The Design and Implementation of the LENS Program Information Framework

Michael O. McCracken

Abstract

This report presents the design and implementation of LENS, a program information manipulation framework that that supports focused program investigation by providing a standard and easily accessible interface to access program information, based on the programs low-level control structure. LENS maintains version and configuration attributes for the data, enabling flexible comparisons across executions and on different systems.

1 Introduction

Writing programs and optimizing them are both complex tasks that can benefit from as much information about the program and target system as can be found. Tools abound that generate interesting information about programs, be it through static analysis, profiling, or instrumentation. Examples include profilers, memory analyzers, security checkers, compilers and static analyzers, and many more. The variety and amount of information available about programs creates problems of interface and scale. Both humans and other programs that could benefit from this information face the difficult task of integrating data from many sources and translating each tool's view of the program into their own.

To address these problems, in this work we recast the program as a data object subject to queries about its behavior, performance, and instantiations into machine code. Viewing the program in this way leads to the observation that all programming tools, including compilers, are producing and consuming data about the program. It is then natural to envision a single uniform interface to this data, both to expose more data to other tools and to programmers, and to make the task of writing cooperating tools easier.

The main design problem of such an interface is how to represent the data in a way that will map easily both to most tools' internal representations and to source code, which is ultimately the most natural interface for programmers. We believe that using the low-level control structure of a program while maintaining source code line number information provides enough detail and flexibility to support a wide variety of useful data storage and access needs.

We have designed and implemented LENS, a focused program investigation framework which is intended to serve the purpose outlined above by providing a single interface to program data, a standard naming scheme for low-level program components and a standard query mechanism for selectively requesting information about a program.

This enables tools to store data in a way that makes it easy for other tools to use and display. For example, using LENS, compilers can easily support an unlimited number of sources of information about program behavior to inform optimizations. Use of focused selective queries will allow much more detailed reporting about transformations and other actions that affect program behavior without creating an unmanageable amount of data. Standardizing access to program information reveals many opportunities for automating common programming tasks and quickly developing custom program investigation tools.

When a programmer is investigating software behavior, the benefit of a single interface to all program information is the ability to display that information and issue requests for it from the same environment. From the perspective of interaction with the programmer, LENS is designed to be the back-end for an integrated development environment *(IDE)* such as Eclipse [3], which would display program metrics along with source code, together with an interface for adding queries based the selection in the text. For example, a contextual-menu click could present the programmer with a selection of macro-like sequences of queries that represent common paths of investigation, such as timing an important inner loop, then looking into what compiler transformations were performed on it, and what opportunities were missed because an optimization could not be performed.

This work is a step towards a much more productive tool set and environment for programmers. The contributions of this paper represent the first parts of that overall vision, including the design and implementation of the LENS framework, as well as modifications to the LLVM [7] compiler framework. The modifications include generating the basic data format, and support for responding to selective requests for information on a program transformation and a register allocation pass. We also discuss an example of a custom programming tool built with the framework, and show three scenarios that highlight the benefits of our design.

1.1 Paper Organization

This paper is organized as follows: Section 1.2 presents further motivation for the work presented in the rest of the paper. In section 2, we introduce the LENS framework, discussing the high-level design in section 2.1 and the current implementation in section 2.5. Section 3 shows an example program and the resulting LENS file in XML.

1.2 Motivation

This section explores some of the problems that motivated the design of the LENS framework, and describes how a system of compilers and tools that work with the framework would address them.

Programmers are now faced with complicated machine architectures and many optimization choices that can affect performance. Furthermore, most programmers are not experts in assembly code, optimization, or compiler systems. In the past, programmers interested in performance were willing to investigate the outcome of a small number of significant optimizations such as auto-vectorization for vector processors. Those systems were easy to work with because there was an easily understood, single transformation that was necessary and sufficient for high performance, and programmers learned techniques to write their code in ways that enabled that transformation. Compilers such as those from Cray, Inc [2] gave clear reports on both which sections of codes had been auto-vectorized and, perhaps more importantly, which had not, with indications of why not. Programmers were thus quickly pointed in the right direction and could then focus on restructuring the code to vectorize, with resulting big performance gains. Unfortunately, the easy to follow report describing auto-vectorization has been replaced in many modern systems by incomplete or overwhelming information from the compiler and the inability to form a reasonable expectation about performance resulting from certain optimizations, as described in [6].

