The Camino Compiler Infrastructure

Chunling Hu John McCabe Daniel A. Jiménez Ulrich Kremer

Department of Computer Science
Rutgers University
Piscataway, NJ 08854
{chunling,jomccabe,djimenez,uli}@cs.rutgers.edu

Abstract

This paper introduces the Camino Compiler Infrastructure. Camino implements several types of profiling, including basic block counts, edge profiling, interprocedural path profiling, and a special technique that allows using a SimPoint-like methodology to do efficient and precise fine-grained power behavior characterization. It also supports a growing set of code placement optimizations such as branch alignment and pattern history table partitioning. In its current implementation, Camino works as a post-processor for the Gnu Compiler Collection (GCC). The goal of Camino is to serve as a testbed for various low-level performance optimizations as well as power and energy optimizations. It currently supports the x86 instruction set.

1. Introduction

This paper introduces the Camino Compiler Infrastructure, a work in progress at the Camino Lab in the Department of Computer Science at Rutgers. The goal of Camino is to serve as a testbed for various low-level optimizations. It is currently used to study performance optimizations as well as power and energy optimizations. Camino currently supports the x86 instruction set. Support for ARM is in our ongoing work. *Camino* is the Spanish word for "path," representing our lab's focus on control-flow-oriented optimizations.

Camino can be used as a static instrumentation tool for profiling. It parses the assembly code generated by GCC and distinguishes data and code, thus bypassing one of the serious disadvantages of static binary instrumentation [7]. Camino implements several types of profiling, including basic block counts, edge profiling, and interprocedural path profiling. It also includes a special technique that allows using a SimPoint-like [10] methodology to efficiently characterize the fine-grained power behavior of an application with very low overhead by sampling specially chosen intervals.

Camino also supports a growing set of code placement optimizations such as branch alignment [2] and pattern history table partitioning [6]. Branch alignment improves instruction fetch bandwidth by reordering code such that most conditional branches to be not taken and thus do not incur a discontinuous fetch penalty. Pattern history table partitioning improves branch predicton accuracy through a feedback-directed placement of conditional branches that reduces the likelihood that they will interfere destructively with one another in branch prediction tables.

The Camino infrastructure is currently implemented as a post-processor to GCC. It can theoretically handle programs in any language that can be compiled by GCC. Currently it supports compiling C, C++, and FORTRAN 77.

The rest of the paper is organized as follows: Section 2 describes some related work in profiling methods, SimPoint, and code placement. Section 3 presents an overview of Camino, including its execution, internal representation, and output. Section 4 describes the various types of profiling implemented in Camino. Section 5 describes the current optimizations supported by Camino, including the unique pattern history table partitioning optimization. Finally, Section 6 concludes and gives directions for future work.

2. Related Work

Before stepping into the details of Camino, we discuss some previous work related to program behavior profiling tools, SimPoint, and code placement.

2.1. Program Behavior Profiling Tools

There is a long history of program profiling tools, including static instrumentation tools, dynamic instrumentation tools, simulators, and built-in hardware monitors.

Static instrumentation tools modify a program prior to its execution with the purpose of monitoring the behavior of the program during execution. Instrumentation can be done on source code, assembly code, or binary code. ATOM is a commonly used static binary instrumentation tool [11]. An instrumentation file and an analysis file are needed to instrument a program through ATOM. Due to its dependence on the huge gap between data segment and code segment in memory address space, ATOM is not very portable. Another static binary instrumentation tool is FIT [1], which has better portability. As an instrumentation tool, our infrastructure, Camino, instruments assembly code.

Dynamic instrumentation tools insert profiling code in executable image during program execution. They can instrument dynamic generated code, which is impossible for static instrumentation tools. Changes in the profiling method does not require recompiling the instrumented program. Possible disadvantages are imprecise mapping between the profiling result and the instrumented program, and high instrumentation overhead. Pin [9] uses a just-in-time compiler for

dynamic instrumentation and does not change code and data addresses when instrumenting a program. It enables users to observe runtime processor state. Since runtime power behavior is time-dependent and sensitive to energy consumption overhead, we do not want to use dynamic instrumentation for power behavior measurement of specific intervals.

Simulators are widely used in research for nonexistent architectures. simulation is like dynamic instrumentation, but much slower. It provides precise mapping between the profiling result and the simulated program, but the difference between a real system and the modeled one makes it infeasible in evaluating some low-level optimizations.

