McGill University
School of Computer Science
Sable Research Group

STOOP: The Sable Toolkit for Object-Oriented Profiling

Sable Technical Report No. 2001-2

Rhodes Brown, Karel Driesen, David Eng, Laurie Hendren,
John Jorgensen, Clark Verbrugge and Qin Wang

November 26, 2001

Contents

List of Figures

List of Tables

Abstract

The performance and behaviour of large object-oriented software systems can be difficult to profile. Furthermore, existing profiling systems are often limited by a fixed set of events that can be profiled (usually time and space consumption), and a fixed set of visualizations. This paper presents the Sable Toolkit for Object-Oriented Profiling (STOOP), which we developed to support a wide variety of profiling needs. The toolkit makes it simple to build new profilers which can be used for both standard and less conventional profiling tasks.

At the core of the STOOP toolkit is its event pipe. A profiling agent produces events for the event pipe and a visualizer consumes events from the pipe. To provide a general purpose event pipe we have defined a language, STEP-DL, for defining events and a compiler, `stepc`, which automatically produces the event pipe interfaces for the specified events. These interfaces provide the methods for both writing to the event pipe and reading from it. Many different profiling agents can produce events for the event pipe, including JVMPI agents, instrumented VMs and instrumented bytecode. This paper illustrates the process with instrumented bytecode, but we have used our system with all three types of profiling agents.

It is also simple to adapt a variety of visualizers to read events from the event pipe. We have worked with two visualizers, EVolve and JIMPLEX. We summarize each visualizer and show how easy it was to adapt them to use events produced from STOOP.

1 Introduction and Overview

It is often very difficult to understand the performance and behaviour of object-oriented programs, particularly for large, complex software systems consisting of many packages and classes. Our research groups are interested in tools that will reveal the behaviour of Java programs and help us to: 1) develop new compiler optimizations, 2) design new memory management strategies, 3) understand Java virtual machine requirements and 4) evaluate new hardware features. For example, understanding the behaviour of hot data fields and the relationships between field accesses may help us to develop new strategies to make better use of data caches.

Although there are tools available to profile and visualize Java programs, those we have encountered are limited by the fixed set of data they collect, and the fixed set of visualizations they provide to their users. Frustrated by this limitation, we developed STOOP: the Sable Toolkit for Object-Oriented Profiling, a framework for creating tools to collect and visualize arbitrary profile data. STOOP provides high-level facilities for rapidly constructing tools to collect and visualize profile data from the execution of object-oriented programs. We have implemented STOOP in Java and applied the toolkit to a variety of tasks for profiling Java.

As illustrated in Figure 1, a profiler based on STOOP consists of three major pieces: a profiling agent, an event pipe, and a visualizer. A profiling agent collects the desired profile data and feeds them to the event pipe as a stream of events. We support a wide variety of profiling agents including agents using the Java Virtual Machine Profiling Interface (JVMPI), instrumented virtual machines, and instrumented bytecode. A visualizer reads the stream of profile events leaving the pipe and presents the data to the profiler end-user.

The STOOP event pipe is the core of our system, serving as the high-level interface between the front-end profiling agent and the back-end visualizer. It converts profile data to a binary format, buffers (and potentially compresses) the data, then forwards it to the visualizer. In order to provide a clear and flexible interface, profile events are described using a specification language, STEP-DL. STEP-DL allows profiler developers to specify their events at a highlevel, without worrying about low-level encoding issues. Furthermore, STEP-DL makes it simple to add new record types and to extend existing types using inheritance. A compiler, `stepc`, generates the code for the interface to the event pipe. A profiling agent uses the generated routines to write to the event pipe, while a visualizer uses them to read from the pipe. Our experiments show that it is easy to define events using STEP-DL and that the generated binary event traces are compact and very compressible using off-the-shelf compression programs.

This paper is organized as follows. First, in Section 2, we describe the event pipe architecture (STEP), with a particular emphasis on the STEP-DL specification language. Then we discuss how a complete profiling system can be built by adding a profiling agent as the front-end and a visualizer as the back-end. In Section