Selective queries and a single user interface for investigation, in cooperation with compilers, can help recover some of the earlier understandability of performance programming by revealing the compiler's actions, by giving the programmer control over the amount of data presented, and by keeping version-labeled records for easy experimentation. Information about why optimizations were *not performed* can guide a new set of techniques that will allow the programmer to work with the compiler instead of guessing at its actions.

For example, consider the task of a programmer who must diagnose a performance problem in an important inner loop. This task is defined poorly - the programmer knows that a loop should be faster and may have some idea of what changes they could make to improve it, but it is not clear whether code changes or compiler flags are the appropriate course of action, or how much improvement to expect. Previous experience might suggest that the loop's memory performance is at fault, so a cache profiling tool could be used to determine that it is missing a high percentage of memory accesses. In this case, perhaps unrolling the loop to allow for instruction scheduling to increase cache latency tolerance is warranted, but we need to know whether the compiler is already attempting this, and if so, whether it succeeds or if it is necessary to unroll by hand. This requires investigating an assembly-code dump of the program, which can be confusing and difficult to understand. It may not even be clear from that if unrolling was applied.

In comparison to a difficult task that requires directly using multiple tools, several data files that quickly become lost or stale, and reading assembly code, the workflow that a tool built using LENS enables is much simpler. One prototype tool we have build using LENS is loop-inspect, a tool that reports on important loop optimizations performed by the LLVM system. With loop-inspect, a simple command:

```
> loop-inspect unrolling <pmf-file>
```

inserts the appropriate queries, runs the optimizer, and shows right away the answers to all the questions stated above in a concise form. Once the information about where the optimization was and was not successful is displayed, the programmer can choose to change compilation options or make code changes, re-analyze the performance, and then easily compare the results of compilation with the previous settings, also using the loop-inspect tool.

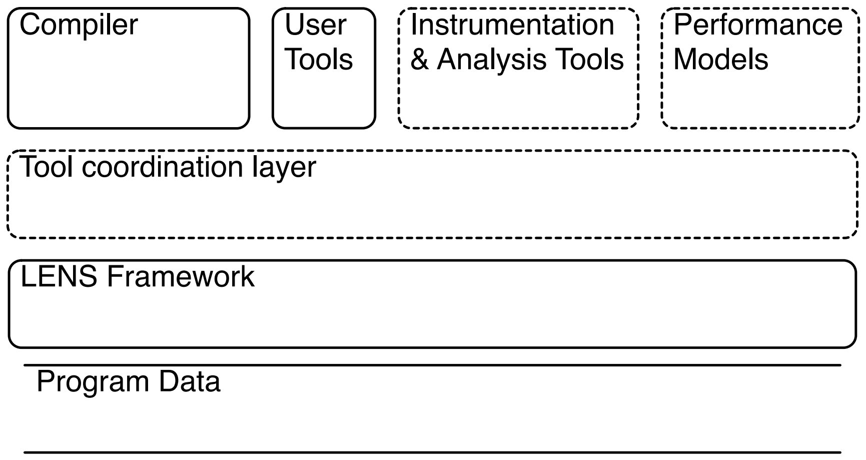

Figure 1: The LENS Framework in context.

With trends in architecture pointing toward ever more complicated processors, the number of transformations that can have significant effects on performance is growing. LENS provides the ability to report about all optimizations and analyses performed by the compiler, not just a selected few, and the incremental cost of adding new information capabilities is lower than producing equally useful new kinds of traditional diagnostic output. The framework's selectivity features are very important to avoid overload in cases where many optimization passes can produce pertinent information. LENS can store and facilitate access to all the information needed in a more complex development world by providing a uniform interface to more sources of information while encouraging focus and historical comparison.

2 LENS: A Framework for Program Information Manipulation

This section describes the LENS framework design and implementation, including the versioned, control structure-based data format, global naming schemes for program structure and program data, and the use of explicit, generic queries for accessing the information. Figure 1 shows an overall diagram of where the LENS framework fits among the various kinds of tools and data sources involved in programming. The portions of that plan that

▼ **procedure** `main` lines 100 – 162

- ▶ **block** “A”
- ▼ **loop** lines 110 – 119
  - ▼ **block** “B” lines 110 – 115
    metric `/papiex/instr-completed` = 158004227
    metric `/*/wallclocktime` = 1000 μsec
  - ▶ **block** “C” line 116
  - ▶ **block** “D” lines 117-119

