ETAPS 2011

European Joint Conferences on Theory and Practice of Software

26 March – 03 April, 2011

Saarbrücken, Germany

BYTECODE 2011

6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation

Pierre Ganty and Mark Marron (Eds.)

http://www.etaps.org
http://etaps2011.cs.uni-saarland.de

Universität des Saarlandes

Complete and Platform-Independent Calling Context Profiling for the Java Virtual Machine

Aibek Sarimbekov Philippe Moret Walter Binder[1]

Faculty of Informatics
University of Lugano
Lugano, Switzerland

Andreas Sewe Mira Mezini[2]

Software Technology Group
Technische Universität Darmstadt
Darmstadt, Germany

Abstract

Calling context profiling collects statistics separately for each calling context. Complete calling context profiles that faithfully represent overall program execution are important for a sound analysis of program behavior, which in turn is important for program understanding, reverse engineering, and workload characterization. Many existing calling context profilers for Java rely on sampling or on incomplete instrumentation techniques, yielding incomplete profiles; others rely on Java Virtual Machine (JVM) modifications or work only with one specific JVM, thus compromising portability. In this paper we present a new calling context profiler for Java that reconciles completeness of the collected profiles and full compatibility with any standard JVM. In order to reduce measurement perturbation, our profiler collects platform-independent dynamic metrics, such as the number of method invocations and the number of executed bytecodes. In contrast to prevailing calling context profilers, our tool is able to distinguish between multiple call sites in a method and supports selective profiling of (the dynamic extent of) certain methods. We have evaluate the overhead introduced by our profiler with standard Java and Scala benchmarks on a range of different JVMs.

Keywords: Calling Context Profiling, JP2, Bytecode Instrumentation, Dynamic Metrics

[1] Email: `firstname.lastname@usi.ch`
[2] Email: `lastname@st.informatik.tu-darmstadt.de`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

1 Introduction

Calling context profiling is a common profiling technique that helps analyse the dynamic inter-procedural control flow of applications. It is particularly important for understanding and optimizing object-oriented software, where polymorphism and dynamic binding hinder static analyses. Calling context profiling hereby collects statistics separately for each calling context, such as the number of method invocations or the CPU time spent in a calling context.

Both the dynamic call graph (DCG) and the Calling Context Tree (CCT) are well-known data structures often used for performance characterization and optimization [1]. The nodes in the respective data structures are associated with profiling information. Such a profile can include a wide range of dynamic metrics, e.g., execution times or cache misses. Platform-independent dynamic metrics such as the number of method invocations or executed bytecodes are of particular interest in the area of performance characterization. These metrics are reproducible,[3] accurate, portable, and comparable [12,4].

Unlike a context-insensitive DCG, a CCT in principle is capable of capturing the *complete* context of a call. Still, CCTs generated by state-of-the-art profilers [17] are missing one key bit of information present in the well-known labelled variant of DCGs: information about the individual site at which a call is made. In other words, while keeping track of numerous methods in entire call chains, many calling context profilers are unable to distinguish between multiple call sites within a single method.

In this paper, we introduce JP2, a call-site-aware profiler for both platform-independent and complete calling context profiling. The profiler is based on portable bytecode instrumentation techniques for generating its profiling data structures at runtime. Besides two counters for the number of method executions and number of executed bytecodes, each calling context tracks the current bytecode position; this enables JP to distinguish between the call sites within a single method.

While several of these features were already present in our earlier JP tool [6], this paper makes several unique contributions:

- A detailed description of JP2, the first call-site aware profiler to capture complete CCTs.
- A description of how JP2 can temporary disable profiling for the current thread without breaking the CCT's structure, hence collecting only appropriate profiles.
- A rigorous evaluation of JP2's performance on 3 virtual machines and 22 Java and Scala benchmarks [7,21].

[3] For deterministic programs with deterministic thread scheduling.

This paper is structured as follows: Section 2 gives background information on platform-independent dynamic metrics and CCTs. Section 3 describes the tool's design. Section 4 details our performance evaluation on a range of benchmarks and virtual machines. Section 5 discusses related work, before Section 6 concludes.

2 Background

In the following we give a brief overview of both platform-independent dynamic metrics and the Calling Context Tree data structure.

2.1 Platform-independent Dynamic Metrics

Most state-of-the-art profilers rely on dynamic metrics that are highly platform-dependent. In particular, elapsed CPU or wallclock time are metrics commonly used by profilers. However, these metrics have several drawbacks: For the same program and input, the time measured can differ significantly depending on the hardware, operating system, and virtual machine implementation. Moreover, measuring execution time accurately may require platform-specific features (such as special operating system functions), which limits the portability of the profilers. In addition, it is usually impossible to faithfully reproduce measurement results.

For these reasons, we follow a different approach that uses only *platform-independent* dynamic metrics [12,4], namely the number of method invocations and the number of executed bytecodes. The benefits of using such metrics are fourfold:

(i) Measurements are *accurate*; profiling itself does not affect the generated profile and will not cause measurement perturbations.

(ii) Profilers can be implemented in a *portable* way; one can execute them on different hardware, operating systems, and virtual machines.

(iii) The profiles made in different environments are *comparable*, as they rely on the same set of platform-independent metrics.

(iv) For deterministic programs, measurements are *reproducible*.

Although information on the number of method invocations is a common metric supported by many available profiling tools, some profilers do not differentiate between different calling contexts or keep calling contexts only up to a pre-defined depth. In contrast, our approach is able to associate both the number of method invocations and the number of executed bytecodes with calling contexts of arbitrary depth.

2.2 The Calling Context Tree (CCT)

The Calling Context Tree (CCT) [1] is a common data structure to represent calling context profiles at runtime [1,2,25,22,26,9]. Each node in a CCT corresponds to a calling context and keeps the dynamic metrics measured for that particular calling context; it also refers to the method in which the metrics were collected. The parent of a CCT node represents the caller's context, while the children nodes correspond to the callee methods. If the same method is invoked in distinct calling contexts, the different invocations are thus represented by distinct nodes in the CCT. In contrast, if the same method is invoked multiple times in the same calling context *and* from the same call site, the dynamic metrics collected during the executions of that method are kept in the same CCT node. The CCT thus makes it possible to distinguish dynamic metrics by their calling context. This level of detail is useful in many areas of software engineering such as profiling [1], debugging [3], testing [20], and reverse engineering [15].

It should be noted that the data structure itself does not impose any restrictions on the number and kind of dynamic metrics kept in the CCT nodes; in particular, these metrics may be platform-dependent (CPU time, number of cache misses) or platform-independent (number of method invocations [4], number of executed bytecodes, number of object allocations). In the following, we will restrict the discussion to two platform-independent metrics: the number of method invocations and the number of executed bytecodes. Fig. 1 exemplifies such a CCT data structure, which stores both metrics.

CCTs are most useful if they faithfully represent overall program execution. We thus require that a *complete* CCT contains all method invocations made after an initial JVM bootstrapping phase [5], where either the caller or the callee is a Java method, i.e., a method not written in native code. The following method invocations must therefore be present within the CCT:

(i) A Java method invoking another Java method.

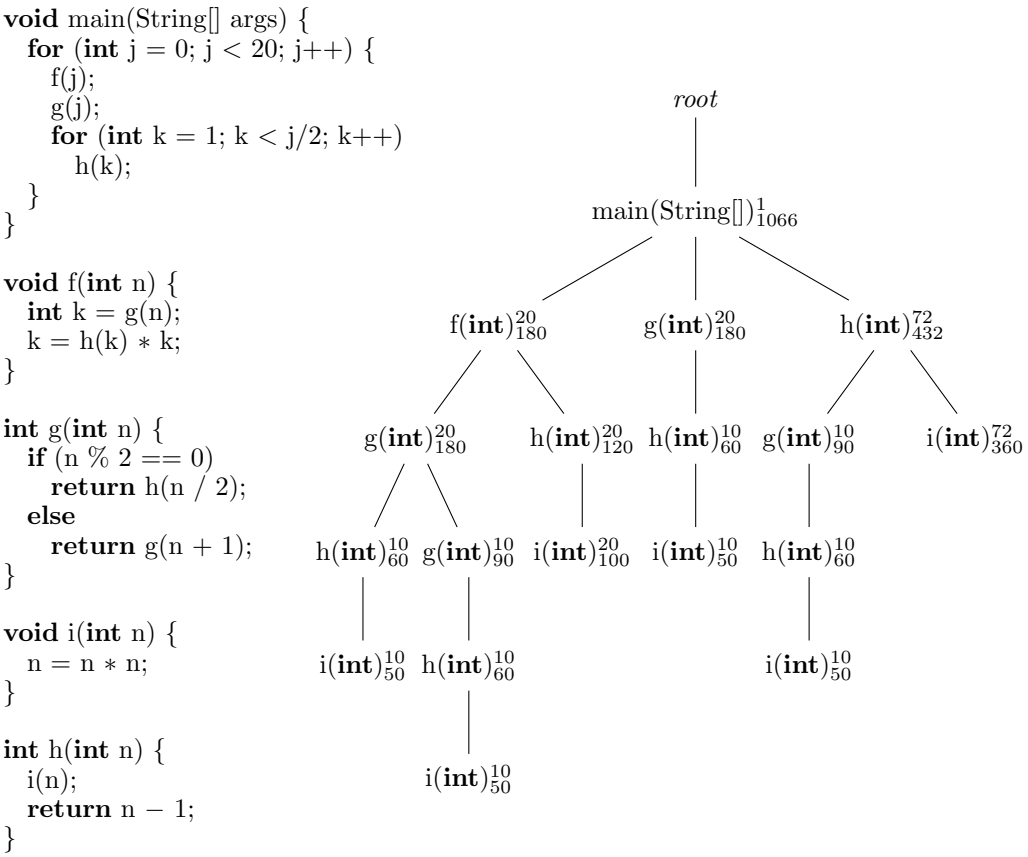
(ii) A Java method invoking a **native** method.

(iii) Native code invoking a Java method (e.g., callback from native code into bytecode through the Java Native Interface (JNI), class loading, or class initialization)

Regarding method invocation through reflection, the above definition of completeness implies that any method called through reflection (Method.invoke(...), Constructor.newInstance(...)) must be represented in the CCT.

[4] In this paper we do not distinguish between methods and constructors; 'method' denotes both 'methods' and 'constructors'.

[5] At the latest, this phase ends with the invocation of the program's main(String[]) method.



(a) Sample code (b) Generate CCT (conceptual representation)

Fig. 1. Sample Java code and the CCT generated for one execution of method main(String[]). As dynamic metrics, each CCT node stores the number of method invocations (^{m}) and the number of executed bytecodes ($_{n}$) in the corresponding calling context.

There are several variations of the CCT supported by our tool. For instance, calls to the same method from different call sites in a caller may be represented by the same node or by different nodes in the CCT. Moreover, in the case of recursion, the depth of the CCT may be unbounded, representing each recursive call to a method by a separate CCT node, or alternatively recursive calls might be stored in the same node, limiting the depth of the CCT and introducing back-edges into the CCT [1].

3 Tool Design

In this section we describe the design and the architecture of our tool. First, Section 3.1 discusses both the design and the weaknesses of a previous version of the tool. Next, Section 3.2 presents our new design implemented in the JP2 profiler. Finally, Section 3.3 explains how JP2 deals with native methods.

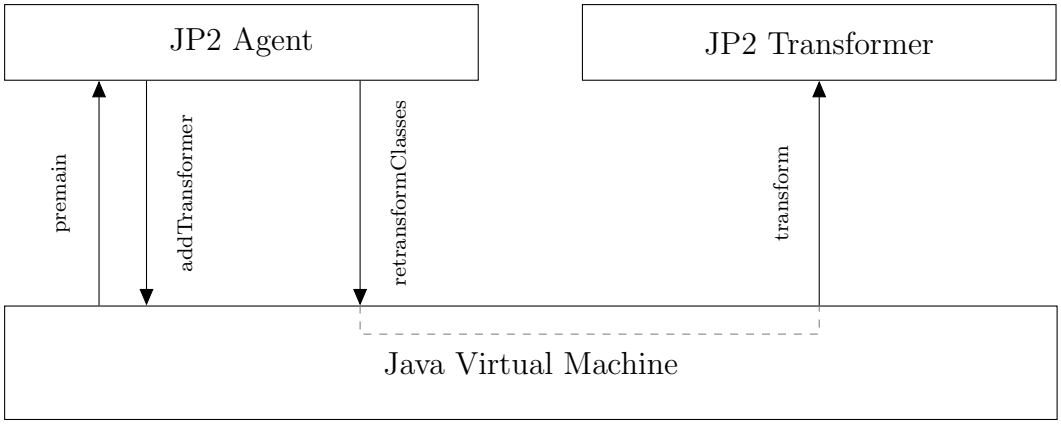


Fig. 2. The architecture of JP2.

3.1 Old Design (JP)

Our previous profiler, JP [17,6], is based on the generic bytecode instrumentation framework FERRARI [5], which allows us to statically instrument the Java class library and apply load-time instrumentation to all other classes. JP extends method signatures in order to pass a CCT node reference from the caller to the callee as an argument. Therefore, each method invocation keeps a reference to its corresponding CCT node in a local variable. Furthermore, for each method a static field is added to hold the corresponding method identifier and the class initializer is instrumented to allocate those method identifiers. For compatibility with native code, wrapper methods with unmodified signatures are added to allow native code, which is not aware of the additional argument, to invoke the instrumented method through the JNI.

This approach introduces compatibility problems with recent JVMs because some methods cannot be overloaded in this fashion. Moreover, the additional stack frames introduced by wrapping methods can break stack introspection. Furthermore, static instrumentation of the Java class library is time consuming and inconvenient to the user. Finally, JP is unable to distinguish between different call sites and one cannot selectively enable or disable metrics collection for a certain calling context, e.g., for a benchmark's harness.

3.2 New Design (JP2)

Fig. 2 depicts the architecture of JP2, our revised design which addresses all of JP's weaknesses mentioned above. JP2 includes two main components with the following responsibilities:

(i) The *JP2 Agent* is a Java programming language agent; it initializes JP2 upon JVM startup.

(ii) The *JP2 Transformer*, which is based on the ASM bytecode engineering

```
BI      void f() {
          while (true) {
            if (i <= 10) {
5:            h();
7:            g(i);
              ++i;
            } else {
                return;
              }
          }
        }
```

(a) Before Instrumentation

```
void f() {
  int callerBI = JP2Runtime.getBI();
  CCTNode caller = JP2Runtime.getCurrentNode();
  CCTNode callee = caller.profileCall("f()", callerBI);
  try {
    callee.profileBytecodes(2);
    while (true) {
      callee.profileBytecodes(3);
      if (i <= 10){
        callee.profileBytecodes(5);
        JP2Runtime.setBI(5);
        h();
        JP2Runtime.setBI(7);
        g(i);
        ++i;
      } else {
          callee.profileBytecodes(1);
          return;
        }
    }
  } finally {
    JP2Runtime.setCurrentNode(caller);
    JP2Runtime.setBI(callerBI);
  }
}
```

(b) After Instrumentation

Fig. 3. Example of Java code instrumented by JP2.

libary [6], is responsible for the necessary bytecode instrumentation.

Upon startup, the JVM invokes the premain() method of the JP2 Agent before any application code is executed. The agent registers the transformer using the standard java.lang.instrument API. Through this API, the agent then triggers retransformation of classes that have already been loaded during JVM startup. During retransformation, the JVM calls back the transformer, which instruments the classes. The JP2 Transformer receives transform() requests both for classes to be retransformed, as well as for newly loaded classes (see Fig. 2).

In contrast to JP, JP2 does not need to extend any method signatures in order to pass a CCT node; instead, it uses thread-local variables to store references to them. Moreover, instead of adding static fields storing the method identifiers, JP2 uses string constants which simply reside in the class file's constant pool; thus, there is no need for extending the class initializer anymore. Fig. 3 illustrates the instrumentation the JP2 Transformer applies. Depicted to the left is the method f() before transformation; depicted to the right is

[6] See `http://asm.ow2.org/`.

```
public class JP2Runtime {
  public static int getBI() {...}
  public static void setBI(int callerBI) {...}
  public static CCTNode getCurrentNode() {...}
  public static void setCurrentNode(CCTNode n) {...}
}

public interface CCTNode {
  CCTNode profileCall(String methodID, int callerBI);
  void profileBytecodes(int i);
}
```

Fig. 4. Runtime classes used by JP2.

the corresponding instrumented version.[7] The transformer inserts invocations to static methods in class JP2Runtime shown in Fig. 4, which are explained below.

setBI(int callerBI), getBI() Store the caller's bytecode position in a thread-local variable, respectively load the stored bytecode position (BI) from the thread-local variable.

setCurrentNode(CCTNode n), getCurrentNode() Store the current thread's current CCT node in a thread-local variable, respectively load the current CCT node from a thread-local variable.

Hereby, CCTNode is an interface shown in Fig. 4, whose methods perform the following functions:

profileCall(String methodID, int callerBI) Return the callee of the method in the CCT; if there is no such node, register it.

profileBytecodes(int) Update the counter keeping the number of executed bytecodes.

It is crucial to restore the caller's bytecode position in the **finally** block because class loading and class initialization may be triggered between a call to setBI(**int**) in a caller and the subsequent method call, which may in turn update the thread-local variable.

JP2 counts bytecodes per basic block using the same algorithm as JP [6]: only bytecodes that may change the control flow non-sequentially (i.e., jumps, branches, return of method or JVM subroutine, exception throwing) end a basic block. This algorithm creates rather large basic blocks, such that the number of updates to the bytecode counter is kept low. This reduces runtime overhead without significantly affecting accuracy [6].

JP2 provides a mechanism to temporarily disable the execution of instrumentation code for each thread. Assume that instrumentation code itself uses

[7] To improve readability, all transformations are shown in Java-based pseudo code, although JP2 works at the JVM bytecode level.

```
native boolean foo(int x);
```

(a) Before wrapping

```
boolean foo(int x) {
  return wrapped_foo(x);
}

native boolean wrapped_foo(int x);
```

(b) After wrapping

Fig. 5. Example of a native method wrapped by JP2.

methods from the Java class library, which has already been instrumented. This will cause infinite recursions. To sidestep this issue, JP2 uses code duplication within method bodies in order to keep the non-instrumented bytecode version together with the instrumented code, and inserts a conditional upon the method entry in order to select the version to be executed. [16]

JP2 allows to selectively activate and deactivate the collection of dynamic metrics for each thread without breaking the structure of the CCT. In Section 4 we use this feature to collect proper profiles only for the execution of the benchmarks, excluding the execution in the harness and the JVM's startup and shutdown sequences.

3.3 Native Methods

To gather complete profiles, JP2 has to keep track of all native method invocations as well as callbacks from those native methods. Since native methods do not have any bytecode representation, they cannot be instrumented directly. As illustrated by Fig. 5, JP2 thus adds simple wrapper methods with unmodified signatures. Native method prefixing [23], a functionality introduced in Java 6, is used to rename native methods and introduce a bytecode implementation with the name of the original native method. However, certain limitations prevent JP2 from applying the transformation at runtime to classes loaded during JVM bootstrapping. While class redefinition may change method bodies, the constant pool and attributes, it cannot add, remove or rename fields or methods, and change the signatures of methods. Therefore, JP2 is accompanied by a static tool, whose sole purpose is to add those wrappers to the Java class library.

This is the only circumstance under which JP2 has to resort to static instrumentation, which should be done before any dynamic instrumentation. Later, the wrapped Java class library is added at the beginning of the boot class path. Since the JVM needs to invoke native methods upon bootstrapping, JP2 has to make it aware of the added prefix. Therefore, a JVMTI agent, which only informs the JVM about the prefix, needs to be passed as a command line option to the JVM.

4 Evaluation

In order for a profiling tool to be universally useful, it has to be *stable* and *portable*. Furthermore, it must not impose prohibitive measurement overhead, i.e., slow down the application by orders of magnitude. In our evaluation we show that JP2 has all three properties; when running a diverse selection of benchmarks on a set of production JVMs it imposes acceptable runtime overhead.

To this end, we have evaluated the runtime overhead incurred by JP2 using two different benchmark suites: the DaCapo 9.12-bach benchmark suite [7] and a DaCapo-based benchmark suite consisting of Scala programs, which is under active development by one of the authors [21]. In either case, the measurements exclude the startup and shutdown of both JVM and benchmark harness. JP2 itself has also been configured to collect dynamic metrics only for the benchmark proper, of whose iterations it is notified using the callback mechanisms provided by the benchmark harness (`Callback`).

To both show that JP2 is portable and to assess the effect a given JVM can have on the runtime overhead incurred by JP2, we have performed all measurements using three configurations representative of modern production JVMs: the HotSpot Server VM [8], the HotSpot Client VM, [8] and the JRockit VM [9]. All benchmarks have been run on a 2.33 GHz Core 2 Duo dual core E6550 processor with 2 GB of main memory, 32 KB L1 data and instruction caches, and 4096 KB L2 cache; its entire main memory has been available to the JVM (`-Xmx2G`). During benchmarking, the computer was operating in single-user mode under Ubuntu Linux 9.10 (kernel 2.6.31).

Fig. 6 depicts the overhead incurred by JP2 during the first iteration of the 14 DaCapo 9.12 benchmarks when using the three virtual machines mentioned above. As can be seen, on most virtual machines the overhead is moderate; the slowdown is less than one order of magnitude. The only exception from this is the HotSpot Client VM. Here, JP2 incurs significantly higher overheads, as the VM's client compiler [13] copes less well than the server compiler [19] with the instrumentation inserted by JP2. But as JP2 produces mostly platform-independent profiles, it is often possible to reduce overheads to acceptable levels simply by choosing a different virtual machine that copes better with JP2's instrumentation; the resulting CCTs will differ only within the platform-specific part of the given Java class library, not within the application.

