STEP: A FRAMEWORK FOR THE EFFICIENT ENCODING OF GENERAL TRACE DATA

by

Rhodes Hart Fraser Brown

School of Computer Science
McGill University, Montréal

December 2002

a thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

Abstract

Program tracing is a common technique employed by software and hardware developers who are interested in characterizing the dynamic behavior of complex software systems. However, despite the popularity of trace-driven analyses, there are surprisingly few options for encoding trace data in a standard format.

In the past, many developers have resorted to creating their own ad-hoc trace encoding solutions, tailored specifically to the data they are considering. Such efforts are usually redundant, and in many cases lead to an obscure and poorly documented trace format which ultimately limits the reuse and sharing of potentially valuable information.

The STEP system was created to address this problem by providing a standard method for encoding general program trace data in a flexible and compact format. The system consists of a trace data definition language along with a compiler for the language and an encoding architecture that implements a number of common trace compaction techniques. The system simplifies the development and interoperability of trace clients by encapsulating the encoding process and presenting the data as an abstract object stream.

This thesis presents a detailed description of the STEP system and evaluates its utility by applying it to a variety of trace data from Java programs. Initial results indicate that compressed STEP encodings are often substantially more compact than similarly compressed naïve formats.

Résumé

Le traçage des programmes est une technique couramment employée par les développeurs de logiciels et matériel informatique intéressés à caractériser le comportement dynamique de systèmes logiciels complexes. Par contre, en dépit de la popularité des analyses effectuées à l'aide de traces, il existe étonnament peu d'options pour l'encodage des traces utilisant un format standardisé.

Dans le passé, plusieurs développeurs ont eu recours à la création de leur propre solutions spécifiques quant à l'encodage de traces, conçues sur mesure pour pour les données examinées. De tels efforts sont habituellement rendondants, et mènent dans plusieurs cas à un format obscur et piètrement documenté, ce qui, ultimement, limite la réutilisation et le partage d'information possiblement pertinente.

Le système STEP à été créé dans le but d'aborder ces problèmes en fournissant une méthode standardisée pour l'encodage de données générales provenant de traçage de programme en un format flexible et compact. Le système est consitué d'un langage de définition des données de traces, ainsi que d'un compilateur pour ce langage et d'une architecture d'encodage qui implémente plusieurs techniques de compaction de traces répandues. Le système simplifie le développement and l'interopérabilité des clients de traces en encapsulant le processus d'encodage et en présentant les données en un train d'objects abstrait.

Cette thèse présente une description détaillée du système STEP et évalue son utilité en l'appliquant à une variété de traces provenant de programmes Java. Les résultats préliminaires indiquent que les données encodées et compressées par STEP sont souvent substantiellement plus compactes que des formats similaires utilisant une méthode de compression naïve.

Acknowledgments

Constructing Step and this thesis was a personal struggle for me, one that would not have been completed without a great deal of assistance and support. I owe a great deal to my supervisor, Laurie Hendren. Her patience and confidence in me has been truly amazing and I will be indebted to her for a long time because of it. I would like to extend my thanks to all of the members McGill's Sable Research Group who were always full of good ideas and good cheer. Individual thanks are due to a number of individuals who helped with the development of Step: Qin Wang and David Eng, whose visualization projects were the driving force behind the need for a general, compact trace format; Karel Driesen, for his continued focus on the elements that were truly important; John Jorgensen, for his pragmatic viewpoint and help with some of the early trace formats; Clark Verbrugge, for his keen eye and focus on the semantics of Step-DL; and Bruno Dufour, for his indispensable JVMPI profiling agent which was used to collect much of the data presented here. My tenure as a graduate student at McGill, and thus the development of Step, was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC). Finally, a special thanks to my family and friends who were nothing but supportive while I endeavored to complete this thesis.

Rhodes H. F. Brown

McGill University,
December 2002

Dedicated to my parents, all 4 of them.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Efficiency is a quality pursued by virtually all software, hardware, and compiler developers. In many cases, an exact definition of the concept depends on what developers consider relevant. Some use raw computing speed as a measure, while others use a broader definition of resource overhead. Some even choose to frame their definition in terms of more abstract notions such as interoperability and maintainability. Yet all such definitions of efficiency have one element in common: they are a measure of some dynamic quality of software systems.

Historically, attempts to characterize the dynamics of software systems have focused their attention on analyses of low-level program events. The analyses are often based on a simulated recreation of a program's execution using a *trace* of the relevant event data. Trace-driven analyses are generally favored over simple statistical sampling methods since the goal is often to characterize sequential patterns in the data rather than just summarize the occurrences of events.

Some trace analyses can be performed on-line—streaming the data directly from a running program to the analysis routines—but, in many cases, it is preferable to record the data for use in several off-line analyses. Recording traces is problematic though, since they often contain huge amounts of data. A variety of lossy and lossless methods for reducing the size of traces have been proposed, but the research has focused almost entirely on simple, restricted forms of data such as the target address of load and store operations. A handful of “standard” trace file formats have been suggested [SCM+95, HKWZ00, JHBZ01] but most are wholly insufficient for capturing the range of events that occur in modern software systems. The lack of adequate trace encoding systems has led many developers to create their own ad-hoc solutions which are frequently tailored specifically to the data they are considering. This approach often results in a trace format with little or no documentation and trace data that goes unpublished, consequently limiting the sharing and reuse of potentially valuable information. Such efforts are also usually redundant since, as this thesis demonstrates, a general approach to trace encoding is possible.

STEP[1] is a system created to address the need for a standard method of encoding general program trace data in a flexible and compact format. The system provides a new trace data definition language, a compiler for the language, run-time support for the annotation features of the language, and an encoding engine that implements a number of common trace compaction techniques. Together, the features of the system simplify the development and interoperability of trace clients by encapsulating the encoding process and presenting trace data as an abstract object stream.

The development of STEP was motivated by a desire to collect a variety of trace data from Java programs and provide an interface to the data that is compatible with a variety of analysis tools. A review of the related literature did not reveal any solutions that were capable of capturing the full range of events of interest that occur during the execution of a Java program. However one approach, the MetaTF system [CJZ00], did offer a useful starting point by suggesting that traces be defined with a specialized definition language. Definitions specified with such a formal language can serve two purposes, namely acting as an explicit document of the trace format and providing a means for automatically generating routines to read and write the trace data files.

Early attempts to build on MetaTF's general approach to trace definition and encoding demonstrated that a better solution could be engineered by focusing on support for three main features: a richer set of event record types, integration with other tracing tools, and a more robust and extensible encoding architecture. STEP is an attempt to meet these requirements by providing a new data definition language,

[1] The name STEP derives from its original incarnation as the STOOP Trace Event Protocol [BDE+01]. A number of other equally appropriate expansions have been suggested.

Step-DL; a compiler for the language, `stepc`; and an object-oriented encoding architecture.

This thesis provides a detailed description of the Step system, elaborating on the design rationale and highlighting a number of the issues encountered during the development of the system. The utility of the system is evaluated by examining its application to the encoding of a variety of trace data collected from a set of well-known Java benchmark programs. The results are promising, indicating that compressed Step trace files are often substantially more compact than similarly compressed naïve formats.

1.1 Motivation

To understand the design and implementation of Step, it is best to start with a discussion of the role the system was intended to play. In many ways, such a discussion helps to frame the boundaries of the project and reveals influences that are echoed in a number of the design and implementation choices.

Step was conceived to act as the core of a system for developing new methods of improving the performance of Java programs based on analyses of both program and run-time environment dynamics. This is not a trivial aspiration considering the multitude of ways that trace information from Java programs can be exploited:

- The run-time overhead of Java's automatic memory management—garbage collection (GC)—is a common concern among developers interested in building fast or precisely-timed applications. Modern GC implementations, such as variants on the generational approach, can be effective for a range of programs but are still based on broad, statistical assumptions and, thus, are not truly adaptive solutions. Recognizing this deficiency, a variety of approaches have been developed that use trace data to either guide the selection of a particular GC algorithm [FT00], or target particular object types for special treatment [Har00].

- Program performance can often be significantly affected by the physical layout of object data values. Highly composite objects incur the cost of frequent dereferencing steps, while the spatial and temporal locality of object references can affect the performance of hardware data caches. Field access traces can be used in a range of type restructuring optimizations: in-lining high-use sub-field definitions [LH02], and re-ordering field positions, possibly segregating low-use fields into separate objects [RBC02].

- A number of Java program transformations such as devirtualization [SHR+00], array bounds check removal [QHV02], and synchronization removal [Ruf00] have been demonstrated as effective optimizations. However, as proposed, these transformations are based on static approximations of a program's call graph—approximations that are often costly to generate. Trace data could be used to accelerate such analyses by highlighting call sites that exhibit definite polymorphism as well as those that are apparently monomorphic.

- Trace data can be used to build statistical models of a program's control flow. The information can be used to estimate branching behavior [WL94] which is, in turn, useful for performing condition or loop inversions that cooperate with local caching and branch prediction hardware. More generally, the statistics can be used for so-called probabilistic data flow analyses [MS00] to estimate the cost/benefit ratio of applying a given transformation to a particular code segment.

- Trace data can be used to quantify various benchmark program dynamics such as polymorphism, concurrency and memory usage [DDHV02]. Such measurements are of clear importance to developers of compilers and run-time systems, in that they provide a basis for meaningful evaluation and comparison of approaches.

This list is, by no means, exhaustive but it does exhibit the diversity of potential Java trace data as well as the diversity of potential trace-driven analysis clients.

The primary assertion behind the development of STEP is that there is value in recording the data alluded to above in a single, universal trace format. There are two advantages to such an approach. First, the existence of a universal (and compact) trace format encourages developers and researchers to publish the data they collect. As is common in other scientific endeavors, such publicly available data can then be used as the basis for validation, extension, adaptation, and meaningful comparison of various analyses and applications. The second advantage is that recording a broad range of data in a single trace file allows results from several independent analyses to be meaningfully collated and related to one another.

1.2 Requirements

Having established that a system for encoding traces in a common format is desirable, a logical next step is to derive a series of requirements that such a system should meet. Based on the motivations outlined above, a number of criteria present themselves:

Expressiveness:
A complete range of event data structures should be supported: essentially, any serializable form composable from common basic types.

Flexibility:
The system should provide a flexible trace format that is not bound to any particular set of data records. Specifically, augmenting a trace with new records should not affect compatibility with existing tools.

Compactness:
Standard methods for trace compaction should be integral to the system. However, the default encoding should be lossless, deferring lossy reductions to a post-processing filter.

Efficiency:
Given that traces often comprise a tremendous amount of data, the serialization process, including compaction, should operate in linear (or near-linear) time in the size of the input data.

Documentation:

To ensure proper use of the data, the trace files should be accompanied by a descriptive document that specifies both the form and interpretation of the data records. The document should provide sufficient detail to create a tool capable of reading the encoded trace and reconstructing the event data.

Encapsulation:

To facilitate the interoperability of tracing and analysis tools, the system should provide a library of routines for reading/writing trace files that encapsulates the encoding process so as to separate the production and consumption of events from their serialized off-line representation.

Portability:

Both the encoding software and format should be platform independent so that traces collected on one architecture can be decoded and analyzed on any other architecture that the system supports.

Extensibility:

Anticipating the development of new trace-based analyses or compaction schemes, the system should provide an open architecture that allows the addition of new event types and/or encoding strategies with minimal modification.

In addition to these criteria, it also reasonable to request that the system provide a simple and intuitive client interface and that its design adhere to commonly accepted software design principles.

1.3 System Overview

STEP is an attempt to capture the motivations outlined in section 1.1 and to embody the requirements proposed in section 1.2. The system consists of a specialized trace definition language, STEP-DL, and an object-oriented (OO) framework, implemented in Java, that provides compiler and run-time support for STEP-DL definitions, as well as an encoding engine that includes a number of standard trace encoding strategies.

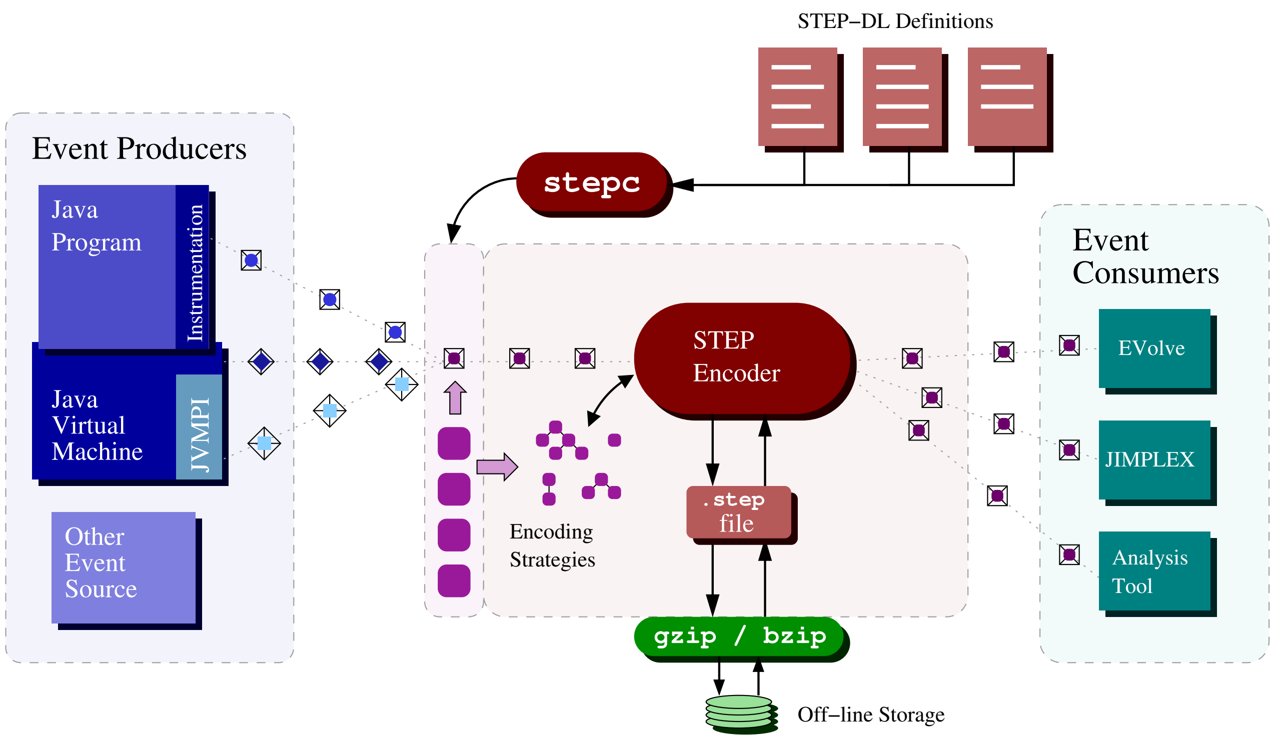


Figure 1.1: An overview of the STEP framework

The design of STEP synthesizes a number of ideas related to tracing, resulting in an entirely unique implementation approach.

STEP is often depicted as a mediator, connecting a variety of trace data sources to a variety of analysis and consumer tools. Clients interested in encoding data in the STEP format begin by defining the format of the data records with the STEP definition language, STEP-DL. The definitions supply both a structural description of the records as well as various annotations that can be used to automatically generate methods for manipulating the data; for example, to serialize it into the STEP binary format. As figure 1.1 indicates, the definitions are converted by the `stepc` compiler into equivalent Java class definitions that act as the interface layer that both producer and consumer clients interact with.

Some example trace data producers are indicated on the left side of figure 1.1. In the case of Java programs, trace data can arise from several sources. Standard data, such as method usage and allocation events can be collected through the Java Virtual Machine Profiler Interface (JVMPI). Information about the inner workings of a particular Java Virtual Machine (JVM)—such as information about a specific Just-In-Time (JIT) compilation strategy or GC algorithm—is often obtained by directly modifying the VM. Still more information, such as basic block or field uses, can be obtained by adding custom instrumentation to the target Java bytecode.

Trace producers use the Java object definitions generated from STEP-DL to convert the trace data from its original format into the STEP object format and pass the data on to the STEP encoding system for serialization. Individual records are encoded in a platform independent, binary format according to an adaptive set of strategies generated from annotations associated with their type. The resulting STEP trace file is usually passed to a standard compression tool such as `gzip` or `bzip2` to further compact the data for off-line storage. Traces are unpacked by the encoder, regenerating the original data stream in the common object format. A variety of analysis tools (depicted on the right of figure 1.1) can consume the data; in the figure, the EVolve [WWB+02] and JIMPLEX [Eng02] visualization tools are highlighted.

The STEP encoding engine is not specific to any particular value type. Instead the system uses annotations supplied in the STEP-DL record definitions—specifically, those referred to as `encoding` attributes—to generate a set of encapsulated algorithms (strategies) for encoding each record type. The strategies are constructed hierarchically, deferring to sub-strategies in order to encode each field value. The strategies employ a number of established techniques that exploit known characteristics in the data to adaptively encode individual values, achieving an amortized reduction in the number of bytes required to encode a given type.

The STEP definition language has a simple and intuitive syntax that supports a reasonably diverse set of record structures. The most significant contribution of the language is its attention to the need for structured annotation of trace record definitions. The basic concept is that, in addition to defining a record's structural composition, other supplemental information can be associated with the various elements of the record. Examples include descriptive labels, basic characteristics of the data, and information specific to a particular application (such as encoding).

The language supports two forms of extension: common structural inheritance and contextual refinement of attributes.

Conceptually, Step is quite straightforward. The system is a bridge, connecting trace producers and consumers through an interface generated from simple record definitions. However, as subsequent chapters highlight, there are a number of subtleties to implementing the system as described. Step-DL's extension mechanisms give rise to a number of interesting syntactic and semantic issues, while the encoding architecture must implement a sophisticated mechanism for handling values that deviate from an expected norm.

1.4 Contributions

The Step framework offers a number of contributions to the research community. Primarily, the system facilitates the development of new tracing tools by separating the process of encoding from those of collection and analysis. A consequence of the approach is that it provides a common trace format with explicit documentation of the trace contents in the form of Step-DL definitions. This, in turn, encourages the publication and reuse of traces, allowing research results to be scrutinized, evaluated and related to one another. Among the secondary contributions of the project are solutions for implementing a combination of structural and interpretive type extensions, and the creation of an open and flexible platform for the studying and testing various trace reduction and compaction techniques. It is also worth noting that, while Step is primarily intended as a trace encoding system, the system does not make any assumptions about the incoming data itself other than the regularity characteristic common to program traces. Thus, the system may provide a basis for encoding other forms of highly-regular, sequential data.

This thesis contributes a number of perspectives on the design and implementation of Step. Its primary purpose is to document the features of Step-DL and the functioning of the encoding architecture. It provides links to a broad spectrum of research related to tracing, and considers the applicability of a number of existing approaches to the task of tracing Java programs. The results presented in chapter 6 also provide a basis for evaluating the effectiveness of the approach.

1.5 Thesis Overview

The primary goal in presenting this thesis is to convey a factual description of STEP. A secondary goal, as in all theses, is to explain and formalize the intuitions behind particular design and implementation choices. The discussion begins in chapter 2 with a survey of related approaches and applications. The review is reasonably cursory and somewhat eclectic, serving as much to underscore the motivations presented in section 1.1 as it does to present existing approaches to trace encoding. Readers may find it useful to skim the chapter on a first reading, returning for a second pass after covering the main content chapters. The actual presentation of STEP begins in chapter 3 with an examination of STEP-DL. The chapter covers the main language features, details some of the semantic consequences, and presents a brief historical review of the language's evolution. Chapter 4 continues the discussion by describing the approach to compiling STEP-DL definitions and the internal, run-time representations used in both the `stepc` compiler and encoding architecture. The encoding process is then detailed in chapter 5. The complications introduced by adaptive encoding and type inheritance are touched upon. Following that, a categorization of various trace encoding strategies is presented. Experiences in using the system are presented in chapter 6 with a focus on the effectiveness of various encoding techniques and the compressibility of trace data from a range of Java benchmark programs. The discussion concludes with a summary of the presentation and comments on the utility of the approach. A final section suggests future directions for the development and use of the system.

Chapter 2
Related Work

The design of Step draws from a broad range of research related to program tracing. The discussion that follows begins with the most relevant influences, namely the current approaches to trace encoding. Section 2.2 provides a brief survey of the plethora of tracing applications. The survey is not intended as a comprehensive review but is instead meant to underscore the diversity of trace data sources and forms. The focus is on off-line tracing, since that is the area most relevant to Step. Readers interested in on-line approaches are directed, as a starting point, towards systems such as Morph [ZWG+97] or that of Anderson *et al.* [ABD+97], or for a Java specific approach to the Jikes RVM project (formerly Jalapeño [AAB+00]). Section 2.3 concludes the chapter with a short discussion of other data file definition languages.