Hardware monitor measures resource utilization during program execution. It has separate pieces of equipment that are attached to the system component being monitored. It does not consume system sources and has low overhead. Hardware monitors include probes, performance counters, and logic elements.

2.2. SimPoint

SimPoint [10] partitions a program's execution into intervals, clusters the intervals into phases based on the similarity of their Basic Block Vectors (BBV), and selects a representative interval for each phase, called a simpoint. It is independent of the underlying microarchitecture and provides an idea to estimate whole program metrics from the behavior of the simpoints. Sherwood *et al.* show the use of this idea in estimating instructions per cycle (IPC), branch prediction, instruction cache, data cache, and unified L2 cache miss rates of the SPEC 2000 benchmarks [10]. Hu *et al.* show that SimPoint can also be used to estimate the power consumption of a program through simulation its simpoints [5]. We implement a method in Camino to profile BBVs during program execution, do phase classification, and select the simpoints. Physical CPU power measurement is performed for the simpoints on a real system. Also, we propose a SimPoint-like method that also does phase classification and selects representative intervals, but results in very low overhead in power measurement.

2.3. Code Placement

Many papers have presented code-reordering methods to improve locality or minimize cache conflict misses [4, 3, 8]. Calder and Grunwald present branch alignment [2]. This algorithm reorders code to minimize number of taken branches, such that the not path through a procedure is laid out in a straight line.

3. Camino Overview

In this section, we explain the basic organization and operation of the Camino Compiler Infrastructure.

3.1. Using Camino

Camino provides three drivers, *ccc*, *ccpp*, and *cf77*, for the compilation of programs in C, C++, and FORTRAN 77, respectively. The driver wraps GCC compilation and the post-processing provided by Camino. A user can invoke the driver with arguments at the command line to start compiling a program. The driver invokes the corresponding front-end from GCC with the `-S` option to generate assembly language output from the source file. It then invokes the Camino post-processor, described later, on the assembly language file. The driver parses the command line arguments, passing Camino-specific arguments to Camino and other options to the GCC front-end program. These specific arguments control what kind of profiling will be inserted, as well as which optimizations will be applied to the assembly code.

3.2. Internal Representation

The Camino post-processor is a C++ program that reads the assembly code generated by GCC and the arguments passed by the driver. It first parses the assembly code into three basic abstractions: procedures, basic blocks, and lines.

Functions implemented in an assembly program are parsed into a Standard Template Library (STL) list of procedures. For each procedure, the control-flow is analyzed and a control-flow graph (CFG) is maintained with STL lists of basic blocks with pointers. There are two distinguished basic blocks: *entry* and *exit* nodes. *entry* is the first executable basic block in the procedure.

The basic block is probably the most important abstraction in the compiler. The intraprocedural structure of the code is completely represented within basic blocks. Each CFG node is the data structure of a basic block. It has a lists of lines and a number of pointers to predecessor and successor nodes, the targets of a conditional branch at the end of the basic block, basic blocks that this block dominates and post-dominates, and the list of loops in which this basic block appears. Also, each basic block has an edge profile and a list of path profiles that may be read from a path profile file generated by certain profiling.

Camino distinguishes data and code from directives in the assembly output. Non-text items such as string constants and other data are kept in special basic blocks that are included in the nearest procedure but are not part of any CFG. Each executable basic block is assigned a hash value calculated based on the name of the program, the name of the procedure this basic block belongs to, and its sequence number in this procedure. This value is used by the profiling instrumentation for reference. This value may be considered unique for most purposes. Although collisions are possible, our tests show that they occur very infrequently with no impact on the quality of profiling.

Lines in a basic block are represented with a list of *line* objects. Each line consists of an optional label, an optional x86 opcode or assembler pseudo-op, and an optional set of operands. Camino also has the ability to determine the byte offset of any given instruction in the final executable modulo a moderate power of two. This capability is useful in certain code placement optimizations [6] where knowing the lower bits of the address of an instruction is important in predicting how the microarchitecture will treat this instruction. We currently use this information for a branch prediction optimization, but it could also be useful for instruction cache optimizations.