Also, part of the runtime overhead is incurred by JP2 only upon class-loading, i.e., when newly loaded classes are instrumented. Fig. 7 illustrates this fact; the absolute overhead diminishes over the course of several iterations

[8] JRE build 1.6.0_22-b04, JVM build 17.1-b03
[9] JRE build 1.6.0_20-b02, JVM build R28.0.1-21-133393-1.6.0_20-20100512-2126-linux-ia32

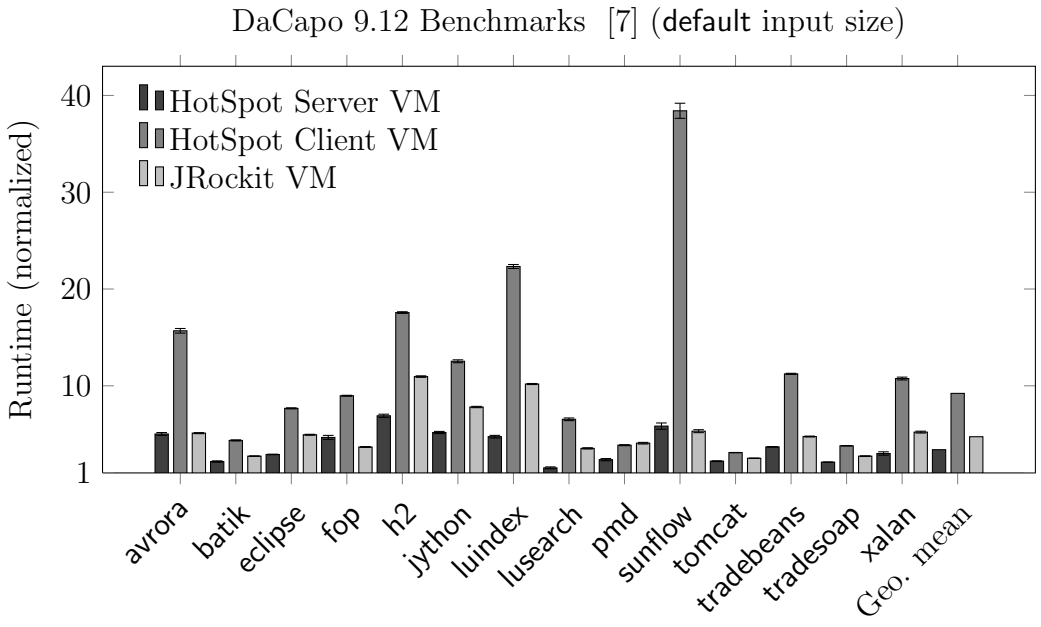


Fig. 6. Runtime overhead (5 invocations, arithmetic mean ± sample standard deviation) incurred by JP2 during the first iteration of 14 Java benchmarks on 3 different JVMs.

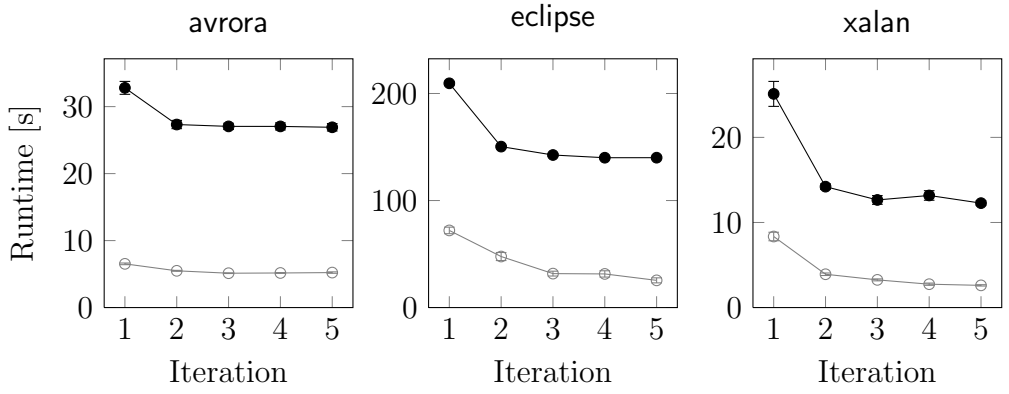


Fig. 7. Runtime (5 invocations, arithmetic mean ± sample standard deviation) with (—●—) and without (—○—) JP2 over serveral iterations of three benchmarks on the HotSpot Server VM.

of a benchmark or during long-running applications. The relative overhead, however, increases, as the more advanced optimizations performed by the just-in-time compiler during later iterations are hindered by the instrumentation inserted by JP2.

Fig. 8 depicts the overhead incurred by JP2 on a set of Scala benchmarks. As the CCTs generated for several benchmarks (kiama, scalac, and scaladoc) exceed the heap's capacity of 2 GB, only the small input size has been used for those benchmarks. But when compared to the Java benchmarks of Fig. 6, the overhead incurred by JP2 on the Scala benchmarks is remarkably similar: For only two benchmarks (scalaxb, tmt), the HotSpot Server VM, which per-

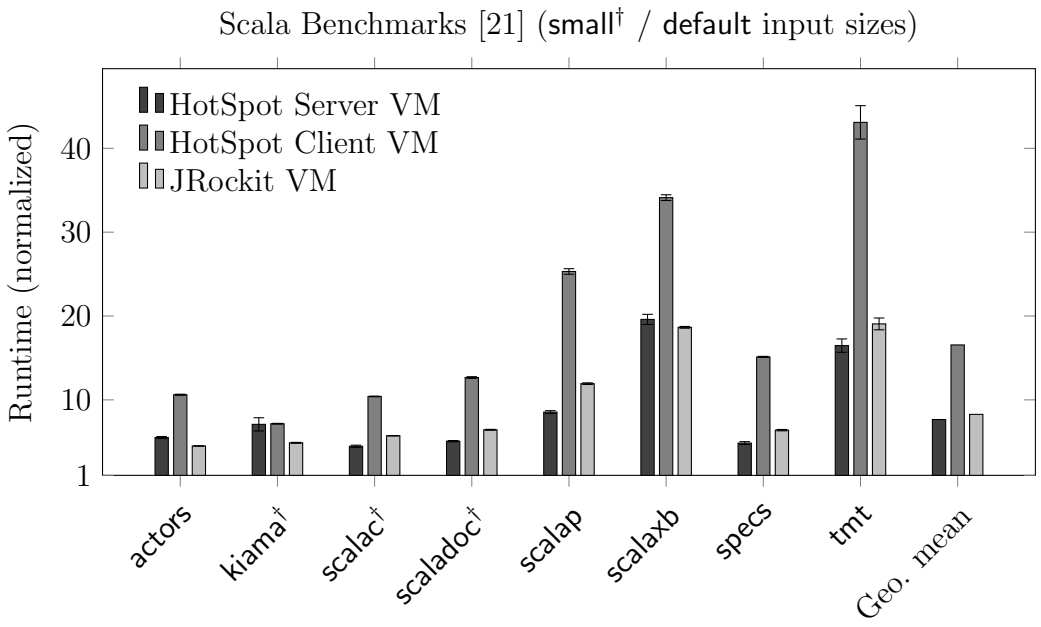


Fig. 8. Runtime overhead (5 invocations, arithmetic mean ± sample standard deviation) incurred by JP2 during the first iteration of 8 Scala benchmarks on 3 different JVMs.

forms best with JP2 on the Java benchmarks, experiences more than moderate performance degradation on the Scala benchmarks.

Fig. 9 shows two key properties of the CCTs generated for various benchmark programs: the number of unique methods called and the number of CCT nodes that result therefrom. As JP2 has to keep the CCT in memory, benchmarks with millions of CCT nodes (the Java benchmarks eclipse, jython, and pmd; the Scala benchmarks scalac, specs, and tmt) naturally put additional pressure on the JVM's garbage collector, which has to trace a large data structure that never dies till VM shutdown. Nevertheless, as Fig. 9 shows, JP2 is able to deal with large programs consisting of tens of thousands of methods.

5 Related Work

Calling context profiling has been explored by many researchers. Existing approaches that create accurate CCTs [22,1] suffer from considerable overhead. Sampling-based profiles promise a seemingly simple solution to the problem of large profiling overheads. However, as Mytkowicz et al. have recently shown [18], implementing sampling-based profilers correctly such that the resulting profiles are at least "actionable" if not accurate is an intricate problem which many implementations fail to solve. JP2 sidesteps this issue by focussing on machine-independent metrics, which it measures both accurately and with moderate profiling overhead.

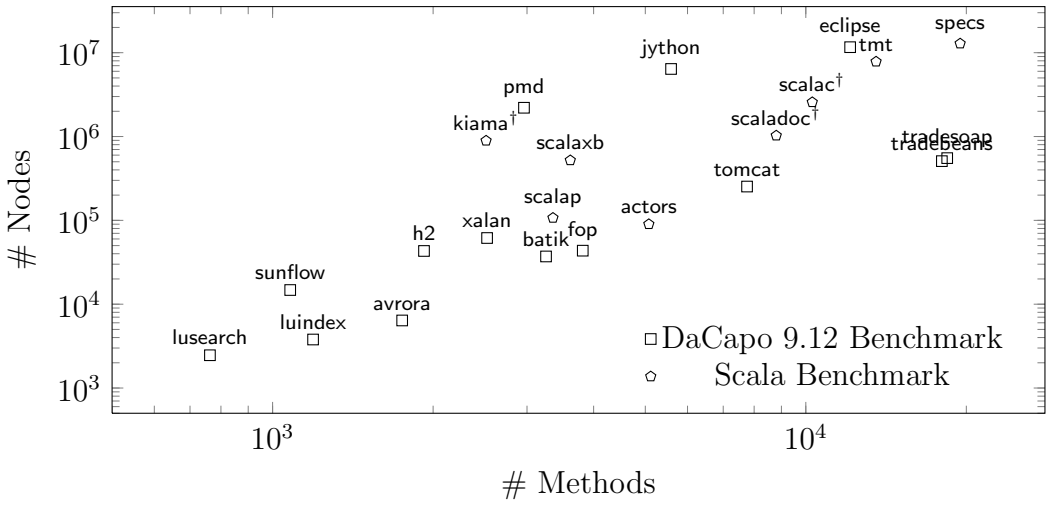


Fig. 9. Number of called methods and number of generated CCT nodes for different benchmarks. (For some Scala benchmarks, only the small† input size was measured.)

Dufour et al. [11] present a variety of dynamic metrics, including bytecode metrics, for selected Java programs, such as the SPEC JVM98 benchmarks [24]. They introduce a tool called *J [12] for the metrics computation. *J relies on the JVMPI [14], a profiling interface for the JVM, whose use is known to cause high overhead when recording, e.g., method entry and exit events like JP2 does [10]. Furthermore, JVMPI is an interface no longer supported as of the Java 6 release (late 2006).

The NetBeans Profiler [11] integrates Sun's JFluid profiling technology [10] into the NetBeans IDE. JFluid exploits dynamic bytecode instrumentation and code hotswapping in order to turn profiling on and off dynamically, for the whole application or just a subset of it. However, this tool needs a customized JVM and is therefore only available for a limited set of environments. In contrast, JP2 works with any standard JVM without customization.

The Probabilistic Calling Context (PCC) approach due to Bond et al. [9] continuously maintains a probabilistically unique value representing the current calling context. As this value can be efficiently computed, the approach causes rather low overhead, if supported by a customized virtual machine. But due to its probabilistic nature PPC does not always produce completely accurate profiles. Recent research, however, has shown that is often possible to reconstruct a significant amount of context offline [8].

[10] For the fop Java benchmark, e.g., *J increases runtime by a factor of 33.
[11] See `http://profiler.netbeans.org/`.

6 Conclusion

In this paper we presented JP2, a new tool for complete platform-independent calling context profiling. JP2 is conceived as a software development tool and is not meant to support profiling of systems in production. JP2 relies on bytecode transformation technique in order to create CCTs with platform-independent dynamic metrics, such as the number of method invocations and the number of executed bytecodes. In contrast to prevailing profilers, JP2 is able to distinguish between multiple call sites in a method and supports selective profiling of certain methods. We have evaluated the overhead caused by JP2 with standard Java and Scala benchmarks on a range of different JVMs.

Acknowledgements

This work has been supported by the Swiss National Science Foundation and by CASED (www.cased.de).

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1997.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[3] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making Software Failures Reproducible by Preserving Object States. In J. Vitek, editor, *ECOOP '08: Proceedings of the 22th European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 542–565, Paphos, Cyprus, 2008. Springer-Verlag.

[4] W. Binder. A portable and customizable profiling framework for Java based on bytecode instruction counting. In *Third Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *Lecture Notes in Computer Science*, pages 178–194, Tsukuba, Japan, Nov. 2005. Springer Verlag.

[5] W. Binder, J. Hulaas, and P. Moret. Advanced java bytecode instrumentation. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM.

[6] W. Binder, J. Hulaas, P. Moret, and A. Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39(1):47–79, 2009.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[8] M. D. Bond, G. Z. Baker, and S. Z. Guyer. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 13–24, New York, NY, USA, 2010. ACM.

[9] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming, systems and applications*, pages 97–112, New York, NY, USA, 2007. ACM.

[10] M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150. ACM Press, 2004.

[11] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, Nov. 2003.

[12] B. Dufour, L. Hendren, and C. Verbrugge. *J: A tool for dynamic analysis of Java programs. In *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 306–307, New York, NY, USA, 2003. ACM Press.

[13] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5:7:1–7:32, May 2008.

[14] S. Liang and D. Viswanathan. Comprehensive profiling support in the Java virtual machine. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-99)*, pages 229–240, Berkeley, CA, May 3–7 1999. USENIX Association.

[15] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[16] P. Moret, W. Binder, and E. Tanter. Polymorphic bytecode instrumentation. In *International Conference on Aspect-Oriented Software Development '11*, Porto de Galinhas, Pernambuco, Brasil, March 21-25 2011. Publication forthcoming.

[17] P. Moret, W. Binder, and A. Villazón. CCCP: Complete calling context profiling in virtual execution environments. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 151–160, Savannah, GA, USA, 2009. ACM.

[18] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 187–197, New York, NY, USA, 2010. ACM.

[19] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[20] A. Rountev, S. Kagan, and J. Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In *FASE '05: Fundamental Approaches to Software Engineering*, volume 3442 of *Lecture Notes in Computer Science (LNCS)*, pages 282–297, April 2005.

[21] A. Sewe. Scala $\stackrel{?}{\equiv}$ Java mod JVM. In *Proceedings of the Work-in-Progress Session at the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ 2010)*, volume 692 of *CEUR Workshop Proceedings*, 2010.

[22] J. M. Spivey. Fast, accurate call graph profiling. *Software: Practice and Experience*, 34(3):249–264, 2004.

[23] Sun Microsystems, Inc. JVM Tool Interface (JVMTI), Version 1.0. Web pages at `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`, 2004.

[24] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`, 1998.

[25] J. Whaley. A portable sampling-based profiler for Java Virtual Machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 78–87. ACM Press, June 2000.

[26] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 263–271, New York, NY, USA, 2006. ACM.

Implementing a Language with Flow-Sensitive and Structural Typing on the JVM

David J. Pearce[1]

School of Engineering and Computer Science
Victoria University of Wellington, NZ

James Noble[2]

School of Engineering and Computer Science
Victoria University of Wellington, NZ

Abstract

Dynamically typed languages are flexible and impose few burdens on the programmer. In contrast, static typing leads to software that is more efficient and has fewer errors. However, static type systems traditionally require every variable to have one type, and that relationships between types (e.g. subclassing) be declared explicitly. The Whiley language aims to hit a sweet spot between dynamic and static typing. This is achieved through structural subtyping and by typing variables in a flow-sensitive fashion. Whiley compiles to the JVM, and this presents a number of challenges. In this paper, we discuss the implementation of Whiley's type system on the JVM.

1 Introduction

Statically typed programming languages (e.g. Java, C#, C++, etc) lead to programs which are more efficient and have fewer errors [11,2]. Static typing forces some discipline on the programming process. For example, it ensures at least some documentation regarding acceptable function inputs is provided. In contrast, dynamically typed languages are more flexible which helps reduce overheads and increase productivity [37,47,34,7]. Furthermore, in recent times, there has been a significant shift towards dynamically typed languages [38].

[1] Email: djp@ecs.vuw.ac.nz
[2] Email: kjx@ecs.vuw.ac.nz

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Numerous attempts have been made to bridge the gap between static and dynamic languages. Scala [45], C#3.0 [6], OCaml [43] and, most recently, Java 7 all employ local type inference (in some form) to reduce syntactic overhead. Techniques such as gradual typing [46,50], soft typing [11] and hybrid typing [20] enable a transitory position where some parts of a program are statically typed, and others are not. Alternatively, type inference can be used (in some situations) to reconstruct types "after the fact" for programs written in dynamic languages [3,23].

Whiley is a statically-typed language which, for the most part, has the look and feel of a dynamic language. This is achieved with an extremely flexible type system which utilises the following features:

- **Flow-sensitive types**, which are adopted from flow-sensitive program analysis (e.g. [22,30,14]) and allow variables to have different types at different points.
- **Structural Subtyping**, where subtyping between data types is implict and based on their structure.

Taken together, these offer several advantages over traditional nominal typing, where types are named and subtyping relationships explicitly declared.

1.1 Contributions

The contributions of this paper are:

(i) We discuss the flow-sensitive and structural typing system used in the Whiley language.

(ii) We detail our implementation of these features on the JVM, and identify a number of challenges.

An open source implementation of the Whiley language is freely available from `http://whiley.org`. Finally, a detailed formalisation of Whiley's type system, including some discussion of JVM implementation, can be found here [39].

2 Whiley

In this section, we present a series of examples showing Whiley's key features. In the following section, we'll discuss their implementaiton in the JVM.

2.1 Implicit Declaration

Most contemporary statically typed languages require variables be explicitly declared (FORTRAN is one exception here). Compared with dynamically typed languages, this is an extra burden for the programmer, particularly when a variable's type can be inferred from assigned expression(s). In Whiley, local variables are declared by assignment:

```
int average([int] items):
    v = 0
    for i in items:
        v = v + items[i]
    return v / |items|
```

Here, `items` is a list of **`int`**s, whilst `|items|` returns its length. The variable `v` is used to accumulate the sum of all elements in the list. Variable `v` is declared by the assignment from `0` and, since this has type **`int`**, `v` has type **`int`** after the assignment.

2.2 Union Types

Nullable references have proved a significant source of error in languages such as Java [29]. The issue is that, in such languages, one can treat *nullable* references as though they are *non-null* references [40]. Many solutions have been proposed which strictly distinguish these two forms using static type systems [18,17,41,35].

Whiley's type system lends itself naturally to handling this problem because it supports *union types* (see e.g. [5,31]). These allow variables to hold values from different types, rather than just one type. The following illustrates:

```
null|int indexOf(string str, char c):
    ...

[string] split(string str, char c):
    idx = indexOf(str,c)
    if idx ~= int:
        // matched an occurrence
        below = str[0..idx]
        above = str[idx..]
        return [below,above]
    else:
        return [str] // no occurrence
```

Here, `indexOf()` returns the first index of a character in the string, or **`null`** if there is none. The type **`null|int`** is a union type, meaning it is either an **`int`** or **`null`**.

In the above example, Whiley's type system seamlessly ensures that **`null`** is never dereferenced. This is because the type **`null|int`** cannot be treated as an **`int`**. Instead, we must first perform a runtime type test to ensure it is an **`int`**. Whiley automatically retypes `idx` to **`int`** when this is known to be true, thereby avoiding any awkward and unnecessary syntax (e.g. a cast as required in [4,35]).

2.3 Flow-Sensitive Typing

The following code shows the definition of a simple hierarchy of Shapes in Whiley, and a function that returns the area of any of these three types of Shapes.

```
define Circle as {int x, int y, int radius}
define Square as {int x, int y, int dimension}
define Rectangle as {int x, int y,
                     int width, int height}

define Shape as Circle | Square | Rectangle

real area(Shape s):
    if s ~= Circle:
        return PI * s.radius * s.radius
    else if s ~= Square:
        return s.dimension * s.dimension
    else:
        return s.width * s.height
```

A `Shape` is a union type — either a `Circle`, `Square` or `Rectangle` (which are all themselves record types). The code employs a runtime type test, "`s ~= Circle`", to distinguish the different kinds of `Shapes`. This is similar to Java's **`instanceof`** or Eiffel's reverse assignment. Unlike Java, Whiley retypes `s` to be of type `Circle` on the true branch of the **`if`** statement, so there is no need to cast `s` explicitly to `Circle` before accessing the `Circle`-specific field `radius`. Similary, on the false branch, Whiley retypes `s` to the union type `Square|Rectangle`, and then to `Square` or `Rectangle` within the next **`if`**.

Implementing these Shapes in most statically-typed languages would be more cumbersome and more verbose. In modern object-oriented languages, like Java, expressions must still be *explicitly* retyped. For example, after a test such as `s` **`instanceof`** `Circle`, we must introduce a new variable, say `c`, with type `Circle` as an alias for `s`, and use `c` whenever we wanted to access `s` as a circle.

2.4 Structural Subtyping

Statically typed languages, such as Java, employ *nominal typing* for recursive data types. This results in rigid hierarchies which are often difficult to extend. In contrast, Whiley employs *structural subtyping* of records [10] to give greater flexibility. For example, the following code defines a `Border` record:

```
define Border as {int x, int y, int width, int height}
```

Any instance of `Border` has identical structure to an instance of `Rectangle`. Thus, wherever a `Border` is required, a `Rectangle` can be provided and vice-

versa — even if the `Border` definition was written long after the `Rectangle`, and even though `Rectangle` makes no mention of `Border`.

The focus on structural, rather than nominal, types in Whiley is also evident in the way instances are created:

```
bool contains(int x, int y, Border b):
    return b.x <= x && x < (b.x + b.width) &&
           b.y <= y && y < (b.y + b.height)

bool example(int x, int y):
    rect = {x: 1, y: 2, width: 10, height: 3}
    return contains(x,y,rect)
```

Here, function `example()` creates a record instance with fields `x`, `y`, `width` and `height`, and assigns each an initial value. Despite not being associated with a name, such as `Border` or `Rectangle`, it can be freely passed into functions expecting such types, since they have identical *structure*.

2.5 Value Semantics

In Whiley, all compound structures (e.g. lists, sets, and records) have *value semantics*. This means they are passed and returned by-value (as in Pascal, or most functional languages) — but unlike functional languages (and like Pascal) values of compound types can be updated in place.

Value semantics implies that updates to the value of a variable can only affect that variable, and the only way information can flow out of a procedure is through that procedure's return value. Furthermore, Whiley has no general, mutable heap comparable to those found in object-oriented languages. Consider the following:

```
int f([int] xs):
    ys = xs
    ys[0] = 1
    ...
```

The semantics of Whiley dictate that, having assigned `xs` to `ys` as above, the subsequent update to `ys` does not affect `xs`. Arguments are also passed by value, hence `xs` is updated inside `f()` and this does not affect `f`'s caller. That is, changes can only be communicated out of a function by explictly returning a value.

Whiley also provides strong guarantees regarding subtyping of primitive types (i.e. integers and reals). In Whiley, **`int`**s and **`real`**s represent unbounded integers and rationals, which ensures **`int`** $\leq$ **`real`** has true subset semantics (i.e. every **`int`** can be represented by a **`real`**). This is not true for e.g. Java, where there are **`int`** (resp. **`long`**) values which cannot be represented using **`float`** (resp. **`double`**) [26, §5.1.2].

2.6 Incremental Construction

A common pattern arises in statically typed languages when a structure is built up piecemeal. Usually, the pieces are stored in local variables until all are available and the structure can be finally created. In dynamic languages it is much more common to assign pieces to the structure as they are created and, thus, at any given point a partially complete version of the structure is available. This reduces syntactic overhead, and also exposes opportunities for code reuse. For example, the partial structure can be passed to functions that can operate on what is available. In languages like Java, doing this requires creating a separate (intermediate) object.