2.1 Trace Collection and Storage

Step is a conceptual extension of the MetaTF system developed by Chilimbi, Jones and Zorn [CJZ00]. Their approach was developed during the process of trying to establish a common trace format for dynamic storage allocation (DSA) events. They proposed a format called the Heap Allocation Trace Format (HATF) as an instantiation of a meta-level trace specification, dubbed MetaTF. The original presentation of MetaTF defined a language for specifying trace formats as a collection of records, then used the language to define the records contained in the HATF format. In a subsequent report, Jones [Jon01] discusses some modifications to the MetaTF language, explains the reasons for switching from the original binary file format to a text-based approach, and details how the MetaTF specifications are compiled into Java components for reading and writing trace files.

While it appears that MetaTF is the only previous attempt to develop a trace definition language, a number of trace formats have been publicly defined. Humphries *et al.* proposed the POSSE format [HKWZ00] for recording the behavior of persistent object systems (POS). Scheuerl *et al.* suggest the MaStA I/O format [SCM+95] for recording database input and output operations. New Mexico State University's TraceBase provides address reference streams in the PDATS format [JH94, Joh99, JHBZ01]. As with Step, these trace encoding systems do not provide methods for actually collecting trace data. Instead interested users are directed towards program instrumentation systems such as `pixie` [Smi91], ATOM [SE94] and EEL [LS95]. Java trace data can be obtained by explicitly modifying a JVM, providing a JVMPI [Sun] collection agent, or instrumenting the target bytecode with a tool such as Soot [VRGH+00].

2.1.1 Compaction Methods

PDATS and MetaTF integrate methods for compacting trace data. The methods are generally derivatives of the *difference technique* developed by Samples [Sam89] for his Mache system. MetaTF provides methods for indicating that values should be recorded as the difference from a base offset or previous value (delta), or as following a regular stride pattern. PDATS focuses on address values, combining the difference technique with run-length encoding. Fox and Grün [FG96] propose a method called recovered program structure (RPS) that improves on the effectiveness of the PDATS approach by relating instruction address values to the structure of a program (i.e., the composition of its various instruction blocks). Elnozahy [Eln99] suggests an apparently similar approach, but defers details to a patent application.

Address traces seem particularly amenable to compaction and have attracted a variety of research efforts. An early study by Hammerstrom and Davidson [HD77] reveals that address streams often have very low entropy, an intuitive result in light of the commonly accepted 90-10 rule of computing behavior which suggests that programs often spend most of their effort in limited code and memory segments. Becker, Park and Farrens [BPF91] continue this work by investigating higher order entropy measures. Pleszkun [Ple94] builds on these studies to develop a lossless reduction scheme based on measuring the second-order entropy of address streams; essentially resulting in a path-based Huffman encoding of the sequence. Realizing that many applications of address traces tolerate the deletion of a portion of the statistically deviant values, Smith [Smi77] initiated the research into lossy address trace reduction methods. A number of other lossy approaches [AH90, PG95, KSW99] have followed since then.

Reiss and Renieris [RR01, RR00] suggest applying the SEQUITUR hierarchical inference algorithm [NMW97] to compact general trace data. The approach is further developed by Chilimbi [Chi01] and Larus [Lar99] who describe methods for deriving lossy reductions.

2.2 Applications of Tracing

Collecting event traces to study the dynamic behavior of programs is a technique that has been in use for many years. In fact, Smith [Smi82] points to uses that date back as far as 1966.

Probably the most prominent use of tracing is in the study of caching and paging architectures. Uhlig and Mudge [UM97] survey much of this work and also provide a good summary of the various approaches to address trace reduction. More recently, hardware developers have also used tracing in the development of branch prediction architectures [EPCP98, FFW98, Wu02].

Software interactions with caching and branch-prediction hardware are addressed in a variety of trace-based research. Chilimbi *et al.* look at several issues regarding the layout [CDL99] and positioning [CHL99] of data structures in memory. The concepts are used to develop trace-driven compiler optimizations [RBC02]. Ball and Larus [BL94] suggest efficient methods for tracing the execution path of programs.

Wu and Larus [WL94] subsequently use the traces to isolate various families of branching behavior and develop branch and loop optimizations that cooperate with local branch-prediction hardware. Applications of statistical execution models have recently evolved beyond simple optimizations to drive a number of compile-time transformations based on so-called probabilistic data flow analysis [MS00].

Traces have been used in a variety of work related to memory management. Several approaches have been developed based on object lifetime measurements. Seidl and Zorn [SZ98] investigate how allocator efficiency can be improved by segregating heap objects according to their lifetimes. Harris [Har00] identifies objects for pre-tenuring in generational garbage collected systems. Shaham, Kolodner and Sagiv [SKS01] use lifetime measurements to identify so-called dragged objects (those that outlast their utility). In more general work, Johnstone [Joh97] used traces to compare statistical assumptions about garbage collected systems with their actual behavior. Fitzgerald and Tarditi [FT00] used traces to select appropriate GC algorithms for particular programs.

Efforts to characterize program dynamics have used a variety of visualization techniques to reveal relations and patterns in execution and memory usage. Several tools for visualizing Java trace data are available, they include: IBM's Jinsight [IBM], Borland's Optimizeit^{TM} Suite [Bor], and Sitraka's JProbe^{TM} [Sit]. Many of the visualizations these tools support are derived from the ideas of Jerding, Stasko and Ball [JSB97]. Reiss and Renieris build on their work to develop several complex visualization systems [Rei98, RR99], culminating in a system called BLOOM [Rei01]. Shende *et al.* developed a similar system called TAU [SMC+98] for visualizing the execution of parallel, scientific applications. STEP was created to support two customizable visualization systems: EVolve [WWB+02] and JIMPLEX [Eng02].

Other miscellaneous applications of tracing abound. Informal tracing is often used to debug programs; Netzer and Weaver [NW94] present some formal approaches. Ernst *et al.* [ECGN99] describe a method that combines data and execution tracing to locate program characteristics that remain invariant across some computation window; they use the information to reveal various implicit design contracts. Colcombet and Fradet [CF00] describe a system that modifies code to ensure that it conforms to a particular family of traces (i.e. the runtime behavior is restricted). Dufour *et al.* [DDHV02] use traces to develop various metrics to formally categorize benchmark programs as memory intensive, numeric, polymorphic, etc.

2.3 Data Definition Languages

The approach of using a special definition language to define data file formats is definitely not new. A 1978 proposal by Norton [Nor78] uses a language to define the format of medical data files and suggests how the definitions can be used to automatically generate tools for manipulating the data. A recent extension of this approach was proposed by Haines, Mehrotra and Van Rosendale. Their SmartFile system [HMVR95] imported the object-oriented concept of inheritance to increase the interoperability and extensibility of scientific data files. Their DAta File Type (DAFT) definition language also used specialized annotations to indicate characteristics of the data such as units and coordinate systems.

Many fully general data definition systems exist, including international standards such as ASN.1 [ISO90] and SGML [ISO86], as well as the now ubiquitous XML [W3C00]. However, specialized languages such as MetaTF, DAFT and Step-DL are often preferred by developers for their concise, easily-read and application-specific syntax.

Chapter 3
The Step Definition Language

This chapter presents the Step Definition Language (Step-DL). The purpose of the language is to provide clients of the Step system with a simple and concise means for defining records to be encoded in the Step trace format. The definitions are translated (compiled) into structures that are used to represent trace data as objects and to convert a stream of such objects to and from the Step binary format. The capabilities of the language (i.e., the forms of data it is capable of expressing) provide the design foundation upon which the rest of the Step architecture is built.

A Step-DL record definition includes both a structural description of the type (its name, its fields, the ordering of the fields and their type, etc.) as well as annotations that indicate various supplemental information such as which encoding strategy to use for a particular field element. The definitions are processed by the `stepc` compiler to generate Java class definitions that represent the record types. The annotations associated with a record are used to generate related structures both during compilation of the definition and when the type is loaded at run-time. An example is the encoder objects that are used to encapsulate the strategies for serializing records into the Step trace format.

Step-DL is a simple language—its only purpose is type definition—however, its support for features such as inheritance and contextual refinement of attributes leads to a surprisingly complex semantics. The sections that follow describe the details of the language and the reasoning behind the particular choice of features and syntax. Some of the semantic implications of STEP-DL definitions are touched upon, deferring more discussion of the compile-time and run-time issues to chapters 4 and 5 respectively. The chapter concludes by presenting a brief history of the evolution of STEP-DL, highlighting some of the original influences that shaped the language and its implementation.

3.1 Goals

A collection of STEP-DL definitions provides a so-called document type definition (DTD) for a given trace. The definitions are used both as a basis for generating code to represent and manipulate the data, and also as a formal document that describes the structure and content of the data. To serve in both these capacities the language has been designed to support both a range of types and expressions about types (and their instances) while maintaining a syntax that is concise, precise and easy-to-read.

3.2 Approach

The design of STEP-DL seeks to improve upon that of the MetaTF language [CJZ00, Jon01] by providing a new syntax that offers increased generality, flexibility, and extensibility. Specifically, STEP-DL uses a more familiar type definition syntax than MetaTF and borrows from the DAFT language [HMVR95] to introduce type inheritance and a more intuitive annotation syntax. Some specific contrasts between MetaTF, DAFT, and an earlier version of STEP-DL are presented in section 3.5.

The fundamental concept in STEP-DL is the separation of type structure from interpretation. The language permits generalized annotations, in the form of *attributes*, that contain essentially arbitrary text data. To provide some form of structure to the attributes, the values are partitioned into *groups*, where the group defines the permissible values the attribute may contain. In this sense, the language is somewhat open-ended. Each attribute group defines a new sub-component of the language with its own independent syntax and interpretation. A number of attribute groups, such as the `encoding` group, are built in to the system but developers are free to add their own groups.

3.3 Language Features

An initial appraisal might place Step-DL as somewhere between MetaTF and a general mark-up approach such as XML [W3C00], however Step-DL's specific purpose (trace record definition) and its specialized notions of extension give rise to a unique and curious language.

The syntax of Step-DL is described formally in figure 3.1. The following sections proceed to elaborate on the various language features. To begin with, some of the basic structural elements are presented, then the various forms of annotation are introduced. Sections 3.3.5 and 3.3.6 exhibit the mechanisms for extending existing definitions. Readers are encouraged to relate the simple examples that follow to the complete, real-life examples provided in appendix C.

3.3.1 Language Basics

The syntax of Step-DL is reminiscent of Java in several respects. Step-DL files are written in standard ASCII text and are composed of a sequence of record definitions.

Several standard comment forms are supported. Single-line comments begin with either '`//`' or '`#`'. Comments that begin with '`#`' are supported to allow the introduction of C-Preprocessor macros. Multi-line comments are written with the familiar '`/*`' and '`*/`' delimiters.

Figure 3.2 provides examples of some basic Step-DL record definitions. The `record` keyword begins a definition and is followed by an identifier representing the record's name. Identifiers in Step-DL are of the standard $letter|underscore$, $letter|number|underscore*$ form. Each record is composed of zero or more fields, the definitions of which are bounded by the standard '{' and '}' symbols. Fields are specified starting with their type, followed by a comma separated list of field names, terminated by the '`;`' character.

```
file              →  definition*
definition        →  package identifier '{' definition* '}'
                  |  record identifier string-literal?
                     ( extends '!'? item-name )?
                     '{'
                        string-literal*
                        attribute*
                        field-definition*
                        field-modifier*
                     '}'
field-definition  →  type attribute* identifier string-literal* attribute*
                     ( ',' string-literal* attribute* )* ';'
field-modifier    →  ( '~' | '!' ) item-name attributes* ';'
attribute         →  '<' identifier ':' string-literal '>'
type              →  int
                  |  float
                  |  data
                  |  string
                  |  item-name
                  |  type '[]'
item-name         →  identifier ( '.' identifier )*
```

Figure 3.1: Step-DL 1.1 Syntax

```
record A {
    int     i, j;  // two integers
    float   f;     // a floating-point value
    string  s;     // a text value
    data    d;     // byte data
}

record B {
    A[]   a;       // an array of ‘A’ records
}
```

Figure 3.2: Basic STEP-DL definitions

type	*description*
`int`	Integer values in the range -2^{63} .. $2^{63}-1$
`float`	Double precision floating-point values
`string`	Variable-length text values
`data`	Variable-length raw byte data

Table 3.1: The primitive STEP-DL types

The primitive field types are `int`, `float`[1], `string`, and `data`; they are summarized in table 3.1. The STEP-DL `int` type abstracts over the range of Java integer types (`byte`, `short`, `int`, and `long`), internally representing all values as 64 bit `long`s. The STEP-DL `float` type similarly abstracts over the Java single-precision (`float`) and double-precision (`double`) floating point types. STEP-DL `string` values wrap Java `String` values. The primitive `data` type is provided to hold arbitrary byte data. It is used most commonly as the basis for `MetaRecords` (see section 5.3.1).

Fields may be specified as arrays. As in Java, STEP arrays are first-class objects.

[1]Only basic support for the `float` type is provided in the initial public version of STEP

```
package p1 {
    record A { }       // p1.A
}

package p2 {
    package p3 {
        record B {     // p2.p3.B
            p1.A  a;
        }
    }
}

package p1 {           // continue p1
    record C {         // p1.C
        A  a;
    }
}
```

Figure 3.3: Package scopes and qualified names

The size of arrays are defined at run-time, and multi-dimensional arrays may be ragged. If an array field is always of a fixed size, this fact can be indicated with a relative `encoding` attribute modifier (see section 3.3.5).

Once defined, a record type may be used to define the fields of other records.

3.3.2 Packages

Record definitions may be partitioned into various *packages*. A package scope is indicated with the `package` keyword, followed by the name of the package and delimited by the '{' and '}' symbols. As indicated in figure 3.3, packages may be nested or specified in several portions. In the context of multiple packages, record

```
record A  "multi-word label for A"  {

    "description explaining what A records signify"
    "...a continuation of the description"

    int  x  "description of x values";
}
```

Figure 3.4: Labels and descriptions

names are specified relative to the current package or by their absolute name, using the '`.`' symbol as a qualifier. The example illustrates fields defined with both relative and absolute references to the `A` record type.

The intention of Step-DL packages is simply to provide a name-space partitioning that allows a convenient grouping of record types, or a resolution of potentially ambiguous type names (e.g., a `Class` record which, when implemented in Java, needs to be distinguished from the standard `java.lang.Class` type).

3.3.3 Labels and Descriptions

The simplest forms of annotation in Step-DL are *labels* and *descriptions*. Examples of both forms are illustrated in figure 3.4. A label indicates an alternate name for a record that is not restricted to the identifier syntax. Descriptions provide an elaboration on what a record or field value represents. The various platform specific methods for terminating a text line are addressed by having multi-part descriptions, where a break implies a line-break in the description text.

The reason for providing labels and descriptions is so that a tool reading the data can access the annotations to automatically generate on-line help (or some other form of elaboration) regarding the given record type.

```
record A {
    <g:"record attribute">

    int  x <g:"field attribute 1">, y <g:"field attribute 2">;

    int  <g:"distributed field attribute"> i, j, k;
}
```

Figure 3.5: Attribute placement

3.3.4 Attributes

Anticipating that Step-DL definitions could be used to interpret Step record data in a variety of contexts (encoding, visualization, conversion, etc.), the language provides a means for generalized annotation in the form of *attributes*. Figure 3.5 illustrates Step-DL attributes in a variety of placements. In the example, only a single attribute is present in each location, however the grammar permits any number of attributes (written in sequence) to be associated with a given item.

Attributes are divided into two parts, a group identifier and a text value. For both simplicity and generality, the examples in this section use the fictitious group identifier `g` (see appendix B for a description of the real attribute groups, such as `encoding`). The group identifier provides a context for the attribute value. For example, an attribute value of “`constant`” could indicate that all instances of a field have the same value or that the field represents a well known quantity. By qualifying the attribute with the `encoding` group identifier, the meaning is refined to be: “all subsequent instances can be cloned from the initial field instance value.”

Attributes are delimited by the ‘`<`’ and ‘`>`’ symbols, a feature which suggests their analogy to mark-up tags from languages such as XML. An important difference however is the choice of the ‘`:`’ symbol which separates the group name from the value.

The symbol is chosen to contrast the meaning of STEP attributes from that implied in the common '`key=value`' syntax. A STEP-DL attribute specification indicates, not an assignment, but an addition: "add value v to the group g."

There are two varieties of attributes, record attributes and field attributes. Although no strict requirements are placed on the meaning of attributes, record attributes generally indicate information that applies to the type as a whole as opposed to details concerning individual record values. For example, a common record attribute is `<property:"event">` which indicates that the record represents an actual event as opposed to other records which are used as auxiliary data types. Such attributes highlight the intended separation of structure from interpretation. Whether or not a record should be interpreted as an event is a secondary feature of the data and by deferring the information to an attribute, clients can choose to act on the information at their discretion. In contrast, `encoding` attributes do not appear as record attributes because doing so would imply a uniform strategy for all instances, when clearly the encoding of records may vary by context.

Record attributes are placed after any description elements, before the field definitions. Field attributes are applied to a specific field by placing them immediately after the field name in a definition. In the example, the `x` and `y` fields have specific attributes. Attributes that appear between the type and field name, as with the `i`, `j`, and `k` fields in the example, are distributed to each field in the list as if they preceded any specific field attributes. The syntax for distributed field attributes is provided purely as a convenience. It is often the case that a set of attributes are applicable to several fields, the distributed notation simply allows for such redundancy to be collapsed. The syntax also allows the definition of C-Preprocessor macros that represent common combinations of type and attributes. This technique is applied in the examples listed in appendix C.

The division of attributes into groups is intended as a method for distributing the complexity of STEP-DL definitions. The basic language is quite simple and so are the built-in sub-components (i.e., attribute groups). If the language directly included elements for specifying encoding it would not only become more complicated, but also more rigid. The current syntax allows any developer to introduce new interpretive extensions to the language in a way that does not break other clients: consulting the attributes is purely optional.

Step-DL's concept of annotation relates to the common notion of so-called meta-data. However, the term *meta-data* is used in a specific sense in this document, thus it is useful to refine the definition with respect to Step. The colloquial definition of meta-data can be paraphrased as "data about data." In the Step system data about data occurs in both static and dynamic contexts. The static context is embodied by Step-DL annotations, they are information about the data instances. As section 3.3.5 explains, attribute annotations can be extended and overridden. The annotations are however fixed for all instances of a given type.[2] On the other hand, the encoding process, presented in chapter 5, makes use of the term meta-data to describe data regarding the changing encoding policy that is embedded in the encoding stream. To avoid confusion, the term meta-data (in the context of Step) is only used to refer to the dynamic variant encountered in the encoding process. Static forms are referred to as annotations, and in most cases, simply as attributes.

3.3.5 Field Modifiers

Following the field definitions in a record, any number of *field modifiers* may be specified with the '`~`' (extend) and '`!`' (override) operators. In the case of extension modifiers ('`~`' followed by the field name), the attributes that follow are appended to the currently defined set. Override modifiers (those that use the '`!`' operator) indicate that all previous attributes should be discarded, the defaults (if any) restored, and new attributes appended subsequently.

Figure 3.6 illustrates both the extension and override modifiers. The first statement in the definition is a definition of the `x` field, which assigns a base attribute for the group `g`. The first modifier extends the attributes of the `x` field, appending `<g:"extended attribute">`. The override modifier discards the two previous attributes and leaves `x` with the single attribute `<g:"overridden attribute">`. Override modifiers are often useful for experimentation, since they allow the attributes

[2]Section 5.3.2 describes a subtle exception to this rule.

```
record A {
    int  x <g:"base attribute">;

    ~x <g:"extended attribute">;

    !x <g:"overridden attribute">;
}
```

Figure 3.6: Field modifiers

for a given item to be reset and updated without adjusting other records in the type hierarchy.

Modifiers were originally developed as a method for indicating changes to inherited fields (see section 3.3.6), however they can also be used to modify fields within the same definition to enhance the readability of the definition. Basic attributes can be specified in the definition of a field and other more specialized or experimental attributes deferred to later in the definition. This technique is used in some of the examples in appendix C, separating encoding strategy attributes from the main definitions.

Modifiers can also apply to sub-components of a field. Common examples are `string`, `data`, or array fields that have a fixed length. This characteristic is indicated, as in figure 3.7, by specifying an `encoding` attribute. In the example, `A` records are defined as having a `string` field, `x`, where the values of `x` always have the same length.[3] Another record, `B`, contains an `A` record as one of its fields and modifies the attributes for the length of the `x` sub-component to specify that in the particular context of `B.a` values, the values "mostly" have the same length (see section B.1.1 for details on the `constant` and `default` strategies).

[3]`string`, `data`, and array types have an implicit `length` sub-field. A modifier may also be applied to the base type of an array by referring to its `element` field.

```
record A {
    string  x;
    ~x.length <encoding:"constant">;   // all same length
}

record B {
    A  a;
    ~a.x.length <encoding:"default">;  // most same length
}
```

Figure 3.7: Relative modifiers

Modifiers can only be applied to attributes, and not to labels and descriptions. The reasoning is that labels and descriptions are fixed concepts that do not vary with context, while other attributes may require refinement based on context.

3.3.6 Inheritance

Step-DL supports inheritance from a single parent type with the use of the familiar `extends` keyword. Figure 3.8 illustrates a simple example where the type `B` derives from `A`, inheriting the field `x` and adding a new field `y`.