In this internal representation, instrumentation and/or optimizations may be performed on various level, from procedure to instruction. Since each basic block has a "unique" reference value and the lines of a basic block are stored, a user can instrument or optimize only the basic blocks that

satisfy some special condition, instead of all basic blocks. This is also true for instructions. Instrumentation using Camino is very simple. Only two routines are required, an instrumentation routine and an analysis routine. The instrumentation routine inserts a call to the analysis routine at proper positions in each basic block. The analysis routine is normally implemented as a library function linked to the instrumented program at the last step of the compilation. This sort of instrumentation is used to implement various types of profiling as well as triggering power and energy measurement by an external device.

3.3. Output

Once Camino is finished with its transformations, it extracts the modified assembly code from the internal representation and overwrites the original assembly file. Then the driver calls the appropriate GCC component to complete the process of assembling the compilation unit and possibly linking the program.

4. Program Profiling Supported in Camino

Camino currently implements three types of profiling: 1) basic block and edge counts, 2) interprocedural path profiling, and 3) profiling in support of obtaining basic block vectors for SimPoint-like clustering. Because of its clear internal representation on multiple levels, it is easy for a user to insert instrumentation at proper positions, or just change the analysis routine, to implement a new type of profiling.

4.1. Basic Block and Edge Counts

This type of profiling combines basic block counts and edge counts. For each basic block not containing a conditional branch, a record is kept for the number of times it is encountered. For basic blocks ending in a conditional branch, a basic block count is kept as well as a count of the number of times the branch was taken. When this information is read back into the compiler later to recompile the program with the guide of the profiling result, it is converted into counts of the number of times CFG edges were traversed through conditional branches. The analysis code also has the ability to simulate a simple branch predictor and record the number of times the branch was correctly predicted; however, this option is usually turned off for efficiency.

4.2. Interprocedural Path Profiling

Many branch predictors use history tables. Camino implements a special form of path profiling whose goal is to determine the path corresponding to the global history used by certain types of branch predictors.

Interprocedural path profiling is implemented through the instrumentation of branch instructions. The analysis routine invoked during program execution maintains a record for a global path of a given fixed length. The frequency with which this history is encountered, the frequency with which this path is taken, and a sequence of branches identifiers along this path are recorded. The taken and not taken information of a branch is stored in the profile data structure of the basic block corresponding to this branch. Each time a branch is executed, the analysis routine updates this information and determines if this branch should be shifted into the global path record. If the frequency of a global path is higher than a given threshold, the profile of this path is temporarily stored in some table. At the end of program execution, all of the recorded path profiles are written to an output file. Camino provides a method for reading in the output file for use in path-based optimizations.

4.3. A SimPoint-like Method for Physical Measurement of Power Behavior

Detailed power behavior of a program is useful in both evaluating power optimizations and observing power optimization opportunities. For example, an optimization to limit the maximum dynamic CPU, or smooth the change of CPU current with time. Total power consumption estimation through simulation or coarse physical measurement can not give us enough information to evaluate the benefits from such optimizations. Detailed power simulation takes a long time, and a real system is normally more complex than the abstracted one to simulate. Detailed power measurement is faster and objective, but generates huge power data file and it is hard to know which part of the file is useful in our evaluation.

Section 2 shows that SimPoint provides a method to characterize the whole program power behavior by measuring only the simpoints of the program. In order to verify the feasibility of SimPoint in estimating program power behavior on a real system, we implement a static basic block instrumentation for BBV profiling and power measurement of selected intervals in Camino. We also implement a SimPoint-like method that has low instrumentation overhead on power behavior measurement, which is very useful when detailed program power behavior is needed. All of the profiling and measurement are performed on a Pentium 4 machine running Linux 2.6.9 and GCC 3.4.2. Benchmarks are from the members of SPEC CPU2000 that can be compiled by Camino successfully. Our system's motherboard has a separate power cable for CPU and its voltage is 12V. We measure the current on this cable using an Tektronix oscilloscope, and calculate the power consumption using voltage, current, and time.

4.3.1. Basic Block Vector Profiling

As described in Section 2.2, instrumentation at the basic block level using Camino is very simple. We instrument each basic block before its first instruction for BBV profiling.