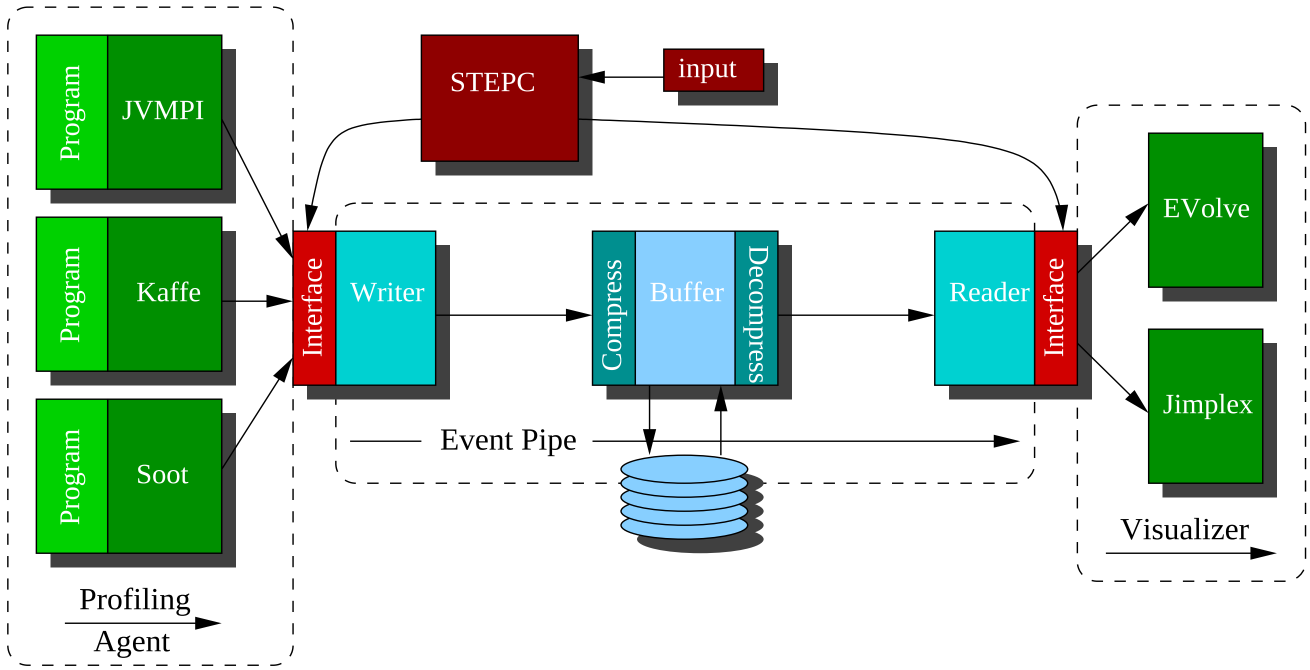

Figure 1: Overview of the Stoop framework

3 we discuss how we have built a variety of profiling agents to produce events using the interface routines generated by `stepc`. In Section 4 we describe two very different visualizers, EVolve and Jimplex, and show how they use the events produced by the event pipe. We conclude and mention some related systems in section 5.

2 The Event Pipe

At the core of the Stoop framework is the *event pipe*. The event pipe is a transport layer which decouples the event producer, the profiling agent, from the event consumer, the visualization agent. The event pipe provides an abstraction that allows a variety of producer and consumer agents to be connected to a profiler interchangeably.

In the rest of section 2 we motivate the the need for a separate transport layer, then we detail the event pipe architecture, dubbed Step. In short, Step provides a mechanism for abstracting trace data records. It is this abstract interface that facilitates the interchange of producer and consumer agents.

2.1 Motivation & Background

The Stoop framework was initially motivated by our own need to simplify the processing of trace data. We had developed several ad-hoc systems for collecting various profile data and were seeking a method to feed this data to a common visualization system. In developing this common interface, we encountered the need to address several important issues, namely: 1) variability in the type of data, 2) variability in the format of the data and 3) extremely large quantities of data. For example, our traces contained different types of data (e.g. object allocations, method entries and exits, address loads and stores, etc.) and the data was often obtained from different sources (e.g. an instrumented Kaffe[14] Virtual Machine running on Linux, a JVMPI agent running on Solaris, etc.). Furthermore, the data was stored in an inefficient format that often required several gigabytes to record a single program run. Realizing that these issues are common to many, if not all, profiler-developers, we began to create a framework that would simplify the development process

Benchmark	Trace Size	Record Count	Fields/Record	Bytes/Field
_202_jess	136.17 MB	29677410	4.43	1.09
_213_javac	42.75 MB	8236998	4.41	1.23
_222_mpegaudio	70.73 MB	13256375	4.75	1.18
_228_jack	53.13 MB	11819550	4.52	1.04

Table 1: STEP encoding efficiency

by encapsulating such details within a more intuitive abstraction.

During the early development of the event pipe, we were influenced by the attempts of Chilimbi, Jones and Zorn[3] to develop a standard for heap allocation traces. The standard, called HATF, was written using Meta-TF, their language for defining the content and interpretation of trace data records. Chilimbi *et al.* focus on providing a simple, yet highly compressed format, and provide a convincing argument for the use of a specialized format over a more general system such as XML.

The notion of using a definition language to describe the trace format appealed to us. In fact, we had hoped to use Meta-TF directly as part of the event pipe system, however there are some drawbacks to the language that prevented us from doing so. In particular, Meta-TF views all record fields as integer data, deferring any interpretation of the data to the user level. This deferral makes any system built on top of such a trace format brittle; if the interpretation of the format changes, so must its producer and consumer. To balance the concerns of flexibility and re-usability, we instead engineered the STOOP event pipe to provide both a definition language and an architecture that encapsulates the details of data representation.

2.2 The STOOP Trace Event Protocol

At the heart of the event pipe architecture is the STOOP Trace Event Protocol (STEP). STEP provides an intuitive abstraction of the trace as a stream of record objects, defined with a specialized definition language, STEP-DL, described in section 2.3. The record definitions are compiled to Java or C code that encapsulates the encoding details by building records from a set of basic field types.

The basic STEP field types have two important features. First, their encoding removes issues of byte-order translations between producer and consumer systems. Second, the STEP encoding process attempts to automatically minimize the number of bytes required to represent a given basic field value. Currently, the basic fields that STEP supports are: integer, string, data and identifier. Integer and string fields are straightforward extensions of their natural forms, encoded with a minimum number of bytes. Data fields are similar to strings, but instead hold an arbitrary array of bytes. Identifier fields are a specialized variant of string fields for recurring values.

2.2.1 Identifier Fields

Early in the development of STEP, we noticed that the vast majority of fields in our traces could be classified as *identifiers*. An identifier is a value that is used repeatedly as a tag or label. For example, an allocation record might contain identifiers such as the allocated type and the allocation site. Clearly it is wasteful to store the full representation of such values (e.g. "`spec.benchmarks._213_javac.UnsignedShiftRight-Expression`") in each record. With an identifier field, users see the full representation, but the value is actually encoded using a compact integer id. To maintain compact ids, separate mappings from id to value exist for each field of each record type. For example, if a trace contains both allocation and object field access records, both records may contain a "type" identifier, indicating the allocated type in an allocation record and the declaring (base) object type in a field access record. Since the distribution of "type" values is likely to differ for the two record types, separate mappings are kept to minimize the number of bytes needed for the id values.