In Whiley, structures can also be constructed piecemeal. For example:

```
BinOp parseBinaryExpression():
    v = {} // empty record
    v.lhs = parseExpression()
    v.op = parseOperator()
    v.rhs = parseExpression()
    return v
```

After the first assignment, `v` is an empty record. Then, after the second it has type `{Expr lhs}`, after the third it has type `{Expr lhs, Op op}`, and after the fourth it has type `{Expr lhs,Expr rhs,Op op}`. This also illustrates the benefits of Whiley's value semantics with update: the value semantics ensure that there can never be any alias to the value contained in `v`; while updates permit `v` to be built imperatively, one update at a time.

2.7 Structural Updates

Static type systems normally require updates to compound types, such as list and records, to respect the element or field type in question. Whiley's value semantics also enables flexible updates to structures without the aliasing problems that typically arise in object-oriented languages. For example, assigning a **`float`** to an element of an **`int`** array is not permitted in Java. To work around this, programmers typically either clone the structure in question, or work around the type system using casting (or similar).

In Whiley, updates to lists and records are always permitted. For example:

```
define Point as {int x, int y}
define RealPoint as {real x, real y}

RealPoint normalise(Point p, int w, int h):
    p.x = p.x / w
    p.y = p.y / h
    return p
```

Here, the type of `p` is updated to `{real x,int y}` after `p.x` is assigned, and `{real x,real y}` after `p.y` is assigned. Similarly, for lists we could write:

```
[real] normalise([int] items, int max):
    for i in 0..|items|:
        items[i] = items[i] / max
    return items
```

Here, the type of `items` is updated to `[real]` by the assignment. Thus, Whiley's type system permits an in-place update from integer to real without requiring any explicit casts, or other type system abuses (e.g. exploiting raw types, such as `List`, in Java).

3 Implementation on the JVM

The Whiley language compiles down to Java bytecode, and runs on the JVM. In this section, we outline how Whiley's data types are represented on the JVM.

3.1 Numerics

Whiley represents numbers on the JVM in a similar fashion to Clojure [28]. More specifically, **int**s and **real**s are represented using custom `BigInteger` and `BigRational` classes which automatically resize to prevent overflow. Whiley requires that integers are truly treated as subtypes of reals. For example:

```
real f(int x):
    if x >= 10:
        x = 9.99
    return x
```

At the control-flow join after the **if** statement, `x` holds either an **int** or a **real**. Since **real** values are implemented as `BigRational`s on the JVM, we must coerce `x` from a `BigInteger` on the false branch.

3.2 Records

The implementation of Whiley's record types must enable structural subtyping. A simple approach (used in many dynamic languages), is to translate them as `HashMap`s which map field names to values. This ensures that record objects can be passed around freely, provided they have the required fields.

One issue with this approach, is that each field access requires a `HashMap` lookup which, although relatively fast, does not compare well with Java (where field accesses are constant time). Whiley's semantics enable more efficient implementations. In particular, the type `{int x}` is a record containing *exactly* one

field x. Thus, records can have a static layout to give constant time access [40]. For example, records can be implemented on the JVM using arrays of references, rather than HashMaps. In this approach, every field corresponds to a slot in the array whose index is determined by a lexicographic ordering of fields.

To implement records using a static layout requires the compiler to insert coercions to support subtyping. Consider the following:

```
define R1 as {int x, int y}
define R2 as {int y}

R1 r1 = {x: 10, y: 20}
R2 r2 = r1
```

Let us assume our records are implemented using a static layout where fields are ordered alphabetically. Thus, in R1, field x occupies the first slot, and field y the second. Similarly, field y corresponds to the first slot of R2 and, hence, R1 is not compatible with R2. Instead, we must convert an instance of R1 into an instance of R2 by constructing a new array consisting of a single field, and populating that with the second slot (i.e. field y). This conversion is safe because of Whiley's value semantics — that is, two variables' values can never be aliased.

3.3 Collections

Whiley provides first-class lists, sets and maps which are translated on the JVM into ArrayLists, HashSets and HashMaps respectively. Of course, all these collection types must have value semantics in Whiley. Recall that, in the following, updating ys does not update xs:

```
int f([int] xs):
    ys = xs
    ys[0] = 1
    ...
```

A naive translation of this code to the JVM would clone() the ArrayList referred to by xs, and assign this to ys. This can result in a lot of unnecessary copying of data and there are several simple strategies to reduce this cost:

(i) Use a CopyOnWriteArrayList to ensure that a full copy of the data is only made when it is actually necessary.

(ii) Use an intraprocedural dataflow analysis to determine when a variable is no longer used. For example, in the above, if xs is not live after the assignment to ys then cloning it is unnecessary.

(iii) Exploit compiler inferred read-only modifiers for function parameters. Such modifiers can be embedded into the JVM bytecode, and used to identify situations when an argument for an invocation does not need to be cloned.

Currently, we employ only `CopyOnWriteArrayLists` to improve performance, although we would like to further examine the benefits of those other approaches.

3.4 Runtime Type Tests

Implementing runtime type tests on the JVM represents something of a challenge. Whilst many runtime type tests translate directly using appropriate **`instanceof`** tests, this is not always the case:

```
int f(real x):
    if x ~= int:
        return x
    return 0
```

Although Whiley **`int`**s are implemented as Java `BigIntegers`, this test cannot be translated to "`e` **`instanceof`** `BigInteger`". This is because of Whiley's subtyping rules: `x` will be implemented by an instance of `BigRational` on entry, but because Whiley **`int`**s are subtypes of Whiley **`real`**s, the subtype check should succeed if the actual real passed in is actually an integer. The test is therefore translated into a check to see whether the given `BigRational` instance corresponds to an integer or not.

Union types also present a challenge because distinguishing the different cases is not always straightforward. For example:

```
define data as [real] | [[int]]

int f(data d):
    if d ~= [[int]]:
        return |d[0]|
    else:
        return |d|
```

Since variable `d` is guaranteed to be a list of some sought, its type on entry is translated to `List` on the JVM. Thus, Whiley cannot use an **`instanceof`** test on `d` directly to distinguish the two cases. Instead, we must examine the first element of the list and test this. Thus, if the first element of the list is itself a list, then the true branch is taken [3].

The following example presents another interesting challenge:

```
int f([real] d):
    if d ~= [int]:
        return d[0]
    return 0
```

[3] Note that in the case of an empty list, then type test always holds

To translate this test, we must loop over every element in the list and check whether or not it is an integer. Furthermore, if this is the case, we must convert the list into a list of `BigInteger`, rather than `BigRational`.

Finally, records present an interesting issue. Consider the following example:

```
define Rt as {int x, int y} | {int x, int z}

int unpackSecond(Rt r):
    if r ~= {int x,int y}:
        return r.y
    return r.z
```

Since variable `r` is guaranteed to be a record of some sort, its type on entry is translated as `Object[]`. Thus, implementing the type test using **`instanceof`** does not make sense, as this will not distinguish the two different kinds of record. Instead, we must check whether `r` has fields `x` and `y`, or not. To support this, the representation of records must also associate the corresponding field name with each slot in the `Object[]` array. This is achieved by reserving the first slot of the array as a reference to an array of field names, each of which identifies the name of a remaining slot from the outer array.

Distinguishing between different record types can be optimised by reducing the number of field presence checks. For example, in the above, there is little point in checking for the presence of field `x`, since it is guaranteed in both cases. The Whiley compiler generates the minimal number of field checks to be certain which of the possible cases is present.

4 Related Work

In this section, we concentrate primarily on work relating to Whiley's flow-sensitive type system.

4.1 Dataflow Analysis

Flow-sensitive dataflow analysis has been used to infer various kinds of information, including: *flow-sensitive type qualifiers* [21,22,12,17,27,13,1,41,4,35], *information flow* [30,44,36], *typestates* [49,19,8], *bytecode verification* [33,32] and more.

Type qualifiers constrain the possible values a variable may hold. CQual is a flow-sensitive qualifier inference supporting numerous type qualifiers, including those for synchronisation and file I/O [21,22]. CQual does not account for the effects of conditionals and, hence, retyping is impossible. The work of Chin *et al.* is similar, but flow-insensitive [12,13] JQual extended these systems to Java, and considered whole-program (flow-insensitive) inference [27]. AliasJava introduced

several qualifiers for reasoning about object ownership [1]. The `unique` qualifier indicates a variable holds the only reference to an object; similarly, `owned` is used to confine an object to the scope of its enclosing "owner" object.

Fähndrich and Leino discuss a system for checking non-null qualifiers in the context of C# [18]. Here, variables are annotated with `NonNull` to indicate they cannot hold **`null`**. Non-null qualifiers are interesting because they require variables be retyped after conditionals (i.e. retyping `v` from `Nullable` to `NonNull` after `v!=`**`null`**). Fähndrich and Leino hint at the use of retyping, but focus primarily on issues related to object constructors. Ekman *et al.* implemented this system within the JustAdd compiler, although few details are given regarding variable retyping [17]. Pominville *et al.* also briefly discuss a flow-sensitive non-null analysis built using SOOT, which does retype variables after `!=`**`null`** checks [41]. The JACK tool is similar, but focuses on bytecode verification instead [35]. This extends the bytecode verifier with an extra level of indirection called *type aliasing*. This enables the system to retype a variable `x` as `@NonNull` in the body a **`if`**`(x!=`**`null`**`)` conditional. The algorithm is formalised using a flow-sensitive type system operating on Java bytecode. JavaCOP provides an expressive language for writing type system extensions, including non-null types [4]. This system is flow-insensitive and cannot account for the effects of conditionals; as a work around, the tool allows assignment from a nullable variable `x` to a non-null variable if this is the first statement after a `x!=null` conditional.

Information Flow Analysis is the problem is tracking the flow of information, usually to restrict certain flows based for security reasons. The work of Hunt and Sands is relevant here, since they adopt a flow-sensitive approach [30]. Their system is presented in the context of a simple While language not dissimilar to ours, although they do not account for the effect of conditionals. Russo *et al.* use an extended version of this system to compare dynamic and static approaches [44]. They demonstrate that a purely dynamic system will reject programs that are considered type-safe under the Hunt and Sands system. JFlow extends Java with statically checked flow annotations which are flow-insensitive [36]. Finally, Chugh *et al.* developed a constraint-based (flow-insensitive) information flow analysis of JavaScript [15].

Typestate Analysis focuses on flow-sensitive reasoning about the state of objects, normally to enforce temporal safety properties. Typestates are finite-state automatons which can encode usage rules for common APIs (e.g. a file is never read before being opened), and were pioneered by Strom and Yellin [48,49]. Fink *et al.* present an interprocedural, flow-sensitive typestate verification system which is staged to reduce overhead [19]. Bodden *et al.* develop an interprocedural typestate analysis which is flow-sensitive at the intra-procedural level [9]. This is a hybrid system which attempts to eliminate all failure points statically, but uses dynamic checks when necessary. This was later extended to include a backward propagation step that improves precision [8].

Java Bytecode Verification requires a flow-sensitive typing algorithm [33]. Since locals and stack locations are untyped in Java Bytecode, it must infer their types to ensure type safety. Like Whiley, the verifier updates the type of a variable after an assignment, and combines types at control-flow join points using a least upper bound operator. However, it does not update the type of a variable after an **`instanceof`** test. Furthermore, the Java class hierarchy does not form a join semi-lattice. To deal with this, the bytecode verifier uses a simplified least upper bound operator which ignores interfaces altogether, instead relying on runtime checks to catch type errors (see e.g. [32]). However, several works on formalising the bytecode verifier have chosen to resolve this issue with intersection types instead (see e.g. [25,42]).

Gagnon *et al.* present a technique for converting Java Bytecode into an intermediate representation with a single static type for each variable [24]. Key to this is the ability to infer static types for the local variables and stack locations used in the bytecode. Since local variables are untyped in Java bytecode, this is not always possible as they can — and often do — have different types at different points; in such situations, a variable is split as necessary into multiple variables each with a different type.

Dubochet and Odersky [16] describe how structural types are implemented in Scala in some detail, and compare reflexive and generative approaches to implementing methods calls on structural types. They recognise that structural types always impose a penalty on current JVMs, but describe how both techniques generally provide sufficient performance in practice — about seven times slower than Java interface calls in the worse case.

5 Conclusion

The Whiley language implements a flow-sensitive and structural type system on the JVM. This permits variables to be declared implicitly, have multiple types within a function, and be retyped after runtime type tests. The result is a statically-typed language which, for the most part, has the look and feel of a dynamic language. In this paper, we have discussed various details relating to Whiley's implementation on the JVM. Finally, an open source implementation of the Whiley language is freely available from `http://whiley.org`.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *Proc. OOPSLA*, pages 311–330, 2002.

[2] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proc. DLS*, pages 53–64. ACM Press, 2007.

[3] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *Proc. ECOOP*, volume 3586 of *LNCS*, pages 428–452. Springer-Verlag, 2005.

[4] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proc. OOPSLA*, pages 57–74. ACM Press, 2006.

[5] F. Barbanera and M. Dezani-CianCaglini. Intersection and union types. In *Proc. of TACS*, volume 526 of *LNCS*, pages 651–674, 1991.

[6] G. Bierman, E. Meijer, and M. Torgersen. Lost in translation: formalizing proposed extensions to C#. In *Proc. OOPSLA*, pages 479–498, 2007.

[7] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strnisa, J. Vitek, and T. Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Proc. OOPSLA*, pages 117–136, 2009.

[8] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proc. ICSE*, pages 5–14, 2010.

[9] E. Bodden, P. Lam, and L. J. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proc. ESEC/FSE*, pages 36–47. ACM Press, 2008.

[10] L. Cardelli. Structural subtyping and the notion of power type. In *Proc. POPL*, pages 70–79. ACM Press, 1988.

[11] R. Cartwright and M. Fagan. Soft typing. In *Proc. PLDI*, pages 278–292. ACM Press, 1991.

[12] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Proc. PLDI*, pages 85–95. ACM Press, 2005.

[13] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *Proc. ESOP*, 2006.

[14] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proc. POPL*, pages 232–245. ACM Press, 1993.

[15] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proc. PLDI*, pages 50–62, 2009.

[16] G. Dubochet and M. Odersky. Compiling structural types on the JVM. In *ECOOP 2009 Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICLOOPS)*, 2009.

[17] T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *JOT*, 6(9):455–475, 2007.

[18] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. OOPSLA*, pages 302–312. ACM Press, 2003.

[19] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM TOSEM*, 17(2):1–9, 2008.

[20] C. Flanagan. Hybrid type checking. In *Proc. POPL*, pages 245–256. ACM Press, 2006.

[21] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proc. PLDI*, pages 192–203. ACM Press, 1999.

[22] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proc. PLDI*, pages 1–12. ACM Press, 2002.

[23] M. Furr, J.-H. An, J. Foster, and M. Hicks. Static type inference for Ruby. In *Proc. SAC*, pages 1859–1866. ACM Press, 2009.

[24] E. Gagnon, L. Hendren, and G. Marceau. Efficient inference of static types for java bytecode. In *Proc. SAS*, pages 199–219, 2000.

[25] A. Goldberg. A specification of java loading and bytecode verification. In *Proc. CCS*, pages 49–58, 1998.

[26] J. Gosling, G. S. B. Joy, and G. Bracha. *The Java Language Specification, 3rd Edition*. Prentice Hall, 2005.

[27] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for java. In *Proc. OOPSLA*, pages 321–336. ACM Press, 2007.

[28] S. Halloway. *Programming Clojure*. Pragmatic Programmers, 2009.

[29] T. Hoare. Null references: The billion dollar mistake, presentation at qcon, 2009.

[30] S. Hunt and D. Sands. On flow-sensitive security types. In *Proc. POPL*, pages 79–90. ACM Press, 2006.

[31] A. Igarashi and H. Nagira. Union types for object-oriented programming. *JOT*, 6(2), 2007.

[32] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.

[33] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

[34] R. P. Loui. In praise of scripting: Real programming pragmatism. *IEEE Computer*, 41(7):22–26, 2008.

[35] C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @NonNull types. In *Proc. CC*, pages 229–244, 2008.

[36] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. POPL*, pages 228–241, 1999.

[37] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998.

[38] L. D. Paulson. Developers shift to dynamic programming languages. *IEEE Computer*, 40(2):12–15, 2007.

[39] D. J. Pearce and J. Noble. Flow-sensitive types for whiley. Technical Report ECSTR10-23, Victoria University of Wellington, 2010.

[40] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[41] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proc. CC*, pages 334–554, 2001.

[42] C. Pusch. Proving the soundness of a java bytecode verifier specification in isabelle/hol. In *Proc. TACAS*, pages 89–103, 1999.

[43] D. Remy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[44] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. CSF*, pages 186–199, 2010.

[45] The scala programming language. http://lamp.epfl.ch/scala/.

[46] J. Siek and W. Taha. Gradual typing for objects. In *Proc. ECOOP*, volume 4609 of *LNCS*, pages 151–175. Springer-Verlag, 2007.

[47] D. Spinellis. Java makes scripting languages irrelevant? *IEEE Software*, 22(3):70–71, 2005.

[48] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, 12(1):157–171, 1986.

[49] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE TSE*, 19(5):478–485, 1993.

[50] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Proc. POPL*, pages 377–388. ACM Press, 2010.

Treegraph-based instruction scheduling for stack-based virtual machines

Jiin Park[a,2] Jinhyung Park[a,1], Wonjoon Song[a,1], Songwook Yoon[a,1], Bernd Burgstaller[a,2], Bernhard Scholz[b,3]

[a] *Department of Computer Science*
Yonsei University
Seoul, Korea

[b] *School of Information Technologies*
The University of Sydney
Sydney, Australia

Abstract

Given the growing interest in the JVM and Microsoft's CLI as programming language implementation targets, code generation techniques for efficient stack-code are required. Compiler infrastructures such as LLVM are attractive for their highly optimizing middleend. However, LLVM's intermediate representation is register-based, and an LLVM code generator for a stack-based virtual machine needs to bridge the fundamental differences of the register and stack-based computation models.
In this paper we investigate how the semantics of a register-based IR can be mapped to stack-code. We introduce a novel program representation called treegraphs. Treegraph nodes encapsulate computations that can be represented by DFS trees. Treegraph edges manifest computations with multiple uses, which is inherently incompatible with the consuming semantics of stack-based operators. Instead of saving a multiply-used value in a temporary, our method keeps all values on the stack, which avoids costly store and load instructions. Code-generation then reduces to scheduling of treegraph nodes in the most cost-effective way.
We implemented a treegraph-based instruction scheduler for the LLVM compiler infrastructure. We provide experimental results from our implementation of an LLVM backend for TinyVM, which is an embedded systems virtual machine for C.

Keywords: bytecode instruction scheduling, stack-code, treegraphs, DAGs, LLVM

[1] Authors contributed equally.

[2] Research partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0005234), and the OKAWA Foundation Research Grant (2009).

[3] Research partially supported by the Australian Research Council through ARC DP grant DP1096445.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

1 Introduction

Stack-code provides a compact instruction encoding, because instructions only encode the operation being performed, while its operands are implicitly consumed from the stack. Likewise, result-values of operations are implicitly produced on the stack. Network computing is facilitated by a compact program representation, because smaller programs take less time to transmit. For this reason the Java virtual machine (JVM, [12]) and various virtual machines (VMs) for wireless sensor networks, e.g., [14,19,9] employ a stack-based instruction set. Stack-based VMs are an attractive target for code generation, because several non-trivial tasks such as register allocation and callstack management are not required. All-in-all, 58 language implementations targeting the JVM and more than 52 programming languages targeting Microsoft's .NET's common language runtime (CLR, [17]) have been reported [24,23].

The renewed interest in stack-machines requires compiler support for the generation of efficient stack-code. The LLVM [11,10] compiler infrastructure is attractive for its highly-optimizing middle-end, but LLVM uses static single assignment form (SSA, [4]) as its intermediate representation (IR). Targeting LLVM to a stack-machine is complicated by the fact that SSA assumes an underlying register machine. Operands in registers can be used several times, whereas stack-based operations consume their operands.

Because the last-in, first-out stack protocol cannot support random operand access and unlimited operand lifetimes, operands that are not immediately used can be temporarily stored as local variables (temporaries) instead of keeping them on the stack. Generating bytecode to both minimize code size and improve performance by avoiding temporaries has been referred to as stack allocation [8,13]. A stack allocation method is required to provide the operands of each operation on the top of stack (TOS) when operations need them. If the underlying stack machine provides instructions to manipulate the stack, operands can be fetched from below the TOS. E.g., the JVM provides `SWAP` and `DUP` instructions to manipulate the stack. If the cost of a stack manipulation is less than the costs from stores and loads from a temporary, keeping a value on the stack is profitable.

In this paper we introduce a novel intermediate representation called treegraphs to facilitate stack allocation from register-based IRs. Treegraph nodes encapsulate computations that are compatible with a stack-based computation model. Edges between treegraph nodes represent dependencies on operands that have multiple uses. Multiple operand uses are inherently incompatible with the consuming semantics of stack-based operators. Instead of saving multiply-used values in temporaries, our method keeps all values on the stack. We perform stack allocation for treegraphs through scheduling of treegraph-nodes such that (1) dependencies between nodes are satisfied, and (2) the

number of nodes that don't require stack manipulation because their operands are already on the TOS is maximized.

The remainder of this paper is organized as follows. In Section 2 we provide background information and survey the related work. Section 3 introduces our treegraph IR and the basic treegraph scheduling algorithm. In Section 4 we discuss the minimization of overall stack manipulation costs. Section 5 contains experimental results. We draw our conclusions in Section 6.

2 Background and Related Work

Generating register-code for arithmetic expressions was first studied by Andrei Ershov [5]. Sethi and Ullman used Ershov numbers to devise an algorithm that they prove to generate optimal code for arithmetic expressions [18]. Aho and Johnson used dynamic programming to generate optimal code for expression trees on CISC machines [1].

Compiler back-ends for stack machines perform a DFS traversal of expression trees (the abstract syntax tree or other IR equivalent) and generate code as a side-effect. Common subexpression elimination finds expressions with multiple uses. The resulting DAGs complicate stack code generation: unlike values in registers, operands on the stack are consumed by their first use and are thus unavailable for subsequent uses. Fraser and Hanson's LCC [6] compiler for C comes with its own backend for a stack-based virtual machine [4]. LCC converts DAGs to trees by storing multiply-used values in temporaries and replacing references by references to the corresponding temporaries. This approach is taken with the LLVM backend for Microsoft .NET MSIL code, and it can be observed with code produced by Sun's javac compiler.

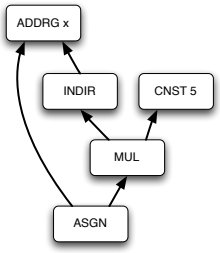
A significant amount of research has been conducted to reduce the number redundant store/load combinations by post-compilation transformations of bytecode [8,13,21,22]. In [15], dynamic instruction scheduling is performed to reduce the stack usage of a JVM. These methods are different from our approach in that they operate on the generated bytecode itself. Our approach avoids stores to temporaries altogether by keeping all values on the VM's evaluation stack.