Type inheritance is supported because it is a familiar and established method for promoting reuse. It also has the secondary benefit of allowing the records to be arranged into hierarchical groupings which are often useful for accounting purposes. An important consequence of supporting inheritance is that it allows traces to be extended without breaking existing tools. For example, a `Class` record might initially be defined with only a `name` field. The record could then later be extended to include information such as the size of its instances, the interfaces it implements and/or a variety of other information. By deriving the new, extended version of the record (vs. creating a completely new definition), a tool designed to read the original record type

```
record A {
    int  x;
}

record B extends A {
    int  y;
}
```

Figure 3.8: Simple inheritance

would still function, accepting the new extended type in its place.

It is often useful to view attributes as analogous to methods from other object-oriented systems. In essence, they dictate ways (methods) in which the record data might be used. The `encoding` attributes, for example, provide rules for assembling a functor object (i.e., an encoder) to be applied to instances of the type. The `property` attribute group supplies more nebulous information, but still acts to define how the data might be used. With this analogy in mind, it is then natural to see why attributes might share the same notions of inheritance that methods do: what applies to the parent version should, in most cases, also apply to the child, however there are exceptions which may require a slight modification (in the case of attributes, the act of extension) or a complete redefinition (in the case of attributes, overriding). Figure 3.9 illustrates the various STEP-DL modifier operators in the context of inheritance. In the case of the `B` type, both record attributes and field attributes for the `x` field are inherited from `A`. For `C` records, the '`!`' operator included in the `extends` expression discards the record attributes normally inherited from `A`. A second override modifier is used to discard and update the attributes associated with the `x` field.

```
record A {
    <g:"record attribute for A">
    int x <g:"field attribute for A.x">;
}

record B extends A {
    // inherit: <g:"record attribute for A">
    // inherit: x <g:"field attribute for A.x">

    ~x <g:"extend x's attributes">;
}

record C extends !A {
    // do not inherit: <g:"record attribute for A">
    // inherit: x <g:"field attribute for A.x">

    !x <g:"override x's attributes">;
}
```

Figure 3.9: Inheritance of attributes

3.4 Semantic Issues

Although Step-DL is not a programming language, there are still several semantic issues that influence the resulting translations of the definitions. For the most part, the significant issues result from the ordering of various elements. Records may appear in any order in a Step-DL file, however the order in which they are parsed does have consequences for the type resolution process, described in chapter 4. The ordering of fields within a definition has ramifications in that the generated Java class definitions define equality, hash-code and iteration methods that operate on the fields in the order of their definition (i.e., top down, with inherited fields first).

The semantics of attribute ordering are more subtle. As with fields, there is an implied linearity to the order of attributes. Attributes distributed from a field type precede those that follow individual fields. Subsequent extensions append to the previous list of attributes. However, as previously indicated, the meaning of attributes technically lies in the definition of the particular group to which they belong. The built-in `encoding` and `property` attributes define a top-down interpretation with the most recent (bottom) value taking precedence. Having said that, some attribute values, as is the case with `encoding` attributes, have non-overlapping semantics. A review of section B.1 reveals how some attributes have a layered interpretation. In such cases, a modifier may introduce a new dominant attribute in one layer that does not affect the others.

Step-DL permits recursive type definitions. An example of why such a feature might be useful can be drawn from the Java type system. A `Class` is a `Type` and so is an `Interface`. Each `Class` may implement zero or more `Interfaces` and thus may be represented as a Step record with an `Interface[]` field. Thus a `Type` (`Class`) contains other `Types`, namely the `Interfaces` the type implements. In this case, the recursive definition is acceptable since inheritance is a directed relation, thus the record values should not directly or indirectly refer to themselves. Recursive type definitions are problematic for Step only when they are used to instantiate data values that actually contain circular references. Since there is no simple method for detecting such values, they will cause the serialization process to continue expanding

```
tag.width = 1;
size.width = 4;
size.interpretation = none;
address.width = 4;
address.interpretation = none;

alloc : (tag, size, address, vfield) {
    tag.value = 4;
}
```

Figure 3.10: A simple MetaTF specification

sub-structures infinitely.

3.5 History

The current version of STEP-DL is the result of many iterations. In particular the approach to attributes went through a number of phases. Early on, the attributes related purely to encoding issues and thus their syntax was more integrated with the base language. As development proceeded, it became clear that attributes could be used to automate (or at least explicitly document) the interface with other tools. This led to the partitioning of attributes into groups, each with a specific purpose.

To place the syntax and features of STEP-DL in some sort of context it is useful to contrast the language with its primary influences, namely MetaTF and DAFT.

Figure 3.10 shows an example MetaTF definition, extracted from the DTD for HATF and simplified somewhat. The example shows the definition for an `alloc` record with four fields, `tag`, `size`, `address`, and `vfield`. The example highlights MetaTF's implicit typing, as well as a misleading attributing syntax that suggests attributes are sub-fields. The typing is adjusted to be somewhat more explicit in

```
filetype len_t = {
    parameter n;
    field Length[n] : double
        <units=meters>;
}

filetype flen_t : len_t = {
    field Length[n] : double
        <units=feet>;
}
```

Figure 3.11: A simple DAFT specification

version 1.2 of MetaTF, however the use of fixed record tag values persists.

The DAFT language uses a significantly different syntax, which is illustrated in figure 3.11. Since DAFT definitions equate a single record type with a file, the example actually defines two file formats, where each record in the file is a fixed size array of `double` values. Despite the limited focus of DAFT, the language does offer some interesting features. It supports inheritance (one file type derives from another) and a more familiar annotation syntax that uses the '`<`' and '`>`' symbols. One confusing element of DAFT is the approach to overriding attributes. To override an attribute, the field must be redefined in a sub-type with new attributes. A similar mechanism was implemented in early versions of Step-DL, however it quickly became apparent that such an approach led to definitions where it was unclear which elements were inherited and which were being defined for the first time. To alleviate this problem, the modifier syntax was introduced.

As depicted in figure 3.12, early versions of Step-DL bore a significant resemblance to DAFT definitions. In particular, the type is seen as following the name of a field and the attribute syntax uses the '`key=value`' form. One notable feature

```
record Allocation {
    type : identifier;
    size : int <width=1>;
}
```

Figure 3.12: An early Step-DL specification

```
record Allocation {
    string  type <encoding:"identifier">;
    int     size <encoding:"size=1">;
}
```

Figure 3.13: A modern Step-DL definition

of early Step-DL is the existence of an `identifier` type. The *identifier* strategy (discussed in several subsequent sections) was prevalent enough in early experiments to suggest that it form a separate type. However, once the separation of structure from interpretation was refined, it was clear that the "identifier" characteristic was independent of the type.

Contrasting the `Allocation` record defined in figure 3.12 with a more modern expression in Step-DL illustrates the clear partitioning of encoding information into explicit annotations.

Chapter 4
The `stepc` Compiler

The `stepc` compiler provides a mechanism for processing Step-DL definitions. The current version is capable of parsing the definitions, validating certain built-in attributes and generating output in the form of Java class definitions suitable for interfacing with the Step encoding system. The discussion that follows focuses primarily on the internal representation (IR) for Step types and how this internal form is initially generated and subsequently regenerated from the compiler outputs. While an understanding of the run-time representation of Step types is helpful for understanding elements of the Step-DL language and Step encoding architecture, it is not an essential part of the presentation of Step. The material presented in this chapter will, however, be of interest to those who wish to create extensions to the encoding architecture or extensions that define and utilize new attribute categories.

4.1 Goals

The `stepc` compiler exists as a platform for working with type representations specified in Step-DL. The aim is to provide a modular solution where the aspects of the compilation process (parsing, IR construction, validation, and output) operate reasonably independently. Clearly, the primary goal is to have a method for generating elements to interface with the Step encoding system however, the hope is that the design of `stepc` will also open the door to a variety of extensions for processing Step-DL definitions.

4.2 Approach

A Step-DL input stream is processed by creating a compiler object that completes the parsing of the input and builds the intermediate type representations. Once the resolution of the types is complete, a series of attribute verifiers can be applied to the definitions, followed by a series of emitters that generate output from the definitions.

The mechanism for representing (and creating) type objects in `stepc` is the same one used to create the definitions within the run-time environment in which Step records are instantiated. The reason for using the same representation is that the definitions remain compatible: information available at compile-time is also available at run-time.

4.3 Step-DL → Intermediate Form

Generating the intermediate, object representation of Step-DL record definitions with `stepc` proceeds in four phases:

1. The input Step-DL files are parsed to generate an abstract syntax tree (AST) representation of the input.

2. Skeleton type structures are built from the AST representation.

3. The type structures are resolved in two sub-phases. The first resolves structural information such as the existence of referenced types and inherited field information for derived types. The second sub-phase applies the attribute modifier statements to complete the set of attributes associated with various elements.

4. For attribute groups that are known to the compiler, the resolved definitions are checked to ensure valid attribute values.

Each of the four phases is applied to the set of input files in sequence. Types defined, resolved and validated from earlier inputs persist in the run-time environment and thus may be referenced (without definition) in subsequent input files.

4.3.1 Parsing Step-DL

The first phase in the compilation of Step-DL files involves parsing the input to create an AST representation of the definitions. The parsing mechanism used in `stepc` is generated using the SableCC [Gag98] compiler generator tool. The tool generates Java classes for scanning and parsing a given syntax and also provides skeleton methods for traversing the subsequent ASTs using the *visitor* design pattern.

4.3.2 Building Step Record Definitions

Once an AST has been created for a given input file, the `stepc` compiler then traverses the tree constructing object representations of the Step record definitions. Step's internal type representations are conceptually similar to Java's `Class` objects in that they contain an abstraction of the type as various field and attribute objects. The reason for creating object representations of the record types is twofold. First, they serve as a useful intermediate form that various back-ends of the `stepc` compiler can query to generate output. The second role of the object type definitions is their use in the run-time generation of elements for processing Step records. In particular, the encapsulated encoder objects (described in chapter 5) are generated, not at compile-time, but on-demand as various records are passed to the encoding system. The encoders are created by inspecting a definition object that is equivalent to the one generated at compile-time. The definition object is recreated in the run-time environment by code that accompanies the Java source output for the type definition. This process is best illustrated with a simple example.

Figure 4.1 presents a simplified Step-DL definition for JVMPI method entry events (one extracted from those presented in section C.1). In the second processing phase of `stepc`, a variant of the *builder* pattern is used to assemble type objects by traversing the AST and adding elements to the definitions as they are encountered.

```
package jvmpi {
  record METHOD_ENTRY2 "Method Entry" extends MethodEvent {
    int <property:"address"><encoding:"size=4">
      targetObjId "Target Object Address";
    ~targetObjId <encoding:"cache=65536">;
  }
}
```

Figure 4.1: STEP-DL for a method-use record

```
...
DEFINITION = RecordDef.builder().newRecordDef("METHOD_ENTRY2", "jvmpi")
  .setParent("jvmpi.MethodEvent", false)
  .setLabel("Method Entry")
  .addField(FieldDef.builder().newFieldDef("targetObjId", "step.StepInt")
    .addDescriptionLine("Target Object Address")
    .addAttribute(new Attribute("property", "address"))
    .addAttribute(new Attribute("encoding", "size=4"))
    .addAttribute(new Attribute("encoding", "cache=65536"))
    .makeFieldDef())
  .makeRecordDef();
...
```

Figure 4.2: Java code for building an internal STEP type definition

An equivalent sequence of build operations can then be distilled from the completed type and included in the generated Java output for the type. A segment of the output Java code for building a representation of `METHOD_ENTRY2` records is illustrated in figure 4.2. The sequence of construction operations (`newRecordDef()`, `setParent()`, `addField()`, etc.) acts on a builder object (the singleton `RecordDef.builder()`), appending various items to the current build state. When all the necessary elements have been set/added the builder combines the information to create a new definition object. The example also shows the use of a subordinate builder (`FieldDef.builder()`) to create individual field definitions. One step in the build process that is not shown in figure 4.2 is the addition of modifiers (such as the one that modifies `targetObjId` in the example). Local modifications are expanded by `stepc`, and thus only modifications to inherited fields or sub-fields appear in the generated Java output.

Note that newly constructed type definitions only have indirect references to other types; specifically, the parent and field types are referenced only by name.

4.3.3 Resolving Type Information

After the traversal of the AST and construction of the skeleton type definitions, `stepc` assumes that all the necessary type information has been loaded and then proceeds to resolve the dangling references alluded to in the previous section. The resolution phase iterates over the definition objects, completing their type information in two steps. The first resolves references to other types, ensuring that fields have types that are defined, and in the case of derived records importing the inherited attribute and field structures. The second resolution step applies the various modifiers included in the original Step-DL definition.

Types are resolved by `stepc` roughly in the order that they were defined in the Step-DL input. Out-of-order resolutions arise from two requirements. First, in the case of derived types, it is necessary for the parent type to be complete prior to importing the inherited structures. Thus, if a derived type precedes its parent in the input, the parent will be resolved on-demand before the derived type. The second case where a type may be resolved on-demand occurs in the application of relative

```
record A {
    B  b;
    ~b.x <encoding:"size=1..">;
}

record B {
    int  x <encoding:"size=1+">;
}
```

Figure 4.3: A field that requires an alternate type definition

modifiers (see section 3.3.5).

Figure 4.3 illustrates a situation where a relative modifier leads to on-demand resolution. In the example, the modifier applies to a sub-component of the `b` field, thus a complete definition of the `B` type is needed to interpret the modification. Since the definition for `B` follows after `A`, it is resolved ahead-of-time during the resolution of `A`.

The example also provides an introduction to the concept of *alternate* type definitions. Without the modifier, the definition of `A`'s `b` field would simply require a reference to the `B` type. However, the inclusion of a relative modifier requires that the field be defined in terms of a refined version of the `B` record type. To implement the refinement, the definition of `b` to refers to an alternate version of the `B` type, where the `x` field is extended to include the additional `<encoding:"size=1..">` attribute. The alternate type definition is created by copying the definition of `B` and adding the extra attribute to the copy's `x` field.

4.3.4 Attribute Verification

The fourth stage in the compilation of Step-DL definitions provides developers with an opportunity to support verification of the content of various attribute values. The current version of `stepc` only checks `encoding` attributes, essentially ensuring that the values are syntactically valid (according to the definitions in section B.1) and that they are applied to the correct types of fields.

4.4 Compilation Errors

Errors in Step-DL definitions that `stepc` is capable of detecting can essentially be divided into categories that follow the four phases of compilation. The error messages that are produced follow the SableCC approach of specifying the location of the error with a line number and offset (e.g., `[3,12] expecting: ';'`). Parse errors and those that occur during the building of the type object (e.g., redefinition of a local field) can often be associated with a specific location, whereas errors in the resolution and attribute verification phases are associated with a given type definition, and thus identified by the start of the corresponding Step-DL record definition. Some example error messages and their interpretation are presented in table 4.1.

4.4.1 Circular Dependency Errors

The multi-phase type resolution in `stepc` provides the flexibility needed for defining directly and indirectly recursive types. However, as a byproduct, this flexibility also creates the potential for circular dependency errors. Examples of the two kinds of circular dependencies are illustrated in figure 4.4. The first error is common to all languages that support type inheritance, namely that a type cannot derive from itself. The second error is specific to Step-DL. The problem occurs in recursive types when a relative field modifier is applied to a sub-component of the type, where the sub-component has the same type as that being resolved. In the example, the type `C` has a field `d` of type `D` which in turn has a sub-component `c` of type `C`. To apply the modifier `!d.c` during the resolution of `C`, it is necessary to have a complete definition for the type of `d`, namely `D`. However, resolving `D` results in a similar dependence on `C`, completing the cycle and preventing the resolution of either type. These examples are rather trivial, but it is reasonable to think a larger and more complex definition

Error Message	*Description*
`Unknown token: é`	**Lexical Error:** The character ‘é’ is not allowed in Step-DL
`expecting: ';'`	**Parse Error:** The statement requires a terminating ‘;’
`redefinition of type "T"`	**Semantic Error:** The record type `T` was already defined earlier
`parent type "T" is undefined`	**Semantic Error:** A record is defined as inheriting from a non-existent type named `T`
`redefinition of field "f"`	**Semantic Error:** A field named `f` was already defined in the current or parent record
`the type of field "f" (T) is undefined`	**Semantic Error:** The type `T` is not defined, thus `f` cannot be defined as being of type `T`
`can't modify field "f", it is not defined for "T" types`	**Semantic Error:** A modifier (‘~’ or ‘!’) was applied to a field that is not defined in the current or parent record
`circular inheritance; "T" derives from itself`	**Semantic Error:** Type `T` is defined by inheriting directly or indirectly from itself
`circular dependency detected`	**Semantic Error:** A modifier requires the complete definition of the current type (which is impossible without the completed modifier)
`field "f" has illegal encoding attribute: "size = -1"`	**Attribute Error:** The attribute value is unknown or incorrect for the `encoding` group

Table 4.1: Interpreting stepc compile errors

```
record A extends B {
}

record B extends A {
}
```

(a) Circular inheritance

```
record C {
    D  d;  !d.c;
}

record D {
    C  c;  !c.d;
}
```

(b) Circular relative modifiers

Figure 4.4: Circular type dependencies

hierarchy could contain such errors through several levels of indirection.

4.5 Generated Java Output

The Java class definitions generated from the internal type representation are reasonably simple. Code for recreating the definition object is included in the static initializer of the class. Various instance methods are generated for iterating over a record's fields, comparing two records for equality, and computing a hash-code for the record. The equality and hash-code methods are necessary for records to be compatible with the *identifier* strategy. Finally, since the encoding engine must be able to create instances of the records without directly invoking a specific constructor, the Java output includes an inner factory class that can be used to instantiate records from a set of field data.

Java output for the METHOD_ENTRY2 record defined in figure 4.1 would appear as follows:

```
package jvmpi;

import step.*;
import step.typedef.*;

public class METHOD_ENTRY2 extends jvmpi.MethodEvent
{
  public static final RecordDef DEFINITION;

  static
  {
    DEFINITION = RecordDef.builder().newRecordDef("METHOD_ENTRY2", "jvmpi")
      .setParent("jvmpi.MethodEvent", false)
      .setLabel("Method Entry")
      .addField(FieldDef.builder().newFieldDef("targetObjId", "step.StepInt")
        .addDescriptionLine("Target Object Address")
        .addAttribute(new Attribute("property", "address"))
        .addAttribute(new Attribute("encoding", "size=4"))
        .addAttribute(new Attribute("encoding", "cache=65536"))
        .makeFieldDef())
      .setFactory(new Factory())
      .makeRecordDef();
  }
  public StepInt targetObjId;

  public METHOD_ENTRY2(StepInt envId, StepInt methodId, StepInt targetObjId)
  {
    super(envId, methodId);
    this.targetObjId = targetObjId;
  }

  public FieldIterator fieldIterator()
  {
    StepObject[] fields = { envId, methodId, targetObjId };
    return new FieldIterator(fields);
  }

  public boolean equals(Object o)
  {
    METHOD_ENTRY2 rhs = (METHOD_ENTRY2) o;
    return envId.equals(rhs.envId)&&methodId.equals(rhs.methodId)&&
           targetObjId.equals(rhs.targetObjId);
  }

  public int hashCode()
  {
    return envId.hashCode()+methodId.hashCode()+targetObjId.hashCode();
  }

  private static class Factory implements RecordFactory
  {
    public StepRecord newRecord(StepObject[] fieldData)
    {
      return new METHOD_ENTRY2(
        (StepInt) fieldData[0],
        (StepInt) fieldData[1],
        (StepInt) fieldData[2]
      );
    }
  }
}
```

Chapter 5
Encoding Architecture

Chapters 3 and 4 established how to define trace data records and create the type structures associated with the records. This chapter focuses on the main function of Step, namely encoding a stream of such records in a compact file format. The discussion reveals the inner workings of the central portion of figure 1.1, detailing how the run-time record definitions are used to create modular, adaptive encoding policies for each data element. Some of the subtleties of the approach are described including how so-called meta-data regarding the encoding process is embedded in the output stream, and how dynamic polymorphism is handled by the encoding policies. Several common adaptive compaction strategies are discussed and a classification of the various techniques is presented. The chapter concludes with a discussion that touches on some of the other design factors that influenced the particular approach of Step.

5.1 Goals

The primary goal of the encoding engine is to embody a method that, in most cases, produces an encoding that is significantly more compact than a naïve serialization of the data. This goal must be balanced against the need to provide a solution that is efficient both in the time and memory needed to encode a trace. Specifically, the approach should be constrained to operate in linear time, $O(n)$, and use a bounded,

$O(1)$, amount of memory, both with respect to the input trace size. Furthermore, an effective (or at least desirable) solution should also provide an open, flexible architecture that makes few assumptions about the forms of data to encode and the strategies used to implement the encoding. Finally, to address the portability requirement, both the encoding system and format should be compatible with various platforms.