Appropriate use of identifier fields contributes to a particularly efficient encoding format. Table 1 shows the effectiveness of the STEP encoding on traces gathered from several of the SPECjvm98 benchmarks. The traces were produced by instrumenting the application classes using SOOT[13, 15], a bytecode transformation tool. They contain records for method entries and exits, allocations and field accesses. As the table indicates, the average size of a field is close to one byte. These results, although preliminary, indicate that the simple technique of translating identifiers to id values leads to a nearly optimal byte-level encoding. Section 2.2.4 elaborates further on the opportunities for compressing the STEP format.

2.2.2 Translators

Consider a trace composed of various record types, each consisting of a different collection of fields. How is such a stream to be tokenized into its constituent records and fields? An approach based on a mark-up language such as XML would wrap each data element with delimiting tags. Such an approach is very flexible and allows the trace to contain virtually any serializeable data. The addition of tagging information can, however, significantly increase the size of a trace file. A second approach, the one taken by Meta-TF (mentioned in section 2.1), is to avoid mark-up tags by defining the record formats ahead of time. Incorporating the record definitions directly into the trace reader and writer allows records to be stored using a single identification field to indicate their type. Unfortunately, a Meta-TF specification requires a particular instantiation (such as HATF) to actually provide the reading and writing mechanism, and it cannot easily be extended to handle new record types: adding a new record requires modification of the reader and writer to handle the new type. Furthermore, the record identification tags are assigned explicitly in the Meta-TF definition thus rendering two instances that use the same tag value incompatible.

Although the Meta-TF system has drawbacks, it does provide the useful notion of "interpretation" attributes for each field. These attributes indicate whether a particular transformation needs to be applied to the field in order to regenerate the original value. Meta-TF only considers interpretations that apply to integer values. STEP attempts to generalize this idea by introducing the notion of a *translator* that is associated with each record and field type. Translators underlie the STEP encoding, encapsulating the rules for transforming a data element from its surface form to its binary form.

As records are passed to a STEP writer, the writer defers the construction of a binary representation to the translator associated with the incoming record type. The record translator may, in turn, defer to sub-translators to construct the individual field elements. The writer then outputs the value returned from the translator along with a dynamically allocated identification tag. A STEP reader performs the inverse process; identifying the record type, then deferring construction of the record to the appropriate translator.

It is the ability to defer the encoding and decoding of records to the appropriate translator object that makes the STEP system extensible. Adding a new record type involves simply providing an appropriate translator. This process is further simplified through the use of the `stepc` compiler (described in section 2.3), which automatically generates a STEP record definition along with the necessary translator object.

2.2.3 Metarecords

The most important feature of the STEP encoding is that it is dynamic. For example, consider a string field used to record class names within a class-load record. Most class names are less than 255 characters, and thus require only a single byte to encode their length. However, the length field of the "classname" string cannot be fixed at a single byte since the length of some class names may exceed 255 characters, requiring at least two bytes to encode the length. The STEP system addresses this issue by allowing field attributes to vary as needed. The changes are indicated through the use of *metarecords*, an idea also used by the Meta-TF system.

STEP uses metarecords internally to signal updates to the various translator objects. As described above, metarecords can be used to signal a change in field attributes or interpretation. The most common use of metarecords occurs in the implementation of identifier fields (described in section 2.2.1). When a new identifier value is encountered, a metarecord is generated to specify the translation from id to value.

Benchmark	Trace Size	`gzip` Size	ratio (%)	`bzip2` Size	ratio (%)
_202_jess	136.17 MB	1237.55 KB	0.89	247.11 KB	0.18
_213_javac	42.75 MB	1180.74 KB	2.70	521.06 KB	1.19
_222_mpegaudio	70.73 MB	825.69 KB	1.14	313.87 KB	0.43
_228_jack	53.13 MB	670.91 KB	1.23	168.49 KB	0.31

Table 2: Compression of STEP trace files

2.2.4 Compressing STEP Data

As mentioned in section 2.2.1, the STEP encoding is quite efficient, using roughly one byte per data field on average. Nevertheless, it is still easy to produce STEP trace files consisting of several hundred megabytes for a single program run. The initial STEP design allowed for a specialized compression layer to further reduce trace sizes. However, during development we were pleased to discover that our trace files responded remarkably well to standard compression methods. Table 2 shows that the STEP format is highly compressible by the `gzip`[5] and `bzip2`[2] compression tools, which are available on most GNU/Linux systems. For example, a trace from `jess` was reduced to only 0.18% of its original size using `bzip2`[1] thus effectively increasing the encoding efficiency to roughly 0.002 bytes/field. We suspect that the use of identifier fields are the key to achieving such impressive compression ratios, since they introduce a high degree of regularity to the data.

In light of these results, we will likely to continue to use standard compression tools to reduce STEP traces, though we continue to investigate the potential for a specialized compression layer. One particularly promising technique is the SEQUITUR[9] hierarchical compression algorithm, which has been shown to be effective for compressing address traces[4].

2.3 The STEP Definition Language

The STEP Definition Language (STEP-DL) simplifies the process of developing new STEP record types. Using STEP-DL and its associated `stepc` compiler, profiler-developers can easily generate the code to represent a record and its translator.

STEP-DL is conceptually similar to the Meta-TF language but, differs in several significant respects. The most obvious difference is the syntax. STEP-DL borrows much of its syntax from the DAFT language[6], a specification language for defining extensible scientific data file formats. We chose to adopt the DAFT syntax, as opposed to that of Meta-TF, since it provides several important notions, namely typing and inheritance. Typing is used to build records in STEP-DL from the set of basic fields, described in section 2.2. The inheritance mechanism, adapted from DAFT, allows record definitions to be extended to support new fields as needed. Inheritance is an important part of STEP-DL, since it is only natural that a development framework such as STOOP should provide features that promote reuse.