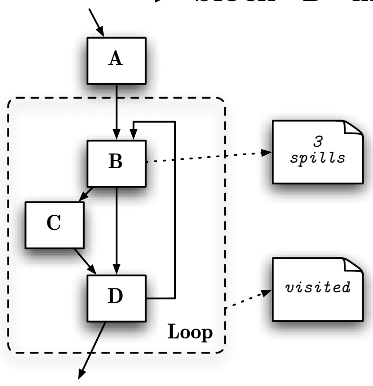

Figure 2: Example Hierarchy of Structural Nodes (in **bold**) and Attached Metric Nodes (in *italics*)

are represented in this paper include compiler support, the framework and its data organization and some simple but powerful user tools.

2.1 A Control Structure Based Data Format for Program Information

LENS stores information about a program in a separate file from the generated object file to ensure compatibility with different systems and avoid negative impact on the efficiency of the generated code. Each file contains a complete description of the program's control structure, with a *Structural Node* for each module, procedure, loop and basic block in the program. This is an abstraction of common low-level compiler structures which was chosen specifically to allow representation of arbitrary programming languages, and to simplify the process of modifying compilers to add support for the format. The entire program structure is stored because it is the foundation of the naming scheme for program components when constructing information requests, and to provide structure information to tools that may need it. Loop structure is made explicit, with a loop node that contains the blocks that form the loop. Doing this makes it easier for tools to store information about loops without needing to perform their own analysis of the source code or binary.

2.2 Structural Nodes

The structural nodes in LENS represent the constructs of an object file in procedural languages. In addition to the program's modules, procedures and blocks, which are represented directly in object files, loops are also explicitly stored in LENS files. This makes the task of storing information relevant to loops simpler for tools which do not have access to source or binary control flow analysis.

All structural nodes store the line numbers of the source code that they were generated from. This information is stored in a manner that allows multiple ranges of line numbers from separate files.

The nodes are laid out in the file according to the order they were generated by the compiler. Changing the order of basic blocks or procedures in the object file would represent a new version of the code, and should be represented as such in the LENS file.

2.2.1 Module Nodes

Module nodes are parent nodes to all the other structure nodes as well as some metadata nodes, including paths to the relevant source and object files, and the version nodes.

2.2.2 Procedure Nodes

Procedure nodes have a label that should be as much as possible the human-readable name of the procedure. These nodes contain loop and block nodes for the contents of the represented procedure, and have an attribute that says which child block is the entry block for the procedure.

2.2.3 Loop Nodes

Loop nodes contain the block nodes that represent a loop's contents. They can also contain other loops to represent loop nests. Loop nodes do not have a label of their own. They share the label of the header block for the loop, and store that label as an attribute of the loop.

2.2.4 Block Nodes

Block nodes represent basic blocks, and can be children of procedure or loop nodes. They have a label which should be unique within their procedure. They contain nodes which list their predecessor and successor blocks, including the relevant condition for the successor blocks.

2.3 Query and Metric nodes

Requests for program information, or *Queries*, are also explicitly stored in the file. Queries are saved in the file to enable a history of the investigation process that led to an action taken to alter the program, either by a tool or by the user. They also implement a layer of abstraction between producers and consumers of program information. *Query Nodes* are the primary means of communication between tools using the framework. Tools, including user tools like an IDE, request information from other tools by inserting queries into the file, and contribute information by inserting metric nodes in response to queries. Queries use metric names to specify what information the query is requesting. The system for matching queries and metrics is very flexible, and is further described in section 2.4.6.

The following is a sample query that asks for a cooperating tool to fill in metrics explaining for each loop in the program, why it was not unrolled, if that optimization is unsuccessful. In real use, a query like this would potentially result in too much resulting data, and a more specific query that only specified some loops of interest would be preferable.

```
<query  requestingToolName="lensh"
        label="loopUnrollQuery"
        date="2004-05-01-19:31gmt-4"
        versionLabel="version1"
        xpath="/pmf/module/procedure/loop"
        metricName="/llvm/optimizations/loopunroll/reasons/"/>
```

Associated program information is stored in *Metric Nodes*, which are attached to the specific structural nodes to which they refer. Metric nodes can contain arbitrary information, and a structural node can have any number of metric nodes attached to it. Metrics are labeled using a standard scheme, which is further described in section 2.4.6. Figure 2 shows a conceptual picture of a set of structural nodes, the control flow graph they represent and sample metrics. The control flow graph diagram includes metrics shown as notes linked to the structures they describe.

