Loop Recognition in C++/Java/Go/Scala

Robert Hundt
Google
1600 Amphitheatre Parkway
Mountain View, CA, 94043
rhundt@google.com

***Abstract*—In this experience report we encode a well specified, compact benchmark in four programming languages, namely C++, Java, Go, and Scala. The implementations each use the languages' idiomatic container classes, looping constructs, and memory/object allocation schemes. It does not attempt to exploit specific language and run-time features to achieve maximum performance. This approach allows an almost fair comparison of language features, code complexity, compilers and compile time, binary sizes, run-times, and memory footprint.**

While the benchmark itself is simple and compact, it employs many language features, in particular, higher-level data structures (lists, maps, lists and arrays of sets and lists), a few algorithms (union/find, dfs / deep recursion, and loop recognition based on Tarjan), iterations over collection types, some object oriented features, and interesting memory allocation patterns. We do not explore any aspects of multi-threading, or higher level type mechanisms, which vary greatly between the languages.

The benchmark points to very large differences in all examined dimensions of the language implementations. After publication of the benchmark internally at Google, several engineers produced highly optimized versions of the benchmark. We describe many of the performed optimizations, which were mostly targeting run-time performance and code complexity. While this effort is an anecdotal comparison only, the benchmark, and the subsequent tuning efforts, are indicative of typical performance pain points in the respective languages.

I. Introduction

Disagreements about the utility of programming languages are as old as programming itself. Today, these "language wars" become increasingly heated, and less meaningful, as more people are working with more languages on more platforms in settings of greater variety, e.g., from mobile to datacenters.

In this paper, we contribute to the discussion by implementing a well defined algorithm in four different languages, C++, Java, Go, and Scala. In all implementations, we use the default, idiomatic data structures in each language, as well as default type systems, memory allocation schemes, and default iteration constructs. All four implementations stay very close to the formal specification of the algorithm and do not attempt any form of language specific optimization or adaption.

The benchmark itself is simple and compact. Each implementation contains some scaffold code, needed to construct test cases allowing to benchmark the algorithm, and the implementation of the algorithm itself.

The algorithm employs many language features, in particular, higher-level data structures (lists, maps, lists and arrays of sets and lists), a few algorithms (union/find, dfs / deep recursion, and loop recognition based on Tarjan), iterations over collection types, some object oriented features, and interesting memory allocation patterns. We do not explore any aspects of multi-threading, or higher level type mechanisms, which vary greatly between the languages. We also do not perform heavy numerical computation, as this omission allows amplification of core characteristics of the language implementations, specifically, memory utilization patterns.

We believe that this approach highlights features and characteristics of the languages and allows an almost fair comparison along the dimensions of source code complexity, compilers and default libraries, compile time, binary sizes, run-times, and memory footprint. The differences along these dimensions are surprisingly large.

After publication of the benchmark internally at Google, several engineers produced highly optimized versions of the benchmark. We describe many of the performed optimizations, which were mostly targeting run-time performance and code complexity. While this evaluation is an anecdotal comparison only, the benchmark itself, as well as the subsequent tuning efforts, point to typical performance pain points in the respective languages.

The rest of this paper is organized as follows. We briefly introduce the four languages in section II. We introduce the algorithm and provide instructions on how to find, build, and run it, in section III. We highlight core language properties in section IV, as they are needed to understand the implementation and the performance properties. We describe the benchmark and methodology in section V, which also contains the performance evaluation. We discuss subsequent language specific tuning efforts in section VI, before we conclude.

II. The contenders

We describe the four languages by providing links to the the corresponding wikipedia entries and cite the respective first paragraphs from wikipedia. Readers familiar with the languages can skip to the next section.

C++ [7] is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language. It is regarded as a "middle-level" language, as it comprises a combination of both high-level and low-level language features. It was developed by Bjarne Stroustrup starting in 1979 at Bell Labs as an enhancement to the C language and originally named C with Classes. It was renamed C++ in 1983.

Java [9] is a programming language originally developed by James Gosling at Sun Microsystems (which is now a sub-

sidiary of Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to byte-code (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere". Java is currently one of the most popular programming languages in use, and is widely used from application software to web applications.

Go [8] is a compiled, garbage-collected, concurrent programming language developed by Google Inc. The initial design of Go was started in September 2007 by Robert Griesemer, Rob Pike, and Ken Thompson, building on previous work related to the Inferno operating system. Go was officially announced in November 2009, with implementations released for the Linux and Mac OS X platforms. At the time of its launch, Go was not considered to be ready for adoption in production environments. In May 2010, Rob Pike stated publicly that Go is being used "for real stuff" at Google.

Scala [10] is a multi-paradigm programming language designed to integrate features of object-oriented programming and functional programming. The name Scala stands for "scalable language", signifying that it is designed to grow with the demands of its users.