In figure 2, we present a simple STEP-DL definition. Here, we define a record for recording object field accesses. The record has three fields, one to identify the type of object whose field is being accessed, one to identify the particular field being accessed and one to indicate whether the access is a read or a write. This example illustrates several important features of STEP-DL. First, STEP-DL provides the notion of packages to allow namespace partitioning. In this example, the `FieldAccess` record is contained within the `example` scope. Another important notion is that of attributes. As with Meta-TF and DAFT, STEP-DL provides the ability to associate attributes with a particular field or record. Attributes indicate various encoding details to be considered by a record's translator. In this particular example, a "width" attribute indicates that only a single byte should be used to encode the `readWrite` field. Another common use of attributes is to indicate a default value for a particular field. The example also illustrates the use of STEP-DL's inheritance mechanism. Here, a `FieldAccess` record is extended to form a `ThreadFieldAccess` record that adds a new field to indicate the thread in which the access occurred.

[1] `bzip2` uses a Burrows-Wheeler[1] block sorting compression algorithm.

```
package example
{
  record FieldAccess
  {
    declaringType, fieldName : identifier;
    readWrite : int <width = 1>;
      // use a single byte for a boolean
  }

  // an extended record that includes the
  // thread that performed the field access
  record ThreadFieldAccess : FieldAccess
  {
    thread : identifier;
  }
}
```

Figure 2: A STEP-DL Definition

```
public class FieldAccess implements StepRecord {
  public final IdField  declaringType;
  public final IdField  fieldName;
  public final IntField readWrite;

  public FieldAccess(IdField declaringType, IdField fieldName, IntField readWrite) {
    // initialize field data...
  }

  private static class Translator extends RecordTranslator {
    public byte[] makeBytes(Object target) {
      // transform a FieldAccess record into bytes...
    }

    public Object readFrom(InputStream is) {
      // read a FieldAccess from the InputStream of bytes...
    }
  }

  public Translator newTranslator(StepSession session) {
    return new Translator( ... );
  }
}
```

Figure 3: A Generated STEP Record

The STEP-DL definition of a record is passed to the `stepc` compiler to generate code that can be integrated with existing STOOP-based profilers. Figure 3 shows an example of Java code generated from the STEP-DL specification in figure 2. The generated `FieldAccess` class is a simple aggregation of the basic identifier and integer field types. The example also shows the interface to the record's translator. In section 3, we discuss how the generated STEP records can be integrated with a profiling agent.

The examples above have illustrated how STEP-DL can be used to generate STEP records. We now review the main features of STEP-DL:

- Records defined with STEP-DL are composed of a set of fields. Each field may be either a singular or array type. Initially, the records are defined in terms of the basic field types: integer, string, data and identifier. Once a record is defined, it may be used as a field type for subsequent records.
- STEP and STEP-DLsupport the notion of sessions as a means for sharing context and parameters among several record types. Sessions may be used to encapsulate specialized translation processes, such as high-level compression schemes. Sessions may also be used to limit the lifetime of records within a trace, allowing identifier values to be reused. A session may contain parameter values that

are shared among the translators of records defined within the session. The STEP framework provides a default global session that is used to apply translator changes to all records.

- All STEP data types have an associated list of attributes. STEP-DL separates the attributes into a distinct syntactic construct that can be attached to any field, record or parameter definition. Certain attributes, such as the width of an integer field or the encoding of a string field[2] are built-in. Users are free to add their own attributes to any value.

- STEP-DL promotes reuse by providing an object-oriented style single inheritance mechanism. Both records and sessions may be inherited from existing STEP-DL definitions. A derived record inherits fields from its parent, whereas a derived session inherits parameters. An important feature of STEP-DL inheritance is the ability to override the attribute values of elements inherited from the parent record or session.

- STEP-DL provides a namespace mechanism to avoid name clashes between various record and session definitions. This name partitioning is particularly useful when adapting a profiler to accept inputs from several profiling agents. In this case, the data may be similar but not entirely compatible. For example, an allocation record from a custom instrumentation may contain information not available to a JVMPI agent; however, both are still an "allocation" and should be named as such.

- STEP-DL definitions may be compiled to either C or Java code to be integrated with a STOOP-based profiler. Java is the recommended output format, since STEP-DL features map naturally onto an object-oriented language.

3 Profiling Agents

Profiling agents provide raw data to the rest of the STOOP framework. Data may originate from a wide variety of sources. For example, traces of method invocations or object instantiations might come from a agent communicating with a virtual machine via the Java Virtual Machine Profiling Interface (JVMPI), maps of object addresses in memory can be produced by an instrumented garbage collector, and arbitrary application-specific data might come from custom instrumentation compiled into the profiled program.

To avoid restricting the set of possible data sources, STOOP imposes minimal requirements on profiling agents. Any information source – including existing trace data and sources as yet unforeseen – may feed a STOOP-based profiling agent, so long as the data can be converted to the STEP data record format. STOOP facilitates this process by providing a specification language, STEP-DL, and a compiler, `stepc` to generate the necessary interface elements.

We have created several profiling agents for use with the STOOP framework. To date they include:

- a JVMPI agent which reports class loading and method entry/exit;

- an instrumented copy of the Kaffe Virtual Machine, which reports field accesses as well as class loading and method entry/exit;

- instrumented Java class files produced from existing programs using SOOT, a framework for analyzing and transforming Java bytecode. The transformed classes report their own field accesses as well as method entries and exits.

In figure 4 we show a simplified version of the profiling agent used to collect trace events from bytecode instrumented with the SOOT tool. After every field access, a message is passed to the singleton `Collector` object, which in turn writes a record to the `StepOutputStream` (the `_stepOut` field in the example). Record objects are created using the generated interface shown in figure 3. The example shows how a `FieldAccess` record is created by passing new instantiations for the identifier (`IdField`) and integer (`IntField`) fields.