The following is a sample metric node

```
 <metric answeringTool="llvm"
         name="/llvm/optimizations/loopunroll/reasons/toomany_blocks"
         versionLabel="label 16:28:51"
         inResponseTo="unrollQuery">
Loop too large to unroll
</metric>
```

For the planned applications of the LENS framework, we expect the volume of stored data to be small enough that the current data storage mechanism will be appropriate. The ability to selectively query data at a very fine level should allow the user to narrow their investigation scope quickly with high-level timing queries, and then use selective queries for data that would otherwise be overwhelming. The framework is intended to store only data that is directly useful as an answer to a question, so large amounts of profile data are less appropriate for storage than the results of analysis on that data. However, if during our investigation we find that data size is a problem, it would be straightforward to extend the framework to include separate data storage for larger metrics, with no change to the interface seen by tools or users.

2.4 Metadata Nodes

Every structural, query, and metric node is annotated with a *version*. Each version contains information about the current system configuration and descriptions of each tool which contributed queries or metrics. This allows historical comparison, because a tool responding to the same query several times will result in a set of metrics marked with their date and time and context. This information can be very useful in diagnosing bugs and performance problems which depend on system parameters. A uniform means for such historical comparison of arbitrary program metrics is an important step toward improving programmer productivity and consistency.

The metadata nodes provide a flexible way of tracking the environment in which a structural node or metric node was recorded.

2.4.1 Tool Nodes

Tool nodes include data about a particular program involved in generating information about the program of interest. Examples include compilers and instrumentation tools.

2.4.2 Platform Nodes

Platform nodes represent the hardware and system configuration that was present for recording metrics and structure nodes. The platform descriptor should contain enough information to uniquely describe an individual computer, such as an IP address.

2.4.3 Configuration Nodes

Configuration nodes represent the configuration of a particular system at the time a metric or structure node was generated. In the case of parallel systems, each configuration contains multiple platform nodes, each of which describe the individual computer on which information about the program was generated.

2.4.4 Version Nodes

Version nodes combine a configuration and a time to represent the unique environment that resulted in particular information being generated. There are likely to be comparatively many more version nodes than configuration nodes, as each new set of metrics generated on a single system will result in a new version being created that links to the same configuration.

2.4.5 A Standard Naming Scheme for Program Components

We have defined a common naming scheme for the program structural nodes, to permit tools to communicate with each other about pieces of the program even if they have different internal representations of the program. The naming scheme is simple, and resembles UNIX directory paths. For example, a block named '`b1`' in a procedure named '`init`' is identified by the following path: `/module/procedure[@name='init']/block[@name='b1']`. Because the naming system is actually implemented using XPath, more complicated names can be used, including referring to sets of structural nodes or selecting nodes that match a simple predicate testing the represented source line number.

Structural nodes store source line number information for the pieces of the program they represent to allow the linking of structural nodes to source code. This mapping is important for tools which may not have the same detailed access to code structure as compilers or debuggers, or for tools which display program information alongside source code. This approach degrades somewhat gracefully when line numbers are not available, with only those tools which rely on the line numbers being affected.

2.4.6 A Hierarchical Naming Scheme for Program Metrics

Tools using the framework must also share a common naming scheme for metrics if they are to read each other's output. For this task, we view the entire space of possible measured entities about *each separate location* in a program as a hierarchy of metrics. The top level of the hierarchy names the tools, the second level names metrics those tools support, and lower levels if necessary refer to aspects of metrics that can be specified to further focus the investigation.

For example, metric names relating to the loop unrolling pass in LLVM would look like the following:

```
/llvm/loop-unroll/reasons/too-large
```

which would indicate a simple metric stating that the loop was not unrolled because it was too large, and provides as the metric data how large the loop was, and what the cutoff threshold was. When looking for queries matching a particular metric name, queries specifying more general metric names also match. For example, the high-level metric name `/llvm/loop-unroll/` is currently answered by many metrics, including the above one.

2.5 Framework Implementation

In this section we discuss the current implementation of the LENS framework. We recognize that the real utility and success of this kind of a platform depends on having a sufficient number of participating tools. Therefore, we have attempted to make it easy to add support for the system to existing tools by providing a high-level object-oriented interface that does not reveal the details of the storage mechanism.