TinyVM is a stack-based embedded systems VM for C [3]. TinyVM's instruction set closely resembles the instruction set of the bytecode backend of the LCC compiler [6]. TinyVM's instruction set is given in the appendix. Examples throughout this paper use the TinyVM instruction set.

[4] Different from the JVM

3 Treegraph IR and Treegraph Scheduling

Our optimization for the stack allocation problem is a *local* optimization, i.e., it is restricted to *single* basic blocks (BBs), where a BB is a sequence of statements with a single entry and a single exit [2].



(a)

```
1 ADDRG x
2 ADDRG x
3 INDIR
4 CNST 5
5 MUL
6 ASGN
```

(b)

```
1 ADDRL t0
2 ADDRG x
3 ASGN
4 ADDRL t0
5 ADDRL t0
6 INDIR
7 CNST 5
8 MUL
9 ASGN
```

(c)

```
1 ADDRG x
2 DUP
3 INDIR
4 CNST 5
5 MUL
6 ASGN
```

(d)

Fig. 1. Dependence-graph and instruction schedules for assignment statement `x = 5 * x`

An *acyclic* dependence graph $G(V, E, L, a)$ consists of a set of vertices V that represent instances of operations, and a set of edges $E \subseteq V \times V$ representing data dependencies among operations in V. Each each operation $o \in O$ has an arity[5] $a : O \to \mathbb{N}_0$. The mapping $L : V \to O$ assigns an operation to a vertex. In abuse of notation, we will refer to node u and $L(u)$ synonymously. A data dependency $(u, v) \in E$ is defined as a dependency between operation u and v such that operation u depends on operation v. The set of successors $S_u = \{w : (u, w) \in E\}$ of vertex u constitutes the operands of u, and they are totally ordered by relation $\prec_u \subseteq S_u \times S_u$, i.e., $v_1 \prec_u \ldots \prec_u v_k$ for successors $\{v_1, \ldots, v_k\}$. Note that the total order is given by the expected order of the operands on the stack. A dependence graph V is *well-formed* if $|S_u| = a(L(u))$, for all $u \in V$. The set P_u that denotes the predecessors of vertex u is defined similarly. We refer to operations u whose set of predecessors P_u is empty as *result* nodes of G. Figure 1(a) depicts a dependence graph for assignment statement `x=5*x`.

A stack machine can execute a sequence of instructions $\langle i_1, \ldots, i_l \rangle$ where an instruction belongs to one of the following instruction classes:

- Instruction $\texttt{op}_o$ is an instruction that performs operation $o \in O$ and has the signature $E^{a(o)} \to E$. The instruction pops $a(o)$ elements from the stack, generates one element as result, and pushes the result on top of the stack. If there are less than $a(o)$ elements on the stack, the stack machine will go into the *error state* and will terminate the execution.
- Instruction `DUP` k duplicates the top of stack element k times. Omitting argument k is equivalent to `DUP` 1. If there are no elements on the stack, the

[5] The arity is the number of operands of an operation.

stack machine will go into the error state and will terminate the execution.

- Instruction `FETCH` k duplicates element k (counted from the top) and pushes the duplicate onto the TOS. The stack size is incremented by one. If there are less than k elements on the stack, the machine will go into the error state and will terminate the execution.

A sequence of instructions $\langle i_1, \ldots, i_l \rangle$ is *admissible*, if for an empty stack none of the statements make the machine go into the error state and terminate the execution. Graph G is computed by the sequence of instructions $\langle i_1, \ldots, i_l \rangle$ iff the resulting stack contains the correct result.

Problem statement

Input: An acyclic dependence graph $G(V, E, L, a)$.
Output: code for a stack machine that performs the computations of G at minimum cost, i.e., with the minimum number of `DUP` and `FETCH` stack reordering instructions.

3.1 Special Case: G is a Tree

If G forms a tree, there exists only a single result node $r \in V$ which represents the root node of the tree. We can perform a depth-first search (DFS) as shown in Algorithm 1 where $operand(u, i)$ gives the i-th element of the total order $\prec_u$.

Algorithm 1: $DFS\,(G, u)$

```
1 foreach i ∈ {1, ..., |S_u|} do
2     DFS (G, operand(u, i))
3 emit op_{L(u)}
```

Proposition 3.1 *Sequence $DFS\,(G, r)$ is admissible and code optimal.*

Proof. The sequence is optimal because for each vertex a single instruction is emitted and this is a lower bound on the number of computations. The generated sequence is admissible and correct: This can be shown by structural induction, i.e., for each sub-tree it is shown that it is admissible and correct. The induction start are leaves of the tree. □

Note that the DFS sequence is unique in the absence of commutative operators. For a commutative operator like addition or multiplication, we get two unique trees, depending on which subtree of the commutative operator is traversed first by Algorithm 1.

Theorem 3.2 *No other instruction sequence for G is code optimal.*

Proof. The other sequence is either longer or destroys correctness. □

3.2 Code Generation for DAGs

If G is a DAG, there are no cycles but there exists a vertex v that has more than one predecessor in G.

Definition 3.3 Cut set $C \subseteq E$ is the set of all in-coming edges of nodes v which have more than one predecessor, i.e., $C = \{(u, v) \in E : |P_v| > 1\}$.

Lemma 3.4 *Graph $G(V, E - C)$ is a forest F.*

Proof. By definition every node in a forest has at most one predecessor. Hence, the lemma follows. □

The resulting forest is not well-formed in the sense that some of the operations depend on computations that are not performed inside their tree. To generate code for a DAG, we find the roots $r_1, \ldots, r_m$ for all trees $T_1, \ldots, T_m$ in F, i.e., this is the set of all nodes which have more than one predecessor in G or have no predecessors in G. We construct a tree-graph $H(F, A)$ whose set of vertices $F = \{T_1, \ldots, T_m\}$ are the trees of the forest and whose arcs $A \subseteq F \times F$ denote data dependencies between the trees, i.e., $(T_i, T_j) \in A$ iff there exists an edge $(u, v) \in E$ such that $u \in T_i$ and $v \in T_j$.

Lemma 3.5 *The tree-graph is a DAG.*

Proof. The properties of DAG G can only be destroyed by adding an edge that introduces a cycle. Condensating DAG G to a tree-graph $H(F, A)$ does not introduce additional edges, because every tree of forest F becomes a single node representing the root of the tree. The edges of the tree-graph A correspond to the cut-set C. □

For a tree $T_i \in F$ the set of successors are denoted by $\tilde{S}_{T_i}$. The successors are ordered by the depth-first-search order of T_i considering the root nodes of the successors in the order as well.

Lemma 3.6 *All trees $T_i \in F$ have either no predecessor or more than one predecessor in H.*

Proof. By construction of H. □

Algorithm 2 generates code for a treegraph H given a topological order R of H. The treegraph is traversed in reverse topological order. For every treegraph node T_i, the operands are fetched onto the TOS in reverse DFS order (lines 2–4) [6]. Then code for the tree represented by T_i is generated. The tree represented by T_i is a sub-graph of G and hence DFS gives the

[6] Fetching a value onto the TOS does not increase stack space requirements compared to storing values in temporaries, because at any one time at most one copy of a stack value is fetched to the top. Misplaced operands are popped after a treegraph has been executed, using a `POP k` instruction provided for this purpose.

Algorithm 2: Treegraph scheduler

```
1 foreach T_i ∈ F in reverse topological order R of H do
2     foreach T_j ∈ S̃_{T_i} in reverse DFS order of S̃_{T_i} do
3         if result value of T_j not in correct stack slot then
4             emit FETCH x_j
5     DFS(G, r_i);
6     if |P_{r_i}| > 0 then
7         N =Oracle (T_i, H, R) − 1;
8         if N > 0 then
9             emit DUP N
```

optimal code (line 5). Note that execution of the code generated from T_i will produce exactly one value ν on the stack unless T_i is a result node. If node T_i has predecessors (i.e., it is not a result node), then $P_{r_i} > 1$, i.e., node T_i has at least 2 predecessors (by the definition of treegraphs).

For all but the last predecessor v for which the value ν computed by T_i is in the correct stack slot, we duplicate ν (line 9). The last predecessor will consume ν itself. All other predecessors will have to fetch a copy of ν to the TOS before they are scheduled (line 4 for each of those predecessors). Clearly, the number of predecessors for which the value ν is in the correct stack slot depends on (1) the treegraph H, (2) the topological order R of H, and (3) the treenode $T_i \in H$. Algorithm 2 receives this number via an oracle (line 7). In Section 4, we will discuss our realization of this oracle via a symbolic execution of the stack state for given T_i, H, R.

As an example, consider the dependence graph of Figure 1(a). The corresponding treegraph consists of two nodes, one representing `ADDRG x`, and one representing the remaining dependence graph nodes. The reverse topological sorting for this example schedules `ADDRG x` first. The value ν produced by this treegraph node constitutes the memory address of global variable x (see the instruction set specification of TinyVM in the appendix). Value ν is already in the correct place for both uses of ν, so we duplicate once. The resulting TinyVM bytecode is depicted in Figure 1(d).

Figures 1(b) shows the bytecode where the multiply-used value is recomputed, and Figure 1(c) shows the approach where the multiply-used value is stored/loaded from a temporary variable `t0` (this is the approach chosen e.g., by the LCC compiler).

Theorem 3.7 *There exists a topological sort order R^* of H, that generates an admissible and code optimal instruction sequence.*

Proof. Local property: trees can only be generated code optimal by DFS,

i.e., for any re-ordering of instructions inside trees we can find an instruction sequence that does better (the DFS search), in which case the solution can be improved locally. □

Optimality will be achieved by a topological sort order where the number of values computed in the correct stack slot will be maximized. Section 4 will present an enumeration algorithm over all topological sortings of a treegraph.

4 Minimizing Stack Manipulation Costs

To enumerate all topological sorts of a given basic block's treegraph H, we use Knuth and Szwarcfiter's algorithm from [7]. For each topological sort, Algorithm 3 is invoked. This algorithm computes the number of `DUP` and `FETCH` instructions induced by a topological sort. The sum of the number of `DUP` and `FETCH` instructions (line 31) constitutes the cost of a topological sort. The second piece of information computed by Algorithm 3 is the oracle that tells for each treegraph node how many times the value ν computed by this node must be duplicated on the stack (the oracle was introduced with Algorithm 2).

Algorithm 3 maintains the state of the stack in variable `Stack`. When a treegraph node is about to get scheduled, the stack is consulted to check the positions of the treegraph's operands. We distinguish operations with one and two operands. Operands that are on the TOS are consumed (lines 4–5 for the two-operand case, and lines 6 and 10 for the one-operand case. For each operand not in the required position on the TOS we increase the overall counter for the required number of `FETCH` instructions by one (lines 7, 12 and 21).

To maintain the oracle-function that tells the number of times a treegraph node's result ν needs to be duplicated on the stack, the algorithm optimistically pushes one copy of ν on the stack for each predecessor. This number is saved with the oracle (lines 24–27). Every time we encounter an operand value which is not in the correct stack position, we decrement this operand's duplication counter in the oracle (lines 9, 15–16 and 23). The superfluous copy is then removed from the stack (lines 8, 13–14 and 22).

The topological sort R of minimum cost and the associated oracle are then used in Algorithm 2 to schedule treegraph nodes and emit bytecode.

We used a timeout to stop enumeration for basic blocks with too many topological sorts. For those cases the solution of minimum cost seen so far was used.

Algorithm 3: computeBasicBlockCosts

```
   Input: topsort sequence Seq of treegraph nodes
   Output: cost of Seq, Oracle[...] for number of dupped values
 1 Oracle [...] ← {}; Dup ← 0; Fetch ← 0; Stack ← empty;
 2 foreach treegraph node SU in reverse Seq do
 3     if NumberOfOperands(SU) = 2 then
 4         if Stack.top() = SU.op[1] and Stack.second() = SU.op[0] then
 5             Stack.pop(); Stack.pop()
 6         else if Stack.top() = SU.op[0] then
 7             Fetch = Fetch + 1;
 8             Stack.eraseOne(SU.op[1]);
 9             Oracle[SU.op[1]] − −;
10             Stack.pop();
11         else
12             Fetch = Fetch + 2;
13             Stack.eraseOne(SU.op[0]);
14             Stack.eraseOne(SU.op[1]);
15             Oracle[SU.op[0]] − −;
16             Oracle[SU.op[1]] − −;
17     else if NumberOfOperands(SU) = 1 then
18         if Stack.top() = SU.op[0] then
19             Stack.pop();
20         else
21             Fetch = Fetch + 1;
22             Stack.eraseOne(SU.op[0]);
23             Oracle[SU.op[0]] − −;
24     if SU computes result ν then
25         for 1 to SU.NumberOfPreds do
26             Stack.push(SU)
27         Oracle[SU] = SU.NumberOfPreds;
28 foreach treegraph node SU in Seq do
29     if Oracle[SU] > 1 then
30         Dup = Dup + 1;
31 return Dup + Fetch, Oracle[...];
```

5 Experimental Results

Figure 2 shows our experimental setup. C source code was compiled to LLVM IR by llvm-gcc ("Frontend-end"). LLVM's optimizing middle-end was by-

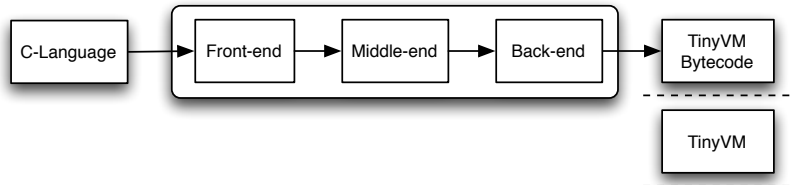


Fig. 2. Overview of the experimental setup

passed for this experiment. We implemented an LLVM backend for the generation of TinyVM bytecode from LLVM IR, based on LLVM version 2.5. The generated bytecode was then benchmarked on TinyVM.

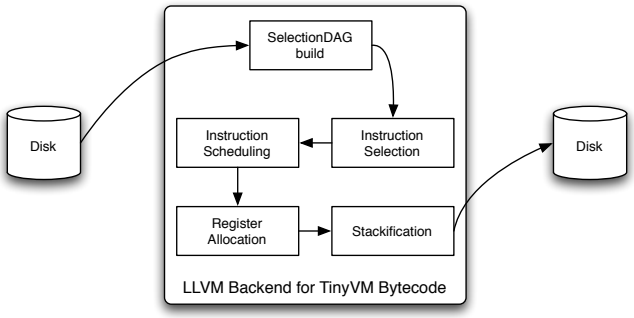


Fig. 3. LLVM Backend Structure

Figure 3 depicts the structure of our TinyVM bytecode backend. We implemented the "Instruction Selection" pass to lower LLVM IR to TinyVM bytecode instructions[7]. These bytecode instructions use pseudo-registers to accommodate SSA virtual registers. At this stage, the IR consists of DAGs where each DAG-node corresponds to one scheduable unit of bytecode instructions (i.e., a sequence of instructions that should be scheduled together). "Instruction Scheduling" denotes our treegraph instruction scheduler. For each basic block DAG we create the corresponding treegraph and run our treegraph scheduling algorithm. We use LLVM's 'local' register allocator to ensure that there are no live ranges between basic blocks. (Our optimization works on a per basic block basis; we did not consider global stack allocation yet.) Our "Stackification" pass converts pseudoregister code to stack code. This is a straight-forward conversion where operands that correspond to stack slots are dropped. For example, `ADDRG R1 x` would become `ADDRG x`. For further details we refer to [16].

All experiments were performed on an Intel Xeon 5120 server running Linux CentOS 5.4 with kernel version 2.6.18. We selected 24 C benchmark programs from the testsuite that comes with the LLVM compiler infrastruc-

[7] "Lowering" denotes the change of abstraction-level by converting LLVM IR to TinyVM bytecode.

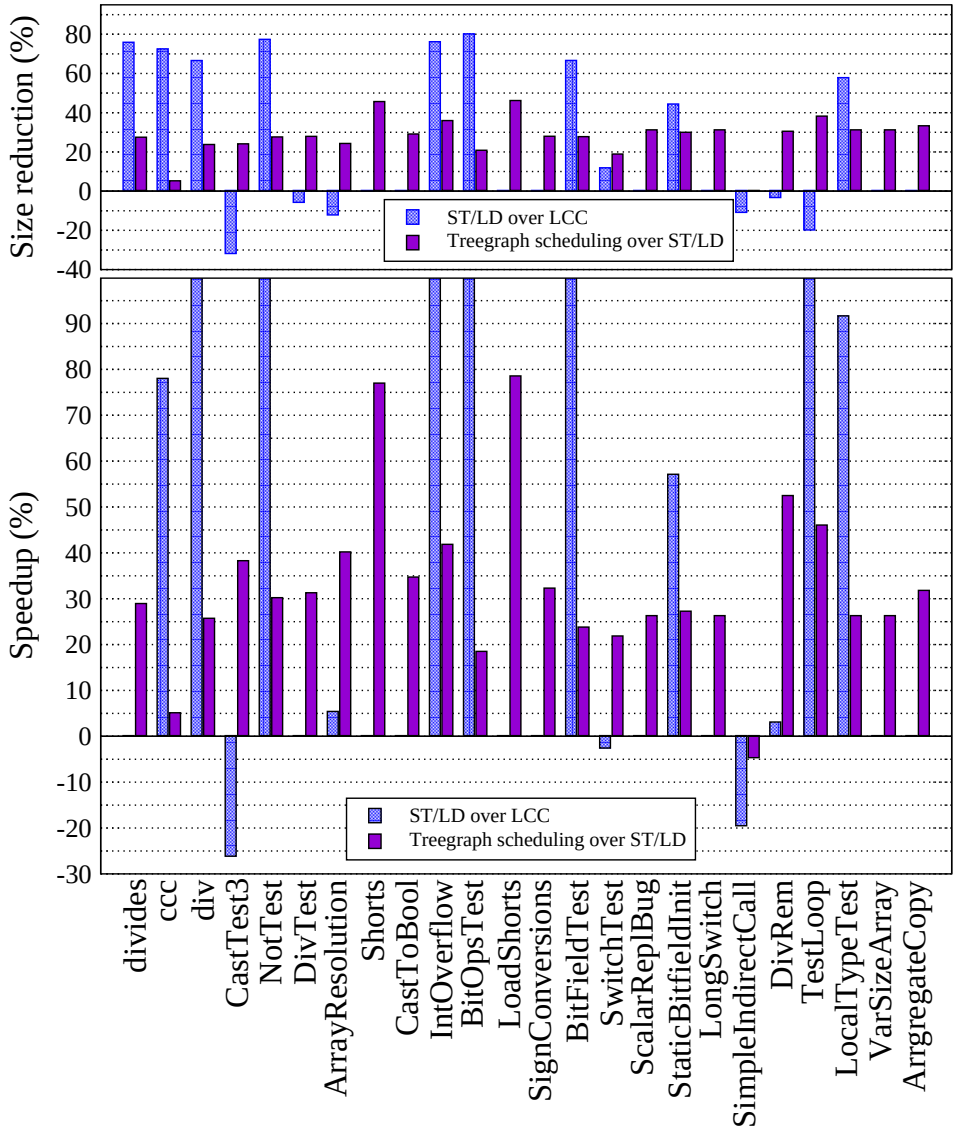


Fig. 4. Speedup and size reductions for ST/LD over LCC, and for treegraph scheduling over ST/LD. No bar with ST/LD over LCC means that the corresponding benchmark was not ANSI C and thus could not be compiled with LCC.

ture [20]. Our TinyVM backend cannot handle floats and struct args yet, which is reflected in our selection of benchmarks.

Figure 4 contains the size reductions and the speedups obtained for our benchmarks programs. Size reductions where computed as $1 - \frac{\text{newsize}}{\text{oldsize}}$.

As a yardstick for our comparison, we implemented the store/load mechanism to temporaries for our LLVM backend (i.e., every multiply-used value is written to a temporary and loaded from there when the value is required). Figure 4 depicts the improvements of our store/load mechanism over the LCC bytecode. It should be noted that LCC also applies store/load of temporaries and that the improvements are largely due to the superior code quality achiev-

able with LLVM. Note also that benchmarks with 0 improvement for ST/LD denote cases where a benchmark was not ANSI C and thus not compilable by LCC. The "Treegraph scheduling" data in Figure 4 denotes the improvement of our treegraph scheduling technique over ST/LD.

For 93% of all basic blocks our treegraph scheduler could derive the optimal solution, for the remaining 7% the enumeration of topological sorts hit the 2 second timeout. For 86% of basic blocks the solve-time was below 0.08 seconds.

6 Conclusions

In this paper we investigated how the semantics of a register-based IR can be mapped to stack-code. We introduced a novel program representation called treegraphs. Treegraph nodes encapsulate computations that can be represented by DFS trees. Treegraph edges manifest computations with multiple uses. Instead of saving a multiply-used value in a temporary, our method keeps all values on the stack, which avoids costly store and load instructions. Values that are in the correct stack slot for (some of) their users are duplicated so that they can be consumed without stack manipulation. All other values are lifted to the top of stack via a FETCH instruction. We implemented our treegraph scheduler with the LLVM compiler infrastructure for TinyVM, a stack-based embedded systems VM for C.

References

[1] Aho, A. V. and S. C. Johnson, *Optimal code generation for expression trees*, in: *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing* (1975), pp. 207–217.

[2] Aho, A. V., M. S. Lam, R. Sethi and J. D. Ullman, "Compilers: principles, techniques, and tools," Addison-Wesley, 2007, second edition.

[3] Burgstaller, B., B. Scholz and M. A. Ertl, *An Embedded Systems Programming Environment for C*, in: *Proc. Euro-Par'06* (2006), pp. 1204–1216.

[4] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Trans. Program. Lang. Syst. **13** (1991), pp. 451–490.

[5] Ershov, A. P., *On programming of arithmetic expressions*, Communications of the ACM **1** (1958).

[6] Hanson, D. R. and C. W. Fraser, "A Retargetable C Compiler: Design and Implementation," Addison Wesley, 1995.

[7] Kalvin, A. D. and Y. L. Varol, *On the generation of all topological sortings*, Journal of Algorithms **4** (1983), pp. 150 – 162.

[8] Koopman, P. J., *A preliminary exploration of optimized stack code generation*, Journal of Forth Applications and Research **6** (1994).

[9] Koshy, J. and R. Pandey, *VMSTAR: Synthesizing Scalable Runtime Environments for Sensor Networks*, in: *SenSys '05: Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems* (2005), pp. 243–254.

[10] Lattner, C. and V. Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in: *Proc. CGO'04* (2004).

[11] Lattner, C. A., *LLVM: an infrastructure for multi-stage optimization*, Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec (2002).

[12] Lindholm, T. and F. Yellin, "The Java Virtual Machine Specification," The Java Series, Addison Wesley Longman, Inc., 1999, second edition.