[2]STEP strings are based on the Java String class, and thus also have the notion of an encoding such as UTF-8 or US-ASCII.

```
/** A simple Java profile agent.
 *  The program being profiled indicates various events by invoking
 *  the corresponding method of the singleton Collector. Calls to the
 *  Collector may be inserted by hand or by a tool such as Soot.
 */
public class Collector {
  private static Collector _theInstance;
  private StepOutputStream _stepOut;

  private Collector(OutputStream os) {
    _stepOut = new StepOutputStream(os);

    Runtime.getRuntime().addShutdownHook(new Thread() {
      public void run() { _stepOut.close(); _stepOut.flush(); }
    });
  }

  public void recordFieldAccess(String declaringType, String fieldName, boolean read) {
    _stepOut.writeRecord(
      new FieldAccess(
        new IdField(declaringType),
        new IdField(fieldName),
        new IntField((read) ? 1 : 0)));
  }

  public void recordMethodEnter(String method) {
    _stepOut.writeRecord(new MethodEnter(new IdField(method)));
  }

  // other event handlers...

  // access the singleton Collector
  public static Collector v() { return _theInstance; }

  public static void main(String[] args) {
    _theInstance = new Collector(new FileOutputStream(args[0]));

    // run code to be profiled ...
  }
}
```

Figure 4: A simple Java profiling agent

Once the record is passed to the STEP writer, the writer invokes the translator of the `FieldAccess` object to generate a sequence of bytes to write to the STEP file. The process is similar for other record types.

We have also developed profiling agents using the JVMPI and an instrumented virtual machine. In both cases, the agent is similar to the one described in figure 4. Events are converted into STEP records and dispatched to a STEP writer.

The flexibility of the STOOP framework allows a profiler developer to use various profile agents interchangeably with the rest of their profile system. The choice of profile agent is thus reduced to determining which agent has the most direct access to the desired data.

4 Visualizers

As well as allowing multiple profiling agents to produce events, STOOP also allows multiple visualizers to consume events. We present two visualizer agents which plug into the back end of the STOOP event pipe:

EVolve: an extensible visualization environment which facilitates the incorporation of new visualization techniques. EVolve was primarily designed to visualize dynamic information and is not targeted to any particular visualization domain.

JIMPLEX: a specialized code visualization framework targeted to facilitate optimization of code at the intermediate representation (IR) level. JIMPLEX represents static information from SOOT and dynamic information from STOOP in a common XML format and displays the data using a configurable HTML interface.

These visualizers have a very different development history: EVolve was designed as part of the STOOP project, whereas JIMPLEX was originally designed only to visualize static information from SOOT. Dynamic information was added to JIMPLEX as an afterthought. Incorporating dynamic STOOP data into JIMPLEX took about a day of programming effort demonstrating STOOP's ease of use.

4.1 EVolve

So far we have illustrated how profile information can be collected with the STOOP framework. In this section we describe EVolve, an extensible visualization toolkit that we developed concurrently with STOOP.

It is our experience that most existing visualization systems lack the ability to add new visualizations easily. Although this lack of extensibility does not greatly affect typical users, the inconvenience is obvious for researchers. Few tools exist to visualize run-time aspects of object-oriented systems such as explored in the hot fields example. To provide extensibility, we developed EVolve, a visualization environment which simplifies the creation and incorporation of new visualizations. Although EVolve has been developed in conjunction with STOOP, the modularity of both frameworks allows the EVolve architecture to remain completely independent.

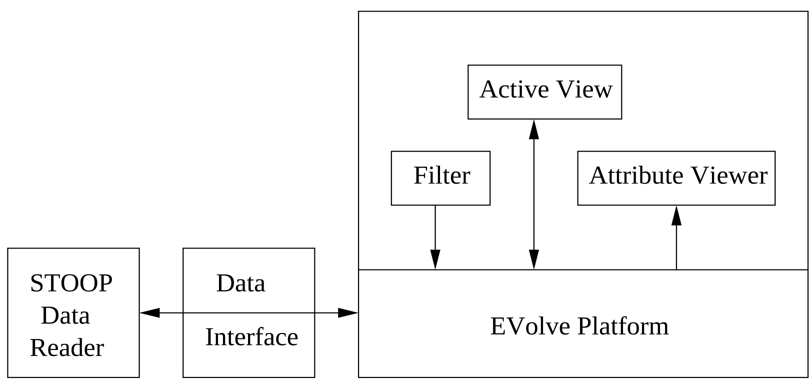

Figure 5: The EVolve Framework

Figure 5 shows the structure of EVolve. An EVolve visualizer is built by combining EVolve components. EVolve provides built-in components, and new components can be created simply through inheritance. Components are classified into three types: *active views*, *filters*, and *attribute viewers*.

The kernel of the EVolve platform takes charge of organizing the components and handling the communication among them. The kernel also provides the connection between the components and the profiling data. In EVolve, every component has its own data interface. Therefore, the user has to provide a data source which implements the data interfaces of all the different EVolve components. For example, to visualize the `FieldAccess` records described in sections 2 and 3 we augment the trace data with the notion of an abstract clock. In this case, one field access represents one time unit.

Among the three types of components, the *active views* display the actual visualizations. Active views also interact with users who highlight or filter selections and thereby influence the selection shown in other active views. EVolve thus enables highly interactive exploration of trace event data. This aids both program understanding and program optimization.

The second type of component, *filters*, work as preprocessors, allowing users to set filtering and coloring criteria before actually reading the data. This is a necessary first step for active view visualization that can reduce the time to build and refresh active views, since doing so requires reading the whole data set. Filtering ensures only useful data will be sent to the active views.