5.2 Approach

The wealth of tracing research highlighted in chapter 2 indicates that trace data is highly compactible. Experience has demonstrated that the compactibility is due to patterns and redundancy that exist on two levels: in the instances values of a particular record type and in the sequence of records within a given data stream. The philosophy of the STEP encoding system is to attack the first form of patterns and redundancy, striving for a near-optimal byte-level encoding of record values. The compression of record sequences is deferred to an established tool such as `gzip` [GAF] or `bzip2` [Sew]. This approach requires special attention to ensure that the record reductions achieved by the STEP encoding system do not adversely affect the overall compaction of traces when combined with sequence compression techniques. The discussion of such issues is continued in section 5.4 and chapter 6.

To address the efficiency goals stated above, the encoding process operates in a single pass, converting the input record stream directly to the binary format without any buffering. To obtain an amortized reduction in the average size of records, the process is implemented by creating encoder objects that encapsulate an encoding policy for a particular type where the encoders adapt their policy based on the sequence of values they encounter. The basic concept is illustrated in figure 5.2. `OBJECT_ALLOC` records (defined in figure 5.1) have a `newObjId` field that contain address values which are likely to be steadily increasing by small increments. These values are passed through a *delta* encoder object that outputs just the difference from the previous value. Policy changes are included in the binary output stream in the form of *meta-data*, that is "data about the encoded data."

The mechanism that implements the encoding process is completely isolated from

```
record OBJECT_ALLOC extends JVMPI_Event {
    int  arenaId;
    int  classId   <property:"address"> <encoding:"size=4">;
    int  arrayType <property:"unsigned"><encoding:"size=1">;
    int  size      <property:"unsigned">;
    int  newObjId  <property:"address"> <encoding:"size=4">;

    ~classId  <encoding:"identifier">;
    ~newObjId <encoding:"delta">;
}
```

Figure 5.1: STEP-DL for an allocation record

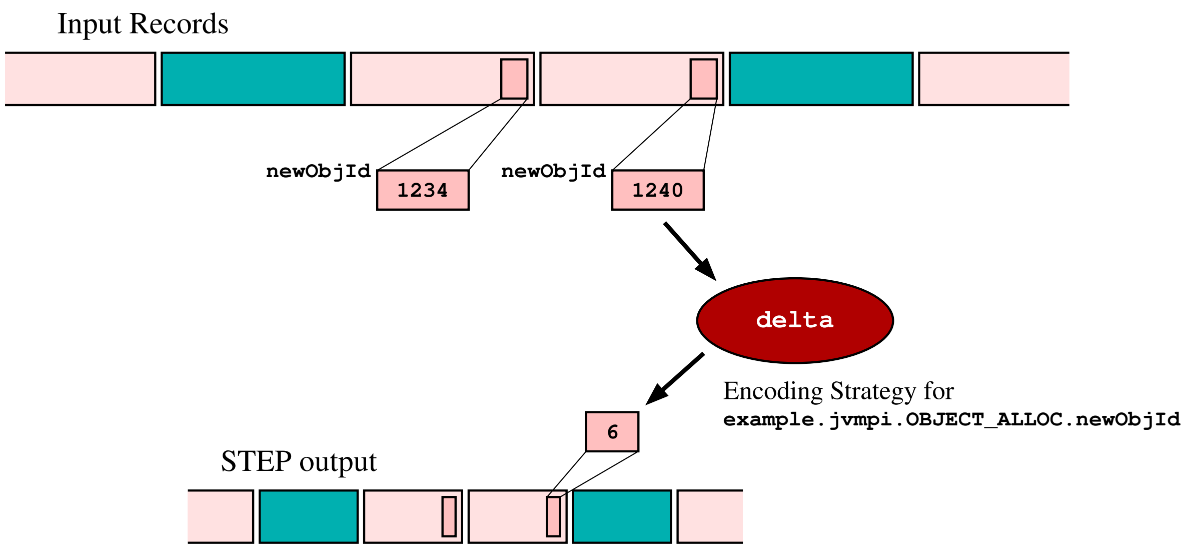


Figure 5.2: The encoding process

```
StepRecordOutput stepOut = new StepEncodedOutput(file);
stepOut.write(new OBJECT_ALLOC(...));
```

(a) Producer

```
StepRecordInput stepIn = new StepEncodedInput(file);
StepRecord record = stepIn.readRecord();
```

(b) Consumer

Figure 5.3: The basic STEP client interface

clients of the system. Figure 5.3 shows the only operations that clients need to be aware of. Trace producers simply create an output stream and write objects to the stream, while consumers just create an input stream an read records from it. The run-time definitions associated with each record (mentioned in previous chapters) are used internally by the encoding system to assemble the encoding policy objects based on the `encoding` attributes associated with the original definition.

5.3 The Encoding Process

The contents of a STEP data file begins with an identification and options header which is followed by a series of `[size][record]` entries. The size, in bytes, of each record is included so that record types that are unknown, or unavailable, during the decoding process can simply be passed over. Each record entry begins with an identification of the subsequent type, where the ID values for each type are assigned automatically, and thus are not bounded or prone to conflict as in the MetaTF [CJZ00] system.

One of the key differences between STEP and other trace encoding systems is its use of an adaptive encoding process. Instead of using a fixed encoding policy (or a dynamic policy with explicit changes, as is possible with MetaTF), the system monitors various characteristics of the input data and, when appropriate, makes adjustments to the encoding policy automatically. The process is implemented by associating each record type with a separate encoder object. Each encoder encapsulates a policy for translating values of the given type to and from the binary representation. Some encoders implement a direct translation, while others implement a more sophisticated transformation based on properties of the underlying values. Encoders are arranged to form a tree- or DAG-like hierarchy, with record encoders deferring to sub-encoders to handle their various fields. The encoders are assembled, as needed at run-time, by a factory object which queries the type definition for a record to get the policy basis from the `encoding` attributes. As records are received by the system, the encoders adjust their internal policy based on the parameters of their particular strategy, communicating their state changes in the form of *meta*-events. When the trace is decoded, the meta-events are applied so as to recreate the same sequence of policy adjustments made by the encoding process.

Encapsulating the encoding policies inside independent objects provides a great deal of flexibility. Encoders may be nested, chained or shared in a variety of ways and their interface makes few assumptions about the data being encoded. The design facilitates experimentation with encoding techniques, as new strategies can be added with only minor modifications. Furthermore, if the definition for a particular trace record is not available during the decoding process, the record is simply skipped with no effect on the other encoders.

5.3.1 Meta-Events and Meta-Records

The term *meta*-data is generally used to refer to information about the content of some underlying data stream. Using this definition, virtually all STEP-DL attribute information could be viewed as meta-data. However, with regards to STEP, the term is used to refer to information about changes in the encoding policy. This information is transmitted in the form of *meta*-events, which are in turn packaged in *meta*-records.

```
record A {
    int[]  x;
    ~x.element <encoding:"size=1+">;
}
```

Figure 5.4: A definition that causes meta-events

To understand the role of meta-events and meta-records in the STEP data stream, it is useful to consider a simple example. Suppose a record `A` is defined as in figure 5.4. The definition indicates that the `x` field is an array of integers, where most of the values are expected to require a single byte to encode but some larger deviants are allowed. Consider then what happens when an `A.x` field holds the data `[3, 10, 1, 13563, 2, 19]`. In this case the first three values conform to the baseline policy (i.e., they only require 1 byte to encode) however, the policy must be adjusted part way through the encoding of `x` to account for the deviant value (the value 13563 requires 2 bytes to encode).

The approach used in the initial version of the MetaTF system is to insert an independent record ahead of the current one—in this example, the `A` record—that expresses the policy adjustment. Thus, the decoding of `A.x` would be adjusted to expect 2 byte values for *all* the array elements. It should be clear from the example that such an approach results in policy changes that are often premature, excessive, and depending on the type of adjustment, potentially even error-prone. Consider an extension of the example. Suppose the `A.x` field also includes the value 78321, which requires $\geq$ 3 bytes to encode. If meta information is automatically generated on-demand and prepended to the current record, which change should apply? In what order?

The STEP approach to meta-data is designed to provide specific targeting of policy adjustments. The basic idea is to introduce a new form of meta-record that bundles

```
data: [3, 10, 1] [13563, 2, 19]
                |
meta: [A.x.element, size:=2]
```

Figure 5.5: Data and meta-data

together both the policy change information and the particular data record that the change applies to. The structure of meta-records is quite simple. The data record is partitioned into sections, where the end of a partition indicates that a meta-event (i.e., a policy adjustment) should be applied before continuing to decode the data. To continue the example above, the `A` record would be packaged in a meta-record as follows: The data is partitioned into 2 segments, one including the data for the `x` field before the adjustment and a second with the data that follows immediately after the adjustment. The meta-record also contains a meta-event record that specifies the policy change, in this case that the next element of an `A.x` field is a 2-byte value. Following the encoding strategy shown in figure 5.4, resizing adjustments of `A.x.element` are elastic, and thus the policy promptly returns to using a single byte encoding for the subsequent element values. Figure 5.5 illustrates the contents of the meta-record.

This definition of meta-records allows several policy changes to be applied throughout the decoding of a given record. The general property of meta-records is that for n data segments, there are $n-1$ meta-events to be applied between each partition. The decoding of meta-records is implemented with a sort of buffer stack. When the current data segment is exhausted, the next meta-event is immediately applied to the current context.

The encoding and decoding of meta-records is complicated somewhat by the fact that the meta-events are implemented as STEP records themselves. This is useful since meta-events often contain data that can benefit from common encoding strategies such as the *identifier* strategy (see section 5.4.4). However, the implication is that, in some cases, one of the meta-data segments of a meta-record—normally a meta-event—can actually be another embedded meta-record, which contains the meta-event and meta-meta-data. The result is that the encoding engine must be structured to allow for such, potentially recursive, record constructions. The implementation of the engine accounts for this while maintaining a low-cost execution path for the common (i.e., non-recursive) case.

This design may seem excessive, however the benefits are twofold. First, by combining data and meta-data into a single record, encoding policy changes can be targeted at the exact byte position that the change is relevant, rather than at the coarse record level. Second, using STEP records to encode meta-events ensures that the amortized size overhead of meta-data is minimized.

Common Meta-Event Types

The current STEP implementation uses two kinds of meta-events. The first, and most often used, is the `IrregularValueEvent`. Such events indicate that the next value to be decoded is a deviation from those normally expected by the so-called regular value strategies discussed in section 5.4. In this case, the decoder defers to some baseline strategy to decode the value. The second, and more general, meta-event is the `EncoderMessageEvent`. These events pass an arbitrary text message to the decoder. In the example above, the message "`size:=2`" is passed to the decoder for elements of `A.x`. In this particular case, the policy is elastic and thus the value `size` returns to `1` immediately after decoding the deviant value.

5.3.2 Accounting for Polymorphism

The fact that STEP allows inherited record types introduces some subtleties to the encoding process that are necessary to handle the case when a derived type is used in place of its parent. The issue is that sub-types must be correctly detected, indicated,

```
record A {
    int  x <encoding:"size=1">;
}

record B extends A {
    ~x <encoding:"size=2">;
}

record C {
    A  a;
    ~a.x <encoding:"size=3">;
}
```

Figure 5.6: Conflicting modifiers

and encoded to prevent so-called object slicing.[1] In other words, the decoding process must have sufficient information to recreate the polymorphic data stream instead of simply generating base-type versions of those that were encoded by the producer.

The presence of polymorphism thus requires that fields that are record types themselves must be encoded with an additional, hidden sub-field that indicates the exact type of each instance value. To minimize the added overhead of identifying the type of each field, encoding strategies are applied to the type values according to one of three variants: the fields are indicated as definitely monomorphic, rarely polymorphic, or often polymorphic. The `encoding` attributes for these variants are described in section B.1.4. The root encoder for all records uses the polymorphic variant, whereas the default strategy for other record fields is to use the rarely polymorphic version.[2]

[1]Object slicing occurs when an object is copied or exported through a reference that views the object as one of its parent types. In such cases, fields that are not defined in the parent get lost (or *sliced*) during the copy process.

[2]In the common case were fields are never polymorphic, the overhead of the "rarely polymorphic" encoding is essentially nil.

One particular difficulty with polymorphism arises when there is a conflict between attribute extensions. Figure 5.6 illustrates a situation where a modifier that applies to an inherited field conflicts with a relative modifier. If a `B` record is used in place of an `A` for the `C.a` field, the modifiers applied to the `x` sub-field differ on how to encode the values. Specifically, it is unclear which of the "`size=2`" or "`size=3`" rules should take precedence. To account for such discrepancies—which can only be detected at run-time—the encoding system implements a mechanism for merging inherited field attributes with relative modifier attributes, where the contextual modifications take precedence (i.e., they follow) the inheritance modifications. The reasoning is that when the rules disagree, the contextual modification is assumed to be the most relevant.

5.4 Encoding Techniques

The strength of the Step system is that it permits a wide range of techniques for reducing the average size of record data. Some of the strategies are based on simple transformations, such as using a minimal number of bytes to encode integer values. However, the most significant reductions can be attributed to strategies that exploit certain "regularity" characteristics of the underlying values.

To be effective, these strategies must address two issues. First, there must be a simple method for handling values that deviate from the expected pattern. The Step encoding process addresses this issue with a solution based on the well known *decorator* design pattern. Strategies are stacked one on top of another. When the top of the stack encounters a deviant value, it simply defers to the next strategy in the stack; the deviant value is signaled by an `IrregularValueEvent` meta-event. Generally, strategies are arranged in two levels, one implementing a complex pattern-based strategy and a second baseline rule to handle deviants. However, the design permits essentially any form of multi-level reduction strategy. The second criteria for effectiveness is that pattern-based reduction strategies must be designed and used carefully so as not to introduce new complex data patterns that will frustrate aggregate compression of the output data. Samples discusses this effect in his work on address trace reduction [Sam89]. For example, if address reference data are separated into separate instruction-read, data-read, and data-write streams, each stream is individually more reducible than the whole. However, this separation requires a fourth stream to identify the category of each reference. It turns out that this fourth stream is highly incompressible, thus resulting in no overall benefit.

The current strategies implemented in the Step system fall into three main categories. Values are either removed entirely, reduced according to some computational rule, or replaced by a smaller representative value. In addition to the descriptions below, the strategies are also described in more detail in section B.1. Section 6.2.2 discusses the application of several of the following strategies to a collection of real traces, and includes examples that express the strategies in Step-DL attributes.

5.4.1 Translation Rules

The basic translation rules implement a direct mapping of data values onto portable byte encodings. The most significant are the various methods for encoding integer data. Integers may be indicated as requiring a fixed number of bytes to encode, where, in some cases, a meta-event can signal a permanent or elastic resizing. The overhead of meta-data can often be avoided by using a variable sized encoding that uses the high bit of each byte to indicate whether there are more bits to follow.

5.4.2 Removal Techniques

Some trace values are extremely repetitive. A good example are thread identifiers for single-threaded programs. In this case, it is sufficient to encode the initial occurrence of the value and then simply assume the same value for all subsequent occurrences—thus effectively reducing the amortized encoding size to 0 bytes.

The `ConstantValueStrategy` encoder implements three versions of this approach. The first assumes that the values are truly constant, encodes the initial value and generates an error if any other value is given. The second approach is similar to the first, but allows some deviant values, signalling the irregularity with a meta-event. This is often the preferable version for data such as thread identifiers (given that the program is single-threaded) because, at least in the Java environment, some other threads used by the virtual machine may still exist and generate a small number of events. The last variant is useful for values that are comprised of long sequences of repeats. Essentially, the default value is reset each time a new value is encountered. These strategies are indicated in STEP-DL as the `encoding` attributes "`constant`", "`default`", and "`repeat`" respectively.

It is worth mentioning that another common trace reduction technique, run-length encoding, is essentially a version of the *repeat* strategy. The idea is to encode the number of repeats along with the initial value, so that changes in the default need not be conveyed with meta-data. The reason STEP does not currently implement run-length encoding is that it would require look-ahead information. To compute run-lengths, the encoder would need to either make two passes over the input or implement a buffer where output records are suspended until the end of a run.

5.4.3 Computational Techniques

Computational reduction techniques can be summarized as those that exploit some known formulaic pattern in the data sequence. In other words, the next value in the input sequence can be reconstructed with knowledge of the previous values and some piece of information that can be encoded with fewer bytes than the actual value. This is a rather broad definition that actually includes the removal strategies mentioned earlier. However, in the current STEP implementation the definition is most applicable to the arithmetic strategies known as *delta* and *stride*.

The delta strategy, often referred to in the literature as the difference technique, is most useful for sequences of numeric values that exhibit an increasing or decreasing pattern. The strategy works by computing the difference between the current value and the previous value. If the absolute value of the difference is below a given threshold then only the difference value is encoded, otherwise the value is considered a deviant, signalled with a meta-event, and encoded with the baseline rule. The difference technique has proven to be a particularly effective way to reduce address values in both allocation and load/store traces. Furthermore, it appears that in many cases the difference values themselves exhibit a high degree of regularity. For example, load/store addresses increase or decrease in multiples of 4 (on a 32-bit architecture), whereas allocators often proceed sequentially through free-space, producing a number of common object sizes. The difference technique is the primary reduction strategy used by both Mache [Sam89] and PDATS [JHBZ01] and is also used in the HATF [CJZ00] format.

Another arithmetic technique is the stride strategy. The stride strategy is, essentially, a version of the difference technique where the delta values are constant. As with the other constant value strategies, only the initial value is encoded and all subsequent values are computed by simply adding the fixed increment to the previous value.

5.4.4 Substitution Techniques

Substitution techniques apply when a more compact representative can be output instead of the complete value. The two examples implemented in STEP are numeric offsets and ID substitutes.

The *offset* strategy outputs the difference between a value and a given fixed base. The *window* strategy is an adaptive variant of the offset strategy where the base is shifted whenever the difference exceeds a given threshold. The two variants of the offset strategy are often useful for encoding address values when no other strategy applies. For example, the window strategy may be a good choice for encoding the addresses of garbage collected objects. In this case, the values are unlikely to exhibit any particular pattern, however it is likely that the collector will reclaim dead objects in the same memory region at roughly the same time.

One encoding strategy, in particular, has a significant effect on the compactness and compressibility of STEP traces. The strategy arose from an early observation that traces often contained fields that are limited to a certain fixed set of values. In many examples, the values are text identifiers used to label various entities (methods, types, threads, etc.). Clearly it is wasteful to store the full representation of such values (e.g. "`spec.benchmarks._213_javac.UnsignedShiftRightExpression`" for a `type` field) in each record. Instead, the *identifier* strategy encodes the values using a compact integer ID, signalling the mapping of value to ID with meta-events. The current version of Step, extends this idea beyond string values to include arbitrary field data that exhibit an identifier distribution. For example, JVMPI data uses an address value to refer to class and method structures, but the number of distinct values is often small enough that values can be indicated using 1 or 2 byte identifier values.

In cases where the total number of distinct values is large and the actual values are reasonably small (as is the case with address data), the identifier strategy can begin to lose its effectiveness. However, if the distribution of values remains limited within a given input window then the *cache* strategy may be a viable alternative. The approach is similar to the identifier strategy in that values are indicated by their cache slot ID. The current implementation of the cache strategy is rather basic, using a simple rotational replacement policy, signalling replacements with a meta-event.

Not many forms of data are suited to the cache strategy, since frequent replacements generate an unacceptable amount of meta-data. Samples encountered this effect in his initial attempts to use a caching strategy for his Mache encoder. Although Samples abandoned the idea in favor of the much simpler (and apparently equally effective) difference technique, caching remains useful for some pernicious forms of data such as the target object address for virtual dispatch calls.

5.5 Other Framework Design Factors

Like the approach of Haines *et al.* [HMVR95], the Step system is designed to embody the major elements of Booch's definition [Boo94] of an *object model*: abstraction (data objects appear uniform, while they exhibit variable encoding), encapsulation (clients are isolated from the encoding process), modularity and hierarchy (interpretation strategies are modular and composable; record types are extensible). The system also embodies some minor elements including typing, and persistence.

Also, an attempt is made to implement the system using a number of common object-oriented design patterns. Constructional approaches such as the *factory* and *builder* patterns are used to create encoder and type definition objects, respectively. The adaptive encoding policy objects are a clear instance of the *strategy* pattern. The hierarchy of encoders is enabled through the use of the *composite* and *decorator* patterns.

Chapter 6
Experiences

The preceding chapters have addressed a number of the requirements for a general trace encoding system established in section 1.2. This chapter completes the discussion by illustrating how STEP addresses its primary objective, namely to generate compact trace representations. The presentation considers the application of STEP to encoding a variety of trace data collected from a number of different Java programs. The results indicate that STEP encodings are significantly more compact than naïve alternatives, and also that in many case the compressibility of the data is improved when encoded in the STEP format. The chapter concludes with a brief discussion of applications that have been developed to consume STEP data.

6.1 Trace Collection

In total, 8 Java programs were used as a source of trace data. They are summarized in table 6.1. The first six are from the standard SPECjvm98 [SPEC98] suite of benchmarks and the last two are adapted from programs included in the Ashes [VRSa] suite of programs. Appendix D offers a slightly more detailed description of the programs, and in some cases their input.