Core properties of these languages are:

- C++ and Go are statically compiled, both Java and Scala run on the JVM, which means code is compiled to Java Byte Code, which is interpreted and/or compiled dynamically.
- All languages but C++ are garbage collected, where Scala and Java share the same garbage collector.
- C++ has pointers, Java and Scala has no pointers, and Go makes limited use of pointers.
- C++ and Java require statements being terminated with a ';'. Both Scala and Go don't require that. Go's algorithm enforces certain line breaks, and with that a certain coding style. While Go's and Scala's algorithm for semicolon inference are different, both algorithms are intuitive and powerful.
- C++, Java, and Scala don't enforce specific coding styles. As a result, there are many of them, and many larger programs have many pieces written in different styles. The C++ code is written following Google's style guides, the Java and Scala code (are trying to) follow the official Java style guide. Go's programming style is strictly enforced – a Go program is only valid if it comes unmodified out of the automatic formatter `gofmt`.
- Go and Scala have powerful type inference, making explicit type declarations very rare. In C++ and Java everything needs to be declared explicitly.
- C++, Java, and Go, are object-oriented. Scala is object-oriented and functional, with fluid boundaries.

```
procedure analyze_loops(G_CF, START)
a:  number vertices of G_CF using depth-first search from START,
        numbering in preorder from 1 to |N_CF| and saving last[*]
b:  for w := 1 to |N_CF| do    // make lists of predecessors
        nonBackPreds[w] := backPreds[w] := ∅
        header[w]       := 1       // default "header" is START
        type[w]         := nonheader
        foreach edge (v, w) entering w do
            if isAncestor(w, v) then add v to backPreds[w]
                                else  add v to nonBackPreds[w]
    header[1] := nil      // START is root of header tree
c:  for w := |N_CF| to 1 step −1 do
        P := ∅
d:      foreach node v ∈ backPreds[w] do
            if (v ≠ w) then add FIND(v) to P
                       else type[w] := self
        worklist := P
        if (P ≠ ∅) then type[w] := reducible
        while (worklist ≠ ∅) do
            select a node x ∈ worklist and delete it from worklist
e:          foreach node y ∈ nonBackPreds[x] do
                y' := FIND(y)
                if not(isAncestor(w, y')) then
                    type[w] := irreducible
                    add y' to nonBackPreds[w]
                else if (y' ∉ P) and (y' ≠ w) then
                        add y' to P and to worklist
        foreach node x ∈ P do
            header[x] := w
            UNION(x, w)      // merge x's set into w's set
```

Fig. 1. Finding headers of reducible and irreducible loops. This presentation is a literal copy from the original paper [1]

III. The Algorithm

The benchmark is an implementation of the loop recognition algorithm described in "Nesting of reducible and irreducible loops", by Havlak, Rice University, 1997, [1]. The algorithm itself is an extension to the algorithm described R.E. Tarjan, 1974, Testing flow graph reducibility [6]. It further employs the Union/Find algorithm described in "Union/Find Algorithm", Tarjan, R.E., 1983, Data Structures and Network Algorithms [5].

The algorithm formulation in Figure 1, as well as the various implementations, follow the nomenclature using variable names from the algorithm in Havlak's paper, which in turn follows the Tarjan paper.

The full benchmark sources are available as open-source, hosted by Google code at `http://code.google.com/p/` in the project `multi-language-bench`. The source files contain the main algorithm implementation, as well as dummy classes to construct a control flow graph (CFG), a loop structure graph (LSG), and a driver program for benchmarking (e.g., `LoopTesterApp.cc`).

As discussed, after an internal version of this document was published at Google, several engineers created optimized versions of the benchmark. We feel that the optimization techniques applied for each language were interesting and indicative of the issues performance engineers are facing in their every day experience. The optimized versions are kept

src	README file and auxiliary scripts
src/cpp	C++ version
src/cpp_pro	Improved by Doug Rhode
src/scala	Scala version
src/scala_pro	Improved by Daniel Mahler
src/go	Go version
src/go_pro	Improved by Ian Taylor
src/java	Java version
src/java_pro	Improved by Jeremy Manson
src/python	Python version (not discussed here)

Fig. 2. Benchmark source organization (at the time of this writing, the cpp_pro version could not yet be open-sourced)

non_back_preds	an array of sets of int's
back_preds	an array of lists of int's
header	an array of int's
type	an array of char's
last	an array of int's
nodes	an array of union/find nodes
number	a map from basic blocks to int's

Fig. 3. Key benchmark data structures, modeled after the original paper

in the *Pro* versions of the benchmarks. At time of this writing, the `cpp_pro` version depended strongly on Google specific code and could not be open-sourced. All directories in Figure 2 are found in the `havlak` directory.

Each directory contains a `Makefile` and supports three methods of invocation:

```
make       # build benchmark
make run   # run benchmark
make clean # clean up build artifacts
```

Path names to compilers and libraries can be overridden on the command lines. Readers interested in running the benchmarks are encouraged to study the very short `Makefiles` for more details.

IV. Implementation Notes

This section highlights a few core language properties, as necessary to understanding the benchmark sources and performance characteristics. Readers familiar with the languages can safely skip to the next section.

A. Data Structures

The key data structures for the implementation of the algorithm are shown in Figure 3. Please note again that we are strictly following the algorithm's notation. We do not seek to apply any manual data structure optimization at this point. We use the non object-oriented, quirky notation of the paper, and we only use the languages' default containers.

1) C++: With the standard library and templates, these data structures are defined the following way in C++ using stack local variables:

```
typedef std::vector<UnionFindNode>  NodeVector;
typedef std::map<BasicBlock*, int>  BasicBlockMap;
typedef std::list<int>              IntList;
typedef std::set<int>               IntSet;
typedef std::list<UnionFindNode*>   NodeList;
typedef std::vector<IntList>        IntListVector;
typedef std::vector<IntSet>         IntSetVector;
typedef std::vector<int>            IntVector;
typedef std::vector<char>           CharVector;
[...]
  IntSetVector       non_back_preds(size);
  IntListVector      back_preds(size);
  IntVector          header(size);
  CharVector         type(size);
  IntVector          last(size);
  NodeVector         nodes(size);
  BasicBlockMap      number;
```

The size of these data structures is known at run-time and the `std::` collections allow pre-allocations at those sizes, except for the map type.