The third type of component, *attribute viewers*, visualize supplemental information that can be obtained without reading the whole data set. For example, the source code associated with a particular class or method. An attribute viewer visualizes information that cannot itself be used as a selector for other active views. Building and refreshing an attribute viewer is also simpler and faster than rebuilding an active view.

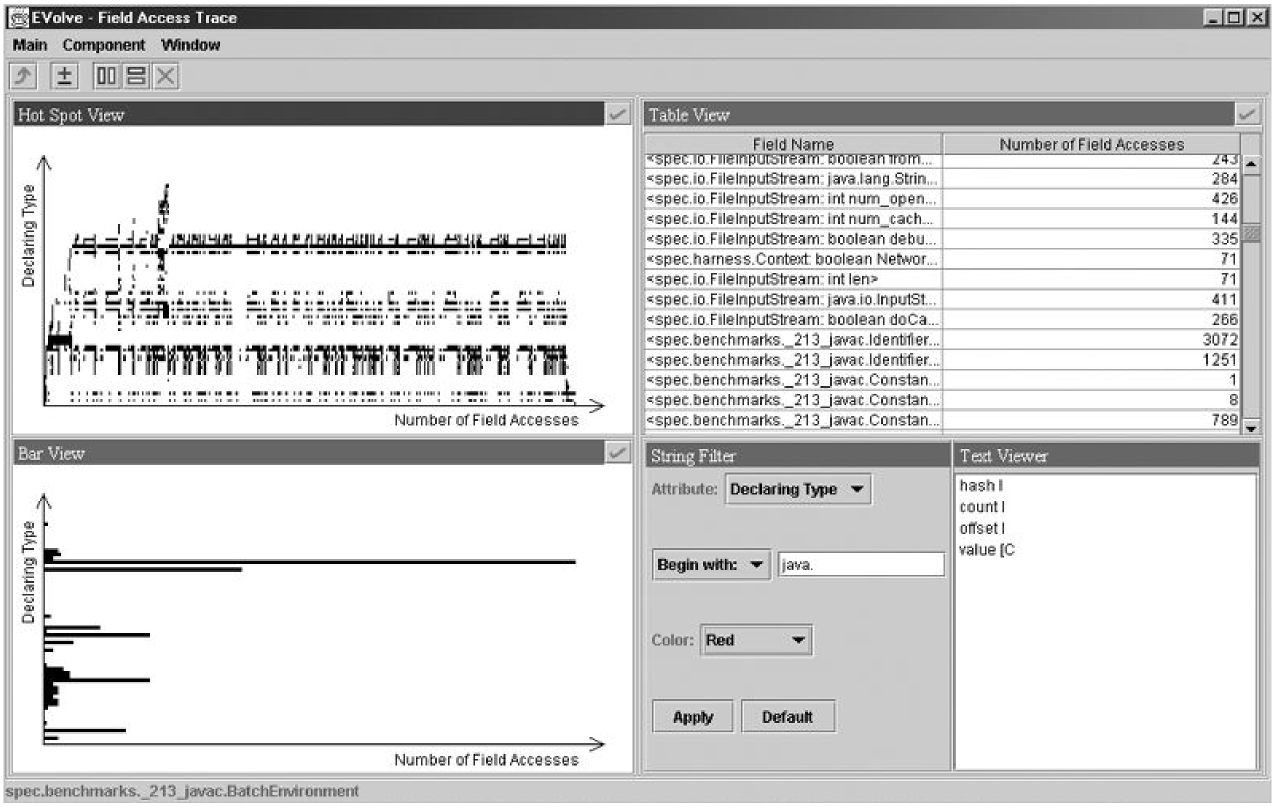

Figure 6: Example of EVolve

Figure 6 shows an example using EVolve to visualize field accesses of `javac`. The data is obtained using the profiling agent described in section 3. Five components are used in this example: three active views (Hot Spot View, Bar View, and Table View), one filter (String Filter) and one attribute viewer (text viewer).

In the Hot Spot View, the X-axis represents time (measured by number of field accesses since start of program) and the Y-axis represents the declaring type (taken from `FieldAccess` records described in figures 2 and 3) of the fields that were accessed during the execution. A point is painted black if the declaring type corresponds to one of the fields accessed within a given time sample, typically several thousands of field accesses per point. From the Hot Spot View, we can easily tell which fields are hot, and can also recognize different phases and patterns in the execution. Due to the large sample size, a Hot Spot View only gives an approximate impression of the total number of field accesses in the entire program run. The second active view, the Bar View, shows this quantity precisely (the Y-axis is identical to that of the Hot Spot View). The third active view, a Table View, shows the same information in table form.

The three active views allow the user to make a selection (lines in the hot spot and bar view or rows in the table view) and this selection is applied to other views for filtering or coloring. This allows a user to select a subset of the information by pointing and clicking in any active view. The String Filter provides similar functionality by allowing a user to define a selection as a regular expression over the field names.

Finally, the Text Viewer shows a list of all the fields in a selected class. Since the Text Viewer is an attribute viewer, it cannot itself be used to make new selections.

In combination, the five views provide the user with more information and navigation facilities than each view would provide in isolation. For example, selecting an often accessed field in the Bar View will highlight the actual access pattern in the Hot Spot View, and the field names involved in the Bar view. By selecting similar field names in the String Filter, the Hot Spot View will show whether these fields also have similar access patterns.

Since EVolve relies on STOOP for data collection, the extensibility and flexibility of EVolve's visualization techniques is coupled to a wide variety of data sources. Initially, EVolve read its own trace format, produced ad-hoc using an instrumented Kaffe Virtual Machine. It took about half an hour to re-specify the trace format in STEP. From then on, EVolve was capable of visualizing data generated using any STOOP-compatible profiling agent.