Three different trace groups were collected to provide a range of data values. The first two were collected through the Java Virtual Machine Profiler Interface [Sun] by writing the records out verbatim and then converting the raw data format to a STEP

program	*description*
`compress`	Lempel-Ziv compression program
`jess`	the Java Expert Shell System
`db`	database simulation
`javac`	Sun's Java compiler from JDK 1.0.2.
`mpegaudio`	MP3 audio decoder
`mtrt`	multi-threaded raytrace program
`jack`	parser generator
`sablecc`	object-oriented compiler compiler
`soot`	Java bytecode transformation framework

Table 6.1: Traced Java programs

encoding. The third set of traces was collected by instrumenting the application bytecode with the SOOT tool [VR00] where the instrumentation wrote the events directly in the STEP format. The number of events collected from each program is summarized in table 6.2.

The first set of traces (subsequently referred to as the "memory" set) consists of heap allocation and free events, garbage collection start and stop events, and class load events. The memory traces provide a heterogeneous mix of records that consist mostly of integer data. Two of the traces (`compress` and `mpegaudio`) have relatively few allocations and thus are given separate treatment in some of the analyses.

The second set of traces (subsequently referred to as the "method" set) consists of method usage events and class load events. The method traces provide large and mostly homogeneous event sequences that permit a statistically meaningful discussion of the average record reduction achieved by the STEP encoding. The traces from `compress`, `mtrt`, `sablecc`, and `soot` were truncated due to limitations of the JVMPI profiling agent.

The third set of traces (subsequently referred to as the "invoke/field" set) consists of invoke (vs. dispatch receiver) and field access events for only the application classes in the benchmarks (i.e., events from the standard library classes are not included).

program	*total events*		
	memory	method	invoke/field
`compress`	20669	165170511	100000000
`jess`	15889150	96347193	100000000
`db`	6428666	117540344	100000000
`javac`	12743801	94254338	100000000
`mpegaudio`	26946	99511817	100000000
`mtrt`	13303965	165169490	100000000
`jack`	11962806	58057376	76468069
`sablecc`	66705675	165159324	100000000
`soot`	29412399	165143277	59416025

Table 6.2: Trace event summaries

The invoke/field traces supply a highly regular, heterogenous sequence of text-based events. In most cases the traces are truncated after encountering 100 million events. The `soot` run is an exception since the benchmark prematurely terminates (after roughly 50% execution) due to a virtual machine error not encountered during the JVMPI runs.

6.2 Encoding

To begin with it is useful to relate the STEP encoding format to other more naïve encodings. Figures 6.1 and 6.2 relate the size of STEP trace files for the memory and method traces to versions that simply record the data in a raw format with no attempt at reduction. The raw format consists of standard JVMPI event structures encoded verbatim (i.e., 4 bytes for an integer, etc.), using a single byte to indicate the event type. On average the STEP format achieves a better that 50% reduction in total trace size. Another perspective, illustrated in figures 6.3 and 6.4, considers the average number of bytes per record (bpr) used by the encoding. The STEP encoding similarly reduces the average record size to less than half of that required by the

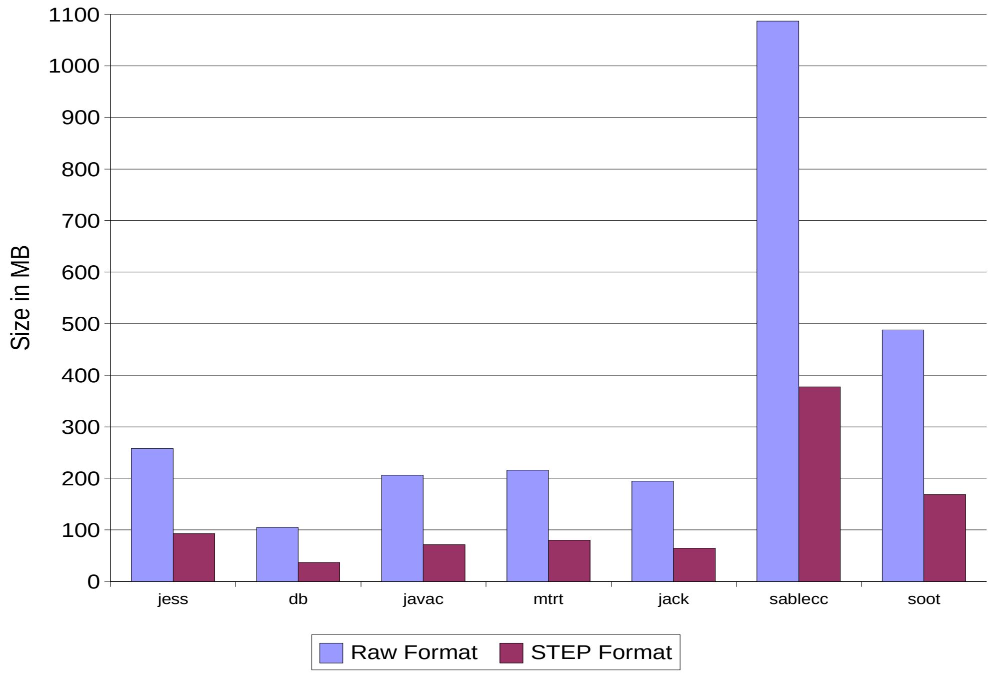


Figure 6.1: Memory trace sizes

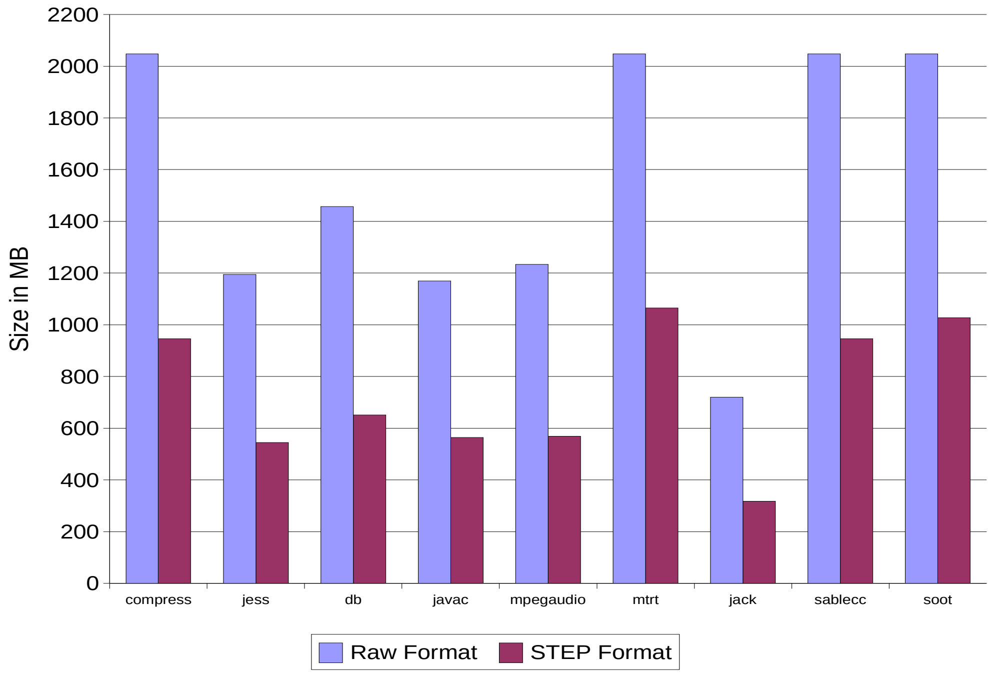


Figure 6.2: Method trace sizes

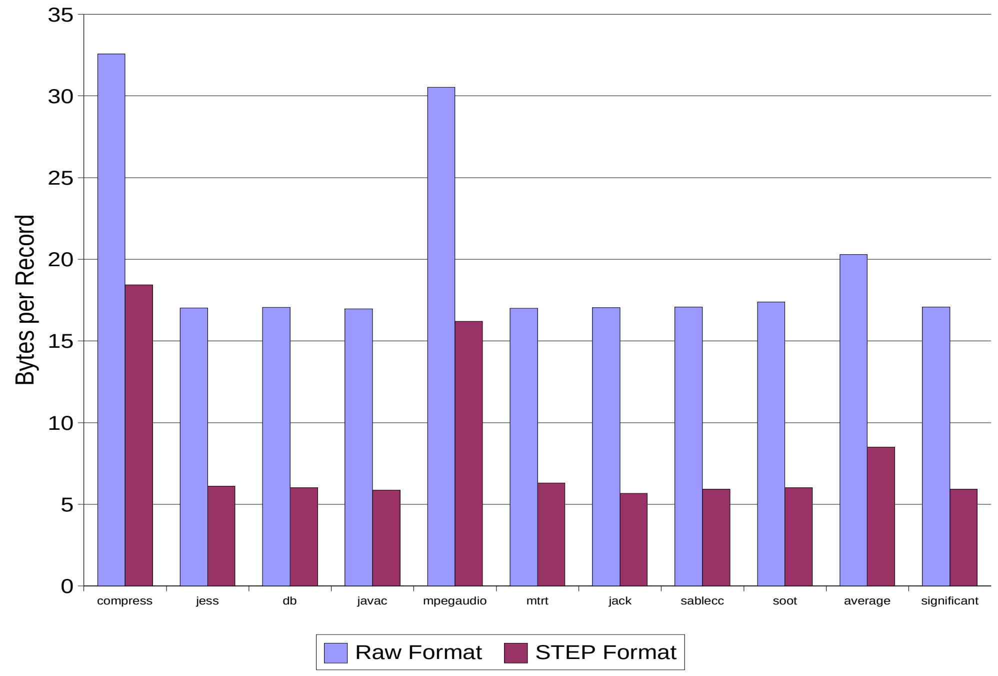


Figure 6.3: Memory trace bpr rates

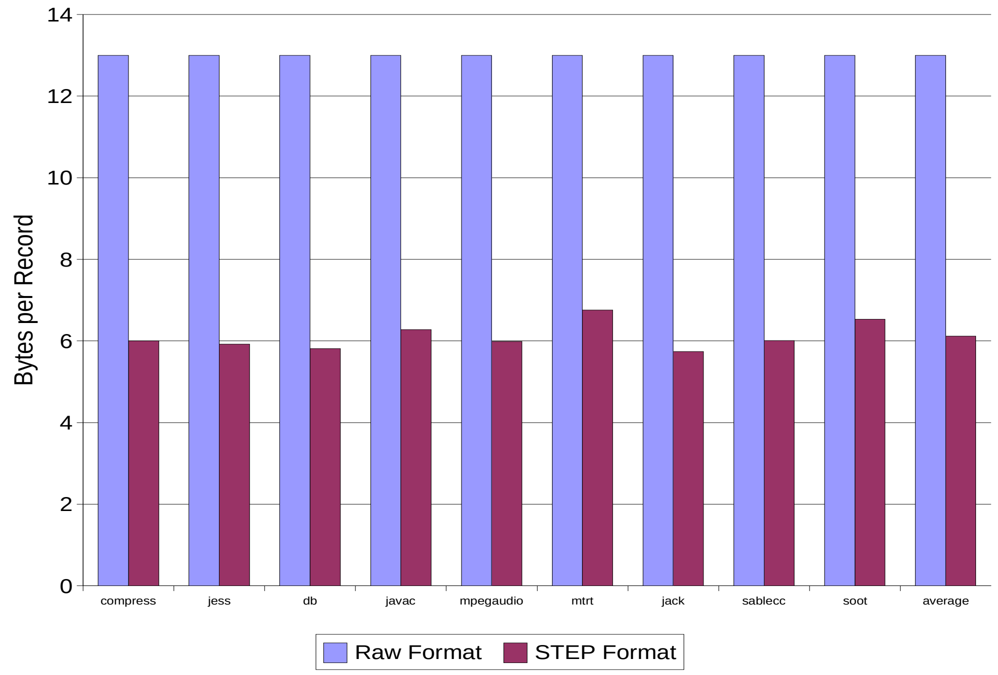


Figure 6.4: Method trace bpr rates

raw format. The results for the method traces also indicate that the Step encoding actually does achieve a nearly optimal byte-level encoding of 6 bytes per record (vs. 13 bpr for the raw format). The encoding is near-optimal in the sense that method-use events, which comprise on average 99% of the data, are composed of five values (record size, record type, thread, method, and target object), where the method ID values require 2 bytes to encode since there are often well over 256 different values.

Since the invoke/field traces were encoded directly in the Step format, an alternate comparison considered a conversion of the traces to a text version of the records. Although the text record format was engineered to be reasonably concise, the frequent occurrence of large text (`string`) field values resulted in a striking contrast when compared to the Step format. On average, the text encoding was nearly 40 times the size of the Step version, with an average record size of 205 bytes (vs. 5.2 bpr for the Step format). These results indicate that decision of Jones *et al.* to modify version 1.2.1 of MetaTF [Jon01] to use text-based encoding should be called into question when considering mostly text-based trace values.

6.2.1 Compression Results

After adjusting the encoding strategies to achieve a reasonably good byte-level encoding of the Step records, the resulting traces were then compressed using the standard `gzip` [GAF] and `bzip2` [Sew] tools, which are freely available for most UNIX platforms. The compressibility of the traces is summarized in table 6.3. Compressibility is expressed as a percentage of the original size of the Step encoding. There are several elements to note in the results. First, the small number of allocations in the `compress` and `mpegaudio` benchmarks skew the compression results for the memory traces since the small traces are more biased by class loading events. Removing the deviants results in average reductions to 6.44% with `gzip` and 2.64% with `bzip2`. The variance in regularity of the method traces is highlighted by the range of compression results, from `db` which is hard to compress to `compress` which is highly compressible. The regularity introduced by eliminating library code events is prominent in the high compressibility of the invoke/field traces.

program	*compression ratios*					
	memory		method		invoke/field	
	`gzip`	`bzip2`	`gzip`	`bzip2`	`gzip`	`bzip2`
`compress`	28.86%	23.82%	1.77%	0.80%	0.97%	0.35%
`jess`	7.54%	2.43%	14.97%	11.51%	1.53%	0.40%
`db`	1.42%	0.71%	26.12%	22.08%	0.82%	0.24%
`javac`	11.06%	5.04%	15.48%	12.91%	4.11%	1.69%
`mpegaudio`	26.73%	21.74%	3.37%	0.91%	2.05%	1.02%
`mtrt`	3.13%	1.34%	19.22%	14.60%	7.88%	2.86%
`jack`	8.15%	3.21%	17.57%	14.74%	2.53%	0.78%
`sablecc`	2.56%	1.13%	1.95%	1.47%	0.66%	0.22%
`soot`	7.88%	3.33%	17.66%	14.67%	3.37%	0.92%
average	10.81%	6.97%	13.12%	10.41%	2.66%	0.94%

Table 6.3: Compression of STEP traces

The results indicate that the block-sorting approach [BW94] of `bzip2` often achieves significantly better compression versus the Lempel-Ziv [ZL77] variant used by `gzip`. However, it is worth noting that empirical observations suggest compressing with `bzip2` often takes an order of magnitude longer that `gzip`. On a dual AMD Athlon 2000MP system `gzip` required roughly 5–10 minutes to compress the traces whereas `bzip2` often required more than an hour. Thus there is a cost/benefit factor to be considered when compressing traces.

Again, comparing the STEP encodings to the raw encodings of the memory and method traces, figures 6.5 and 6.6 show the relative sizes of the compressed raw and STEP traces. In the case of the memory traces, the results are quite dramatic. For example, the `gzip`ped, raw `sablecc` trace is 176MB, while the `gzip`ped STEP version is just 9.6MB. These results are considered further in section 6.2.2. The results are somewhat more variable for the method traces, but still, on average, are an improvement over the naïve encoding. A measurement of the average number bytes per record for the compressed traces is also presented in figures 6.7 and 6.8.

