qiu are supported by NNSFC Grant No. 60718002. Shengchao Qin is supported in part by EPSRC projects EP/E021948/1 and EP/G042322/1.

REFERENCES

[1] M. Hofmann and S. Jost, "Static prediction of heap space usage for first-order functional programs," in *POPL*, 2003, pp. 185–197.

[2] J. Hughes and L. Pareto, "Recursion and dynamic data-structures in bounded space: Towards embedded ml programming," in *International Conference on Functional Programming*. ACM, 1999, pp. 70–81.

[3] R. M. Amadio, S. Coupet-Grimal, S. Dal-Zilio, and L. Jakubiec, "A functional scenario for bytecode verification of resource bounds," in *CSL*, ser. Lecture Notes in Computer Science, J. Marcinkowski and A. Tarlecki, Eds., vol. 3210. Springer, 2004, pp. 265–279.

[4] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark, "Mobile resource guarantees for smart devices," in *CASSIS*, ser. Lecture Notes in Computer Science, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., vol. 3362. Springer, 2004, pp. 1–26.

[5] W.-N. Chin, H. Nguyen, S. Qin, and M. Rinard, "Memory Usage Verification for OO Programs," in *Static Analysis Symposium*, ser. Springer LNCS, London, UK, Sept. 2005.

[6] G. He, S. Qin, C. Luo, and W.-N. Chin, "Memory usage verification using hip/sleek," in *7th International Symposium on Automated Technology for Verification and Analysis (ATVA09)*, ser. Lecture Notes in Computer Science, vol. 5799. Springer, 2009, pp. 166–181.

[7] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy, "Performance specification of software components," in *Symposium on Software Reusability: Putting Software Reuse in Context*, Toronto, Canada, 2001, pp. 3–10.

[8] W.-N. Chin, H. H. Nguyen, C. Popeea, and S. Qin, "Analysing memory resource bounds for low-level programs," in *Proceedings of the 7th International Symposium on Memory Management (ISMM 2008)*. ACM, 2008, pp. 151–160.

[9] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "CodeSurfer/x86-A Platform for Analyzing x86 Executables," in *Intl Symp. on Compiler Construction*, 2005.

[10] J. Gustavsson and J. Svenningsson, "Constraint abstractions," in *Programs as Data Objects II*, Aarhus, Denmark, May 2001, pp. 63–83.

[11] C. Popeea and W. Chin, "Inferring disjunctive postconditions," in *Asian Computing Science Conference (ASIAN)*, ser. Lecture Notes in Computer Science, vol. 4435. Springer, 2006, pp. 331–345.

[12] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann, ""carbon credits" for resource-bounded computations using amortised analysis," in *FM*, ser. Lecture Notes in Computer Science, A. Cavalcanti and D. Dams, Eds., vol. 5850. Springer, 2009, pp. 354–369.

[13] B. Campbell, "Amortised memory analysis using the depth of data structures," in *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, ser. Lecture Notes in Computer Science, vol. 5502. Springer, 2009, pp. 190–204.

[14] V. A. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine, "Parametric prediction of heap memory requirements," in *ISMM*, R. Jones and S. M. Blackburn, Eds. ACM, 2008, pp. 141–150.

[15] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann, "Static determination of quantitative resource usage for higher-order programs," in *POPL*, M. V. Hermenegildo and J. Palsberg, Eds. ACM, 2010, pp. 223–236.

[16] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, "Resource usage analysis and its application to resource certification," in *FOSAD*, ser. Lecture Notes in Computer Science, A. Aldini, G. Barthe, and R. Gorrieri, Eds., vol. 5705. Springer, 2009, pp. 258–288.