[13] Maierhofer, M. and M. Ertl, *Local stack allocation*, in: *Compiler Construction*, 1998 pp. 189–203.
URL http://dx.doi.org/10.1007/BFb0026432

[14] Müller, R., G. Alonso and D. Kossmann, *SwissQM: Next Generation Data Processing in Sensor Networks*, in: *CIDR '07: Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research* (2007), pp. 1–9.

[15] Munsil, W. and C.-J. Wang, *Reducing stack usage in Java bytecode execution*, SIGARCH Comput. Archit. News **26** (1998), pp. 7–11.

[16] Park, J., "Optimization of TinyVM Bytecode Using the LLVM Compiler Infrastructure," Master's thesis, Department of Computer Science, Yonsei University, Korea (2011).

[17] Richter, J., "CLR Via C#," Microsoft Press, 2010, third edition.

[18] Sethi, R. and J. D. Ullman, *The generation of optimal code for arithmetic expressions*, J. ACM **17** (1970), pp. 715–728.

[19] Simon, D., C. Cifuentes, D. Cleal, J. Daniels and D. White, *Java™ on the bare metal of wireless sensor devices: the squawk java virtual machine*, in: *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments* (2006), pp. 78–88.

[20] The LLVM Project, *http://llvm.org/, retrieved 2010.*

[21] Valle-Rai, R., E. Gagnon, L. Hendren, P. Lam, P. Pominville and V. Sundaresan, *Optimizing Java bytecode using the Soot framework: Is it feasible?*, in: *Compiler Construction*, LNCS **1781** (2000).

[22] Vandrunen, T., A. L. Hosking and J. Palsberg, *Reducing loads and stores in stack architectures* (2000).
URL www.cs.ucla.edu/~palsberg/draft/vandrunen-hosking-palsberg00.pdf

[23] Wikipedia, *List of CLI Languages, http://en.wikipedia.org/wiki/List_of_CLI_languages, retrieved Oct. 2010.*

[24] Wikipedia, *List of JVM Languages, http://en.wikipedia.org/wiki/List_of_JVM_languages, retrieved Oct. 2010.*

A TinyVM Instruction Set

Instruction	IS-Op.	Suffixes	Description
ADD SUB	—	FIUP..	integer addition, subtraction
MUL DIV	—	FIU...	integer multiplication, division
NEG	—	FI....	negation
BAND BOR BXOR	—	.IU...	bitwise and, or, xor
BCOM	—	.IU...	bitwise complement
LSH RSH MOD	—	.IU...	bit shifts and remainder
CNST	a	.IUP..	push literal a
ADDRG	p	...P..	push address p of global
ADDRF	l	...P..	push address of formal parameter, offset l
ADDRL	l	...P..	push address of local variable, offset l
BADDRG	index	...P..	push address of mc entity at index
INDIR	—	FIUP..	pop p; push ∗p
ASGN	—	FIUP..	pop arg; pop p; ∗p = arg
ASGN_B	a	.....B	pop q, pop p; copy the block of length a at ∗q to p
CVI	—	FIU...	convert from signed integer
CVU	—	.IUP..	convert from unsigned integer
CVF	—	FI....	convert from float
CVP	—	..U...	convert from pointer
LABEL	—	....V.	label definition
JUMP	target	....V.	unconditional jump to target
IJUMP	—	....V.	indirect jump
EQ GE GT LE LT NE	target	FIU...	compare and jump to target
ARG	—	FIUP..	top of stack is next outgoing argument
CALL	target	....V.	vm procedure call to target
ICALL	—	....V.	pop p; call procedure at p
INIT	l	....V.	allocate l stack cells for local variables
BCALL	—	FIUPVB	mc procedure call
RET	—	FIUPVB	return from procedure call
HALT	—	....V.	exit the vm interpreter
POP	k	....V.	pop k values from the TOS

Table A.1
TinyVM bytecode instruction set

Table A.1 depicts the TinyVM bytecode instruction set. The TinyVM instruction set is closely related to the bytecode interface that comes with LCC [6]. Instruction opcodes cover the leftmost column whereas the column headed "IS-Op." lists operands derived from the instruction stream (all other instruction operands come from the stack). The column entitled "Suffixes" denotes the valid type suffixes for an operand (F=float, I=signed

integer, `U`=unsigned integer, `P`=pointer, `V`=void, `B`=struct).[8] In this way instruction `ADDRG` receives its pointer argument `p` from the instruction stream and pushes it onto the stack. Instructions `ADDRF` and `ADDRL` receive an integer argument literal from the instruction stream; this literal is then used as an offset to the stack framepointer to compute the address of a formal or local variable. Unlike the JVM, TinyVM uses an `ADDR* / INDIR` instruction sequence to load a value onto the stack. To store a value, TinyVM uses the `ASGN` instruction.

[8] Operators contain *byte size* modifiers (i.e., 1, 2, 4, 8), which we have omitted for reasons of brevity.

Handling non-linear operations in the value analysis of COSTA

Diego Alonso [a] Puri Arenas[a] Samir Genaim[a]

[a] *DSIC, Complutense University of Madrid (UCM), Spain*

Abstract

Inferring precise relations between (the values of) program variables at different program points is essential for termination and resource usage analysis. In both cases, this information is used to synthesize ranking functions that imply the program's termination and bound the number of iterations of its loops. For efficiency, it is common to base *value analysis* on non-disjunctive abstract domains such as Polyhedra, Octagon, etc. While these domains are efficient and able to infer complex relations for a wide class of programs, they are often not sufficient for modeling the effect of non-linear and bit arithmetic operations. Modeling such operations precisely can be done by using more sophisticated abstract domains, at the price of performance overhead. In this paper we report on the value analysis of COSTA that is based on the idea of encoding the disjunctive nature of non-linear operations into the (abstract) program itself, instead of using more sophisticated abstract domains. Our experiments demonstrate that COSTA is able to prove termination and infer bounds on resource consumption for programs that could not be handled before.

1 Introduction

Termination and resource usage analysis of imperative languages have received a considerable attention [3,22,20,8,19,13,14]. Most of these analyses rely on a value (or size) analysis component, which infers relations between the values of the program variables (or the sizes of the corresponding data structures) at different program points. This information is then used to bound the number of iterations of the program's loops. Thus, the precision of value analysis directly affects the class of (terminating) programs for which the corresponding tool is able prove termination or infer lower and upper bounds on their resource consumption. Moreover, in the case of resource consumption, it also affects the quality of the inferred bounds (i.e., how tight there are).

Typically, for efficiency, the underlying abstract domains used in value analysis are based on conjunctions of linear constraints, e.g., Polyhedra [10], Octagons [18], etc. While in practice these abstract domains are precise enough for bounding the loops of many programs, they are often not sufficient when the considered program involves non-linear arithmetic operations (multiplication, division, bit arithmetics, etc). This is because the semantics of such operations cannot be modeled precisely with only conjunctions of linear constraints. In order to overcome this limitation,

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

one can use abstract domains that support non-linear constraints, however, these domain typically impose a significant performance overhead. Another alternative is to use disjunctive abstract domains, i.e., disjunctions of (conjunctions of) linear constraints. This allows splitting the behavior of the corresponding non-linear operation into several mutually exclusive cases, such that each one can be precisely described using only conjunctions of linear constraints. This alternative also imposes performance overhead, since the operations of such disjunctive abstract domains are usually more expensive.

In this paper, we develop a value analysis that handles non-linear arithmetic operations using disjunctions of (conjunctions of) linear constraints. However, similarly to [21], instead of directly using disjunctive abstract domains, we encode the disjunctive nature of the non-linear operations directly in the (abstract) program. This allows using non-disjunctive domains like Polyhedra, Octagons, etc., and still benefit from the disjunctive information in order to infer more precise relations for programs with non-linear arithmetic operations. We have implemented a prototype of our analysis in COSTA, a COSt and Termination Analyser for Java bytecode. Experiments on typical examples from the literature demonstrate that COSTA is able to handle programs with non-linear arithmetics that could not be handled before.

The rest of this paper is organized as follows: Section 2 briefly describes the intermediate language on which we develop our analysis (Java bytecode programs are automatically translated to this language); Section 3 motivates the techniques we use for handling non-linear arithmetic operations; Section 4 describes the different components of our value analysis; Section 5 presents a preliminary experimental evaluation using COSTA; and, finally, we conclude in Section 6.

2 A Simple Imperative Intermediate Language

We present our analysis on a simple *rule-based* imperative language [1] which is similar in nature to other representations of bytecode [23,16]. For simplicity, we consider a subset of the language presented in [1], which deals only with methods and arithmetic operations over integers. In the implementation we handle full sequential Java bytecode. A *rule-based program* P consists of a set of *procedures*. A procedure p with k input arguments $\bar{x} = x_1, \ldots, x_k$ and m output arguments $\bar{y} = y_1, \ldots, y_m$ is defined by one or more *guarded rules*. Rules adhere to this grammar:

$$
\begin{array}{rl}
\mathit{rule} ::= & p(\bar{x}, \bar{y}) \ \leftarrow g, b_1, \ldots, b_n \\
g ::= & \mathit{true} \mid e_1 \ \mathit{op} \ e_2 \mid g_1 \wedge g_2 \\
b ::= & x{:=}e \mid x{:=}e - e \mid x{:=}e + e \mid q(\bar{x}, \bar{y}) \\
& x{:=}e * e \mid x{:=}e \ / \ e \mid x{:=}e \ \texttt{rem} \ e \\
& x{:=}e \ \otimes \ e \mid x{:=}e \oplus e \mid x{:=}e \triangleright e \mid x{:=}e \triangleleft e \\
e ::= & x \mid n \\
\mathit{op} ::= & >|<|\leq|\geq|=
\end{array}
$$

where $p(\bar{x}, \bar{y})$ is the *head* of the rule; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_n$ the body of the rule; n an integer; x and y variables and $q(\bar{x}, \bar{y})$ a procedure call by value. The arithmetic operations / and `rem` refer respectively to integer division and remainder. They have the semantics

```
int m(int x,int b) {
  int y=1;
  int z=0;
  if (b > 1) {
    while (y < x) {
      z=z+1;
      y=y*b;
    }
  }
  return z;
}
```

$m(\langle x, b\rangle, \langle r\rangle) \leftarrow$
  $y{:=}1,$
  $z{:=}0,$
  $m_1(\langle x, b, y, z\rangle, \langle r\rangle).$
$m_1(\langle x, b, y, z\rangle, \langle r\rangle) \leftarrow$
  $b \leq 1,$
  $r{:=}z.$
$m_1(\langle x, b, y, z\rangle, \langle r\rangle) \leftarrow$
  $b > 1,$
  $m_2(\langle x, b, y, z\rangle, \langle y_1, z_1\rangle),$
  $r{:=}z_1.$

$m_2(\langle x, b, y, z\rangle, \langle y, z\rangle) \leftarrow$
  $y \geq x.$
$m_2(\langle x, b, y, z\rangle, \langle y_2, z_2\rangle) \leftarrow$
  $y < x,$
  $z_1{:=}z + 1,$
① $y_1{:=}y * b,$
  $m_2(\langle x, b, y_1, z_1\rangle, \langle y_2, z_2\rangle).$

Fig. 1. A Java program and its intermediate representation. Method m computes $\lceil \log_b(x) \rceil$.

of the bytecode instructions `idiv` and `irem` [17]. Operations $\otimes$, $\oplus$, $\triangleleft$ and $\triangleright$ refer respectively to bitwise AND, bitwise OR, left shift and right shift. They have the semantics of the bytecode instructions `iand`, `ior`, `ishl`, and `ishr` [17]. We ignore the overflow behavior of these instruction, supporting them is left for future work.

The key features of this language which facilitate the formalization of the analysis are: (1) *recursion* is the only iterative mechanism, (2) *guards* are the only form of conditional, (3) there is no operand stack, and (4) rules may have *multiple output* parameters which is useful for our transformation. The translation from Java bytecode programs to rule-based programs is performed in two steps. First, a control flow graph (CFG) is built. Second, a *procedure* is defined for each basic block in the CFG and the operand stack is *flattened* by considering its elements as additional local variables. The execution of rule-based programs mimics standard bytecode [17]. Multiple output arguments in procedures come from the extraction of loops into separated procedure (see Example 2.1). For simplicity, we assume that each rule in the program is given in static single assignment (SSA) form [5].

Example 2.1 Figure 1 depicts the Java code (left) and the corresponding intermediate representation (right) of our running example. Note that our analysis starts from the bytecode, the Java code is shown here just for clarity. Procedure m is defined by one rule, it receives x and b as input, and returns r as output, i.e., r corresponds to the return value of the Java method. Rule m corresponds to the first two instructions of the Java method, it initializes local variables y and z, and then passes the control to m_1. Procedure m_1 corresponds to the **if** statement, and is defined by two mutually exclusive rules. The first one is applied when $b \leq 1$, and simply returns the value of z in the output variable r. The second one is applied when $b > 1$, it calls procedure m_2 (the loop), and upon exit from m_2 it returns the value of z_1 in the output variable r. Note that z_1 refers to the value of z upon exit from procedure m_2 (the loop), it is generated by the SSA transformation. Procedure m_2 corresponds to the **while** loop, and is defined by two mutually exclusive rules. The first one is applied when the loop condition is evaluated to *false*, and the second one when it is evaluated to *true*. Note that m_2 has two output variables, they correspond to the values of y and z upon exit from the loop.

$m(\langle x, b\rangle, \langle r\rangle) \leftarrow$
$\quad \{y = 1\},$
$\quad \{z = 0\},$
$\quad m_1(\langle x, b, y, z\rangle, \langle r\rangle).$
$m_1(\langle x, b, y, z\rangle, \langle r\rangle) \leftarrow$
$\quad \{b \leq 1\},$
$\quad \{r = z\}.$
$m_1(\langle x, b, y, z\rangle, \langle r\rangle) \leftarrow$
$\quad \{b > 1\},$
$\quad m_2(\langle x, b, y, z\rangle, \langle y_1, z_1\rangle),$
$\quad \{r = z_1\}.$

$m_2(\langle x, b, y, z\rangle, \langle y, z\rangle) \leftarrow$
$\quad \{y \geq x\}.$
$m_2(\langle x, b, y, z\rangle, \langle y_2, z_2\rangle) \leftarrow$
$\quad \{y < x\},$
$\quad \{z_1 = z + 1\},$
① $\quad \{y_1 = \top\},$
$\quad m_2(\langle x, b, y_1, z_1\rangle, \langle y_2, z_2\rangle).$

Fig. 2. Abstract compilation of the program of Figure 1

3 Motivating Example

Proving that the program of Figure 1 terminates, or inferring lower and upper bounds on its resource consumption (e.g., number of execution steps), requires bounding the number of iterations that its loop can make. Bounding the number of iterations of a loop is usually done by finding a function f from the program states to a well-founded domain, such that if s and s' are two states that correspond to two consecutive iterations, then $f(s) > f(s')$. Traditionally, this function is called *ranking function* [11]. Note that for termination, it is enough to prove that such function exists, while inferring bounds on the resource consumption requires synthesizing such ranking function. For the program of Figure 1, if the program state is represented by the tuple $\langle x, b, y, z\rangle$, then $f(\langle x, b, y, z\rangle) = \mathtt{nat}(x - y)$, where $\mathtt{nat}(v) = \max(v, 0)$, is a ranking function for the **while** loop. Moreover, this function can be further refined to $f(\langle x, b, y, z\rangle) = \log_2(\mathtt{nat}(x - y) + 1)$, which is more accurate for the sake of inferring bounds on the loop's resource consumption.

In this paper we follow the analysis approach used in [1], which divides the value analysis into several steps: (1) an *abstract compilation* [15] step that generates an abstract version of the program, replacing each instruction by an abstract description (e.g., conjunction of linear constraints) that over-approximates its behavior; (2) a fixpoint computation step that computes an abstract semantics of the program; and (3) in the last, we prove termination or infer bounds on resource consumption using the abstract program of point 1 and the abstract semantics of point 2.

Applying the first step on the program of Figure 1 results in the abstract program of Figure 2. It can be observed that linear arithmetic instructions are precisely described by their corresponding abstract versions. For example, $\mathrm{z}_1{:=}\mathrm{z}+1$ updates z_1 to hold the value of $\mathrm{z}+1$, and its corresponding abstract version $\{z_1 = z + 1\}$ is a denotation which states that the value of z_1 is equal to the value of z plus 1. However, in the case of non-linear arithmetic instructions, the abstract description often loses valuable information. This is the case of the instruction $\mathrm{y}_1{:=}\mathrm{y} * \mathrm{b}$ which is annotated with ① in both Figures 1 and 2. While the instruction updates y_1 to hold the value of $\mathrm{y} * \mathrm{b}$, its abstract description $\{y_1 = \top\}$ states that y_1 can take any value. Here $\top$ should be interpreted as any integer value. This abstraction

makes it impossible to bound the number of iterations of the loop, since in the abstract program the function $f(\langle x, b, y, z\rangle) = \mathtt{nat}(x - y)$ does not decrease in each two consecutive iterations.

Without any knowledge on the values of y and b, the constraint $\{y_1 = \top\}$ is indeed the best description for y_1:=y∗b when only conjunctions of linear constraints are allowed. However, in the program of Figure 1 it is guaranteed that the value of y is positive and that of b is greater than 1. Using this context information the abstraction of y_1:=y ∗ b can be improved to $\{y_1 \geq 2 * y\}$, which in turn allows synthesizing the ranking function $f(\langle x, b, y, z\rangle) = \mathtt{nat}(x - y)$ and its refinement $f(\langle x, b, y, z\rangle) = \log_2(\mathtt{nat}(x - y) + 1)$. This suggests that the abstract compilation can benefit from context information when only conjunctions of linear constraints are allowed. However, the essence of abstract compilation is to use only syntactic information, and clearly context information cannot be obtained always by syntactic analysis of the program.

One way to solve the loss of precision when abstracting non-linear arithmetic instructions is to allow the use of disjunctions of linear constraints. For example, the instruction y_1:=y ∗ b could be abstracted to $\varphi_1 \vee \cdots \vee \varphi_n$ where each φ_i is a conjunction of linear constraints that describes a possible scenario. E.g., we could have $\varphi_j = \{y \geq 1, b \geq 2, y_1 \geq 2 * b\}$ in order to handle the case in which $y \geq 1$ and $b \geq 2$. Then, during the fixpoint computation, when the context becomes available, the appropriate φ_i will be automatically selected. However, for efficiency reasons, we restrict our value analysis to use only conjunctions of linear constraints. In order to avoid the use of disjunctive constraints, similarly to [21], we follow an approach that encodes the disjunctive information into the (abstract) program itself. For example, the second rule of m_2 would be abstracted to:

$$
\begin{array}{ll}
m_2(\langle x, b, y, z\rangle, \langle y_2, z_2\rangle) \leftarrow & op_*(\langle a, b\rangle, \langle c\rangle) \leftarrow \{a = 0, c = 0\}. \\
\quad \{y < x\}, & op_*(\langle a, b\rangle, \langle c\rangle) \leftarrow \{a = 1, c = b\}. \\
\quad \{z_1 = z + 1\}, & \qquad\qquad \vdots \\
\textcircled{1}\ \ op_*(\langle y, b\rangle, \langle y_1\rangle), & op_*(\langle a, b\rangle, \langle c\rangle) \leftarrow \{a \geq 2, b \geq 2, c \geq 2 * a\}. \\
\quad m_2(\langle x, b, y_1, z_1\rangle, \langle y_2, z_2\rangle). &
\end{array}
$$

Here, the instruction y_1:=y ∗ b was abstracted to $op_*(\langle y, b\rangle, \langle y_1\rangle)$ which is a call to an auxiliary abstract rule that defines possible abstract scenarios for different inputs. During the fixpoint computation, since op_* is called in a context in which $y \geq 1$ and $b \geq 2$, only the second and last rules of op_* will be selected. Then, these two rules propagate the constraint $y_1 \geq 2 * y$ back, which is required for synthesizing the expected ranking functions, without using disjunctive abstract domains.

4 Value Analysis

In this section we describe the value analysis of COSTA, which is based on the ideas presented in Section 3. The analysis receives as input a program in the intermediate language and a set of initial entries, and, for each (abstract) procedure $p(\bar{x}, \bar{y})$ it infers: (1) A pre-condition (over $\bar{x}$) that holds whenever p is called; and (2) a post-condition (over $\bar{x}$ and $\bar{y}$) that holds upon exit from p. The pre- and post-conditions

are conjunction of linear constraints over the domain of Polyhedra [10]. Later, they can be composed in order to obtain invariants for some program points of interest.

In Section 4.1 we describe the abstract compilation step which translates the program P into an abstract version P^α. In Section 4.2 we describe a standard fixpoint algorithm that is used to infer the pre- and post-conditions. Finally, in Section 4.3 we explain how this information is used for bounding the number of iterations of the program's loops.

4.1 Abstract Compilation

This section describes how to transform a given program P into an abstract program P^α. In the implementation, we support also the abstraction of data-structures using the path-length measure [22] (the depth of a data-structure) and the abstraction of arrays to their length. However, in this paper we omit these features since they do not benefit from the techniques we use for abstracting non-linear arithmetic operations. Given a rule $\mathrm{r} \equiv \mathrm{p}(\bar{\mathrm{x}}, \bar{\mathrm{y}}) \leftarrow \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$, the abstract compilation of r is $r^\alpha \equiv p(\bar{x}, \bar{y}) \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha$, where:

(i) the abstract guard g^α is equal to the (linear) guard g;

(ii) if $\mathrm{b_i} \equiv \mathrm{q}(\bar{\mathrm{z}}, \bar{\mathrm{w}})$, then $b_i^\alpha \equiv q(\bar{z}, \bar{w})$;

(iii) if $\mathrm{b_i} \equiv \mathrm{x{:=}e_1 \diamond e_2}$ and $\diamond \in \{+, -\}$, then $b_i^\alpha \equiv \{x = e_1 \diamond e_2\}$; and

(iv) if $\mathrm{b_i} \equiv \mathrm{x{:=}e_1 \diamond e_2}$ and $\diamond \notin \{+, -\}$, then $b_i^\alpha \equiv op_\diamond(\langle e_1, e_2\rangle, \langle x\rangle)$

Then, $P^\alpha = \{r^\alpha \mid \mathrm{r} \in P\}$. Note that we use the same names for constraint variables as those of the program variables (but in italic font for clarity). This is possible since we have assumed that the rules of P are given in SSA form. In the above abstraction, linear guards (point i) and linear arithmetic instructions (point iii) are simply replaced by a corresponding constraint that accurately model their behavior. Note that $\mathrm{x{:=}e_1 \diamond e_2}$ is an assignment while $\{x = e_1 \diamond e_2\}$ is an equality constraint. In point ii, calls to procedures are simply replaced by calls to abstract procedures. In what follows we explain the handling of non-linear arithmetic (point iv).