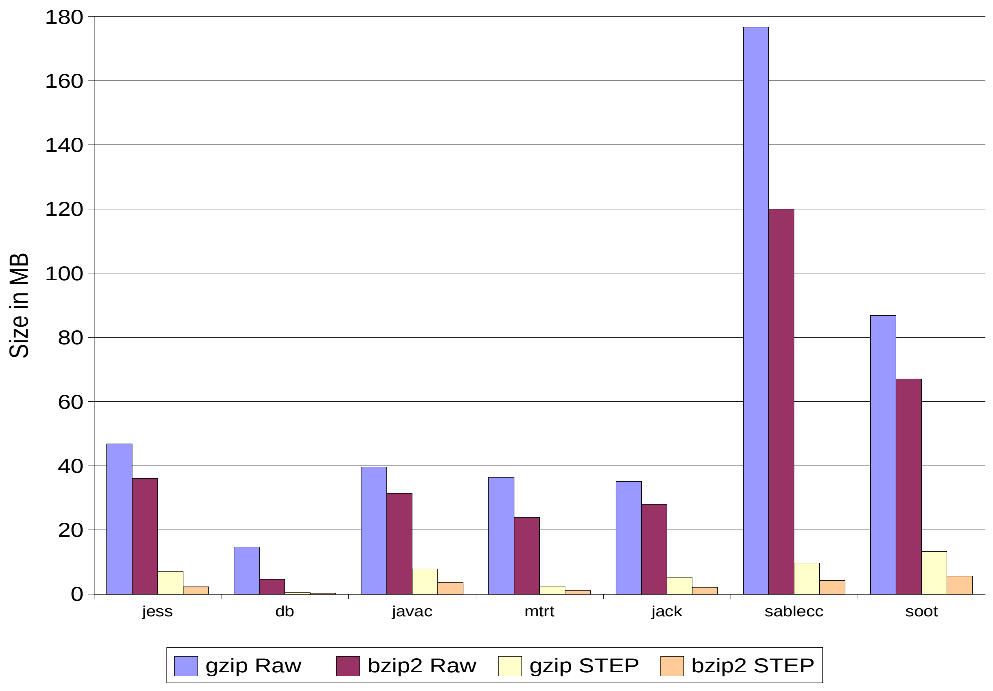


Figure 6.5: Compressed memory trace sizes

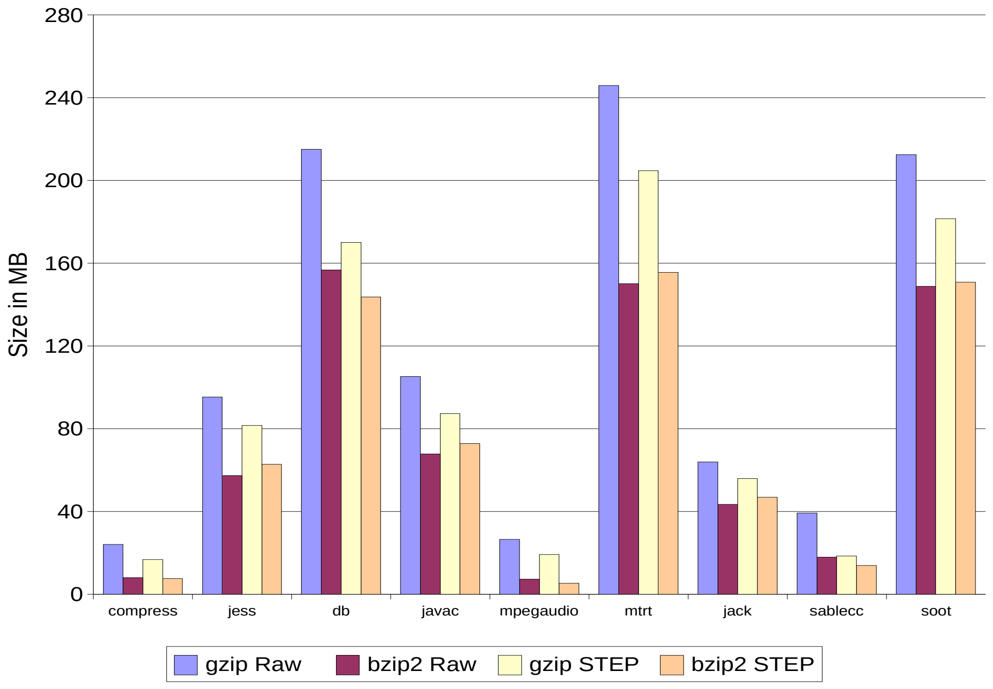


Figure 6.6: Compressed method trace sizes

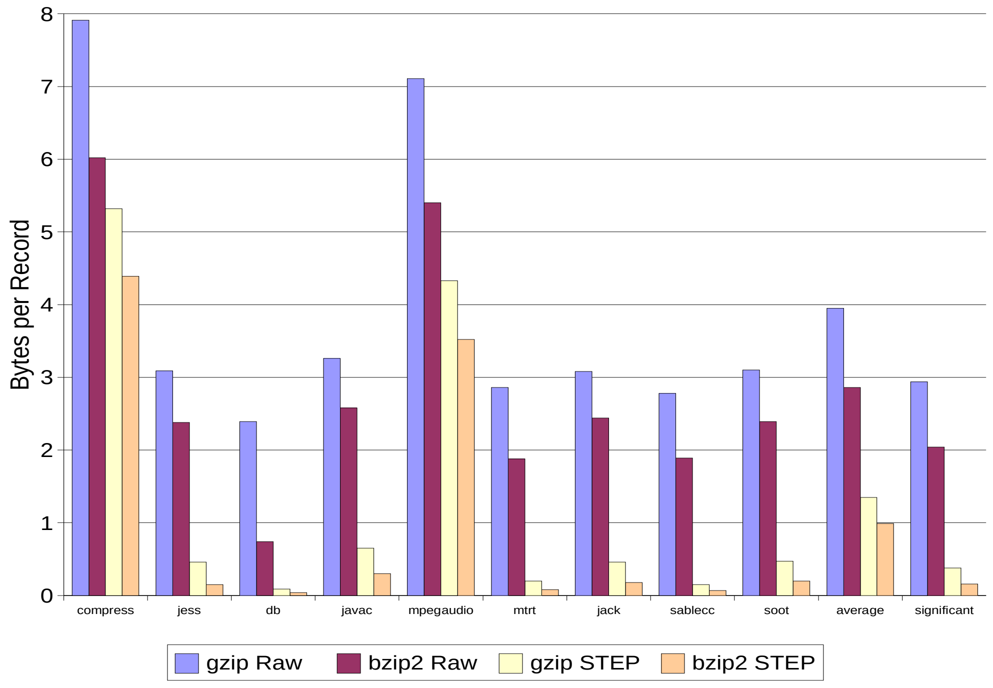


Figure 6.7: Compressed memory trace bpr rates

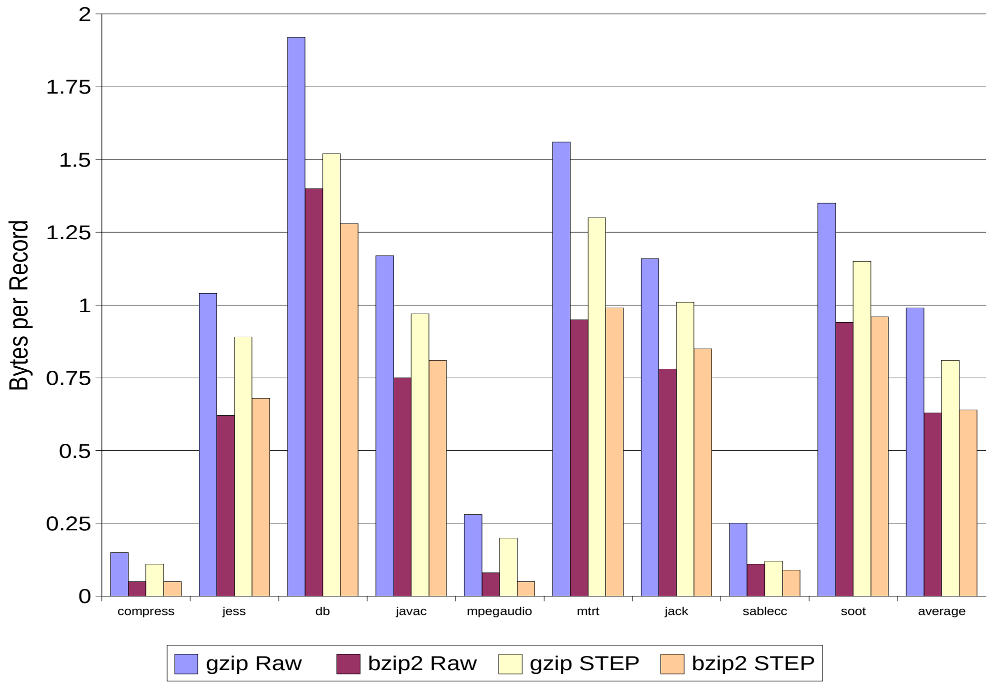


Figure 6.8: Compressed method trace bpr rates

6.2.2 Choosing Appropriate Strategies

Experience has demonstrated that an appropriate choice of encoding strategies can have a significant impact on the both the average record size and the ultimate compressibility of traces encoded with STEP.

Improving Memory Trace Compaction

The memory traces provide an excellent opportunity to study the incremental effect of various strategies. The content of the traces is dominated by the heap allocation and free events, each comprising almost 50% of the data stream. All the records have implicit size and type fields, and also share an explicit environment identifier field (essentially the thread in which the event occurred). The allocation records additionally have fields for the arena of allocation, the type allocated, whether or not the allocation was an array, the size of the allocated object, and the address of the newly allocated object. The free records have one additional field, namely the address of the freed object. STEP-DL for the allocation and free records is shown in figure 6.9.[1] A series of four incremental improvements are applied to improve the encoding of these fields. The reduction in the bytes per record of both the STEP encodings and the compressed STEP encodings are summarized in figures 6.10 and 6.11. (Note, only the traces with a significant number of allocations are considered.)

The baseline measurement considers a version with no intelligent encoding strategies, just appropriate use of the basic integer encoding rules. Already, this encoding offers a 3.5 bpr over the raw version and a 1 bpr improvement in the compressed versions.

The first improvement applies the *repeat* strategy to the environment (thread) identifier values. Since the traces are from single-threaded programs, the 4 bpr improvement is to be expected. The 4-byte address values used for the environment ID are essentially eliminated from the trace.

The second improvement applies the *identifier* strategy to the class ID values in

[1]Some attributes and descriptions have been omitted for clarity. See section C.1 for the complete listing.

```
#define ADDRESS int <property:"address"><encoding:"size=4">

record JVMPI_Event
{
  ADDRESS envId "Environment Identifier";

  ~envId <encoding:"repeat">; // improvement #1
}

record OBJECT_ALLOC "Object Allocation" extends JVMPI_Event
{
  int     arenaId;
  ADDRESS classId;
  int     arrayType <property:"unsigned"><encoding:"size=1">;
  int     size      <property:"unsigned">;
  ADDRESS newObjId;

  ~classId  <encoding:"identifier">; // improvement #2
  ~newObjId <encoding:"delta">;      // improvement #4
}

record OBJECT_FREE "Object Free" extends JVMPI_Event
{
  ADDRESS objId "Freed Object Address";

  ~objId <encoding:"window=8192">; // improvement #3
}
```

Figure 6.9: STEP-DL for object allocation and free records

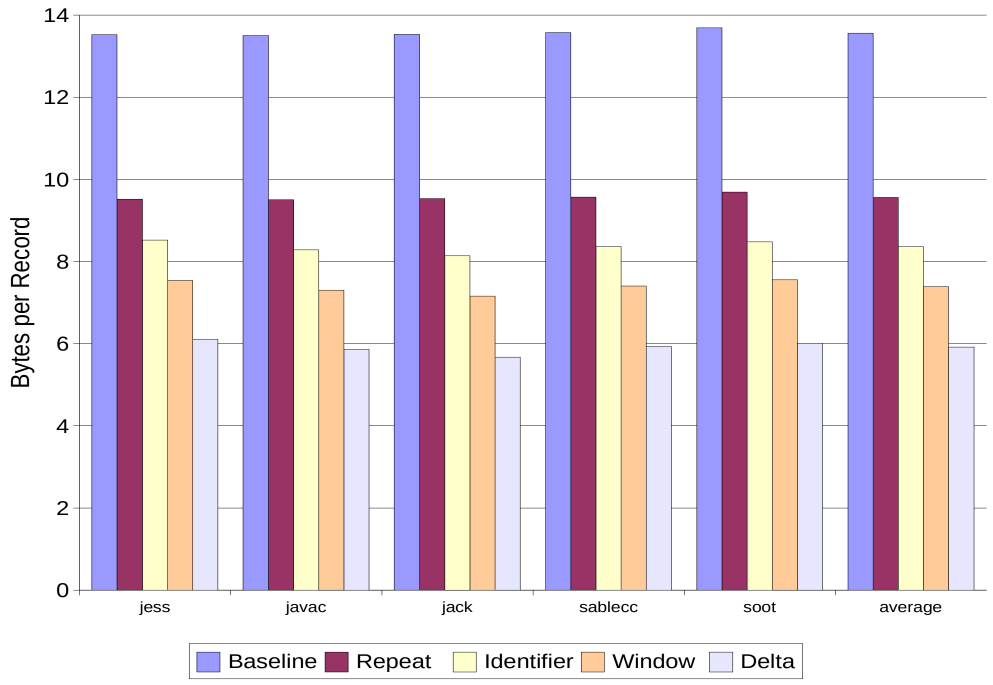


Figure 6.10: Memory trace encodings

the allocation records. The result is another 1.2 bpr improvement, which can actually be read as 2.4 bpr for the allocation records since they comprise roughly half of the stream. In other words, the original 4-byte class ID value (the address of the Java class structure) only requires 1.6 bytes to encode, on average.

The third improvement applies the *window* strategy to the freed object address field of object free records. The technique is surprisingly effective. The strategy used a threshold that restricts the resulting offset values to be encodable in 2 bytes with the variable size "`creep`" integer encoding (i.e., 2×7 usable bits, -1 for the sign $\Rightarrow$ a threshold of $2^{13} = 8192$). The theory is that while the address of objects freed by a Java garbage collector is unlikely to follow any specific pattern, a sweeping or copying collector is likely to free objects in the same memory region at roughly the same time. The conjecture pays off, reducing the encoding by 1 bpr (or 2 bpr for the 50% of records that are free events), thus effectively achieving the desired 2-byte maximum for the encoded version freed object addresses.

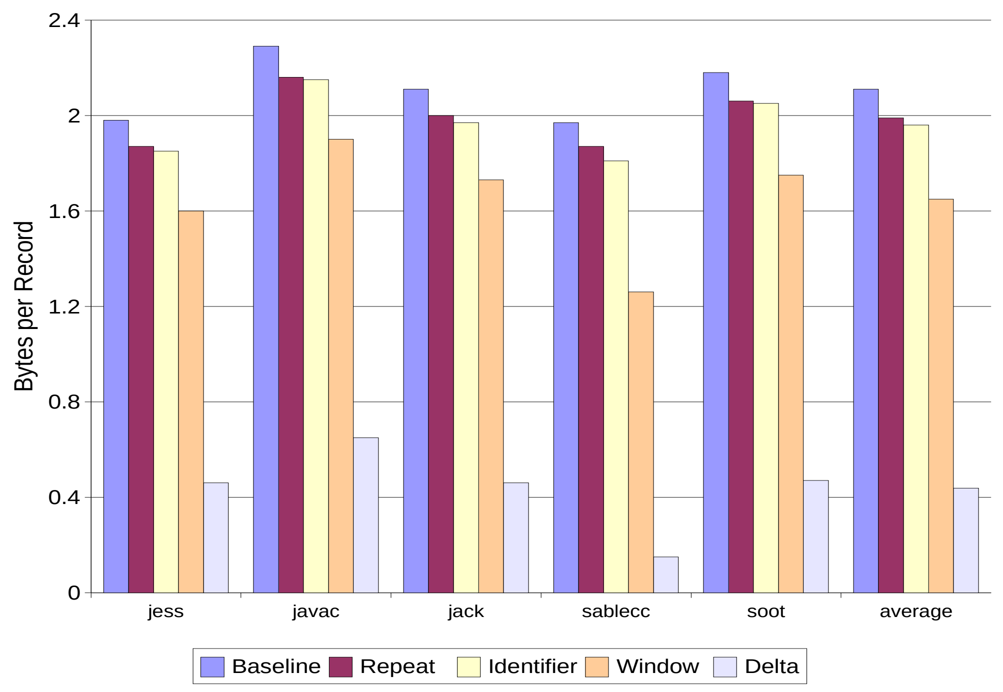


Figure 6.11: Compressed memory trace encodings

The fourth and final improvement yields the most dramatic results. The *delta* strategy is applied to the address values of newly allocated objects. The theory is that an allocator is likely to proceed sequentially through memory allocating objects at steadily increasing memory addresses. Again, the guess pays off and the average record size decreases by another 1.4 bpr (2.8 bpr for allocation records). On its own, the fact that the addresses can be represented with an average of 1.2 bytes is a satisfying result. However the real benefit of the strategy is in the improvement it introduces in the compressibility of the traces. Apparently the delta patterns are significantly more regular than the underlying progression of new object address values. This is an intuitive result since the delta essentially captures the size of the new object (its offset from the last one allocated). A number of researchers have remarked that there are often strong patterns in object allocations sizes.

While the first three improvements do not change any of the patterns in the record values (as the *delta* strategy does in the fourth case), the speculation is that they still produce incremental improvements in the compressibility of the traces by allowing more complete records to fit into the pattern space of a sequential compressor such as `gzip` or `bzip2`. This notion is reinforced by the next set of results that consider improvements to traces from the lone multi-threaded program.

Although the results for the memory traces are not directly comparable to those presented for the HATF format [CJZ00], they do offer some opportunity for a rough comparison with other trace encoding approaches. In both cases the traces are composed of memory management events, and it seems reasonable to assume that allocation and free events dominate the traces in proportions similar to those seen in table 6.5. As with the STEP format, the HATF traces also achieve a near-optimal byte-level encoding. However, the strategies applied to the STEP traces are clearly advantageous with regards to compression, since the STEP encodings are significantly more compressible (with `gzip` ratios of 10.8%, or 6.4% omitting `compress` and `mpegaudio` for the STEP traces vs. 27–33% for the best HATF methods).

Improving Multi-Threaded Trace Compaction

As indicated in section C.1 all of the records used in the memory and method traces derive from a base `JVMPI_Event` type which has one field for the environment (thread) in which the event occurred. For most of the benchmarks, this field is highly redundant since the programs are single-threaded. On the other hand, the `mtrt` benchmark is multi-threaded and warrants a different strategy for encoding the environment ID than is used for the other programs.

Figures 6.12 and 6.13 exhibit two progressions in the encoding of the environment ID field. First, as in the previous section, a baseline version of the encoding is produced where the values are recorded verbatim as a 4-byte addresses. The next version uses the *repeat* strategy in the same way that it is used for the single-threaded traces. Surprisingly, there is reduction of 3 bpr in the STEP encoding, suggesting that while `mtrt` is multi-threaded, long sequences of events occur between context switches. The second improvement applies the *identifier* strategy to the environment ID field, a more natural choice given that there are only a few active threads in the program.

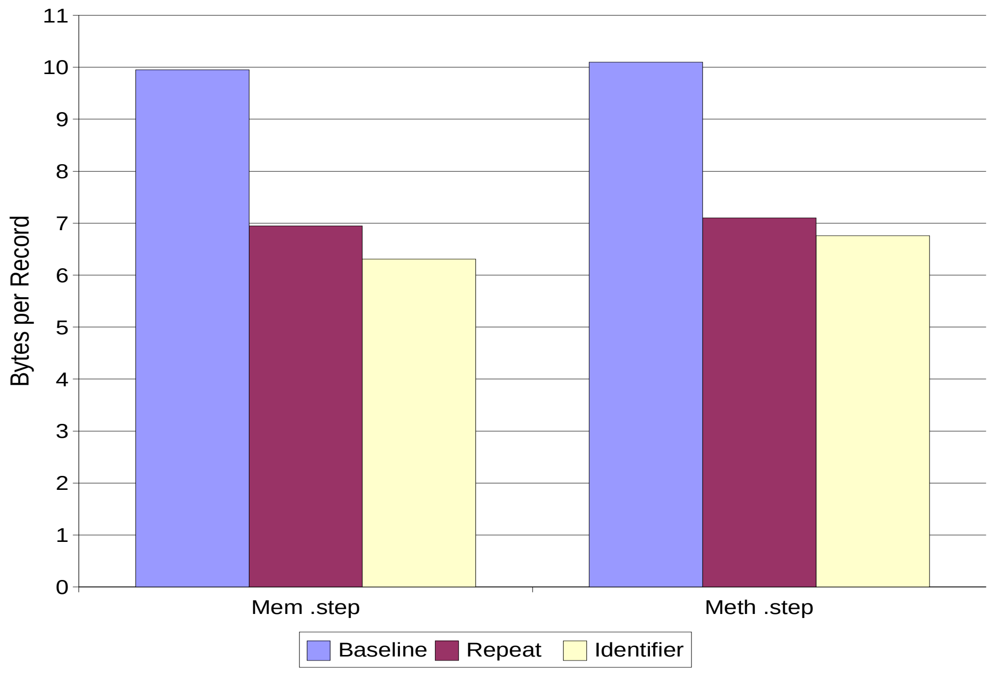


Figure 6.12: Multi-threaded trace encodings

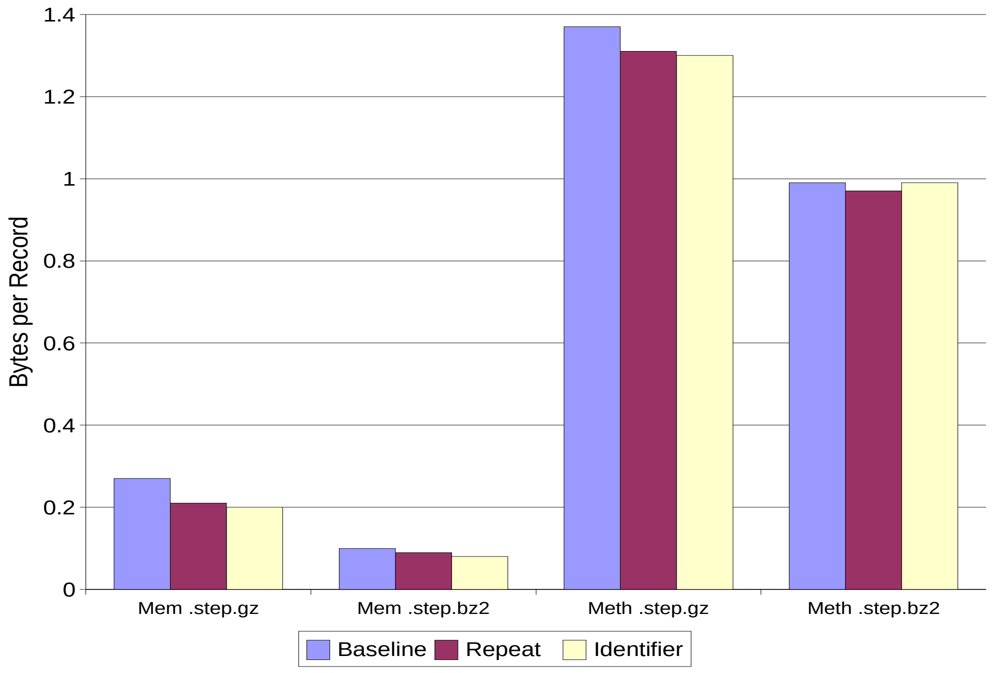


Figure 6.13: Compressed multi-threaded trace encodings

The change leads to another, albeit small, reduction in the average record size.

The strategy changes also improve the compressibility of the traces, as indicated in figure 6.13. The fact that method traces are slightly more compressible when using the *repeat* strategy with `bzip2` lends support to the proposal that squeezing more records into a pattern buffer can result in better compression. The reasoning is that the *repeat* strategy eliminates values whereas the *identifier* strategy compacts them. Thus long sequences without any deviations from the default will locally have a smaller average record size when using the *repeat* variant.

6.2.3 Overhead

Measuring the overhead of encoding STEP traces is not as straightforward as measuring the compactness and compressibility of the traces. However, during the process of encoding the traces a number of informal attempts to monitor execution time, memory use, and meta-data overhead were undertaken.

Encoding Time

Given the variations in hardware, software and system load that factor into the running time of a program, exact timing values are not particularly meaningful. Some empirical observations suggest that converting the raw JVMPI data files to the STEP format requires roughly the same time as does compressing the traces with `gzip`. In the case of the invoke/field traces, it appears that the actual encoding of the trace data increases the execution time by a factor of 5 to 10 times.

Memory Requirements

As expected from the nature of the encoding strategies, the goal of using a bounded, $O(1)$, amount of space is apparently achieved. Table 6.4 indicates the average memory load for decoding traces. The values are obtained by taking an average of the total memory less free memory (after requesting garbage collection), measured at every 10000^{th} record decoded. As the table shows, even decoding the large traces from `mtrt` and `soot` (which exceed 1GB) requires less than 4MB of memory.

program	*average memory use*	
	memory	method
compress	562.34 KB	2720.16 KB
jess	590.32 KB	3795.57 KB
db	562.30 KB	3703.10 KB
javac	688.93 KB	3936.27 KB
mpegaudio	582.41 KB	2790.49 KB
mtrt	573.10 KB	3731.25 KB
jack	570.49 KB	3777.67 KB
sablecc	670.03 KB	3895.02 KB
soot	767.08 KB	4051.07 KB

Table 6.4: Average memory overhead

program	*trace composition*				
	memory			method	
	meta-data	object allocation	object free	meta-data	method enter
compress	4.2%	62.5%	31.6%	$< 0.1\%$	99.9%
jess	0.3%	49.9%	49.9%	0.6%	99.4%
db	0.1%	50.0%	49.9%	2.2%	97.8%
javac	0.3%	49.4%	50.2%	4.0%	96.0%
mpegaudio	2.6%	65.5%	30.6%	$< 0.1\%$	99.9%
mtrt	0.2%	49.9%	50.0%	1.0%	99.0%
jack	0.2%	49.9%	49.8%	0.9%	99.1%
sablecc	0.3%	50.3%	49.4%	0.2%	99.8%
soot	0.2%	52.2%	47.6%	6.1%	93.9%

Table 6.5: Meta-data overhead

Impact of Meta-Data

Informal measurements,[2] summarized in table 6.5, indicate that meta-data often comprise only a small fraction of the data stream. Some notable exceptions are the memory traces from `compress` and `mpegaudio`, where start-up effects are visible in the small traces, and the method traces from `javac` and `soot`, where it is believed that the *cache* strategy used for dispatch targets encounters frequent cache-misses (which result in the generation of meta-events). However, even such poorly behaved traces require no more than 6% of the output stream to be composed of meta-data.