4.2 JIMPLEX

JIMPLEX is a code visualization framework targeted at compiler developers who work with an optimizable intermediate representation (IR). Since optimizations can be applied directly to this IR, the framework is able to support a variety of visualization tools which facilitate the design of optimizing transformations. In addition to displaying the textual code, JIMPLEX explores control flow information and other analysis results which are normally hidden in the IR. JIMPLEX was developed independently of STOOP but, we show that it was a simple process to update the visualizer to use STOOP to obtain run-time profile information.

As a visualization framework, JIMPLEX is composed of two components: an extensible representation of the IR and an interface to this data. Using the SOOT optimization framework, Java bytecode can be converted into a typed 3-address intermediate representation, called Jimple, which is a more suitable target for optimizing transformations. This IR is then augmented with the results of program analyses performed by SOOT. The resulting extended IR is stored as XML, making it portable and modular, while preserving the extensible nature of the data. XML also facilitates the creation of custom visualization interfaces which are independent of the actual data.

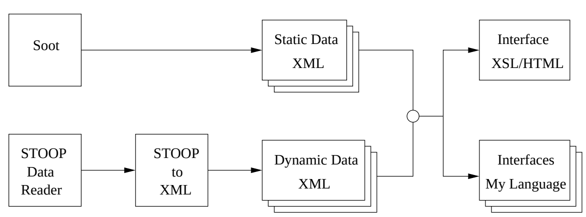

Figure 7: The JIMPLEX Framework

The extensible IR generated by SOOT contains program attributes and control flow information produced by static compile-time analyses. Although these elements provide a good basis for program understanding, many aspects of a program are not exposed until run-time. For example, the effectiveness of optimizations can be evaluated more accurately given actual profiling results, and clues towards new optimizations become more apparent when visualizing this dynamic information. JIMPLEX takes advantage of the open framework provided by STOOP to obtain dynamic profiling information. The addition of run-time data was a logical step forward for JIMPLEX, facilitated by the extensible nature of the STOOP and XML formats.

Using STOOP to generate profiling traces, JIMPLEX converts those traces into the same XML format used to encapsulate the static compile-time data. The open framework of STOOP allows the profiling data to be read and converted to XML by a simple reader, acting as the consumer at the end of the STOOP event pipe. Since the trace events are sent in a high-level, platform independent format, the JIMPLEX trace reader has full control over which trace elements will be included in the XML, and in what format. For example, as the reader receives STEP events from the pipeline, it can calculate temporal data to be integrated into the resulting XML and visualization interface. The translation of STEP data into the JIMPLEX XML format is simply a rendering of a particular interpretation of trace data into another format which is equally extensible and portable. Currently, we are considering extensions to STEP that would provide the XML data directly.

The static program attributes generated by SOOT and the dynamic trace data generated by STOOP are completely independent XML sources. By keeping the data separated, the user can swap in trace results from different profiling runs involving different attributes. Since everything is stored in a common XML format, the static and dynamic data collected for a program can easily be merged on the fly. JIMPLEX does not restrict the design of interfaces. The encapsulation of both static and dynamic aspects of a program in the XML data allows the creation of any number of interfaces. In this case, the extended IR can exist as XML on one node of the network, the interface can exist as XSL on another, and the actual visualization in HTML can take place in any web browser across the network.

Figure 8 shows an example interface. Static and dynamic information is merged: for example, the Methods view shows static information such as method declaration, name and number of statements, and dynamic information such as the number of times each method is called in the visualized program run. Similarly, the Fields view shows static type and modifier declarations, and dynamic counts of reads and writes for each field. The Statements view shows Jimple statements, and such static analysis information as the live local variables flowing into and out of a selected statement.

By combining dynamic run-time data with static program attributes, the interface can present a more comprehensive solution for compiler developers.

5 Conclusions and Related Work

In this paper we have given an overview of the STOOP profiling toolkit. The main goal of the toolkit is to provide a profiler-developer with support for rapid development of custom profilers for object-oriented languages. The toolkit has been designed to support many different profiling agents and visualizers which communicate through an event pipe.

The event pipe is at the core of our approach. Profiling agents write to the event pipe, whereas visualizers read from it. In order to make the event pipe very easy to use we have defined a language, STEP-DL, in which profiler-developers define their own profile events. A compiler, `stepc`, then generates the appropriate interface to the event pipe. STEP-DL specifications can be reused as it is simple to add new record types or to extend existing ones using inheritance. The event pipe architecture provides a high-level interface to the profiler-designer, while at the same time providing a compact and easily compressible trace format. Our initial experiments show that STOOP traces require about one byte per field and that the overall traces are very compressible with off-the-shelf tools like `gzip` and `bzip2`.

Once the events have been defined and the event pipe interface generated with `stepc`, the profiler-developer can hook a profiling agent to the front-end and a visualizer to the back-end of the event pipe. We have experimented with a variety of profiling agents and include one example in this paper. To demonstrate STOOP back-ends we showed how two visualization tools can be used with our framework. The EVolve framework was developed at the same time as STOOP and focuses on visualizing dynamic traces. The JIMPLEX tool was developed completely independently of STOOP and was originally used to annotate a compiler IR (Jimple) with data flow information.

Our experience has shown that separating the tasks of profile collection, transport, storage and visualization into a framework of extensible, independent components greatly simplifies the process of developing new profiling techniques. For example, we were able to quickly develop a profile agent and visualizer to examine hot field data from Java programs. A few simple extensions to the JIMPLEX visualizer allowed it to read the same profile data and present it in a completely new fashion.

Our approach differs from commercial profilers such as OptimizeIt[10], JProbe[8] and Jinsight[7] in that those systems are fixed profiling tools where the profiler presents predetermined program properties. Our tool allows one to develop profilers to view both standard and non-standard program properties, and to view events that come from a variety of sources.