If the elements of the underlying abstract domain consist only in conjunctions of linear constraints, then non-linear operations are typically abstracted to $\top$. As we have seen in Section 3, this results in a significant loss of precision that prevents bounding the loop's iterations. A well-know solution is to use disjunctions of linear constraints which allow splitting the input domain into special cases that can be abstracted in a more accurate way. This can be done by directly using disjunctive abstract domains, however, this comes on the price of performance overhead. The solution we use in our implementation, inspired by [21], is to encode the disjunctions in the (abstract) program itself, without the need for using disjunctive abstract domains. In practice, this amounts to abstracting the non-linear arithmetic instruction $\mathrm{x{:=}e_1 \diamond e_2}$ into a call $op_\diamond(\langle e_1, e_2\rangle, \langle x\rangle)$ to an auxiliary abstract procedure $op_\diamond$, which is defined by several rules that cover all possible inputs and simulate the corresponding disjunction. The rules of $op_\diamond$ are designed by partitioning its input domain and, for each input class, define the strongest possible post-condition. Clearly, the more partitions there are, the more precise are the post-conditions, but the more expensive is the analysis too. Therefore, when designing the rules of

$op_\diamond$ this performance and precision trade-off should be taken into account. For the purposes of termination and resource usage analyzes, the partitioning of the input domain aims at propagating accurate information about constancy, equality and progression (e.g, multiplication by a constant), with the least possible number of rules. In what follows, we explain the auxiliary abstract procedures associated to the non-linear arithmetic operations of our language.

Integer division. The auxiliary abstract rule $op_{\texttt{rem}}$ and $op_{/}$ are defined in terms of $op_{\texttt{dr}}$ which stands for $x = y * q + r$:

$$
\begin{array}{|l|}
\hline
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x = 0, q = 0, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{y = 1, q = x, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{y = -1, q = -x, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x = y, q = 1, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x = -y, q = -1, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x > y > 1, 0 < q \leq \frac{x}{2}, 0 \leq r < y\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{-x > y > 1, \frac{x}{2} \leq q < 0, -y < r \leq 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x > -y > 1, -\frac{x}{2} \leq q < 0, 0 \leq r < -y\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{-x > -y > 1, 0 < q \leq -\frac{x}{2}, y < r \leq 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{|y| > |x|, q = 0, r = x\}. \\
\hline
op_{/}(\langle x, y\rangle, \langle q\rangle) \quad \leftarrow op_{\texttt{dr}}(\langle x, y\rangle, \langle q, \_\rangle). \\
op_{\texttt{rem}}(\langle x, y\rangle, \langle r\rangle) \leftarrow op_{\texttt{dr}}(\langle x, y\rangle, \langle \_, r\rangle). \\
\hline
\end{array}
$$

Note that, in practice, abstract rules that involve $|\cdot|$ are folded into several cases. The sixth rule, for example, states that if $x > y > 1$ then x/y is a positive number smaller than or equal to $\frac{x}{2}$, and x rem y is a non-negative number smaller than y. This rule is also essential for synthesizing logarithmic ranking functions, when the input value is reduced at least by half in every iteration. Note that we ignore the special cases when $x =$ `MIN_VALUE` and $y = -1$, since it is a kind of overflow behavior.

Multiplication. The auxiliary abstract procedure op_* is defined as follows:

$$
\begin{array}{|l|}
\hline
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x = 0, z = 0\}. \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x = 1, z = y\}. \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x = -1, z = -y\}. \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x \geq 2, y \geq 2, z \geq 2 * x, z \geq 2 * y\}. \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x \leq -2, y \geq 2, z \leq 2 * x, z \leq -2 * y\}. \\
op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x \leq -2, y \leq -2, z \geq -2 * x, z \geq -2 * y\}. \\
\hline
\end{array}
$$

We have omitted those rules that can be obtained by swapping the arguments x and y. In this abstraction, we distinguish the cases in which $x = 0$ (constancy), $x = \pm 1$ (equality) and those in which $|x| > 1$ and $|y| > 1$ (progress). Note that, for example, the post-condition $z \geq 2 * x$ is essential for finding a logarithmic ranking function for loops like that of Figure 2. For example, it is not be possible to synthesize such ranking function if we use a weaker, yet sound, post-condition $z > x$.

The bitwise $\otimes$ and $\oplus$. The auxiliary abstract rules $op_{\otimes}$ and $op_{\oplus}$ are defined in terms of op_{ao} as follows:

$$
\begin{array}{|l|}
\hline
op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x = 0, a = 0, o = y\}. \\
op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x = -1, a = y, o = -1\}. \\
op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x = y, a = x, o = x\}. \\
op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x > y > 0, 0 \leq a \leq y, o \geq x\}. \\
op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x > 0, y < -1, 0 \leq a \leq x, y \leq o \leq -1\}. \\
op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x < y < -1, a \leq x, y \leq o \leq -1\}. \\
\hline
op_{\otimes}(\langle x, y\rangle, \langle a\rangle) \quad \leftarrow op_{\mathsf{ao}}(\langle x, y\rangle, \langle a, \_\rangle). \\
op_{\oplus}(\langle x, y\rangle, \langle o\rangle) \quad \leftarrow op_{\mathsf{ao}}(\langle x, y\rangle, \langle \_, o\rangle). \\
\hline
\end{array}
$$

Since these operations are commutative we omit rules derivable by swapping the input arguments. The first two rules describe the cases $x = 0$ and $x = -1$, i.e., vectors in which all bits are respectively 0 or 1. The third rule handles the case $x = y$. The rest of rules are based on that the result of $x \otimes y$ has less 1-bits than either x or y, whereas the result of $x \oplus y$ has more 1-bits than either x or y.

Shift left and right. Although shift operations in Java bytecode accept any integer value as the shift operand, the number of shifted positions is determined only by the five least significant bits, i.e., it is a value between 0 and $2^5 - 1$ (for type `long` it is determined by the six least significant bits). For the shift left operation $\triangleleft$, the auxiliary abstract procedure $op_{\triangleleft}$ is defined as follows:

$$
\begin{array}{|l|}
\hline
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x = 0, z = 0\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{s = 0, z = x\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x > 0, 0 < |s| < 2^5, z \geq 2x\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x < 0, 0 < |s| < 2^5, z \leq 2x\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x > 0, |s| \geq 2^5, z \geq x\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x < 0, |s| \geq 2^5, z \leq x\}. \\
\hline
\end{array}
$$

The above rules provide an accurate post-condition when the shift operand s satisfies $0 \leq |s| < 2^5$. In the last two abstract rules, the post-conditions are respectively $z \geq x$ and $z \leq x$ since we cannot observe the value of the first five bits of s when $|s| \geq 2^5$. Similarly, for the shift right operation $\triangleright$, the auxiliary abstract rule $op_{\triangleright}$ is defined as follows:

$$
\begin{array}{|l|}
\hline
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x = 0, z = 0\}. \\
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x = -1, z = -1\}. \\
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{s = 0, z = x\}. \\
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x > 0, 0 < |s| < 2^5, x > z, x \geq 2z, z \geq 0\}. \\
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x < -1, 0 < |s| < 2^5, x - 1 \leq 2z, z < 0\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x > 0, |s| \geq 2^5, 0 \leq z \leq x\} \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x < 0, |s| \geq 2^5, x \leq z \leq -1\}\}. \\
\hline
\end{array}
$$

Note that when the program includes several non-linear instructions for the same operations, then it might be useful to generate different auxiliary abstract

procedures for them, e.g, op^1_*, op^2_*, etc. This is required mainly when the calling contexts of these instructions are disjoint, and therefore separating their auxiliary abstract procedures avoids merging the calling contexts, which usually results in a loss of precision. In addition, non-linear arithmetic instructions that do not affect the termination of the program can be abstracted as before, i.e., to $\{x = \top\}$, and thus avoid the performance overhead caused by unnecessary auxiliary abstract procedures. These instructions can be identified using dependency analysis, similar to what have been done in [4] for identifying program variables that affect termination.

4.2 Fixpoint algorithm

Algorithm 1 implements the value analysis using a top-down strategy in the style of [7]. It receives as input an abstract program P^α and a set of initial pre-conditions E, and computes pre- and post-conditions for each procedure in P (stored in tables PRE and POST respectively). The meaning of a pre-condition $\mathsf{PRE}[q(\bar{x})] \equiv \varphi$, is that φ holds when calling q, and of a post-condition $\mathsf{POST}[q(\bar{x}, \bar{y})] \equiv \varphi$ is that φ holds upon exit from q.

Procedure FIXPOINT initializes the event queue $\mathcal{Q}$ to $\emptyset$ (L2), initializes the elements of tables PRE and POST to *false* (L4 and L5), processes the initial pre-conditions E by calling ADD_PRE for each one (L6) which in turn adds the corresponding event to $\mathcal{Q}$, and then in the while loop it processes the events of $\mathcal{Q}$ until no more events are available. In each iteration, an event q (a procedure name) is removed from $\mathcal{Q}$ (L8) and processed as follows: the current pre-condition ψ of q is retrieved (L9), each of the rules of q is evaluated in order to generate a post-condition for that specific rule w.r.t. ψ (L11), all post-conditions are joint into a single element δ (using the least upper-bound $\sqcup$ of the underlying abstract domain), and finally δ is added as a post-condition for q by calling ADD_POST. Note that the call to ADD_POST might add more events to $\mathcal{Q}$. The evaluation of a rule (procedure EVALUATE) w.r.t. a pre-condition ψ processes each b^α_i in the rule's body B as follows: if b^α_i is a call $q'(\bar{w}, \bar{z})$, then it registers the corresponding pre-condition by calling ADD_PRE (L16) and adds the current post-condition of q to ψ (L17); otherwise, b^α_i is a constraint and it simply adds it to ψ (L18).

Procedure ADD_PRE adds a new pre-condition for q if it does not imply the current one, and adds the corresponding event to $\mathcal{Q}$. Procedure ADD_POST adds a new post-condition for q if it does not imply the current one, and adds events for all procedures that call q since they might have to be re-analyzed. Note that both procedures use the least upper bound $\sqcup$ of the underlying abstract domain in order to join the new pre- or post-conditions with the current one. Note also that since we use abstract domains with infinite ascending chains, in practice, these procedures incorporate a widening operator in order to ensure termination.

Example 4.1 Consider again the abstract program of Figure 2, where the second abstract rule of m_2 is replaced by

$$m_2(\langle x, b, y, z\rangle, \langle y_2, z_2\rangle) \leftarrow \\ \{y < x\},\ \{z_1 = z + 1\},\ op_*(\langle b, y\rangle, \langle y_1\rangle),\ m_2(\langle x, b, y_1, z_1\rangle, \langle y_2, z_2\rangle).$$

and the initial set of entries $E = \{\langle m(\langle x, b\rangle), true\rangle\}$. Then, the fixpoint algorithm

Algorithm 1 The fixpoint algorithm

1: **procedure** FIXPOINT(P^α, E)
2: $\mathcal{Q} = \emptyset$;
3: **for all** $q(\bar{x}, \bar{y}) \in P$ **do**
4: $\mathsf{PRE}[q(\bar{x})] = \mathit{false}$;
5: $\mathsf{POST}[q(\bar{x}, \bar{y})] = \mathit{false}$;
6: **for all** $\langle p(\bar{x}), \varphi \rangle \in E$ **do** ADD_PRE($p(\bar{x}), \varphi$);
7: **while** $\mathcal{Q}.\mathit{notempty}()$ **do**
8: $q = \mathcal{Q}.\mathit{poll}()$;
9: $\psi = \mathsf{PRE}[q(\bar{x})]$;
10: $\delta = \mathit{false}$;
11: **for all** $q(\bar{x}, \bar{y}) \leftarrow B^\alpha \in P^\alpha$ **do** $\delta = \delta \sqcup$ EVALUATE($q(\bar{x}, \bar{y}) \leftarrow B^\alpha, \psi$);
12: ADD_POST($q(\bar{x}, \bar{y}), \delta$);
13: **function** EVALUATE($q(\bar{x}, \bar{y}) \leftarrow B^\alpha, \psi$)
14: **for all** $b_i^\alpha \in B^\alpha$ **do**
15: **if** $b_i^\alpha \equiv q'(\bar{w}, \bar{z})$ **then**
16: ADD_PRE($q'(\bar{w}), \bar{\exists}\bar{w}.\psi$);
17: $\psi = \psi \sqcap \mathsf{POST}[q'(\bar{w}, \bar{z})]$;
18: **else** $\psi = \psi \sqcap b_i^\alpha$;
19: return $\bar{\exists}\bar{x} \cup \bar{y}.\psi$;
20: **procedure** ADD_PRE($q(\bar{x}), \varphi$)
21: $\psi = \mathsf{PRE}[q(\bar{x})]$;
22: **if** $\varphi \not\models \psi$ **then**
23: $\mathsf{PRE}[q(\bar{x})] = \psi \sqcup \varphi$;
24: $\mathcal{Q}.\mathit{add}(q)$;
25: **procedure** ADD_POST($q(\bar{x}, \bar{y}), \varphi$)
26: $\delta = \mathsf{POST}[q(\bar{x}, \bar{y})]$;
27: **if** $\delta \not\models \varphi$ **then**
28: $\mathsf{POST}[q(\bar{x}, \bar{y})] = \delta \sqcup \varphi$;
29: **for all** $p \in P$ **do**
30: **if** p calls q **then** $\mathcal{Q}.\mathit{add}(p)$;

infers $\mathsf{PRE}[m_2(\langle x, b, y, z \rangle)] = \{z \geq 0, y \geq 1, b \geq 2\}$, $\mathsf{PRE}[op_*(\langle b, y \rangle)] = \{b > 1, y \geq 1\}$, and $\mathsf{POST}[op_*(\langle b, y \rangle, \langle y_1 \rangle)] = \{y_1 \geq 2 * y\}$.

4.3 Bounding the loops

In this section we describe how the abstract program and the pre- and post-conditions are used in order to bound the program's loops, as done in [1]. Briefly, for each abstract rule $p(\bar{x}, \bar{y}) \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha \in P^\alpha$, we generate a set of transitions

$$\left\{ \langle p(\bar{x}) \rightarrow q(\bar{w}), \bar{\exists}\bar{x} \cup \bar{w}.\varphi \rangle \;\middle|\; \begin{array}{l} i \in [1, \ldots, n],\ b_i^\alpha = q(\bar{w}, \bar{z}), \\ \varphi = \ \mathsf{PRE}[q(\bar{x})] \wedge g^\alpha \wedge \phi(b_1^\alpha) \cdots \wedge \phi(b_{i-1}^\alpha) \end{array} \right\}$$

where $\bar{\exists}\bar{x} \cup \bar{w}.\varphi$ is the projection of φ on the variables $\bar{x} \cup \bar{w}$; $\phi(b_i^\alpha) = b_i^\alpha$ if b_i^α is a constraint; and $\phi(b_i^\alpha) = \mathsf{POST}[b_i^\alpha]$ if b_i^α is a call. Then, the set of all transitions

is passed to, for example, the tool of [2], which in turn infers ranking functions for the corresponding loops.

Example 4.2 Using the abstract rule and the pre- and post-conditions of Example 4.1, we generate the transition relation $\langle m_2(\langle x, b, y, z\rangle) \rightarrow m_2(\langle x, b, y_1, z_1\rangle), \varphi\rangle$, where $\varphi = \{z \geq 0, y \geq 1, b \geq 2, x < y, z_1 = z + 1, y_1 \geq 2 * y\}$. Then, the solver of [2] infers the expected ranking functions as explained in Section 3.

5 Experimental Evaluation

We have implemented, in the context of COSTA [3], a prototype of the value analysis described in Section 4. We have performed some experiments on typical examples from the literature that use non-linear and bit arithmetic operations. The benchmarks are available at `http://costa.ls.fi.upm.es/papers/bytecode2011`. Unfortunately, the implementation cannot be tried out via COSTA's web-interface since it has not been integrated in the main branch yet.

COSTA, with the new value analysis, was able to prove termination of all benchmarks. Note that without this value analysis COSTA could not handle any of these benchmarks. We have also analyzed the benchmarks using other termination analyzers for Java bytecode. Julia [1] [22] was not able to prove termination of any of these benchmarks. AProVE [2] [12] could not prove termination of programs with bit arithmetic operations, but could handle programs with non-linear arithmetic operations such as multiplication and integer division, except for the program of Figure 1 for which it could not complete the proof in a time limit of 5 minutes. In what follows we explain the results of our analysis on some of the benchmarks.

EX1: We start with an example borrowed from [9]:

```
void and(int x){
  while( x > 0)
    x = x & x-1;
}
```

$$
\begin{array}{l}
and(\langle x\rangle, \langle\rangle) \leftarrow and_1(\langle x\rangle, \langle\rangle).\\
and_1(\langle x\rangle, \langle\rangle) \leftarrow \{x \leq 0\}.\\
and_1(\langle x\rangle, \langle\rangle) \leftarrow \{x > 0\},\\
\quad \{y = x - 1\},\\
\quad op_{\otimes}(\langle x, y\rangle, \langle x_1\rangle),\\
\quad and_1(\langle x_1\rangle, \langle\rangle).
\end{array}
$$

The code on the right is the abstract compilation of the corresponding intermediate representation of the Java method. In order to bound the number of iterations of the **while** loop, it is essential to infer that the value of x decreases in each iteration. This cannot be guaranteed when considering the instruction x=x & x−1 separately, since, for example, it does not decrease when x=0. Our analysis infers the pre-condition $\mathsf{PRE}[op_{\otimes}(x, y)] = \{y = x-1, x > 0\}$, i.e., the context $x > 0$ is available when calling $op_{\otimes}$, which in turn makes it possible to infer the post-condition $\mathsf{POST}[op_{\otimes}(\langle x, y\rangle, \langle x_1\rangle)] = \{y = x - 1, x > 0, 0 \leq x_1 \leq x - 1\}$. Using this information we generate the transition $\langle and_1(\langle x\rangle) \rightarrow and_1(\langle x_1\rangle), \{x > 0, 0 \leq x_1 \leq x - 1\}\rangle$ for which we synthesize the ranking function $f(\langle x\rangle) = \mathtt{nat}(x)$.

[1] using the online version `http://julia.scienze.univr.it/`
[2] using the online version `http://aprove.informatik.rwth-aachen.de/`

EX2: The next example implements the Euclidean algorithm for computing the greatest common divisor of two natural numbers. It is taken from the Java bytecode termination competition database [3]:

```
int gcd(int a, int b){
  int tmp;
  while(b>0 && a>0){
    tmp = b;
    b = a % b;
    a = tmp;
  }
  return a;
}
```

$$
\begin{array}{l}
gcd(\langle a,b\rangle,\langle r\rangle) \leftarrow gcd_1(\langle a,b\rangle,\langle r\rangle).\\
gcd_1(\langle a,b\rangle,\langle a\rangle) \leftarrow \{a \leq 0\}.\\
gcd_1(\langle a,b\rangle,\langle a\rangle) \leftarrow \{b \leq 0\}.\\
gcd_1(\langle a,b\rangle,\langle r\rangle) \leftarrow\\
\quad \{a>0, b>0\},\\
\quad \{tmp = b\},\\
\quad op_{\texttt{rem}}(\langle a,b\rangle,\langle b_1\rangle),\\
\quad \{a_1 = tmp\},\\
\quad gcd_1(\langle a_1,b_1\rangle,\langle r\rangle).
\end{array}
$$

COSTA was not able to prove termination of this program in the competition of July 2010, mainly because it ignores the calling context when abstracting b=a % b, and therefore it cannot infer that b decreases. Our analysis infers the pre-condition $\mathsf{PRE}[op_{\texttt{rem}}(\langle a,b\rangle)] = \{a>0, b>0\}$, which in turn makes it possible to infer the post-condition $\mathsf{POST}[op_{\texttt{rem}}(\langle a,b\rangle,\langle b_1\rangle)] = \{a>0, b>0, b>b_1\}$. Using this information we generate the transition $\langle gcd_1(\langle a,b\rangle) \rightarrow gcd_1(\langle a_1,b_2\rangle), \{a>0, b>0, b>b_1\}\rangle$ for which we synthesize the ranking function $f(\langle a,b\rangle) = \texttt{nat}(b)$.

EX3: The next example is taken from the method `toString(int i, int radix)` of class `java.lang.Integer`. It is used for writing a number in any numeric base. For simplicity, we have removed code that does not affect the termination, and annotated the loop with a pre-condition that is inferred by our analysis:

```
// {i <= 0, 2 <= radix <= 32}
while (i <= -radix) {
    i = i / radix;
}
```

$$
\begin{array}{l}
p(\langle i, radix\rangle,\langle\rangle) \leftarrow \{i > -radix\}.\\
p(\langle i, radix\rangle,\langle\rangle) \leftarrow\\
\quad \{i \leq -radix\},\\
\quad op_{/}(\langle i, radix\rangle,\langle i_1\rangle),\\
\quad p(\langle i_1, radix\rangle,\langle\rangle).
\end{array}
$$

Due to the pre-condition $\mathsf{PRE}[op_{/}(\langle i, radix\rangle,\langle i_1\rangle)] = \{2 \leq radix \leq 32, i \leq -radix\}$, our analysis infers the post-condition $\mathsf{POST}[op_{/}(\langle i, radix\rangle,\langle i_1\rangle)] = \{2 \leq radix \leq 32, i \leq -radix, \frac{i}{2} \leq i_1 < 0\}$. Using this post-condition we generate the transition $\langle p(\langle i, radix\rangle) \rightarrow p(\langle i_1, radix\rangle), \{2 \leq radix \leq 32, i \leq -radix, \frac{i}{2} \leq i_1 < 0\}\rangle$ for which we synthesize the ranking function $f(\langle i, radix\rangle) = \log_2(\texttt{nat}(-i)+1)$.

EX4: The next example is a variation of a loop from method `toUnsignedString(int i, int shift)` of class `java.lang.Integer`, which is used for writing a number in binary, octal or hexadecimal form:

[3] http://termcomp.uibk.ac.at

```
// { 1 <= shift <= 4 }
while ( i > 0 ) {
    i >>= shift;
}
```

$$
\begin{array}{l}
p(\langle i, \mathit{shift}\rangle, \langle\rangle) \leftarrow \{i \leq 0\}. \\
p(\langle i, \mathit{shift}\rangle, \langle\rangle) \leftarrow \\
\quad \{i > 0\}, \\
\quad op_{\triangleright}(\langle i, \mathit{shift}\rangle, \langle i_1\rangle), \\
\quad p(\langle i_1, \mathit{shift}\rangle, \langle\rangle).
\end{array}
$$

Due to the pre-condition $\mathsf{PRE}[op_{\triangleright}(\langle i, \mathit{shift}\rangle)] = \{i > 0, 1 \leq \mathit{shift} \leq 4\}$, our analysis is able to infer the post-condition $\mathsf{POST}[op_{\triangleright}(\langle i, \mathit{shift}\rangle, \langle i_1\rangle)] = \{i > 0, 1 \leq \mathit{shift} \leq 4, i \geq 2 * i_1, i_1 \geq 0\}$. Using this post-condition we generate the transition $\langle p(\langle i, \mathit{shift}\rangle) \rightarrow p(\langle i_1, \mathit{shift}\rangle), \{i > 0, 1 \leq \mathit{shift} \leq 4, i \geq 2 * i_1, i_1 \geq 0\}\rangle$ for which we synthesize the ranking function $f(\langle i, \mathit{shift}\rangle) = \log_2(\mathtt{nat}(i) + 1)$.