The implementation of the abstract file format described in section 2.1 uses XML [9], while the LENS framework software is built using Python [8], and provides a powerful interface to the file and its structure. Each structural node, metric and query is a Python object that can be manipulated easily. File input and output, document state and consistency maintenance are contained within the framework to insulate tool programmers from those concerns.

Using Python has allowed us to be more productive when working on the system itself, especially due to the Python unit test framework, which has allowed us to build a suite of tests that lend confidence to the system. LENS was built and tested on a Mac OS X system using Python 2.3, but is written in a portable style, and has been used successfully in combination with LLVM on Linux systems.

Implementing LENS in Python also results in an environment for rapid development of customized tools for specific investigation tasks. The loop-inspect tool discussed earlier is one example of such a custom tool. We have also built a more generic command-line tool for listing and manipulating queries and metrics in a file that further insulates users of the framework from the XML data file, and is suitable for use in automated build and testing systems where a direct interface to the framework is not desirable.

We have also built a procedural C interface that bridges a subset of the LENS framework's functionality. The exposed functionality is enough to create the data files and to check for and respond to queries. This relatively simple interface is used in the modifications to LLVM.

2.5.1 Implementation of the Data Representation in XML

We have defined an XML document format for the program data file. Each of the conceptual nodes described in section 2 is implemented as an XML node with a version attribute. The nodes are laid out in order in the document where order is meaningful, and are nested to represent containment – for example, all of the loop nodes belonging to a procedure are nested inside the appropriate procedure node.

The structure of our XML format and the XPath standard [1] for identifying and searching XML nodes leads to a natural scheme for naming program components. The naming scheme is implemented using XPath. For example, query nodes include simple XPath expressions that identify the structural node or set of nodes to which they refer. Sections 2.4.5 contain examples of these expressions. XPath contains many powerful features for specifying sets, all of which are usable within LENS. However, a simple subset is sufficient for defining a common naming scheme for program structures. The naming scheme for LENS is limited to XPath expressions using simple paths, indexes and predicates matching single attributes, in an attempt to hide the complicated details of XML processing.

The metric data names are also inspired by XPath, with one important difference: the system does not build an XML document containing all possible metrics. While the program structural nodes do all exist in the data file, such a document for metrics would be infinite, which precludes the use of XPath libraries. The system currently mimics the behavior of XPath to access the name space of available metrics.

2.5.2 Space and Time Overhead of LENS Data Files

Our current implementation of the LENS framework incurs some overhead due to reading and parsing the XML and building the internal representation of the structural, query and metric nodes. In practice it is not overwhelming, and represents a small percentage of the runtime for large programs. To quantify the space and time overhead of building and loading the LENS files, we modified the LLVM test suite, consisting of many benchmarks and test programs, to also produce LENS structure files. In experiments using a 1.25 GHz Apple G4 Powerbook with 1.5 GB of RAM, we compared the time it took to optimize a program with the default settings of LLVM to the time it took to perform the same optimizations as well as building and loading the LENS file. The following results are from ten application and synthetic benchmarks in the test suite, all of which took more than ten seconds for LLVM to optimize the final linked bytecode. The average time for building the data structures was 1.24 times longer, and 1.16 times longer for loading the file and making the data available for queries. While some overhead is justified for the benefits of extra available information, opportunities do exist to optimize the framework, including rewriting the framework in C++ instead of Python to avoid overhead due to language boundaries.

We also measured the average size of a minimal LENS file consisting of just the structural nodes for the same benchmarks, which was on average 4.6 times the size of the resulting program. We also looked into methods for saving space when working with XML data. One straightforward method is to use a library such as zlib [5], to compress the data for storage, because XML compresses very well. In experiments using gzip [4], we found that such compression would result in data files on average 0.06 times as large as the uncompressed version, which would then be an average of 0.25 times the size of the program itself. Based on these measurements, we believe that the current implementation is sufficient for its intended uses, and extensive optimization of the framework itself will be performed only if it becomes necessary in the future.