6.3 Analyzing Trace Data

STEP was originally developed as part of a larger framework for tracing and analyzing Java programs. Two systems for visualizing trace data were designed to act as consumer clients of STEP.

EVolve [WWB+02] is a customizable event visualization system designed to reveal a wide variety of trace patterns and characteristics. Figure 6.14 shows several graphs generated with the tool based on trace data from a very simple Java application.

The Java Intermediate Language (JIL) [Eng02] was created to augment Java intermediate representations with information from a number of static, compiler analyses. The JIL representation was later extended to incorporate dynamic program characteristics and present the data along side the relevant static results. A JIL document browser called JIMPLEX is illustrated in figure 6.15. The display shows counts of field and method use for a given class.

[2] The measurements do not account for the nesting of meta-data described in section 5.3.1. Thus, the results over-estimate the proportion of meta-data since meta-data bytes may be counted multiple times as meta-records are unpacked.

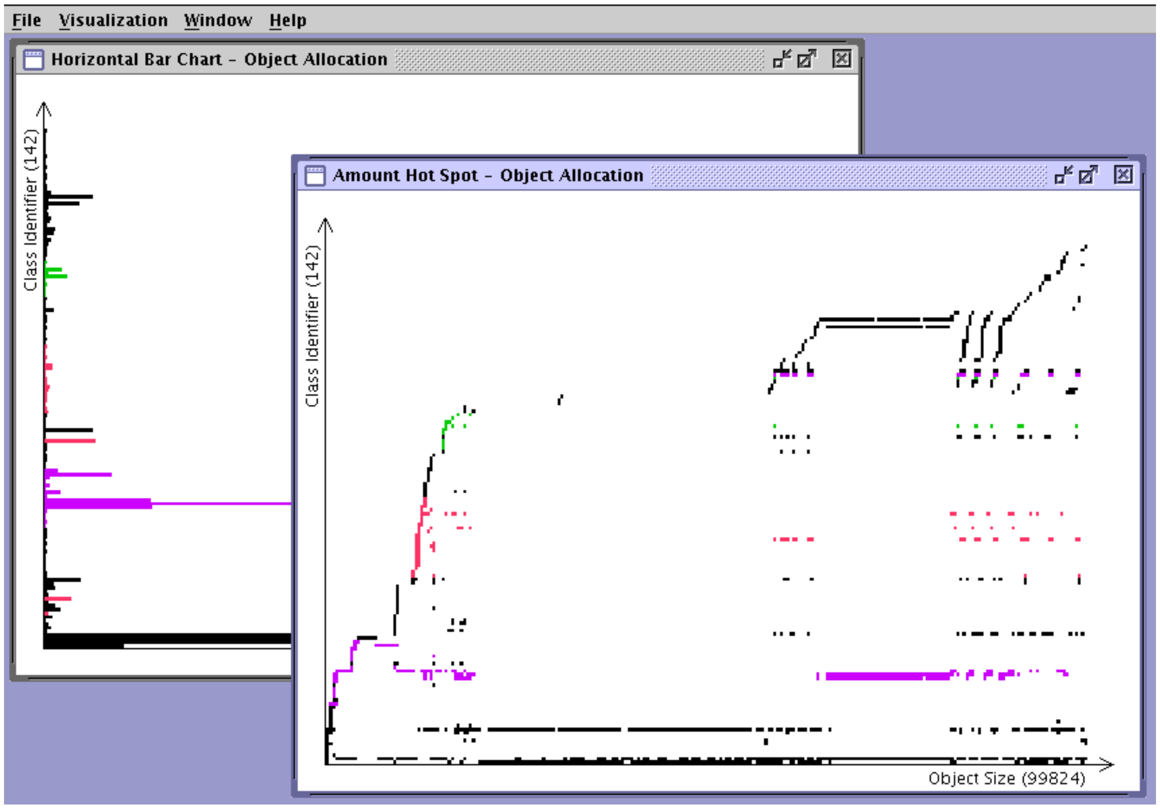
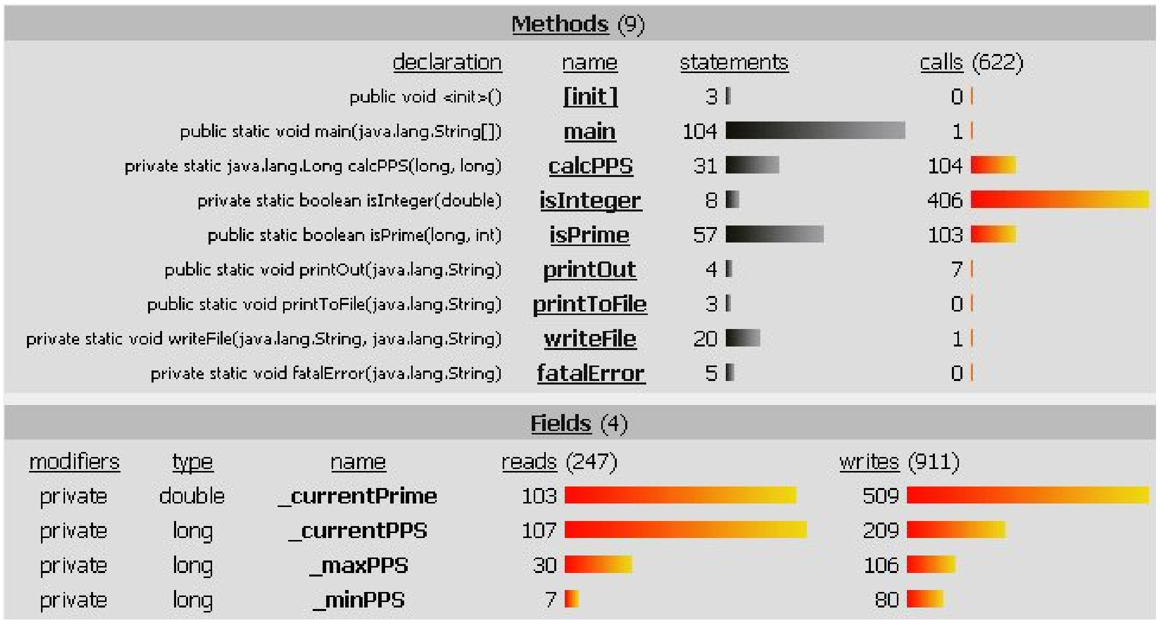


Figure 6.14: Visualizing traces with EVolve



Methods (9)			
declaration	name	statements	calls (622)
public void <init>()	[init]	3	0
public static void main(java.lang.String[])	main	104	1
private static java.lang.Long calcPPS(long, long)	calcPPS	31	104
private static boolean isInteger(double)	isInteger	8	406
public static boolean isPrime(long, int)	isPrime	57	103
public static void printOut(java.lang.String)	printOut	4	7
public static void printToFile(java.lang.String)	printToFile	3	0
private static void writeFile(java.lang.String, java.lang.String)	writeFile	20	1
private static void fatalError(java.lang.String)	fatalError	5	0

Fields (4)				
modifiers	type	name	reads (247)	writes (911)
private	double	_currentPrime	103	509
private	long	_currentPPS	107	209
private	long	_maxPPS	30	106
private	long	_minPPS	7	80

Figure 6.15: Browsing program dynamics with JIMPLEX

Chapter 7
Conclusion

7.1 Summary

This thesis has presented Step, a system designed to facilitate the definition, encoding, and sharing of arbitrary program trace data. The system was motivated by the need to capture the rich variety of events and behaviors exhibited by modern software systems such as Java programs running on a Java Virtual Machine.

The approach uses a new and powerful trace data definition language, Step-DL, that supports features such as type inheritance and generalized annotation. The `stepc` compiler uses the trace definitions to generate Java class definitions for interfacing with the included encoding engine. The design of Step builds on a number of existing approaches to provide a robust and effective solution for encoding general trace data.

The utility of the system was evaluated by encoding a variety of trace data from a range of well-known Java benchmark programs. The discussion considered both the compactness and compressibility of the encoding, as well as the overhead of operating the system.

7.2 A Success?

The preceding chapters have illustrated how the design and implementation of STEP have addressed the requirements outlined section 1.2. The STEP Definition Language is capable of expressing a wide range of trace data types and the annotation features of the language provide a structured method for documenting the data. The encoding architecture that supports the language implements a flexible and portable file format, and offers a simple input/output interface to clients by encapsulating the details of the encoding process. STEP-DL supports extensibility in the form of inheritance and attribute refinement, while the encoding architecture uses modular encoding policies to allow extension and experimentation with new reduction strategies. In combination with standard compression tools, the encoding strategies included with the system provide an efficient method for creating particularly compact trace representations. In several examples, the compressed STEP encodings were less than 1% of the size of equivalent naïve encodings.

The STEP system can be considered a success in the sense that it meets its design goals of flexibility and interoperability while still producing an encoding format that is competitive with other approaches, both in terms of bytes-per-record size and overall compressibility.

7.3 Future Directions

STEP was conceived as an openly extensible framework, and a variety of future enhancement possibilities exist. The STEP-DL attribute partitioning described in chapter 3 was specifically designed so that developers of trace production and consumption tools could add their own attribute groups and extend the `stepc` compiler to generate additional interface components. The current set of encoding strategies is effective but by no means complete or optimal. The *identifier* and *cache* strategies could benefit from more sophisticated implementations, and other refinements on the removal and computational techniques would be a welcome addition. Finally, the most obvious extension would be to integrate sequential compression as part of the STEP encoding.

A number of researchers have shown that the SEQUITUR hierarchical inference algorithm is particularly well suited to compressing trace data. A space-restricted version of the algorithm [NMW98] would be an excellent addition to STEP.

Appendix A
On-line Step Resources

Step is software made publicly available, at no cost, under the terms of the GNU General Public License (version 2, or later). The source code for Step, compiled Java binaries, `javadoc` API documentation, and other related documents (including a copy of this thesis) are available from McGill University's Sable Research Group at: http://www.sable.mcgill.ca/step/.

Appendix B
Step-DL Attribute Groups

B.1 `encoding` Attributes

The `encoding` attribute group is the most prominent and integral to the Step system. The encoding techniques fall into 3 categories: general regularity strategies, which may be applied to any data value; specific regularity strategies, which target a particular data type; and simple, property based rules. The precedence of encoding strategies is based on these three categories. First, the most recent general strategy is applied. If an irregular value is encountered, the next available rule is used: either a targeted strategy or basic rule. Again, if a targeted strategy encounters an irregular value, it defers to the most recently defined basic rules. If no basic rule is given, the encoder factory assigns certain default rules. When a strategy must defer to its subordinate, the irregular value is indicated through the use of a meta-event record.

B.1.1 General Strategies

identifier

This strategy is applied when the values are expected to derive from a relatively small, fixed distribution. As new values are encountered, they are written as $< value, ID >$ pairs. All subsequent occurrences of the value result in only the ID being written to the trace. The decoder reads the IDs and converts them to values based on the initial mapping.

cache=$size$

The cache strategy is similar to the identifier approach. $size$ values are kept in a table. When a value's equivalent exists in the table, only the table index is encoded. New values are placed in the table on a rotational basis. Generally, the cache strategy is only useful when the size of the value distribution is too large for the identifier strategy to be effective.

constant

This strategy assumes that all values for the given field are the same. The value is only written for the initial occurrence. If any subsequent value differs from the initial value, an error is generated.

default

This strategy is similar the the constant strategy, but deviant values are allowed and are signalled with meta-data. This strategy is effective for fields which almost always have the same value.

repeat

The repeat strategy is similar to the default strategy, except that deviant values change the base value. The strategy is useful for data with long repeating sequences, or when the best default is not the initial value.

B.1.2 Integer Strategies

Integer (`int`) field values may be encoded using a variety of targeted strategies and basic rules.

Targeted Strategies

delta | **delta=**$threshold$

This strategy assumes that values are arithmetically "close" to the previous value, and only encodes the difference. In the case when a *threshold* is specified, absolute delta values greater than the threshold cause the deviant value to be transmitted with a meta-event instead. This is a version of Samples' difference technique [Sam89]. The strategy is useful for data such as allocation addresses where the values often exhibit a sequentially increasing pattern.

stride=*increment*

This strategy assumes that values occur with a regular increment from the previous value. In such cases, nothing is written to the trace and the decoder reconstructs the value from the previous value and the increment.

offset | **offset=***base*

This strategy assumes that values are clustered about a given base value and that it is more economical to transmit the difference from the base than the absolute value. If no *base* is specified, the initial value is used as the base.

window=*threshold*

This strategy can be viewed as an adaptive version of the offset strategy. The initial value is used as the base, and subsequent values are encoded as the offset from the initial value. If the difference exceeds the given threshold, the base is shifted.

Basic Rules

size=*fixed* | *start***..** | *min***+** | **creep**

The number of bytes used for an integer value (i.e., its size) can be defined in a number of ways. The rule may state that values always use the same fixed number of bytes. The rule may begin using a particular size, and then grow to use more bytes as larger values are encountered. The resizing may be elastic, in the case where values requiring more than the minimum are rare. Finally, a variable size encoding may be used, where the high bit of each byte is used to signal whether more bytes should be read. The default rule is to use the variable size "`creep`" rule.

signed | unsigned

Values that are always ≥ 0 are indicated as unsigned. The default is to assume signed integer values. Unsigned values are also implied by the `property` attributes "`unsigned`" and "`address`", and is often omitted in favor of the `property` version.

B.1.3 String Rules

String (`string`) types currently have just a single basic rule which states character encoding of the string in bytes.

charset=UTF-8 | US-ASCII | ...

The encoding of `string` values parallels Java's string encoding rules. The default rule is to encode values using the UTF-8 character set.

B.1.4 Record Rules

type=variable | default | constant

Since Step supports inheritance of record types, it is possible that sub-types may be used in the place of a field's defined type. To avoid object slicing, the record encoder must indicate the type of the specific value. The strategy for tagging the type of a record value assumes that either a) the types are uniform, in which case the `default` or `constant` options are appropriate, or that b) a number of different types are used, in which case the `variable` option (based on the `identifier` strategy) is a better choice.

General Notes

- `string`, `data` and array objects write a length field when encoded. The length encoding strategy may be adjusted with a relative modifier (e.g., `~x.length <encoding:"default">`).

- The strategy for elements of an array field can also be changed by applying a modifier to the `element` field.

B.2 `property` Attributes

B.2.1 Record Properties

event | entity

STEP does not make an explicit distinction between records that represent an event or those that are used as auxiliary structures to describe complex entities. Indicating whether a record signifies and event or entity is useful for tools that consume the trace data to distinguish the two forms.

B.2.2 Integer Properties

signed | unsigned

Values that are always ≥ 0 are indicated as unsigned. The default is to assume signed integer values.

address

Address values imply the "`unsigned`" property and also indicate a memory coordinate.

boolean

Values that should be converted to an appropriate `boolean` representation, where non-zero $\Rightarrow$ `true` and zero $\Rightarrow$ `false`.

Appendix C
Step-DL Examples

C.1 Step-DL Definitions for JVMPI Data

This section presents a listing of the Step-DL definitions used to encode the 'method' and 'memory' traces discussed in chapter 6. The definitions parallel the standard JVMPI event record definitions.

```
#define ADDRESS int <property:"address"><encoding:"size=4">

package example {

  package adapt {

    record AdaptHeader {
      int   magic <property:"unsigned"><encoding:"size=4">;
      data  options;
    }

  }

  package jvmpi {

    record JVMPI_Event {
      <property:"event">

      ADDRESS  envId "Environment Identifier";
  ~envId <encoding:"repeat">;  // nearly constant for single-threaded apps
  //~envId <encoding:"identifer">;  // variable for multi-threaded apps
}

record JVM_Entity {
  <property:"entity">
}

record Method "JVM Method" extends JVM_Entity {
  string   methodName "Method Name";
  string   signature  "Method Signature";
  int      startLine  "First Source Line";
  int      endLine    "Last Source Line";
  ADDRESS  methodId   "Method Identifier";
}

record Field "JVM Field" extends JVM_Entity {
  string  fieldName "Field Name";
  string  signature "Field Signature";
}

record Thread "JVM Thread" extends JVM_Entity {
  string   threadName  "Thread Name";
  string   group       "Thread Group";
  string   parent      "Parent Thread";
  ADDRESS  threadId    "Thread Identifier";
  ADDRESS  threadEnvId "New Thread's Environment Identifier";
}

// -- Definition Events ------------------------------------------------------

record ClassEvent extends JVMPI_Event {
  ADDRESS  classId "Class Identifier";
}

record CLASS_LOAD "Class Load" extends ClassEvent {
  "a class was loaded into the VM"

  string    className "Class Name";
  string    source    "Source File";
  int       numInterfaces;
  Method[]  methods;
  Field[]   staticFields, instanceFields;
}

record CLASS_UNLOAD "Class Unload" extends ClassEvent {
  "a class was unloaded from the VM"
}
```

```
// -- Memory Management ----------------------------------------------------

record OBJECT_ALLOC "Object Allocation" extends JVMPI_Event {
  "an object was allocated on the heap"

  int      arenaId   "Allocation Arena Identifier";
  ADDRESS  classId   "Allocated Type Identifier";
  int      arrayType "Array Type"      <property:"unsigned"><encoding:"size=1">;
  int      size      "New Object Size" <property:"unsigned">;
  ADDRESS  newObjId  "New Object Address";

  ~classId  <encoding:"identifier">;
  ~newObjId <encoding:"delta">; // capture patterns in allocation size
}

record OBJECT_FREE "Object Free" extends JVMPI_Event {
  "a heap object was freed"

  ADDRESS  objId "Freed Object Address";

  ~objId <encoding:"window=8192">; // gc freeing objects in nearby locations
}

record ArenaEvent extends JVMPI_Event {
  int  arenaId "Allocation Arena Identifier";
}

record ARENA_NEW "Arena New" extends ArenaEvent {
  "an allocation arena was created"

  string  arenaName "New Arena Name";
}

record ARENA_DELETE "Arena Delete" extends ArenaEvent {
  "an allocation arena was deleted"
}

record GC_START "Garbage Collection: Started" extends JVMPI_Event {
  "a garbage collection cycle has begun"
}

record GC_FINISH "Garbage Collection: Finished" extends JVMPI_Event {
  "a garbage collection cycle has ended"

  int <property:"unsigned">
    usedObjects,
    usedObjectSpace,
    usedTotalSpace;
}

// -- Execution ----------------------------------------------------------

record MethodEvent extends JVMPI_Event {
  ADDRESS  methodId "Method Identifier";

  ~methodId <encoding:"identifier">;
}

record METHOD_ENTRY extends MethodEvent {
}

record METHOD_ENTRY2 "Method Entry" extends MethodEvent {
  ADDRESS  targetObjId "Target Object Address";

  ~targetObjId <encoding:"cache=65536">;
}

record METHOD_EXIT "Method Exit" extends MethodEvent {
}

record JVM_INIT_DONE "JVM Initialization Complete" extends JVMPI_Event {
  "the JVM has finished its initialization phase"
}

record JVM_SHUT_DOWN "JVM Shut Down" extends JVMPI_Event {
  "the JVM is exiting"
}

// -- Concurrency --------------------------------------------------------

record THREAD_START "Thread Start" extends JVMPI_Event {
  "a new thread of execution was started"

  Thread  newThread "New Thread";
}

record THREAD_END "Thread End" extends JVMPI_Event {
  "a thread of execution ended"
}

record MonitorEvent extends JVMPI_Event {
  ADDRESS  lockObjId "Lock Object Address";

  ~lockObjId <encoding:"identifier">; // limited number of locks? cache?
}
```

```
record MONITOR_CONTENDED_ENTER "Monitor Contend: Enter"
  extends MonitorEvent
{
  "attempt to acquire contended monitor lock"
}

record MONITOR_CONTENDED_ENTERED "Monitor Contend: Lock Acquired"
  extends MonitorEvent
{
  "acquired contended monitor lock"
}

record MONITOR_CONTENDED_EXIT "Monitor Contend: Exit"
  extends MonitorEvent
{
  "released contended monitor lock"
}

record MonitorWaitEvent extends MonitorEvent {
  int  timeout "Wait Timeout" <property:"unsigned">;
}

record MONITOR_WAIT "Monitor Wait: Begin" extends MonitorWaitEvent {
  "wait for a monitor lock"
}

record MONITOR_WAITED "Monitor Wait: End" extends MonitorWaitEvent {
  "done waiting for a monitor lock"
}

record RawMonitorEvent extends JVMPI_Event {
  string  monitorName "Monitor Name";
  int     monitorId   "Monitor Identifier" <property:"unsigned">;

  ~monitorName <encoding:"identifier">;
}

record RAW_MONITOR_CONTENDED_ENTER "Raw Monitor Contend: Enter"
  extends RawMonitorEvent
{
  "attempt to acquire contended raw monitor lock"
}

record RAW_MONITOR_CONTENDED_ENTERED "Raw Monitor Contend: Lock Acquired"
  extends RawMonitorEvent
{
  "acquired contended raw monitor lock"
```

```
    }

    record RAW_MONITOR_CONTENDED_EXIT "Raw Monitor Contend: Exit"
      extends RawMonitorEvent
    {
      "released contended raw monitor lock"
    }

  } // jvmpi

} // example
```