2) Java: Java does not allow arrays of generic types. However, lists are index-able, so this code is permissible:

```
List<Set<Integer>>  nonBackPreds =
              new ArrayList<Set<Integer>>();
List<List<Integer>> backPreds =
              new ArrayList<List<Integer>>();
int[]               header = new int[size];
BasicBlockClass[]   type =
              new BasicBlockClass[size];
int[]               last = new int[size];
UnionFindNode[]     nodes = new UnionFindNode[size];
Map<BasicBlock, Integer> number =
              new HashMap<BasicBlock, Integer>();
```

However, this appeared to incur tremendous GC overhead. In order to alleviate this problem we slightly rewrite the code, which reduced GC overhead modestly.

```
nonBackPreds.clear();
backPreds.clear();
number.clear();
if (size > maxSize) {
  header = new int[size];
  type = new BasicBlockClass[size];
  last = new int[size];
  nodes = new UnionFindNode[size];
  maxSize = size;
}
```

Constructors still need to be called:

```
for (int i = 0; i < size; ++i) {
  nonBackPreds.add(new HashSet<Integer>());
  backPreds.add(new ArrayList<Integer>());
  nodes[i] = new UnionFindNode();
}
```

To reference an element of the `ArrayLists`, the `get/set` methods are used, like this:

```
if (isAncestor(w, v, last)) {
        backPreds.get(w).add(v);
      } else {
        nonBackPreds.get(w).add(v);
      }
```

3) Scala: Scala allows arrays of generics, as an array is just a language extension, and not a built-in concept. Constructors still need to be called:

```
var nonBackPreds = new Array[Set[Int]](size)
var backPreds    = new Array[List[Int]](size)
var header       = new Array[Int](size)
var types        =
          new Array[BasicBlockClass.Value](size)
```

```
var last         = new Array[Int](size)
var nodes        = new Array[UnionFindNode](size)
var number       =
   scala.collection.mutable.Map[BasicBlock, Int]()

for (i <- 0 until size) {
    nonBackPreds(i) = Set[Int]()
    backPreds(i)    = List[Int]()
    nodes(i)        = new UnionFindNode()
}
```

With clever use of parenthesis (and invocation of `apply()` accesses become more canonical, e.g.:

```
    if (isAncestor(w, v, last)) {
        backPreds(w) = v :: backPreds(w)
    } else {
        nonBackPreds(w) += v
    }
```

4) Go: This language offers the `make` keyword in addition to the `new` keyword. `Make` takes away the pain of the explicit constructor calls. A map is a built-in type, and has a special syntax, as can be seen in the 1st line below. There is no `set` type, so in order to get the same effect, one can use a map to `bool`. While maps are built-in, lists are not, and as a result accessors and iterators become non-canonical.

```
nonBackPreds := make([]map[int]bool, size)
backPreds := make([]list.List, size)
number := make(map[*cfg.BasicBlock]int)
header := make([]int, size, size)
types := make([]int, size, size)
last := make([]int, size, size)
nodes := make([]*UnionFindNode, size, size)

for i := 0; i < size; i++ {
nodes[i] = new(UnionFindNode)
}
```

B. Enumerations

To enumerate the various kinds of loops an enumeration type is used. The following subsections show what the languages offer to express compile time constants.

1) C++: In C++ a regular `enum` type can be used

```
enum BasicBlockClass {
  BB_TOP,           // uninitialized
  BB_NONHEADER,     // a regular BB
  BB_REDUCIBLE,     // reducible loop
  BB_SELF,          // single BB loop
  BB_IRREDUCIBLE,   // irreducible loop
  BB_DEAD,          // a dead BB
  BB_LAST           // Sentinel
};
```

2) Java: Java has quite flexible support for enumeration types. Specifically, enum members can have constructor parameters, and enums also offer iteration via `values()`. Since the requirements for this specific benchmark are trivial, the code looks similarly simple:

```
public enum BasicBlockClass {
  BB_TOP,           // uninitialized
  BB_NONHEADER,     // a regular BB
  BB_REDUCIBLE,     // reducible loop
  BB_SELF,          // single BB loop
  BB_IRREDUCIBLE,   // irreducible loop
  BB_DEAD,          // a dead BB
  BB_LAST           // Sentinel
}
```

3) Scala: In Scala, enumerations become a static instance of a type derived from `Enumeration`. The syntax below calls and increments `Value()` on every invocation (parameterless function calls don't need parenthesis in Scala). Note that, in this regard, enums are not a language feature, but an implementation of the method `Enumeration.Value()`. `Value` also offers the ability to specify parameter values, similar to the Java enums with constructors.

```
class BasicBlockClass extends Enumeration {
}
object BasicBlockClass extends Enumeration {
  val BB_TOP,       // uninitialized
  BB_NONHEADER,     // a regular BB
  BB_REDUCIBLE,     // reducible loop
  BB_SELF,          // single BB loop
  BB_IRREDUCIBLE,   // irreducible loop
  BB_DEAD,          // a dead BB
  BB_LAST = Value   // Sentinel
}
```