6 Conclusions

In this paper we have described how we handle non-linear arithmetic instructions in the value analysis of COSTA. It is well-know that handling such operations is problematic when the underlying abstract domain allows only the use of conjunctions of linear constraints. It is also well-know that the use of disjunctive abstract domains is a possible solution to this problem, however, on the price of performance overhead. In this paper, instead of using disjunctive abstract domains, we encoded the disjunctive nature of non-linear arithmetic instructions into the abstract program itself. This encoding, when combined with a value analysis that is based on non-disjunctive abstract domains such as Polyhedra or Octagons, makes it possible to dynamically select the best abstraction depending on the context from which the code that correspond to the encoding was reached. Our experiments demonstrate that COSTA is now able to prove termination and infer bound on resource consumption for programs that it could not handle before. For future work, we plan to improve the scalability of the analyzer, support overflow in arithmetic operations, and support floating point arithmetic. Note that, given the latest developments in the Parma Polyhedra Library [6], supporting overflow and floating point arithmetic is relatively straightforward.

Acknowledgement

This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project. Diego Alonso is partially supported by the UCM PhD scholarship program.

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In Gilles Barthe and Frank de Boer, editors, *IFIP International Conference on Formal*

Methods for Open Object-based Distributed Systems (FMOODS'08), volume 5051 of *Lecture Notes in Computer Science*, pages 2–18, Oslo, Norway, June 2008. Springer-Verlag, Berlin.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *6th International Symposioum on Formal Methods for Components and Objects (FMCO'08)*, number 5382 in Lecture Notes in Computer Science, pages 113–133. Springer, 2007.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In *ACM Symposium on Applied Computing (SAC) - Software Verification Track (SV08)*, pages 368–375, Fortaleza, Brasil, March 2008. ACM Press, New York.

[5] A. W. Appel. Ssa is Functional Programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[6] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006. Available at http://www.cs.unipr.it/Publications/. Also published as arXiv:cs.MS/0612085, available from http://arxiv.org/.

[7] Michael Codish. Efficient goal directed bottom-up evaluation of logic programs. *J. Log. Program.*, 38(3):355–370, 1999.

[8] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426, Tucson, Arizona, USA, 2006.

[9] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2010.

[10] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM Symposium on Principles of Programming Languages (POPL'78)*. ACM Press, 1978.

[11] R. W. Floyd. Assigning Meanings to Programs. In J.T Schwartz, editor, *Proceedings of Symposium in Applied Mathematics*, volume 19, Mathematical Aspects of Computer Science, pages 19–32. American Mathematical Society, Providence, RI, 1967.

[12] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated Termination Proofs with AProVE. In *Proc. of 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, volume LNCS 3091, pages 210–220. Springer-Verlag, 2004.

[13] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Symposium on Principles of Programming Languages (POPL'09)*, pages 127–139. ACM, 2009.

[14] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 292–304. ACM, 2010.

[15] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[16] H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In *2nd Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode'07)*, Electronic Notes in Theoretical Computer Science, pages 35–50. Elsevier, 2007.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[18] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[19] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 6–86. Elsevier - North Holland, March 2009.

[20] Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of java bytecode by term rewriting. In Christopher Lynch, editor, *RTA*, volume 6 of *LIPIcs*, pages 259–276. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[21] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static Analysis in Disjunctive Numerical Domains. In *Static Analysis, 13th International Symposium, (SAS'06)*, volume 4134 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2006.

[22] Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32(3), 2010.

[23] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *CASCON*, pages 125–135. IBM, 1999.

Static Resource Analysis for Java Bytecode Using Amortisation and Separation Logic

Damon Fenacci[1] Kenneth MacKenzie[2]

School of Informatics
The University of Edinburgh
Edinburgh, UK

Abstract

In this paper we describe a static analyser for Java bytecode which uses a combination of amortised analysis and Separation Logic due to Robert Atkey. With the help of Java annotations we are able to give precise resource utilisation constraints for Java methods which manipulate various heap-based data structures.

Keywords: Java, JVM, Bytecode, Resource analysis, Amortisation, Separation Logic

1 Introduction

In [4], Robert Atkey shows how methods from amortised complexity analysis can be combined with Separation Logic to obtain a technique for resource analysis of imperative programs which manipulate heap-based data-structures such as trees and linked lists. He shows how to apply this method to a small stack-based virtual machine, similar to the JVM. In this paper we describe an analyser which applies this analysis to real JVM bytecode, using specifications obtained from programmer-supplied annotations in Java source code.

Outline

We begin with an outline of the ideas involved in the analysis. This is followed by a detailed description of the Java annotations used to communicate the

[1] Email: D.Fenacci@sms.ed.ac.uk
[2] Email: kwxm@inf.ed.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

specifications to our analyser, together with examples of the analyser in action. We also describe some of the methods used in the implementation of the analyser.

Acknowledgments

The work described in this paper was carried out in the RESA project (EPSRC Follow-on Fund grant number EP/G006032/1) at the University of Edinburgh. [3] More information can be found in [3]. We would like to thank Robert Atkey for extensive discussions.

2 Specifying resource consumption

Much of this paper is based on previous work of Robert Atkey [4]. This work is somewhat technical from the point of view of non-experts, and in this section we will attempt to give a non-technical overview. We present a fairly simple example here, but we hope to make an online demonstration available on the RESA webpages including more complex examples.

Specifying heap behaviour

Consider the following Java code:

```
class IntList {
   int head;
   IntList tail;

   IntList concat (IntList p, IntList q) {
      if (p == null) return q;
      else {
         IntList t = p;
         while (t.next != null)
            t = t.next;
         t.next = q;
         return p;
      }

   ...
   }
}
```

This defines a simple class of linked lists with integer entries and a method which concatenates two lists `p` and `q`. If `p` is empty then `append` returns `q`, otherwise a pointer `t` traverses `p` until it reaches the final cell, then adjusts its `next` field to point to `q` and returns `p`.

Suppose that we want to describe the behaviour of `concat`. Let us introduce an assertion `lseg(a,b)` which states that there is a well-formed list segment in the heap for which `a` points to the first cell and `b` points to the final

[3] http://groups.inf.ed.ac.uk/resa/

cell. Intuitively, in the heap we have a picture of the form shown in Figure 1, with all cells distinct.

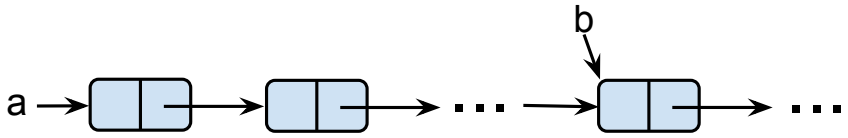


Fig. 1. List segment `lseg(a,b)`.

A first attempt at a specification for the `concat` method as might be as follows [4]:

```
@Requires (lseg(@arg p,null), lseg(@arg q,null))  // precondition
@Ensures (lseg (@ret, null))                      // postcondition
IntList concat (IntList p, IntList q) { ... }
```

The intended meaning of this specification is that *if* when we enter the method, the arguments `p` and `q` point to well-defined list segments, *then* when we reach the end of the method, the return value will also point to a well-defined list segment. The specification may be regarded as a contract with the user: if the inputs to the method satisfy the precondition, then the output is guaranteed to satisfy the postcondition. Ideally, we will be able to *prove* that the implementation of the method does actually guarantee this behaviour.

Unfortunately, there is a problem with the above specification. If `p` and `q` are pointers to the same location in memory (in other words, they are just different names for the same list), then we will end up with a circular structure: we will iterate along to the end of the list pointed to by `p` and then adjust the `next` pointer to point back to the head of the list: see Figure 2.

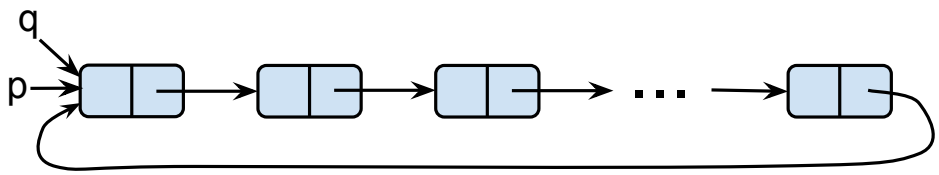


Fig. 2. `concat` result in case of two pointers (`p` and `q`) pointing to the same list segment.

This violates the intended meaning of our `lseg` predicate, and hence the post-condition is false. One might attempt to deal with this by modifying the method to check whether `p==q`, but that would still not work since if the lists pointed to by `p` and `q` share any cells we will still end up with heap structures containing loops (Figure 3). Modifying the method to detect such situations would make it unnecessarily complicated; a better strategy is to amend the assertions to exclude problematic inputs from the outset. The key to this is

[4] For clarity, we have used an idealised annotation syntax here. Restrictions on the syntax of Java annotations mean that the specifications used in practice are somewhat more complex; these will be described in detail in Section 3.

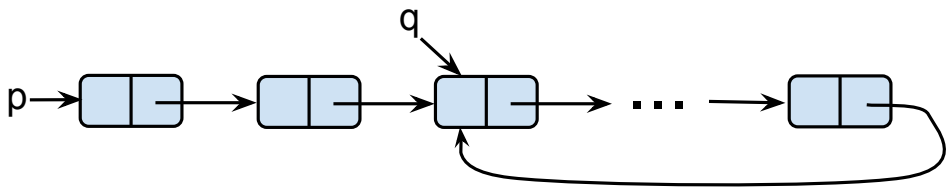


Fig. 3. `concat` result in case of a pointer `q` pointing to an internal element of a list segment pointed by `p`.

to base the assertions on *Separation Logic* [22], a logic which is designed for arguing about non-overlapping structures and has proved very useful in the analysis of heap-allocated data structures in recent years.

Separation Logic has a number of novel logical connectives for arguing about non-overlapping objects. For example, in addition to the usual logical conjunction $\wedge$ ($A \wedge B$ means that A is true and B is true), Separation Logic has the *separating conjunction* $*$: $A * B$ is true if A is true and B is *separately* true. In the context of heap-allocated structures $A * B$ will be interpreted as meaning that A and B are both true, but on *disjoint regions of the heap*. See [22] for full details of Separation Logic.

If we use the separating conjunction to rewrite our original specification as

```
@Requires (lseg(@arg p,null) * lseg(@arg q,null))
@Ensures (lseg (@ret, null))
```

then it becomes valid: if the user supplies two well-formed lists which share no memory cells then the return value is also a well-formed list.

Amortised analysis with Separation Logic

The techniques of the previous section allow us to specify functional properties of methods which manipulate heap-allocated structures. However, our main interest is in the resource consumption of methods, where "resource" refers to some quantity which is consumed by the method: for example, we might consider the number of heap objects allocated by a method, the number of bytecode instructions executed, or the number of network packets sent. In this paper, we will consider the problem of finding the number of times a special method called `consume` is executed, but this can easily be replaced by other methods or bytecode instructions in order to deal with other resources.

In [4], it is shown that assertions in Separation Logic can be neatly extended to include information about resource consumption, and that it is possible both to verify annotations and to infer resource usage for methods where iteration is driven by the processing of heap allocated data structures.

This is based on the idea of *amortised analysis* [25] of algorithms involving data structures. The approach we will take here is to imagine that each node of a data structure is equipped with a number of tokens, and that one of these

is consumed each time the node is processed. Consider our previous example, with some calls to `consume` added:

```
IntList concat (IntList p, IntList q) {
   consume();
   consume();

   if (p == null) return q;
   else {
      IntList t = p;
      while (t.next != null) {
         t = t.next;
         consume();
      }
      t.next = q;
      return p;
   }
```

We see that `consume` is called twice at the start of the method and then once for each node in the list `p`. We can express this by extending our Separation Logic specifications to include costs:

```
@Requires (lseg(1, @arg p,null) * lseg(0, @arg q,null), 2)
@Ensures (lseg (0, @ret, null), 0)
```

The extra numeric annotations are interpreted as saying that if we enter the method with one token for each node of `p` plus two extra tokens then the method can successfully execute and we are left with no tokens at the end; since each call to `consume` requires one token, we see that `consume` is called at most `length(p)+2` times.

We do not require that the annotations specify the minimal number of tokens required, only a number which is sufficient to allow the method to complete. For example

```
@Requires (lseg(7, @arg p,null) * lseg(2, @arg q,null), 5)
```

would also be a valid precondition: if we have the specified number of tokens then the method is still able to complete, but this time we will have some tokens left over which would enable further processing of the result at a later stage. The left-over tokens can be included in the postcondition. For example

```
@Requires (lseg(7, @arg p,null) * lseg(2, @arg q,null), 5)
@Ensures (lseg(2, @ret, null), 3)
```

is also a valid (albeit non-optimal) specification.

Atkey shows that it is possible to use a linear programming technique based on ideas of Hofmann and Jost [15] to verify that resource annotations such as those above are valid; in fact, he shows that it is actually possible to *infer* minimal resource annotations and thus the resource consumption of a method. We will see examples of this later.

3 Amortised Analysis for Java bytecode

We have developed an analyser which implements the ideas of the previous section. Source programs are equipped with Java annotations giving preconditions and postconditions of the type described above; the current version also requires annotations giving loop invariants.

Our analyser works on compiled class files. The Java compiler stores source annotations in the class file, and the analyser retrieves these and uses them to perform the analysis on JVM bytecode.

The technique of analysing the bytecode rather than the source code has several advantages:

- The bytecode is what is actually executed. We do not need any knowledge of the inner workings of a particular compiler, and indeed the analyser is independent of the compiler used.
- This approach extends the JVM verification paradigm. Many Java applications are supplied in the form of compiled classfiles with no source code, and a bytecode analyser can be used to check the behaviour of the class prior to execution. There is no requirement for the user of the code to trust the supplier.
- The analysis is not restricted to bytecode obtained from Java source; in principle our analysis could be made work on bytecode obtained from other languages targeting the JVM (Scala [5], for example), or even on handwritten bytecode.

Java annotations for amortised analysis

We use Java annotations to equip methods with resource-usage specifications. In this section we will expand on the informal description given in earlier sections.

Java annotations are a specific form of metadata that can be added to Java source code. They can be associated with Java classes, fields, methods and method parameters, and are embedded in classfiles when compiling Java code. They are defined using class-like structures in files named after the annotation.

In order to be able to provide amortised analysis on methods we have defined three annotations. These allow user to specify method preconditions and postconditions together with loop invariants which are required by the analyser.

- `@Requires(assertions)` for preconditions

[5] http://www.scala-lang.org/

```
assertions = [exvars] assertion ('||' assertion)* ;
assertion = '{' data-assertions '|'
                heap-assertions '|'
                resource-assertion '}' ;

exvars = 'exists' typedvar + '.' ;
typedvar = id ':' type ;
     (* id is a Java identifier: [A-Za-z_][A-Za-z0-9_]* *)
type = 'int' | 'long' | 'float' | 'double' | 'ref' ;

data-assertions = data-assertion (',' data-assertion)* ;
data-assertion  =
     term '==' term
   | term '!=' term ;

heap-assertions = heap-assertion (',' heap-assertion)* ;
heap-assertion =
     '[' field '->' location ']'
   | 'lseg (' resource-assertion ',' term ',' term ')'
   | 'tree (' resource-assertion ',' term ')' ;


term =
     id             (* existential variable *)
   | '@arg' id (* only @arg variables allowed in preconditions *)
   | '@var' id
   | 'null'
   | '@ret' ;  (* @ret only allowed in postcondition *)

field = '(' term ')' '.' id ':' type ;

location = var | '?' ;

resource-assertion = linear-expression ;
```

Fig. 4. Assertion EBNF

- `@Ensures(assertions)` for postconditions
- `@Invariant(assertions)` for loop invariants

All of them are given in the form of a Java `String` containing assertions. A shortened version of the assertion syntax in EBNF is shown in Fig. 4.

Data assertions consist of comma-separated lists of assertions of the form `term == term` or `term != term` and are used to provide the analyser with information about when references point to the same or different Java objects. These are mostly required in loop invariants. It would be possible to use dataflow analysis to deduce this information, but we have not done this yet.

There are two types of *heap assertion*, which are Separation Logic predicates enriched with indications of field content. The first type of heap assertion indicates that part of the heap forms either a list segment or a tree (e.g. `lseg(r1, @arg x, null)`, `tree(r1, @arg x)`); both of these expand to more complex assertions built from Separation Logic primitives. The second type of heap assertion indicates to the analyser that a particular field points to a particular heap location or to some undetermined location; these are mostly required in preconditions and postconditions, where they facilitate

interprocedural analysis by exposing information about heap structure for use in reasoning with Separation Logic.

Finally, *resource assertions* are linear expressions such as `3*r1 + 5*r2 + r3 + 7` which specify constants and variables which we want the analyser to use to infer the resources associated with the method before and after execution (ie, in the precondition and postcondition); they are also used in list and tree predicates to indicate resources associated with each node of the structure.

We have also introduced a dummy method called `consume` which does nothing except tell the static analyser that at the point where it is introduced, a unit of resource is being used. We can imagine such a dummy method being hidden inside library code in the future, stating the amount of resource used by each method defined in each class, so that programmers will not need to add it explicitly but rather implicitly use it by invoking library methods. In the present implementation though, libraries have not been modified and developers have to specify resource consumption explicitly in their code.

Invariant localisation in Java code

Loop invariants have to be given for each loop for the amortised analysis to be effective but the Java language does not allow the inclusion of annotations inside the code. This is problematic if the method being analysed contains two or more loops, since we need a way to decide which invariant refers to which loop. We could simply give invariants in the same order as the loops appear in the code, but it is possible that a compiler might produce bytecode in which the order in which the loops appear in the bytecode does not correspond to the order in the source code. Our way to obviate this limitation was to identify each loop with an integer identifier. Each loop invariant is given a identifier (`@Invariant(id, assertion)`) and the same identifier has to match the argument of a dummy method `Loop.invariant(id)` placed just before the loop in the code. Thus by searching the code, the analyser can associate the declared invariants with the corresponding loop.

4 Examples and output interpretation

We illustrate the operation of the analyser by returning to our earlier list concatenation example.

```
$ cat examples/IntList.java

import uk.ac.ed.inf.resa.*;

public class IntList {

  private int data;
  private IntList next;
```

```
  ...

  @Requires("{ | lseg(1, @arg p, null) * lseg (0, @arg q, null)| 2 }")
  @Ensures("{ | lseg(0, @ret, null) | 0 }")
  @Invariant("{ @var t != null | lseg (0, @var p, @var t) *"
              + "lseg (1, @var t, null) * lseg (0, @var q, null) | 0}")
  public static IntList concat (IntList p, IntList q) {
    Amortised.consume();
    Amortised.consume();
    if (p==null) return q;

    IntList t = p;
    Loop.invariant (0);
    while (t.next != null) {
      t = t.next;
      Amortised.consume();
    }
    t.next = q;

    return p;
  }
  ...
}
```

We have supplied a loop invariant which describes the state of resource usage as the program progresses. Whenever we reach the head of the `while` loop, we have used up the resources associated with the part of the first input list `p` which has already been processed (between `p` and `t`), we still have one unit of resource available in the remaining part of `p` (between `t` and `null`), and we do not require any resources to process `q`. Note the `@arg` and `@var` annotations attached to variable names. These are used to distinguish between the current value of a variable (`@var`) and the value of a method argument at entry to the method (`@arg`); they are not strictly necessary in this example, but are required in more complex examples where variables representing method arguments are modified. The `@ret` annotation refers to the method return value.

If we compile `IntList.java` and invoke the analyser then the output is as follows.

```
$ resa -amortised examples/IntList concat
...
Solved VCs
Verification successful
```

This shows that the analyser has succeeded in proving that the precondition and postcondition are satisfied. However, if we amend the precondition to say

```
@Requires("{ | lseg(1, @arg p, null) * lseg (0, @arg q, null)| 1 }")
```

then the verification fails because there is only one "constant" unit of resource available, and two are required by the calls to `consume` at the start of the method:

```
LP is infeasible
```

More interestingly, we can supply *generic* annotations and the analyser will infer suitable values for them.

```
@Requires("{ | lseg(x1, @arg p, null) * lseg (x2, @arg q, null)| x3 }")
@Ensures("{ | lseg(y1, @ret, null) | y2 }")
@Invariant("{ @var t != null | lseg (z1, @var p, @var t) *"
            + "lseg (z2, @var t, null) * lseg (z3, @var q, null) | z4}")
```

With these annotations the analyser outputs

```
Optimal solution for resource variables:
x1 = 1, x2 = 0, x3 = 2
y1 = 0, y2 = 0
z1 = 0, z2 = 1, z3 = 0, z4 = 0
```

We can also specify the amount of resource which we require when the method returns: if we replace the postcondition with `@Ensures("{ | lseg(3, @ret, null) | 7 }")` then we get `x1 = 4, x2 = 3, x3 = 9`, so that if the postcondition is to be satisfied then we must have 4 units of resource for each element of `p`, 3 for each element of `q`, and 9 extra units before the method is called.

In addition to this example we have been able to successfully analyse a number of other standard operations on lists, such as reversal, iteration, and deleting and inserting elements, together with similar operations for trees.

5 Implementation details

Our analyser is implemented in OCaml, and Java class files are represented by a collection of datatypes. For program analysis, the most important part of this is the representation of method bytecode. Our design is intended to be fairly general-purpose since we intend to support multiple analysis techniques, including the amortised analysis described earlier.

Java classfiles are initially converted into a low-level OCaml representation which is a fairly faithful representation of structure of the class as described in the JVM specification [19]. This is then converted into a higher-level representation which is more suited to analysis. We will give an outline of this representation here, but space limitations preclude a detailed description.

Bytecode instructions are decompiled into a form where the JVM stack has been eliminated. We keep track of which constants and local variables have been loaded onto the stack, and these are represented by a datatype called `value`, which has constructors for constants of type `int`, `long`, `float`, `double`, `String` and `class`, together with variables. Variables are represented by integer identifiers which are tagged with the type of the corresponding value: this is one of `I`, `L`, `F`, `D`, or `A` (representing integers, long integers, floats, doubles, and addresses). We do not have special types for `boolean`, `byte`, `char` and `short` since the JVM treats all of these as 32-bit integers for most purposes. We also tag all references with the single type `A`, and make no attempt to keep track of the most specific class or interface to which the variable belongs; it would be possible to infer this information, but so far we have not required this.