3 A sample LENS file

The program

```
int main(int argc, char *argv[]) {
  double a[100][100];
  double b[100][100];

  int i,j;

  printf("Loops begin:\n");

  for(i=0;i<99;i++)
    b[i][0] = 47;
    for(j=0;j<99;j++) {
      if(i*j > 1){
a[i][j] = b[i][j] + i*j;
      }else{
a[i][j] = 1.0;
      }
    }

  printf("Loops: a[1][1] = %f\n", a[1][1]);

  return 0;
}
```

The LENS XML file with corresponding structure nodes.

```
<pmf version="0.5">

  <module name="loops.bc">
    <sourcefile path="./loops.c"/>
    <objectfile path="loops.bc"/>
    <version configurationLabel="configuration-1"
             label="llvm-gen"
             date="Tue Feb 14 16:23:32 2006 PST"/>
    <procedure versionLabel="llvm-gen" name="main"
               entryBlockLabel="entry"
               sourceLineRange="(2, 21)">
      <loop versionLabel="llvm-gen" headerBlockLabel="no_exit.0"
            sourceLineRange="(10, 10)">
        <block versionLabel="llvm-gen" label="no_exit.0"
```

```
          sourceLineRange="(10, 10)">
      <predecessor blockLabel="no_exit.0"/>
      <predecessor blockLabel="entry"/>
      <successor condition="True" blockLabel="loopexit.0"/>
      <successor condition="False" blockLabel="no_exit.0"/>
    </block>
  </loop>
  <loop versionLabel="llvm-gen" headerBlockLabel="no_exit.1"
        sourceLineRange="(12, 17)">
    <block versionLabel="llvm-gen" label="no_exit.1"
           sourceLineRange="(12, 12)">
      <predecessor blockLabel="loopentry.1"/>
      <predecessor blockLabel="loopexit.0"/>
      <successor condition="True" blockLabel="then"/>
      <successor condition="False" blockLabel="else"/>
    </block>
    <block versionLabel="llvm-gen" label="loopentry.1"
           sourceLineRange="(-1, 0)">
      <predecessor blockLabel="else"/>
      <predecessor blockLabel="then"/>
      <successor condition="True" blockLabel="loopexit.1"/>
      <successor condition="False" blockLabel="no_exit.1"/>
    </block>
    <block versionLabel="llvm-gen" label="then"
           sourceLineRange="(13, 17)">
      <predecessor blockLabel="no_exit.1"/>
      <successor blockLabel="loopentry.1"/>
    </block>
    <block versionLabel="llvm-gen" label="else"
           sourceLineRange="(15, 17)">
      <predecessor blockLabel="no_exit.1"/>
      <successor blockLabel="loopentry.1"/>
    </block>
  </loop>
  <block versionLabel="llvm-gen" label="entry"
         sourceLineRange="(2, 9)">
    <successor blockLabel="no_exit.0"/>
  </block>
```

```
      <block versionLabel="llvm-gen" label="loopexit.0"
             sourceLineRange="(11, 11)">
        <predecessor blockLabel="no_exit.0"/>
        <successor blockLabel="no_exit.1"/>
      </block>
      <block versionLabel="llvm-gen" label="loopexit.1"
             sourceLineRange="(19, 21)">
        <predecessor blockLabel="loopentry.1"/>
      </block>
    </procedure>
  </module>
  <configuration label="configuration-1">
    <tool name="llvm" role="compiler" version="1.4">
      <arguments/>
      <environment/>
    </tool>
    <platform name="darwin">
      <descriptor>
('Darwin', 'punk.local', '8.3.0',
'Darwin Kernel Version 8.3.0: Mon Oct  3 20:04:04 PDT 2005;
root:xnu-792.6.22.obj~2/RELEASE_PPC', 'Power Macintosh')
      </descriptor>
    </platform>
    <annotation>
Configuration generated by pyPMF
    </annotation>
  </configuration>
</pmf>
```

References

[1] Clark, J., and DeRose, S. J. XML path language (XPath) version 1.0. Recommendation REC-xpath-19991116, W3C, 1999.

[2] Cray Incorporated. Cray compilers and loopmark tool. http://www.cray.com/.

[3] Eclipse Foundation. Eclipse development environment. `http://www.eclipse.org/`.

[4] Gailly, J., and Adler, M. Gzip compression utility. `http://www.gzip.org/`.

[5] Gailly, J., and Adler, M. Zlib compression library. `http://www.zlib.net/`.

[6] Haneda, M., Knijnenburg, P., and Wijshoff, H. Generating new general compiler optimization settings. In *Proceedings of the 19th ACM International Conference on Supercomputing* (2005).

[7] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization* (2004).

[8] Python Software Foundation. Python language. `http://www.python.org/`, 2005.

[9] World Wide Web Consortium. Extensible markup language (XML) 1.1. Recommendation, W3C, 2001.