4) Go: Go has the concept of an `iota` and initialization expression. For every member of an enumeration (constants defined within a `const` block), the right hand side expression will be executed in full, with `iota` being incremented on every invocation. `iota` is being reset on encountering the `const` keyword. This makes for flexible initialization sequences, which are not shown, as the use case is trivial. Note that because of Go's powerful line-breaking, not even a comma is needed. Furthermore, because of Go's symbol exporting rules, the first characters of the constants are held lower case (see comments on symbol binding later).

```
const (
 _             = iota // Go has the iota concept
 bbTop                // uninitialized
 bbNonHeader          // a regular BB
 bbReducible          // reducible loop
 bbSelf               // single BB loop
 bbIrreducible        // irreducible loop
 bbDead               // a dead BB
 bbLast               // sentinel
)
```

C. Iterating over Data Structures

1) C++: C++ has no "special" built-in syntactical support for iterating over collections (e.g., the upcoming range-based for-loops in C++0x). However, it does offer templates, and all basic data structures are now available in the standard library. Since the language support is minimal, to iterate, e.g., over a list of non-backedges, one has to write:

```
// Step d:
IntList::iterator back_pred_iter =
           back_preds[w].begin();
IntList::iterator back_pred_end  =
           back_preds[w].end();
for (; back_pred_iter != back_pred_end;
       back_pred_iter++) {
     int v = *back_pred_iter;
```

To iterate over all members of a map:

```
for (MaoCFG::NodeMap::iterator bb_iter =
     CFG_->GetBasicBlocks()->begin();
    bb_iter != CFG_->GetBasicBlocks()->end();
      ++bb_iter) {
```

```
    number[(*bb_iter).second] = kUnvisited;
}
```

2) Java: Java is aware of the base interfaces supported by collection types and offers an elegant language extension for easier iteration. In a sense, there is "secret" handshake between the run time libraries and the Java compiler. The same snippets from above look like the following in Java:

```
      // Step d:
      for (int v : backPreds.get(w)) {
```

and for the map:

```
    for (BasicBlock bbIter :
           cfg.getBasicBlocks().values()) {
      number.put(bbIter, UNVISITED);
```

3) Scala: Scala offers similar convenience for iterating over lists:

```
    // Step d:
    for (v <- backPreds(w)) {
```

and for maps iterations:

```
    for ((key, value) <- cfg.basicBlockMap) {
      number(value) = UNVISITED
    }
```

These "for-comprehensions" are very powerful. Under the hood, the compiler front-end builds closures and transforms the code into combinations of calls to `map()`, `flatmap`, `filter()`, and `foreach()`. Loop nests and conditions can be expressed elegantly. Each type implementing the base interfaces can be used in for-comprehensions. This can be used for elegant language extensions. On the downside - since closures are being built, there is performance overhead. The compiler transforms the for-comprehensions and therefore there is also has a "secret" hand-shake between libraries and compiler. More details on Scala for-comprehensions can be found in Section 6.19 of the Scala Language Specification [3].

Note that in the example, the tuple on the left side of the arrow is not a built-in language construct, but cleverly written code to mimic and implement tuples.

4) Go: Lists are not built-in, and they are also not type-safe. As a result, iterations are more traditional and require explicit casting. For example:

```
// Step d:
for ll := backPreds[w].Front(); ll != nil;
    ll = ll.Next() {
       v := ll.Value.(int)
```

Since maps are built-in, there is special range keyword and the language allows returning tuples:

```
for i, bb := range cfgraph.BasicBlocks() {
       number[bb] = unvisited
```

Note that lists are inefficient in Go (see performance and memory footprint below). The GO Pro version replaces most lists with array slices, yielding significant performance gains (25%), faster compile times, and more elegant code. We denote removed lines with − and the replacement lines with +, similarly to the output of modern `diff` tools. For example, the LSG data structure changes:

```
type LSG struct {
  root  *SimpleLoop
-       loops list.List
+       loops []*SimpleLoop
 }
```

and references in `FindLoops()` change accordingly:

```
  - backPreds := make([]list.List, size)
  + backPreds := make([][]int, size)
```

and

```
- for ll := nodeW.InEdges().Front(); ll != nil;
-     ll = ll.Next() {
-       nodeV := ll.Value.(*cfg.BasicBlock)

+ for _, nodeV := range nodeW.InEdges {
```

Note that both Go and Scala use _ as a placeholder.

D. Type Inference

C++ and Java require explicit type declarations. Both Go and Scala have powerful type inference, which means that types rarely need to be declared explicitly.

1) Scala: Because of Scala functional bias, variables or values must be declared as such. However, types are inferred. For example:

```
    var lastid = current
```

2) Go: To declare and define a variable lastid and assign it the value from an existing variable current, Go has a special assignment operator :=

```
       lastid := current
```

E. Symbol Binding

The languages offer different mechanism to control symbol bindings:

1) C++: C++ relies on the `static` and `external` keywords to specify symbol bindings.

2) Java: Java uses packages and the `public` keyword to control symbol binding.

3) Scala: Scala uses similar mechanism as Java, but there are differences in the package name specification, which make things a bit more concise and convenient. Details are online at [2] and [4].

4) Go: Go uses a simple trick to control symbol binding. If a symbol's first character is uppercase - it's being exported.