Perhaps closer to our approach is Bloom[11, 12], a profiling system which allows the user to interactively build a queries which may be displayed using a wide range of supported data views. However, even in this system the program traces are predefined and there is no specification language for defining new trace events. Nonetheless, Bloom's query-based visualization techniques may prove useful in visualizing novel profile data

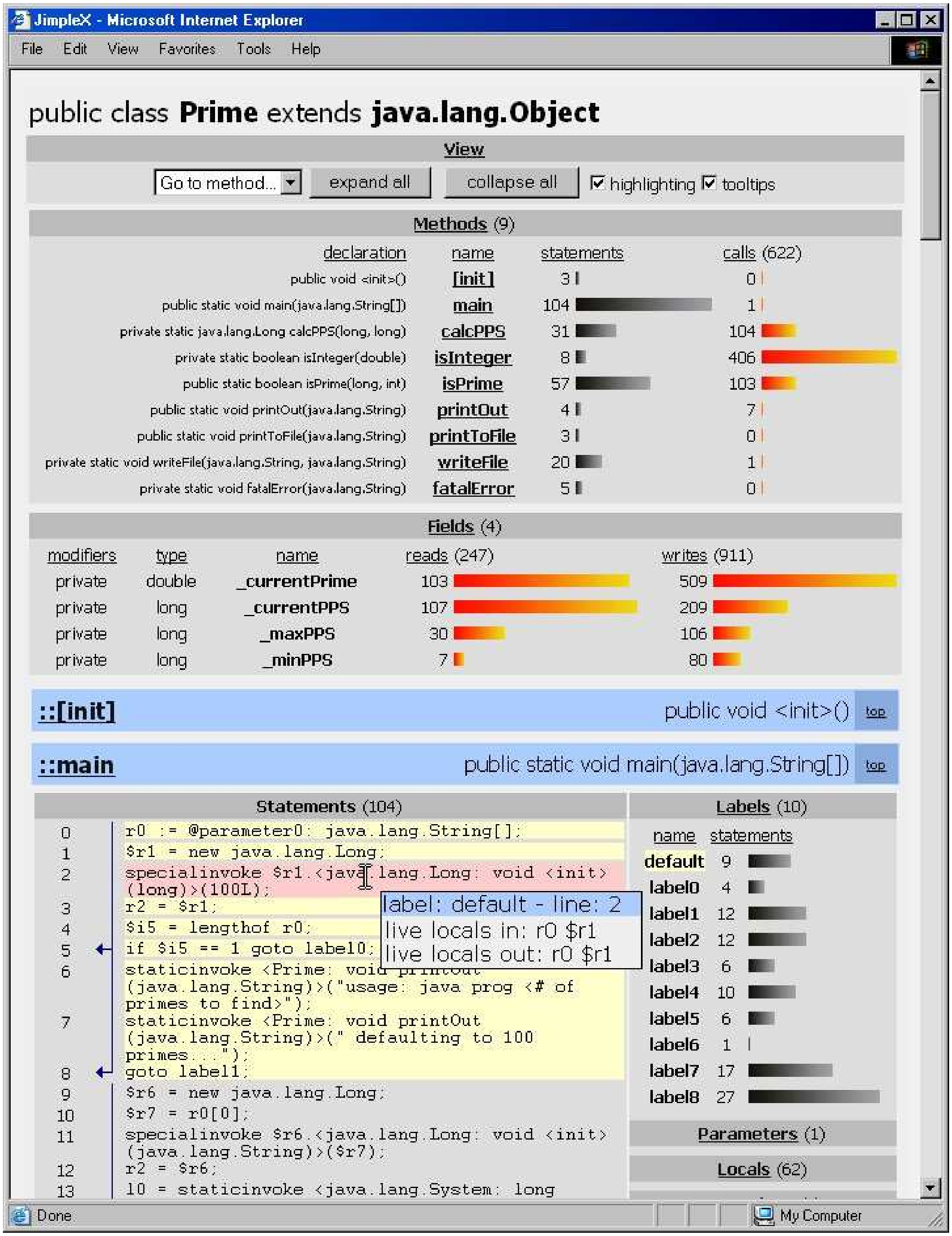

Figure 8: Example JIMPLEX Interface

with Stoop.

References

[1] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital SRC Research, May 1994.

[2] bzip2. http://sourceware.cygnus.com/bzip2/.

[3] T. Chilimbi, R. Jones, and B. Zorn. Designing a trace format for heap allocation events. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM 2000)*, pages 35–49, Minneapolis, MN, USA, October 2000.

[4] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2001)*, pages 191–202, Snowbird, Utah, USA, 2001.

[5] gzip (GNU zip). http://www.gzip.org/.

[6] M. Haines, P. Mehrotra, and J. V. Rosendale. Smart files: An oo approach to data file interoperability. In *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1995)*, pages 453–466, Austin, TX, USA, 1995.

[7] Jinsight. http://www.research.ibm.com/jinsight/.

[8] JProbe. http://www.sitraka.com/software/jprobe/.

[9] C. G. Nevill-Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. In *Proceedings of the IEEE Data Compression Conference (DCC 1997)*, pages 3–11, 1997.

[10] OptimizeIt. http://www.vmgear.com/.

[11] S. P. Reiss. An overview of Bloom. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Anaylsis for Software Tools and Engineering*, pages 71–77, Snowbird, Utah, USA, June 2001.

[12] S. P. Reiss and M. Renieris. Generating Java trace data. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 71–77, San Francisco, CA, USA, June 2000.

[13] Soot – a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[14] Transvirtual. The Kaffe Java Virtual Machine. http://www.kaffe.org.

[15] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In D. A. Watt, editor, *Compiler Construction, 9th International Conference*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, Berlin, Germany, March 2000. Springer.