C.2 Step-DL **Definitions for Java Run-Time Data**

This section presents a listing of the Step-DL definitions used to encode the 'invoke/field' traces discussed in chapter 6. The definitions include a number of static Java entities and run-time events.

```
#define TYPE_ID   example.java.Type       <encoding:"identifier">
#define METHOD_ID example.java.Method     <encoding:"identifier">
#define SITE_ID   example.java.MethodSite <encoding:"identifier">
#define FIELD_ID  example.java.Field      <encoding:"identifier">

package example {

  package java {

    record JavaEntity {
      <property:"entity">
    }

    record Thread extends JavaEntity {
      string  name;
      string  group;
    }

    record Type "Java Type" extends JavaEntity {
      string  name;
    }

    record Method extends JavaEntity {
      string  signature;
      Type    declaringClass;
    }

    record MethodSite "Method Site" extends JavaEntity {
      Method  method;
      int     number <property:"unsigned"><encoding:"size=1+">;
    }

    record Field extends JavaEntity {
      string  name;
      Type    type;
      Type    declaringClass;
      int     isStatic <property:"boolean"><encoding:"size=1">;
    }
```

```
package rt {

  record RuntimeEvent "Runtime Event" {
    <property:"event">

    example.java.Thread  thread;

    ~thread <encoding:"repeat">;        // for single-threaded apps
    //~thread <encoding:"identifier">;  // for multi-threaded apps
  }

  record Allocation extends RuntimeEvent {
    TYPE_ID  allocatedType;
    SITE_ID  allocationSite;
  }

  record MethodEvent extends RuntimeEvent {
    METHOD_ID  method;
  }
  record MethodEnter extends MethodEvent {
  }
  record MethodExit extends MethodEvent {
    SITE_ID  exitSite;
  }

  record FieldAccess extends RuntimeEvent {
    FIELD_ID  field;
    SITE_ID   accessSite;
  }
  record FieldRead  extends FieldAccess {}
  record FieldWrite extends FieldAccess {}

  record Invoke extends RuntimeEvent {
    METHOD_ID  method;
    METHOD_ID  callerMethod;
    SITE_ID    callSite;
  }

  record DispatchInvoke extends Invoke {
    TYPE_ID  targetType;
  }
  record InterfaceInvoke extends DispatchInvoke {}
  record VirtualInvoke   extends DispatchInvoke {}
  record StaticInvoke    extends Invoke {}
  record SpecialInvoke   extends Invoke {}

} // rt
```

```
  } // java

} // example
```

Appendix D
Benchmark Program Descriptions

The programs used to collect the traces discussed in chapter 6 are briefly summarized below. The first six are taken from the well known SPECjvm98 [SPEC98] suite of Java benchmarks. The remaining two (`sablecc` and `soot`) are modified versions of those found in the Ashes [VRSa] benchmark suite. The descriptions of the SPEC programs are adapted from those included with the benchmark bundle.

`_201_compress`

A modified Lempel-Ziv compression method (LZW). Basically, the program finds common substrings and replaces them with a variable size code. The method is deterministic, and can be done on the fly. Thus, the decompression procedure needs no input table, but tracks the way the table was built.

`_202_jess`

JESS, the Java Expert Shell System, is based on NASA's CLIPS expert shell system. In simplest terms, an expert shell system continuously applies a set of if-then statements, called rules, to a set of data, called the fact list. The benchmark workload solves a set of puzzles commonly used with CLIPS. To increase running time, the benchmark problem iteratively asserts a new set of facts representing the same puzzle but with different literals. The older sets of facts are not retracted. Thus, the inference engine must search through progressively larger rule sets as execution proceeds.

`_209_db`

Performs multiple database functions on a memory resident database. Reads in a 1 MB file which contains records with names, addresses, and phone numbers of entities and a 19KB file called `scr6` which contains a stream of operations to perform on the records in the file. The program loops and reads commands till it hits the 'q' (quit) command. The commands performed on the file include, among others:

- add an address
- delete and address
- find an address
- sort addresses

`_213_javac`

This is the Java compiler from the Sun's JDK 1.0.2. [No further details are provided.]

`_222_mpegaudio`

This is an application that decompresses audio files that conform to the ISO MPEG Layer-3 audio specification. As this is a commercial application only obfuscated class files are available. The workload consists of about 4MB of audio data.

`_227_mtrt`

This is a variant of `_205_raytrace`, a raytracer that works on a scene depicting a dinosaur, where two threads each render the scene in the input file time-test model, which is 340KB in size.

`_228_jack`

A Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS). This is an early version of what is now called JavaCC. The workload consists of a file named `jack.jack`, which contains instructions for the generation of `jack` itself. This is fed to `jack` so that the parser generates itself multiple times.

`sablecc`

An object-oriented compiler generator. The tool generates classes to represent lexical and grammatical units, state machines for scanning and parsing the given syntax, and methods for generating and traversing an abstract syntax tree (AST) representation of an input sequence. This benchmark uses SableCC version 2.16.2 and executes on a grammar for version 1.1 of the Java programming language. SableCC is available for download at: http://www.sablecc.org.

`soot`

A Java bytecode transformation and optimization framework. The framework provides a number of static program optimizations that are applied to a 3-address intermediate representation called Jimple. This benchmark uses Soot version 1.2.4.dev.12 and optimizes several large class files from the framework itself (requiring many other context classes to be analyzed in the process). Soot is available for download at: http://www.sable.mcgill.ca/soot/.

Bibliography

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[ABD+97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, October 1997, pages 1–14. ACM Press, New York, New York, USA.

[ACL] The Adaptive Computation Lab, McGill University. <http://www.cs.mcgill.ca/acl/> .

[AH90] Anant Agarwal and Minor Huffman. Blocking: Exploiting spatial locality for trace compaction. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Boulder, Colorado, USA, May 1990, pages 48–57. ACM Press, New York, New York, USA.

[BDE$^+$01] Rhodes Brown, Karel Driesen, David Eng, Laurie Hendren, John Jorgensen, Clark Verbrugge, and Qin Wang. STOOP: The Sable toolkit for object-oriented profiling. Sable Technical Report 2001-2, Sable Research Group, McGill University, Montréal, Québec, Canada, November 2001.

[BDE$^+$02] Rhodes Brown, Karel Driesen, David Eng, Laurie Hendren, John Jorgensen, Clark Verbrugge, and Qin Wang. STEP: A framework for the efficient encoding of general trace data. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Charleston, South Carolina, USA, November 2002, pages 27–34. ACM Press, New York, New York, USA.

[BL94] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, 1994.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Object Technology Series. Addison-Wesley, Reading, Massachusetts, USA, 2nd edition, 1994.

[Bor] Borland Software Corp. OptimizeIt^{TM} suite. <http://www.borland.com/optimizeit/> .

[BPF91] Jeffrey C. Becker, Arvin Park, and Matthew Farrens. An analysis of the information content of address reference streams. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, Albuquerque, New Mexico, USA, November 1991, pages 19–24. ACM Press, New York, New York, USA.

[Bro] Rhodes Brown. STEP: Extensible program trace encoding. <http://www.sable.mcgill.ca/step/> .

[BW94] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital SRC Research, Palo Alto, California, USA, May 1994.

[CDL99] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, USA, May 1999, pages 13–24. ACM Press, New York, New York, USA.

[CF00] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Boston, Massachusetts, USA, January 2000, pages 54–66. ACM Press, New York, New York, USA.

[Chi01] Trishul 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)*, Snowbird, Utah, USA, June 2001, pages 191–202. ACM Press, New York, New York, USA.

[CHL99] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, USA, May 1999, pages 1–12. ACM Press, New York, New York, USA.

[CJZ00] Trishul Chilimbi, Richard Jones, and Benjamin Zorn. Designing a trace format for heap allocation events. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, Minneapolis, Minnesota, USA, October 2000, pages 35–49. ACM Press, New York, New York, USA.

[DA] Karel Driesen and the Adaptive Computation Lab, McGill University. Plumber: A simulation framework for hardware and software based prediction. <http://www.cs.mcgill.ca/acl/plumber/> .

[DDHV02] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for compiler developers. Sable Technical Report 2002-11, Sable Research Group, McGill University, Montréal, Québec, Canada, November 2002.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the ACM SIGSOFT-SIGPLAN/IEEE Computer Society International Conference on Software Engineering (ICSE)*, Los Angeles, California, USA, May 1999, pages 213–224. IEEE Computer Society Press, Los Alamitos, California, USA.

[Eln99] E. N. Elnozahy. Address trace compression through loop detection and reduction. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Atlanta, Georgia, USA, May 1999, pages 214–215. ACM Press, New York, New York, USA.

[Eng] David Eng. Java intermediate language (JIL). <http://www.sable.mcgill.ca/jil/> .

[Eng02] David Eng. Combining static and dynamic data in code visualization. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Charleston, South Carolina, USA, November 2002, pages 43–50. ACM Press, New York, New York, USA.

[EPCP98] Marius Evers, Sanjay J. Patel, Robert S. Chappell, and Yale N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, Barcelona, Spain, June 1998, pages 52–61.

[FFW98] Eitan Federovsky, Meir Feder, and Shlomo Weiss. Branch prediction based on universal data compression algorithms. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, Barcelona, Spain, June 1998, pages 62–72.

[FG96] Alexander Fox and Thomas Grün. Compressing address trace data for cache simulations. Technical Report SFB 124, D4, Universität des Saarlandes, Saarbrücken, Germany, July 1996.

[FT00] Robert Fitzgerald and David Tarditi. The case for profile-directed selection of garbage collectors. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, Minneapolis, Minnesota, USA, October 2000, pages 111–120. ACM Press, New York, New York, USA.

[GAF] Jean-loup Gailly, Mark Adler, and the Free Software Foundation, Inc. The gzip (GNU zip) compression tool. <http://www.gzip.org/> .

[Gag] Étienne M. Gagnon. SableCC: The Sable research group's compiler compiler. <http://www.sablecc.org> .

[Gag98] Étienne Gagnon. SableCC, an object-oriented compiler framework. Master's thesis, McGill University, Montréal, Québec, Canada, March 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, USA, 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[Har00] Timothy L. Harris. Dynamic adaptive pre-tenuring. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*

(ISMM), Minneapolis, Minnesota, USA, October 2000, pages 127–136. ACM Press, New York, New York, USA.

[HD77] D. W. Hammerstrom and E. S. Davidson. Information content of CPU memory referencing behavior. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, 1977, pages 184–192.

[HKWZ00] Thorna O. Humphries, Artur W. Klauser, Alexander L. Wolf, and Benjamin G. Zorn. The POSSE trace format version 1.0. Technical Report CU-CS-897-00, Department of Computer Science, University of Colorado, Boulder, Colorado, USA, January 2000.

[HMVR95] Matthew Haines, Piyush Mehrotra, and John Van Rosendale. SmartFiles: An OO approach to data file interoperability. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Austin, Texas, USA, October 1995, pages 453–466. ACM Press, New York, New York, USA.

[IBM] IBM Research. Jinsight. <http://www.research.ibm.com/jinsight/> .

[ISO86] Standard Generalized Markup Language (SGML). ISO Standard 8879, International Organization for Standardization, 1986.

[ISO90] Abstract Syntax Notation One (ASN.1). ISO Standard 8824, International Organization for Standardization, 1990.

[JH94] Eric E. Johnson and Jiheng Ha. PDATS lossless address trace compression for reducing file size and access time. In *Proceedings of the IEEE International Conference on Computers and Communications*, Phoenix, Arizona, USA, April 1994, pages 213–219. IEEE Press.

[JHBZ01] Eric E. Johnson, Jiheng Ha, and M. Baqar Zaidi. Lossless trace compression. *IEEE Transactions on Computers*, 50(2):158–173, February 2001.

[Joh] Eric E. Johnson. The NMSU tracebase. <http://tracebase.nmsu.edu/tracebase.html> .

[Joh97] Mark Stuart Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, The University of Texas at Austin, 1997.

[Joh99] Eric E. Johnson. PDATS II: Improved compression of address traces. In *Proceedings of the IEEE International Performance, Computing and Communications Conference*, Scottsdale, Arizona, USA, February 1999, pages 72–78. IEEE Press.

[Jon01] Richard Jones. *Specifying trace formats: MetaTF 1.2.1.* Canterbury, Kent, UK, March 2001.

[JSB97] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *Proceedings of the ACM SIGSOFT-SIGPLAN/IEEE Computer Society International Conference on Software Engineering (ICSE)*, Boston, Massachusetts, USA, May 1997, pages 360–370. ACM Press, New York, New York, USA.

[KSW99] Scott F. Kaplan, Yannis Smaragdakis, and Paul R. Wilson. Trace reduction for virtual memory simulations. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Atlanta, Georgia, USA, May 1999, pages 47–58. ACM Press, New York, New York, USA.

[Lar99] James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, USA, May 1999, pages 259–269. ACM Press, New York, New York, USA.

[LH02] Ondrej Lhoták and Laurie Hendren. Run-time evaluation of opportunities for object inlining in Java. In *Proceedings of the ACM SIGPLAN*

Java Grande Conference, Seattle, Washington, USA, November 2002, pages 175–184. ACM Press.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, La Jolla, California, USA, June 1995, pages 291–300. ACM Press, New York, New York, USA.

[LY99] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[MN96] Scott Mahlke and Balas Natarajan. Compiler synthesized dynamic branch prediction. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, Paris, France, December 1996, pages 153–164. IEEE Computer Society Press, Los Alamitos, California, USA.

[MS00] Eduard Mehofer and Bernhard Scholz. Probabilistic data flow system with two-edge profiling. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo)*, 2000, pages 65–72. ACM Press.

[NMW] Craig G. Nevill-Manning and Ian H. Witten. SEQUITUR: inferring hierarchy from sequences. <http://sequence.rutgers.edu/sequitur/> .

[NMW97] Craig G. Nevill-Manning and Ian H. Witten. Linear-time, incremental hierarchy inference for compression. In J. A. Storer and M. Cohn, editors, *Proceedings of the IEEE Data Compression Conference (DCC)*, Snowbird, Utah, USA, March 1997, pages 3–11. IEEE Computer Society Press, Los Alamitos, California, USA.

[NMW98] Craig G. Nevill-Manning and Ian H. Witten. Phrase hierarchy inference and compression in bounded space. In J. A. Storer and M. Cohn, editors, *Proceedings of the IEEE Data Compression Conference (DCC)*, Snowbird, Utah, USA, March 1998, pages 179–188. IEEE Computer Society Press, Los Alamitos, California, USA.

[Nor78] Lewis M. Norton. A program generator package for management of data files-the input language. In *Proceedings of the ACM Annual Conference*, Washington, D.C., USA, December 1978, pages 217–222. ACM Press, New York, New York, USA.

[NW94] Robert H. B. Netzer and Mark H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Orlando, Florida, USA, June 1994, pages 313–325. ACM Press, New York, New York, USA.

[PG95] Vidyadhar Phalke and Bhaskarpillai Gopinath. An inter-reference gap model for temporal locality in program behavior. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Ottawa, Ontario, Canada, May 1995, pages 291–300. ACM Press, New York, New York, USA.

[Ple94] Andrew R. Pleszkun. Techniques for compressing program address traces. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, San Jose, California, USA, 1994, pages 32–39. ACM Press, New York, New York, USA.

[QHV02] Feng Qian, Laurie Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *Proceedings of the International Conference on Compiler Construction*, Grenoble, France, April 2002, Lecture Notes in Computer Science. Springer.

[RBC02] Shai Rubin, Rastislav Bodík, and Trishul M. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings* *of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Portland, Oregon, USA, January 2002, pages 140–153. ACM Press, New York, New York, USA.

[Rei98] Steven P. Reiss. Software visualization in the desert environment. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Montréal, Québec, Canada, June 1998, pages 59–66. ACM Press, New York, New York, USA.

[Rei01] Steven P. Reiss. An overview of BLOOM. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Snowbird, Utah, USA, June 2001, pages 2–5. ACM Press, New York, New York, USA.

[RR99] Manos Renieris and Steven P. Reiss. Almost: Exploring program traces. In *Proceedings of the Workshop on New Paradigms in Information Visualization*, Kansas City, Missouri, USA, 1999, pages 70–77. ACM Press, New York, New York, USA.

[RR00] Steven P. Reiss and Manos Renieris. Generating Java trace data. In *Proceedings of the ACM SIGPLAN Java Grande Conference*, San Francisco, California, USA, June 2000, pages 71–77. ACM Press, New York, New York, USA.

[RR01] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the ACM SIGSOFT-SIGPLAN/IEEE Computer Society International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada, May 2001, pages 221–230. IEEE Computer Society Press, Los Alamitos, California, USA.

[Ruf00] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and*

Implementation (PLDI), Vancouver, British Columbia, Canada, June 2000, pages 208–218. ACM Press, New York, New York, USA.

[Sable] The Sable Research Group, McGill University. <http://www.sable.mcgill.ca/> .

[Sam89] Alain Dain Samples. Mache: No-loss trace compaction. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Berkeley, California, USA, May 1989, pages 89–97. ACM Press, New York, New York, USA.

[SCM+95] S. J. G. Scheuerl, R. C. H. Connor, R. Morrison, J. E. B. Moss, and D. S. Munro. The MaStA I/O trace format. Technical Report CS/95/4, School of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St Andrews, Fife, Scotland, 1995.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Orlando, Florida, USA, June 1994, pages 196–205. ACM Press, New York, New York, USA.

[Sew] Julian R. Seward. The bzip2 compression tool. <http://sources.redhat.com/bzip2/> .

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Minneapolis, Minnesota, USA, October 2000, pages 264–280. ACM Press, New York, New York, USA.

[Sit] Sitraka, Inc. JProbe. <http://www.sitraka.com/software/jprobe/> .

[SKS01] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Heap profiling for space-efficient Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, USA, June 2001, pages 104–113. ACM Press, New York, New York, USA.

[SMC$^+$98] Sameer Shende, Allen D. Malony, Janice Cuny, Peter Beckman, Steve Karmesin, and Kathleen Lindlan. Portable profiling and tracing for parallel, scientific applications using C++. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, Welches, Oregon, USA, 1998, pages 134–145. ACM Press, New York, New York, USA.

[Smi77] Alan Jay Smith. Two methods for the efficient analysis of memory address trace data. *IEEE Transactions on Software Engineering*, SE-3(1):94–101, January 1977.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[Smi91] Michael D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Computer Systems Laboratory, Stanford University, Stanford, California, USA, November 1991.

[SPEC98] Standard Performance Evaluation Corp. SPECjvm98 Java benchmarks. <http://www.spec.org/osg/jvm98/> .

[SS98] Yiannakis Sazeides and James E. Smith. Modeling program predictability. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, Barcelona, Spain, June 1998, pages 73–84.

[Sun] Sun Microsystems Inc. Java^{TM} virtual machine profiler interface. <http://java.sun.com/j2se/1.4/docs/guide/jvmpi/jvmpi.html> .

[SZ98] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, California, USA, October 1998, pages 12–23. ACM Press, New York, New York, USA.

[UM97] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, June 1997.

[VR00] Raja Vallée-Rai. Soot: a Java bytecode optimization framework. Master's thesis, McGill University, Montréal, Québec, Canada, October 2000.

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In David A. Watt, editor, *Proceedings of the International Conference on Compiler Construction*, Berlin, Germany, March 2000, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34. Springer.

[VRSa] Raja Vallée-Rai and the Sable Research Group, McGill University. Ashes: a Java compiler infrastructure.
<http://www.sable.mcgill.ca/ashes/> .

[VRSb] Raja Vallée-Rai and the Sable Research Group, McGill University. Soot: a Java optimization framework.
<http://www.sable.mcgill.ca/soot/> .

[W3C00] Extensible Markup Language (XML) 1.0. W3C recommendation, World Wide Web Consortium, 2000.

[Wan] Qin Wang. EVolve: an extensible software visualization framework.
<http://www.sable.mcgill.ca/evolve/> .

[Wel84] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[WL94] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, San Jose, California, USA, November 1994, pages 1–11. ACM Press, New York, New York, USA.

[Wu02] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, June 2002, pages 210–221. ACM Press, New York, New York, USA.

[WWB+02] Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour, Laurie Hendren, and Clark Verbrugge. EVolve: An open extensible software visualization framework. Sable Technical Report 2002-12, Sable Research Group, McGill University, Montréal, Québec, Canada, December 2002.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.

[ZWG+97] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automatic profiling and optimization. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, October 1997, pages 15–26. ACM Press, New York, New York, USA.