F. Member Functions

1) C++: Google's coding style guidelines require that class members be accessed through accessor functions. For C++, simple accessor functions come with no performance penalty, as they are inlined. Constructors are explicitly defined. For example, for the `BasicBlockEdge` class in the benchmarks, the code looks like the following:

```
class BasicBlockEdge {
 public:
  inline BasicBlockEdge(MaoCFG *cfg,
                        int     from,
```

```
                        int to);

  BasicBlock *GetSrc() { return from_; }
  BasicBlock *GetDst() { return to_; }

 private:
  BasicBlock *from_, *to_;
};
[...]
inline
BasicBlockEdge::BasicBlockEdge(MaoCFG  *cfg,
                               int      from_name,
                               int      to_name) {
  from_ = cfg->CreateNode(from_name);
  to_ = cfg->CreateNode(to_name);

  from_->AddOutEdge(to_);
  to_->AddInEdge(from_);

  cfg->AddEdge(this);
}
```

2) Java: Java follows the same ideas, and the resulting code looks similar:

```
public class BasicBlockEdge {
  public BasicBlockEdge(CFG cfg,
                        int fromName,
                        int toName) {
    from = cfg.createNode(fromName);
    to   = cfg.createNode(toName);

    from.addOutEdge(to);
    to.addInEdge(from);

    cfg.addEdge(this);
  }

  public  BasicBlock getSrc() { return from; }
  public  BasicBlock getDst() { return to; }

  private BasicBlock from, to;
};
```

3) Scala: The same code can be expressed in a very compact fashion in Scala. The class declaration allows parameters, which become instance variables for every object. Constructor code can be placed directly into the class definition. The variables `from` and `to` become accessible for users of this class via automatically generated getter functions. Since parameterless functions don't require parenthesis, such accessor functions can always be rewritten later, and no software engineering discipline is lost. Scala also produces setter functions, which are named like the variables with an appended underscore. E.g., this code can use `from_(int)` and `to_(int)` without having to declare them. The last computed value is the return value of a function, saving on return statements (not applicable in this example)

```
class BasicBlockEdge(cfg      : CFG,
                     fromName : Int,
                     toName   : Int) {
  var from : BasicBlock = cfg.createNode(fromName)
  var to   : BasicBlock = cfg.createNode(toName)

  from.addOutEdge(to)
  to.addInEdge(from)

  cfg.addEdge(this)
}
```

4) Go: Go handles types in a different way. It allows declaration of types and then the definition of member functions as traits to types. The resulting code would look like this:

```
type BasicBlockEdge struct {
       to   *BasicBlock
       from *BasicBlock
}

func (edge *BasicBlockEdge) Dst() *BasicBlock {
       return edge.to
}

func (edge *BasicBlockEdge) Src() *BasicBlock {
        return edge.from
}

func NewBasicBlockEdge(cfg *CFG, from int, to int)
         *BasicBlockEdge {
      self := new(BasicBlockEdge)
      self.to = cfg.CreateNode(to)
      self.from = cfg.CreateNode(from)

      self.from.AddOutEdge(self.to)
      self.to.AddInEdge(self.from)

      return self
}
```

However, the 6g compiler does not inline functions and the resulting code would perform quite poorly. Go also has the convention of accessing members directly, without getter/setter functions. Therefore, in the Go Pro version, the code is becoming a lot more compact:

```
type BasicBlockEdge struct {
       Dst *BasicBlock
       Src *BasicBlock
}

func NewBasicBlockEdge(cfg *CFG, from int, to int)
          *BasicBlockEdge {
       self := new(BasicBlockEdge)
       self.Dst = cfg.CreateNode(to)
       self.Src = cfg.CreateNode(from)

       self.Src.AddOutEdge(self.Dst)
       self.Dst.AddInEdge(self.Src)

       return self
}
```

V. Performance Analysis

The benchmark consists of a driver for the loop recognition functionality. This driver constructs a very simple control flow graph, a diamond of 4 nodes with a backedge from the bottom to the top node, and performs loop recognition on it for 15.000 times. The purpose is to force Java/Scala compilation, so that the benchmark itself can be measured on compiled code.

Then the driver constructs a large control flow graph containing 4-deep loop nests with a total of 76000 loops. It performs loop recognition on this graph for 50 times. The driver codes for all languages are identical.

As a result of this benchmark design, the Scala and Java measurements contain a small fraction of interpreter time. However, this fraction is very small and run-time results presented later should be taken as "order of magnitude" statements only.

Benchmark	wc -l	Factor
C++ Dbg/Opt	850	1.3x
Java	1068	1.6x
Java Pro	1240	1.9x
Scala	658	1.0x
Scala Pro	297	0.5x
Go	902	1.4x
Go Pro	786	1.2x

Fig. 4. Code Size in [Lines of Code], normalized to the Scala version

Benchmark	Compile Time	Factor
C++ Dbg	3.9	6.5x
C++ Opt	3.0	5.0x
Java	3.1	5.2x
Java Pro	3.0	5.0x
Scala scalac	13.9	23.1x
Scala fsc	3.8	6.3x
Scala Pro scalac	11.3	18.8x
Scala Pro fsc	3.5	5.8x
Go	1.2	2.0x
Go Pro	0.6	1.0x

Fig. 5. Compilation Times in [Secs], normalized to the Go Pro version

The benchmarking itself was done in a simple and fairly un-scientific fashion. The benchmarks were run 3 times and the median values are reported. All of the experiments are done on an older Pentium IV workstation. Run-times were measured using wall-clock time. We analyze the benchmarks along the dimensions code size, compile time, binary size, memory footprint, and run-time.

A. Code Size

The benchmark implementations contain similar comments in all codes, with a few minor exceptions. Therefore, to compare code size, a simple Linux `wc` is performed on the benchmarking code and the main algorithm code together, counting lines. The results are shown in Figure 7. Code is an important factor in program comprehension. Studies have shown that the average time to recognize a token in code is a constant for individual programmers. As a result, less tokens means goodness. Note that the Scala and Go versions are significantly more compact than the verbose C++ and Java versions.