There are two kinds of variables: *local* variables and *stack* variables. The former represent values loaded onto the stack from JVM local variables by means of instructions such as `iload`, and the latter represent intermediate values which have been created on the stack by arithmetic operations, method calls and so on. Each variable is marked with an integer which for local variables represents the number of the associated JVM local variable, and for stack variables is simply a counter which is incremented every time a new value is created on the stack and decremented when that value is consumed. Some care is required here since stack operations can duplicate values on the stack. The decompilation process handles this by creating new copies of variables using a special pseudo-instruction called `Copy`. Another complication is that one can load a local variable r onto the stack and then modify the contents of r before the earlier value (still on the stack) has been consumed; again, the `Copy` operation is used to avoid confusion by creating a new variable which represents the earlier value

Instructions which act on the stack are represented by a datatype of operations which take values as arguments. This has 19 constructors which suffice to represent all of the JVM operations (`putfield`, `getfield`, `invokevirtual` and so on) except for those which involve control-flow transfer. Basic blocks are represented by a list of pairs consisting of operations together with the local variable or stack location where the result (if any) of the operation is to be stored. At the end of a basic block we have a member of a `continuation` datatype: this represents various types of jump, comparison, switch, and return operation, together with information specifying which blocks control can flow to after leaving the current block. We do not provide any representation for the `jsr` and `ret` instructions used by exception handlers, since these complicate analysis and are supposedly deprecated in current Java releases (although we have encountered them in a few of the standard API classes in `rt.jar`). If one of these instructions is encountered during decompilation then an exception is thrown and the class is rejected.

The bytecode for an entire method is represented by an array of blocks, stored in preorder. This can be regarded as a graph, and we have a module which computes useful information such as successors, predecessors, and dominators, and can also provide other views of the graph, such as postorder and reverse postorder, which can be useful in some analyses.

Proof search

The most important part of the amortised analysis is the proof search procedure which is used to verify that the precondition of a method implies its postcondition, and also to check or infer the associated resource annotations.

This uses the method described by Atkey in [4], and indeed our implemen-

tation uses an adapted version of code from a prototype implementation by Atkey, which analysed a textual form of a subset of JVM bytecode. The basic idea is to visit the code in postorder, working backwards from the postcondition to infer weakest preconditions for each instruction, and then to prove that the weakest precondition for the first instruction is implied by the (user-supplied) precondition for the method.

Visiting the nodes of the CFG in postorder ensures that when we visit a given node n, the preconditions for all of its successors are already known, and can thus be combined (by logical conjunction) to obtain a postcondition for n. However, this strategy fails when we encounter a back-edge $s \to t$ (ie, when t is at the head of a loop). In this case we will not have visited t when we arrive at s, and it is for this reason that we require annotations for loop invariants: the annotation will provide a precondition for t, and hence will be available to contribute to the postcondition for t.

The proof search procedure also collects linear constraints describing the resource usage of the bytecode instructions, as described in [4]; once the analysis has been completed and the basic Separation Logic annotations have been checked to make sure that the precondition implies the postcondition, the resource constraints are converted into a linear programming problem which is then solved using the Parma Polyhedra Library [5].

6 Conclusions and Further Research

We have shown that it is possible to apply Atkey's amortised analysis technique to realistic JVM bytecode. However, there are a number of issues that would merit further research.

- The annotations are somewhat complex, and it would be desirable to simplify them. A first step would be to dispense with the data equality and non-equality annotations, and we believe that it would be possible to do this using dataflow techniques. A more difficult problem would be to remove the loop invariants, which are generally the most difficult part of the specification. Techniques for automatically inferring loop invariants have been studied, and one possible avenue of attack for the kind of loop invariants used here would be the methods described in [10].
- A minor inconvenience is that assertion syntax is only checked during analysis, and not at compilation time. The Sun `javac` compiler supports plugins for annotation processing [6], and it would be very helpful to use this feature to detect errors in assertion syntax during compilation.
- Amortisation techniques are useful for analysing the resource consumption of loops which process heap-allocated structures, but may be less useful for

[6] See JSR 269: Pluggable Annotation Processing API.

loops which are controlled by numeric quantities. Our analyser can attack such loops by using combinatorial techniques for the enumeration of lattice points in polyhedra (see [6], and [9] for an application of related techniques to resource analysis for Java programs); a preliminary description of this appears in [3], and we will give more details in a subsequent paper. An interesting problem would be to automatically combine the two techniques, so that the analyser can deal with complex methods with minimal intervention from the user.

- The amortised analysis can only handle bounds on resource usage which are linear in the size of the data structures involved. The experience of other authors [8] suggests that this is sufficient in many (but by no means all) practical situations. The linear-programming-based inference technique we use here is intrinsically tied to linear bounds; nevertheless, recent work [14,13] has shown how it can be extended to deduce non-linear bounds in certain specialised cases. There are other inference techniques which can deal with non-linear bounds, and we will discuss some of these later.
- The analyser currently only supports linked lists and trees, and the proof-search implementation depends quite strongly on this. It would be desirable to find more generic annotation and analysis techniques, for example for dealing with data structures in the standard Java API which are accessed via interfaces. One method here might be to attach annotations to existing library classes in order to enable the analysis of client code.

Related work: resource inference

There has been a considerable amount of work on resource inference in recent years, and we now attempt to describe some of this.

The linear-programming-based inference technique used here originated with Hofmann and Jost in [15], and has subsequently been applied to an OCaml-style language [23] and the Hume language for embedded systems [8]. Jost's thesis [17] contains a useful overview of this area of research, and recasts much of the existing work in a consistent and systematic framework. Some more recent developments appear in [14,13].

A number of other resource-inference techniques have been proposed and implemented, for Java and also for other languages. The COSTA system of Albert et al. [1,2] converts JVM bytecode into a collection of "cost equations" which are then solved to obtain a symbolic (and possibly non-linear) bound for resource usage. In contrast with our method, COSTA is fully automatic and does not require the user to supply annotations.

The SPEED project of Gulwani et al. [11,12] uses a number of techniques (including abstract interpretation and SMT solving) to obtain (non-linear and symbolic) bounds for the number of times a program location is visited, and

has been applied to the complexity analysis of C# programs.

Type-theoretic methods for resource usage inference are described in [21] (inference of symbolic bounds for recursive functional programs) and [20] (heap usage for object-oriented programs).

Related work: static analysis of bytecode

A number of representations for JVM bytecode have appeared in the literature. Many of these ([18], [7], and [2] for example) utilise the technique of introducing new variables to represent values on the JVM operand stack. The Soot framework (see [26] for example) contains a number of intermediate representations for JVM bytecode and has been applied to many problems in optimisation and analysis; a large list of related publications can be found at `http://www.sable.mcgill.ca/`. Another framework for JVM bytecode analysis is Julia, which is described in [24]. Finally, another OCaml representation for JVM classfiles is described in [16]; this has much in common with our representation.

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, March 2007.

[2] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. In *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2008.

[3] David Aspinall, Robert Atkey, Kenneth MacKenzie, and Donald Sannella. Symbolic and analytic techniques for resource analysis of Java bytecode. In *Proceedings of the 5th international conference on Trustworthly Global Computing*, TGC'10, pages 1–22, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] Robert Atkey. Amortised resource analysis with separation logic. In *ESOP*, pages 85–103, 2010.

[5] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[6] Alexander Barvinok and James E. Pommersheim. An algorithmic theory of lattice points in polyhedra. In *New perspectives in algebraic combinatorics (Berkeley, CA, 1996–97)*, volume 38 of *Math. Sci. Res. Inst. Publ.*, pages 91–147. Cambridge Univ. Press, Cambridge, 1999.

[7] Lennert Beringer, Kenneth MacKenzie, and Ian Stark. Grail: a functional form for imperative mobile code. In *Foundations of Global Computing: Proceedings of the 2nd EATCS Workshop*, number 85.1 in Electronic Notes in Theoretical Computer Science. Elsevier, June 2003.

[8] Armelle Bonenfant, Christian Ferdinand, Kevin Hammond, and Reinhold Heckmann. Worst-case execution times for a purely functional language. In *Proc. Implementation of Functional Languages (IFL 2006)*, volume 4449 of *Lecture Notes in Computer Science*, 2007. To appear.

[9] Victor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, Jun 2006.

[10] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

[11] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[12] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 292–304, New York, NY, USA, 2010. ACM.

[13] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. In *38th Symp. on Principles of Prog. Langs. (POPL'11)*, 2011. To appear.

[14] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential - A Static Inference of Polynomial Bounds for Functional Programs. In *In Proceedings of the 19th European Symposium on Programming (ESOP'10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.

[15] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, pages 185–197, 2003.

[16] Laurent Hubert, Nicolas Barré, Frédéric Besson, Delphine Demange, Thomas P. Jensen, Vincent Monfort, David Pichardie, and Tiphaine Turpin. Sawja: Static analysis workshop for Java. *CoRR*, abs/1007.3353, 2010.

[17] Steffen Jost. *Automated Amortised Analysis*. PhD thesis, Ludwig-Maximilians-Universität, München, August 2010.

[18] Christopher League, Valery Trifonov, and Zhong Shao. Functional Java bytecode. In *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*, July 2001. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

[19] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[20] Wei ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin Rinard. Memory usage verification for oo programs. In *In SAS 05*, pages 70–86. Springer, 2005.

[21] Álvaro J. Rebón Portillo, Kevin Hammond, Hans-Wolfgang Loidl, and Pedro Vasconcelos. Cost analysis using automatic size and time inference. In *Proceedings of the 14th international conference on Implementation of functional languages*, IFL'02, pages 232–247, Berlin, Heidelberg, 2003. Springer-Verlag.

[22] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[23] Donald Sannella, Martin Hofmann, David Aspinall, Stephen Gilmore, Ian Stark, Lennart Beringer, Hans-Wolfgang Loidl, Kenneth MacKenzie, Alberto Momigliano, and Olha Shkaravska. Mobile Resource Guarantees (project evaluation paper). In *Trends in Functional Programming*, pages 211–226, 2005.

[24] Fausto Spoto. JULIA: A generic static analyser for the Java bytecode. Proceedings of FTfjP'2005, Glasgow, 2005.

[25] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[26] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

LCT: An Open Source Concolic Testing Tool for Java Programs[1]

Kari Kähkönen, Tuomas Launiainen, Olli Saarikivi,
Janne Kauttio, Keijo Heljanko and Ilkka Niemelä

Department of Information and Computer Science
School of Science
Aalto University
PO Box 15400, FI-00076 AALTO, Finland
{Kari.Kahkonen,Tuomas.Launiainen,Janne.Kauttio,Keijo.Heljanko,
Ilkka.Niemela}@tkk.fi, Olli.Saarikivi@iki.fi

Abstract

LCT (LIME Concolic Tester) is an open source concolic testing tool for sequential Java programs. In concolic testing the behavior of the tested program is explored by executing it both concretely and symbolically at the same time. LCT instruments the bytecode of the program under test to enable symbolic execution and collects constraints on input values that can be used to guide the testing to previously unexplored execution paths. The tool also supports distributing the test generation and execution to multiple processes that can be run on a single workstation or even on a network of workstations. This paper describes the architecture behind LCT and demonstrates through benchmarks how the distributed nature of the tool makes testing more scalable.

Keywords: Symbolic execution, concolic testing, constraint solving, bytecode instrumentation.

1 Introduction

Automated testing has the potential to improve reliability and reduce costs when compared to manually written test cases. One such technique that has recently received interest is concolic testing which combines concrete and symbolic execution. Based on this technique, we have developed an open

[1] Work financially supported by Tekes - Finnish Agency for Technology and Innovation, Conformiq Software, Elektrobit, Nokia, Space Systems Finland, and Academy of Finland (projects 126860, 128050 and 139402).

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

source tool called LCT (LIME Concolic Tester) that can automatically explore different execution paths and generate test cases for sequential Java programs.

In this paper we give an overview of LCT and also demonstrate the distributed nature of the tool with benchmarks where several Java programs are tested in a way that multiple instances of the program under test are concurrently taking part in the testing process.

The main improvements in LCT over existing Java concolic testing systems such as jCUTE [10] are the following: (i) the use of bitvector SMT solver Boolector [1] makes the symbolic execution of Java more precise as integers are not considered unbounded, (ii) the twin class hierarchy [4] instrumentation approach of LCT allows the Java core classes to be instrumented, (iii) the tool architecture supports distributed testing; and (iv) the tool is freely available as open source. Distributed constraint solving has been previously employed by the Microsoft fuzzing tool SAGE [6] that uses a distributed constraint solver Disolver while LCT uses a non-distributed constraint solver but can work on several branches of the symbolic execution tree in parallel.

The rest of the paper is structured as follows. Section 2 briefly describes the algorithm behind concolic testing, Section 3 gives an overview of LCT and Sect. 4 discusses the experiments that have been done with the tool.

2 Concolic Testing

Concolic testing [5,11,2] is a method where a given program is executed both concretely and symbolically at the same time in order to explore the different behaviors of the program. The main idea behind this approach is to, at runtime, collect symbolic constraints on inputs to the system that specify the possible input values that force the program to follow a specific execution path. Symbolic execution of programs is typically made possible by instrumenting the system under test with additional code that collects the constraints without disrupting the concrete execution. For a different approach that uses a non-standard interpreter of bytecode instead of instrumentation, see Symbolic Java PathFinder [9].

Concolic testing starts by first executing a program under test with any concrete input values. The execution of the program can branch into different execution paths at branching statements that depend on input values. When executing such statements, concolic testing constructs a symbolic constraint that describes the possible input values causing the program to take the true or false branch at the statement in question. A *path constraint* is a conjunction of these symbolic constraints and new test inputs for the subsequent test runs are generated by solving them. Typically this is done by using SMT (Satisfiability-Modulo-Theories) solvers with integer arithmetic or bit-vectors as the underlying theory. By continuing this process, concolic testing can

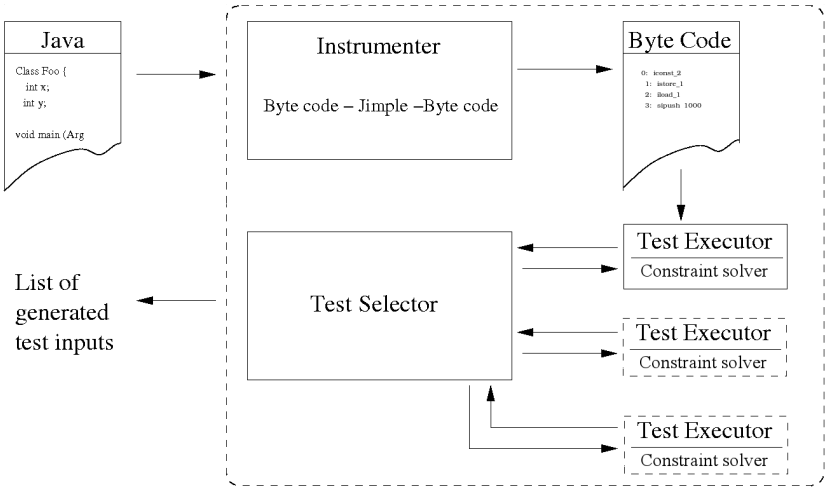


Fig. 1. The architecture of LCT

attempt to explore all distinct execution paths of the given program. These execution paths can be expressed as a *symbolic execution tree* which is a structure where each path from root to a leaf node represents an execution path and each leaf node has a path constraint describing the input values that force the program to follow that specific path.

The concrete execution in concolic testing provides the benefit that it makes available accurate information about the program state which might not be easily accessible when using only static analysis and allows the program to contain calls to uninstrumented native code libraries as symbolic values can always be approximated with concrete ones. Furthermore, as each test is run concretely, concolic testing reports only real bugs.

3 Tool Details

The architecture of LCT is shown in Figure 1 and it can be seen as consisting of three main parts: the instrumenter, the test selector and the test executors. To test a given program, the input locations are first marked in the code using methods provided by LCT. For example, `int x = LCT.getInteger()` generates an int type input value for `x`. After the input variables have been marked in the program, it is given to the instrumenter that enables symbolic execution by instrumenting the program using a tool called Soot [12]. The resulting program is called test executor. The test selector constructs a symbolic execution tree based on the constraints collected by the test executors and selects which path in the symbolic execution tree is explored next. The communication between the test selector and test executors is implemented using TCP sockets. This way it is easy to distribute the testing process. LCT reports uncaught exceptions as defects and the executed tests can also be

archived as a test suite for offline testing.

LCT provides the option to use Yices [3] or Boolector [1] as a constraint solver. LCT uses linear integer arithmetic to encode the constraints when Yices is used and bit-vectors are used with Boolector. LCT has support for all primitive data types in Java as symbolic inputs with the exception of float and double data types as there is no native support for floating point variables in the used constraint solvers. LCT can also generate input objects that have their fields initialized as new input values in a similar fashion to [10].

3.1 Performing Symbolic Execution

To form symbolic constraints, it is necessary to know for each variable its symbolic value in addition to the concrete value. To track the symbolic values, LCT maintains a mapping from variables to their symbolic values. To update these memory maps and to construct path constraints, LCT follows closely the approach described in [5,10]. Every assignment statement in the program is instrumented with symbolic statements to update the symbolic memory maps. Every branching statement must also be instrumented with statements that create the symbolic constraints for both resulting branches so that the necessary path constraints can be constructed. A more detailed description of the instrumentation process can be found in [7].

The test selector maintains a symbolic execution tree based on the symbolic constraints generated by the test executors. LCT supports multiple strategies such as depth-first, breadth-first and randomized searches to explore the tree. For each test run, the test selector chooses an unexplored path from the symbolic execution tree. To explore this path, the test server sends the path constraint to a text executor that solves the constraint and uses the resulting values for a new test run. This way constraint solving is not done in a single centralized place which could cause a performance bottleneck.

3.2 Limitations

LCT has been designed for sequential Java programs. Multi-threading support is currently under development and will be released soon in version 2.0. There are also some cases where LCT can not obtain full path coverage for supported Java programs. LCT is not able to do any symbolic reasoning if the control flow of the program goes to a library that has not been instrumented (e.g., calling a library that has been implemented in a different programming language). Instrumenting Java core classes can also be problematic, therefore we have implemented custom versions of some of the most often required core classes to alleviate this problem. The program under test is then automatically modified to use the custom versions of these classes instead of their original counterparts. This approach can be seen as an instance of the twin

Benchmark	Paths	Runtimes / Speedups		
		1 test executor	10 test executors	20 test executors
AVL tree	3840	16m 57s	2m 6s / 8.1	1m 8s / 15.0
Quicksort (5 elements)	541	3m 11s	21s / 5.2	13s / 8.4
Quicksort (6 elements)	4683	28m 22s	3m 29s / 8.1	1m 39s / 17.2
Greatest common divisor	2070	11m 12s	1m 13s / 9.2	38s / 17.7

Table 1
Results of the experimental evaluation of the distributed architecture of LCT.

class hierarchy approach presented in [4].

LCT also does a similar non-aliasing assumption as the jCUTE tool. The code "`a[i] = 0; a[j] = 1; if (a[i] != 0) ERROR;`" is an example of this. LCT assumes that the two writes do not alias and, thus, does not generate constraint $i = j$ required to reach the `ERROR` label.

4 Experiments

We have evaluated LCT (version 1.1.0) and its distributed architecture by testing multiple Java programs so that varying number of test executors were running concurrently. The tests included: AVL tree (http://cs-people.bu.edu/mullally/cs112/code/GraphicalAvlTree.java), Quick sort (http://users.cis.fiu.edu/~weiss/dsaajava/code/DataStructures/) and Greatest common divisor (GCD) (http://commons.apache.org/). For AVL tree a test driver was used that called either add or remove methods nondeterministically with an integer marked as the input to the methods for five times. For Quick sort the test driver used the algorithm to sort a fixed size array of input integers and checked afterwards that the array had been sorted correctly. Finally, for GCD the test driver called the algorithm with integer inputs that were limited to be between 0 and 50.

The results of our experimental evaluation are shown in Table 4 that shows the number of execution paths explored, total run-times and speedups compared to running only one test executor. The computers used in the experiments had Intel Core 2 Duo processors running at 1.8 GHz together with 2 GB of RAM. For single test executor cases, the test selector and executor were run on the same machine. In our test environment, it made negligible difference whether this single test executor was run on the same or different machine as the test selector. For cases with multiple test executors, each computer was used to run two test executors at the same time while the test server was run on a separate computer (i.e., the test selector was able to utilize two cores). As the results show, LCT is able to take advantage of the increased resources efficiently with speedups ranging from 8.4 to 17.7 in the 20 test executor case. This is because individual test runs are highly independent. We have also

used LCT to compare it with random testing in the context of Java Card applets. In this experiment LCT was used on a large number of mutants of the program under test. Further details of this experiment can be found in [8].

5 Conclusions

This paper introduces the LCT tool that is available together with source code and user guide from: http://www.tcs.hut.fi/Software/lime/ as part of the LIME Interface Test Bench. We have demonstrated how the distributed nature of the tool makes concolic testing more scalable. We are currently adding support for the C language and multi-threaded Java programs to LCT.

References

[1] Brummayer, R. and A. Biere, *Boolector: An efficient SMT solver for bit-vectors and arrays*, in: *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, Lecture Notes in Computer Science **5505** (2009), pp. 174–177.

[2] Cadar, C., V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler, *EXE: automatically generating inputs of death*, in: *Proceedings of the 13th ACM conference on Computer and communications security (CCS 2006)* (2006), pp. 322–335.

[3] Dutertre, B. and L. de Moura, *A Fast Linear-Arithmetic Solver for DPLL(T)*, in: *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, Lecture Notes in Computer Science **4144** (2006), pp. 81–94.

[4] Factor, M., A. Schuster and K. Shagin, *Instrumentation of standard libraries in object-oriented languages: The twin class hierarchy approach*, in: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)* (2004), pp. 288–300.

[5] Godefroid, P., N. Klarlund and K. Sen, *DART: Directed automated random testing*, in: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)* (2005), pp. 213–223.

[6] Godefroid, P., M. Y. Levin and D. A. Molnar, *Automated whitebox fuzz testing*, in: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008* (2008), pp. 151–166.

[7] Kähkönen, K., *Automated test generation for software components*, Technical Report TKK-ICS-R26, Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland (2009).

[8] Kähkönen, K., R. Kindermann, K. Heljanko and I. Niemelä, *Experimental comparison of concolic and random testing for Java Card applets*, in: J. van de Pol and M. W. 0002, editors, *SPIN*, Lecture Notes in Computer Science **6349** (2010), pp. 22–39.

[9] Pasareanu, C. S., P. C. Mehlitz, D. H. Bushnell, K. Gundy-burlet, M. Lowry, S. Person and M. Pape, *Combining unit-level symbolic execution and system-level concrete execution for testing NASA software*, in: *ISSTA*, 2008, pp. 179–180.

[10] Sen, K., "Scalable automated methods for dynamic program analysis," Doctoral thesis, University of Illinois (2006).

[11] Sen, K. and G. Agha, *CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools*, in: *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, Lecture Notes in Computer Science **4144** (2006), pp. 419–423, (Tool Paper).

[12] Vallée-Rai, R., P. Co, E. Gagnon, L. J. Hendren, P. Lam and V. Sundaresan, *Soot - a Java bytecode optimization framework*, in: *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999)* (1999), p. 13.