B. Compile Times

In Figure 5 we compare the static compile times for the various languages. Note that Java/Scala only compile to Java Byte Code. For Scala, the `scalac` compiler is evaluated, as well as the memory resident `fsc` compiler, which avoids re-loading of the compiler on every invocation. This benchmark is very small in terms of lines of code, and unsurprisingly, `fsc` is about 3-4x faster than `scalac`.

Benchmark	Binary or Jar [Byte]	Factor
C++ Dbg	592892	45x
C++ Opt	41507	3.1x
Java	13215	1.0x
Java Pro	21047	1.6x
Scala	48183	3.6x
Scala Pro	36863	2.8x
Go	1249101	94x
Go Pro	1212100	92x

Fig. 6. Binary and JAR File Sizes in [Byte], normalized to the Java version

Benchmark	Virt	Real	Factor Virt	Factor Real
C++ Opt	184m	163m	1.0	1.0
C++ Dbg	474m	452m	2.6-3.0	2.8
Java	1109m	617m	6.0	3.7
Scala	1111m	293m	6.0	1.8
Go	16.2g	501m	90	3.1

Fig. 7. Memory Footprint, normalized to the C++ version

C. Binary Sizes

In Figure 6 we compare the statically compiled binary sizes, which may contain debug information, and JAR file sizes. Since much of the run-time is not contained in Java JAR files, they are expected to be much smaller in size and used as baseline. It should also be noted that for large binaries, plain binary size matters a lot, as in distributed build systems, the generated binaries need to be transferred from the build machines, which can be bandwidth limited and slow.

D. Memory Footprint

To estimate memory footprint, the benchmarks were run alongside the Unix `top` utility and memory values were manually read out in the middle of the benchmark run. The numbers are fairly stable across the benchmark run-time. The first number in the table below represents potentially pre-allocated, but un-touched virtual memory. The second number (Real) represents real memory used.

E. Run-time Measurements

It should be noted that this benchmark is designed to amplify language implementations' negative properties. There is lots of memory traversal, so minimal footprint is beneficial. There is recursion, so small stack frames are beneficial. There are lots of array accesses, so array bounds checking will be negative. There are many objects created and destroyed - this could be good or bad for the garbage collector. As shown, GC settings have huge impact on performance. There are many pointers used and null-pointer checking could be an issue. For Go, some key data structures are part of the language, others are not. The results are summarized in Figure 8.

VI. Tunings

Note again that the original intent of this language comparison was to provide generic and straightforward implementations of a well specified algorithm, using the default language

Benchmark	Time [sec]	Factor
C++ Opt	23	1.0x
C++ Dbg	197	8.6x
Java 64-bit	134	5.8x
Java 32-bit	290	12.6x
Java 32-bit GC*	106	4.6x
Java 32-bit SPEC GC	89	3.7x
Scala	82	3.6x
Scala low-level*	67	2.9x
Scala low-level GC*	58	2.5x
Go 6g	161	7.0x
Go Pro*	126	5.5x

Fig. 8. Run-time measurements. Scala low-level is explained below, it has a hot for-comprehension replaced with a simple `foreach()`. Go Pro is the version that has all accessor functions removed, as well as all lists. The Scala and Java GC versions use optimized garbage collection settings, explained below. The Java SPEC GC version uses a combination of the SPECjbb settings and -XX:+CycleTime

containers and looping idioms. Of course, this approach also led to sub-optimal performance.

After publication of this benchmark internally to Google, several engineers signed up and created highly optimized versions of the program, some of them without keeping the original program structure intact.

As these tuning efforts might be indicative to the problems performance engineers face every day with the languages, the following sections contain a couple of key manual optimization techniques and tunings for the various languages.

A. Scala/Java Garbage Collection

The comparison between Java and Scala is interesting. Looking at profiles, Java shows a large GC component, but good code performance

```
100.0% 14073             Received ticks
 74.5% 10484             Received GC ticks
  0.8%   110             Compilation
  0.0%     2             Other VM operations
```

Scala has this breakdown between GC and compiled code:

```
100.0%  7699             Received ticks
 31.4%  2416             Received GC ticks
  4.8%   370             Compilation
  0.0%     1             Class loader
```

In other words, Scala spends 7699-2416-1 = 5282 clicks on non-GC code. Java spends 14073-10484-110-2 = 3477 clicks on non-GC code. Since GC has other negative performance side-effects, one could estimate that without the GC anomaly, Java would be about roughly 30% faster than Scala.

B. Eliminating For-Comprehension

Scala shows a high number of samples in several `apply()` functions. Scala has 1st-class functions and anonymous functions - the for-comprehension in the DFS routine can be rewritten from:

```
    for (target <- currentNode.outEdges
         if (number(target) == UNVISITED)) {
           lastid = DFS(target, nodes, number,
                        last, lastid + 1)
       }
```

to:

```
  currentNode.outEdges.foreach(target =>
         if (number(target) == UNVISITED) {
           lastid = DFS(target, nodes, number,
                        last, lastid + 1)
         }
       )
```

This change alone shaves off about 15secs of run-time, or about 20% (slow run-time / fast run-time). This is marked as Scala Low-Level in Figure 8. In other words, while for-comprehensions are supremely elegant, a better compiler should have found this opportunity without user intervention.

Note that Java also creates a temporary object for foreach loops on loop entry. Jeremy Manson found that this was one of the main causes of the high GC overhead for the Java version and changed the Java `forall` loops to explicit `while` loops.

C. Tuning Garbage Collection

Originally the Java and Scala benchmarks were run with 64-bit Java, using `-XX:+UseCompressedOops`, hoping to get the benefits of 64-bit code generation, without incurring the memory penalties of larger pointers. Run-time is 134 secs, as shown in above table.

To verify the assumptions about 64/32 bit java, the benchmark was run with a 32-bit Java JVM, resulting in run-time of 290 secs, or a 2x slowdown over the 64-bit version. To evaluate the impact of garbage collection further, the author picked the GC settings from a major Google application component which is written in Java. Most notably, it uses the concurrent garbage collector.

Running the benchmark with these options results in run-time of 106 secs, or a 3x improvement over the default 32-bit GC settings, and a 25% improvement over the 64-bit version.

Running the 64-bit version with these new GC settings, run-time was 123 secs, a further roughly 10% improvement over the default settings.

Chris Ruemmler suggested to evaluate the SPEC JBB settings as well. With these settings, run-time for 32-bit Java reduced to 93 secs, or another 12% faster. Trying out various combinations of the options seemed to indicate that these settings were highly tuned to the SPEC benchmarks.

After some discussions, Jeremy Manson suggested to combine the SPEC settings with `-XX:+CycleTime`, resulting in the fasted execution time of 89 seconds for 32-bit Java, or about 16% faster than the fastest settings used by the major Google application. These results are included in above table as Java 32-bit SPEC GC.

Finally, testing the Scala-fixed version with the Google application GC options (32-bit, and specialized GC settings), performance improved from 67 secs to 58 secs, representing another 13% improvement.

What these experiments and their very large performance impacts show is that tuning GC has a disproportionate high effect on benchmark run-times.

D. C++ Tunings

Several Google engineers contributed performance enhancements to the C++ version. We present most changes via citing from the engineers' change lists.

Radu Cornea found that the C++ implementation can be improved by using hash_maps instead of maps. He got a 30% improvement

Steinar Gunderson found that the C++ implementation can be improved by using `empty()` instead of `size() > 0` to check whether a list contains something. This didn't produce a performance benefit - but is certainly the right thing to do, as `list.size()` is $O(n)$ and `list.empty()` is $O(1)$.

Doug Rhode created a greatly improved version, which improved performance by 3x to 5x. This version will be kept in the `havlak_cpp_pro directory`. At the time of this writing, the code was heavily dependent on several Google internal data structures and could not be open sourced. However, his list of change comments is very instructional:

30.4% Changing `BasicBlockMap` from `map<BasicBlock*, int>` to `hash_map`
12.1% Additional gain for changing it to `hash_map<int, int>` using the BasicBlock's `name()` as the key.
2.1% Additional gain for pre-sizing `BasicBlockMap`.
10.8% Additional gain for getting rid of `BasicBlockMap` and storing `dfs_numbers` on the BasicBlocks themselves.
7.8% Created a `TinySet` class based on `InlinedVector<>` to replace `set<>` for `non_back_preds`.
1.9% Converted `BasicBlockSet` from `set<>` to `vector<>`.
3.2% Additional gain for changing `BasicBlockSet` from `vector<>` to `InlinedVector<>`.
2.6% Changed `node_pool` from a `list<>` to a `vector<>`. Moved definition out of the loop to recycle it.
1.9% Change worklist from a `list<>` to a `deque<>`.
2.2% Changed LoopSet from `set<>` to `vector<>`.
1.1% Changed LoopList from `list<>` to `vector<>`.
1.1% Changed EdgeList from `list<BasicBlockEdge*>` to `vector<BasicBlockEdge>`.
1.2% Changed from using int for the node dfs numbers to a logical `IntType<uint32> NodeNum`, mainly to clean up the code, but unsigned array indexes are faster.
0.4% Got rid of parallel vectors and added more fields to `UnionFindNode`.
0.2% Stack allocated LoopStructureGraph in one loop.
0.2% Avoided some UnionFindNode copying when assigning local vars.
0.1% Rewrote path compression to use a double sweep instead of caching nodes in a list.

Finally, David Xinliang Li took this version, and applied a few more optimizations:

Structure Peeling. The structure `UnionFindNode` has 3 cold fields: `type_`, `loop_`, and `header_`. Since nodes are allocated in an array, this is a good candidate for peeling optimization. The three fields can be peeled out into a separate array. Note the `header_ field` is also dead – but removing it has very little performance impact. The `name_` field in the `BasicBlock` structure is also dead, but it fits well in the padding space so it is not removed.

Structure Inlining. In the `BasicBlock` structure, vector is used for incoming and outgoing edges – making it an inline vector of 2 element remove the unnecessary memory indirection for most of the cases, and improves locality.

Class object parameter passing and return. In Doug Rhode's version, `NodeNum` is a `typedef` of the `IntType<...>` class. Although it has only one integer member, it has non trivial copy constructor, which makes it to be of memory class instead of integer class in the parameter passing conventions. For both parameter passing and return, the low level ABI requires them to be stack memory, and C++ ABI requires that they are allocated in the caller – the first such aggregate's address is passed as the first implicit parameter, and the rest aggregates are allocated in a buffer pointed to by `r8`. For small functions which are inlined, this is not a big problem – but DFS is a recursive function. The downsides include unnecessary memory operation (passing/return stack objects) and increased register pressure – `r8` and `rdi` are used.

Redundant this. DFS's `this` parameter is never used – it can be made a static member function

E. Java Tunings

Jeremy Manson brought the performance of Java on par with the original C++ version. This version is kept in the `java_pro` directory. Note that Jeremy deliberately refused to optimize the code further, many of the C++ optimizations would apply to the Java version as well. The changes include:

- Replaced `HashSet` and `ArrayList` with much smaller equivalent data structures that don't do boxing or have large backing data structures.
- Stopped using `foreach` on `ArrayLists`. There is no reason to do this it creates an extra object every time, and one can just use direct indexing.
- Told the `ArrayList` constructors to start out at size 2 instead of size 10. For the last 10% in performance, use a free list for some commonly allocated objects.

F. Scala Tunings

Daniel Mahler improved the Scala version by creating a more functional version, which is kept in the Scala Pro directories. This version is only 270 lines of code, about 25% of the C++ version, and not only is it shorter, run-time also improved by about 3x. It should be noted that this version performs algorithmic improvements as well, and is therefore not directly comparable to the other *Pro* versions. In detail, the following changes were performed.

The original Havlak algorithm specification keeps all node-related data in global data structures, which are indexed by integer node id's. Nodes are always referenced by id. Similar to the data layout optimizations in C++, this implementation introduces a `node` class which combines all node-specific information into a single class. Node objects are referenced

```
  for (int parlooptrees = 0; parlooptrees < 10;
       parlooptrees++) {
    cfg.CreateNode(n + 1);
    buildConnect(&cfg, 2, n + 1);
    n = n + 1;

    for (int i = 0; i < 100; i++) {
      int top = n;
      n = buildStraight(&cfg, n, 1);
      for (int j = 0; j < 25; j++) {
        n = buildBaseLoop(&cfg, n);
      }
      int bottom = buildStraight(&cfg, n, 1);
      buildConnect(&cfg, n, top);
      n = bottom;
    }
    buildConnect(&cfg, n, 1);
  }
```

Fig. 9. C++ code to construct the test graph

```
(1 to 10).foreach(
      _ => {
        n2
        .edge
        .repeat(100,
                _.back(_.edge
                       .repeat(25,
                               baseLoop(_)))
                .edge)
        .connect(n1)
      })
```

Fig. 10. Equivalent Scala code to construct the test graph

and passed around in an object-oriented fashion, and accesses to global objects are no longer required

Most collection were replaced by using `ArrayBuffer`. This data structure is Scala's equivalent of C++'s `vector`. Most collections in the algorithm don't require any more advanced functionality than what this data structure offers.

The main algorithmic change was to make the main search control loop recursive. The original algorithm performs the search using an iterative loop and maintains an explicit 'worklist' of nodes to be searched. Transforming the search into more natural recursive style significantly simplified the algorithm implementation.

A significant part of the original program is the specification of a complex test graph, which is written in a procedural style. Because of this style, the structure of the test graph is not immediately evident. Scala made it possible to introduce a declarative DSL which is more concise and additionally makes the structure of the graph visible syntactically. It should be noted that similar changes could have been made in other languages, e.g., in C++ via nested constructor calls, but Scala's support for functional programming seemed to make this really easy and required little additional supporting code.

For example, for the construction of the test graph, the procedural C++ code shown in Figure 9 turns into the more functional Scala code shown in Figure 10:

VII. Conclusions

We implemented a well specified compact algorithm in four languages, C++, Java, Go, and Scala, and evaluated the results along several dimensions, finding factors of differences in all areas. We discussed many subsequent language specific optimizations that point to typical performance pain points in the respective languages.

We find that in regards to performance, C++ wins out by a large margin. However, it also required the most extensive tuning efforts, many of which were done at a level of sophistication that would not be available to the average programmer.

Scala concise notation and powerful language features allowed for the best optimization of code complexity.

The Java version was probably the simplest to implement, but the hardest to analyze for performance. Specifically the effects around garbage collection were complicated and very hard to tune. Since Scala runs on the JVM, it has the same issues.

Go offers interesting language features, which also allow for a concise and standardized notation. The compilers for this language are still immature, which reflects in both performance and binary sizes.

VIII. Acknowledgments

We would like to thank the many people commenting on the benchmark and the proposed methodology. We would like to single out the following individuals for their contributions: Jeremy Manson, Chris Ruemmler, Radu Cornea, Steinar H. Gunderson, Doug Rhode, David Li, Rob Pike, Ian Lance Taylor, and Daniel Mahler. We furthermore thank the anonymous reviewers, their comments helped to improve this paper.

References

[1] Havlak, P. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst. 19*, 4 (1997), 557–567.

[2] Odersky, M. Chained package clauses. http://www.scala-lang.org/docu/files/package-clauses/packageclauses.html.

[3] Odersky, M. The Scala language specification, version 2.8. http://www.scala-lang.org/docu/files/ScalaReference.pdf.

[4] Odersky, M., and Spoon, L. Package objects. http://www.scala-lang.org/docu/files/packageobjects/packageobjects.html.

[5] Tarjan, R. Testing flow graph reducibility. In *Proceedings of the fifth annual ACM symposium on Theory of computing* (New York, NY, USA, 1973), STOC '73, ACM, pp. 96–107.

[6] Tarjan, R. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[7] Wikipedia. C++. http://en.wikipedia.org/wiki/C++.

[8] Wikipedia. Go. http://en.wikipedia.org/wiki/Go_(programming_language).

[9] Wikipedia. Java. http://en.wikipedia.org/wiki/Java_(programming_language).

[10] Wikipedia. Scala. http://en.wikipedia.org/wiki/Scala_(programming_language).