Our verification of SimPoint in power behavior estimation includes 6 steps:

1. Camino statically instruments each basic block to profile the number of executed instructions of each basic block.

2. During program execution, the analysis routine builds a BBV for each interval (10 million instructions).

3. Offline analysis based on SimPoint 2.0 selects the simpoints, records the hash values of the starting ending basic blocks of each simpoint, and counts the execution times of such basic blocks before the beginning and the end of each simpoint.

4. The basic blocks recorded in step 3 are statically instrumented. The analysis routine counts the execution times of these basic blocks, determines where is the beginning or the end of each simpoint, and tells the oscilloscope to record the power behavior of each simpoint.
5. The instrumented application is compiled and executed. The power behavior of each simpoint measured by the oscilloscope is recorded.
6. Total power consumption of each benchmark is estimated from the recorded power behavior of each simpoint and its corresponding weight. The error rate compared to the measured whole-program power consumption is calculated.

The first two columns in each group in Figure 1 show the error rates when SimPoint is used to estimate whole program energy consumption. From the first column, we can see that for most benchmarks, we can get the power consumption of a benchmark through measuring power of the simpoints. But the error rates compared to the uninstrumented benchmarks are very high. This is because we instrument all basic blocks from the above step 3, although some of them may execute many times and result in heavy instrumentation overhead.

4.3.2. Phase Identification Based on Infrequent Basic Blocks

To reduce the impact of instrumentation overhead on the detailed power behavior of simpoints, we implement a SimPoint-like method in Camino. It also profiles BBVs during program execution but has very light instrumentation impact on the physical measurement result of simpoints.

Before profiling BBVs, we instrument each benchmark to count the frequency with which each basic block is encountered. The infrequent basic blocks are selected based on some user-specified threshold. The following steps are the same as in the last section, except that only these infrequent basic blocks can be the first basic block of each interval during program execution, instead of using a fixed interval size, thus only some of these basic blocks are instrumented for the final physical mensurement. This method results in varying interval size, but much lighter instrumentation impact on physical measurement result. K-means clustering used in SimPoint also works in this phase identification.

The last two columns in each group in Figure 1 show the error rates when our SimPoint-like method is used for phase identification. The error rate compared to the instrumented benchmarks is sometimes a little bit higher than when SimPoint is used, but the one compared to the uninstrumented benchmarks is much lower. This is very important when we want to analyze the exact runtime program power behavior.

Figure 2 demonstrates the total execution time of each instrumented benchmarks when SimPoint and our new method is used respectively. The execution time of the uninstrumented ones is used to show the low overhead of our SimPoint-like method.

Figure 3 shows the ratio of the execution time of the 30 simpoints to the total benchmark execution time. Our method does not increase the program execution fragment to measure. One part of our ongoing work is to enable users to instrument a program to record executed instructions of each simpoint. This is helpful when users want to see the corresponding source code of a power behavior curve.

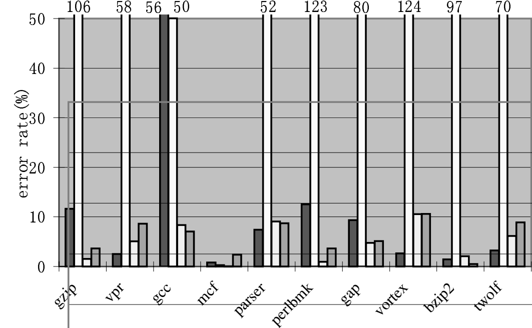

Figure 1: Error rates when SimPoint and our infrequent BB-based method are used for phase classification. From left to right: SimPoint-instrumented, SimPoint-uninstrumented, Infrequent BB-instrumented, Infrequent BB-uninstrumented

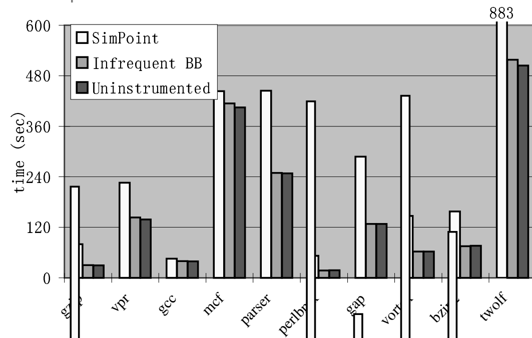

Figure 2: Execution time of instrumented benchmarks using different phase classification methods

5. Optimizations

This section describes control-flow optimizations implemented in Camino.

Camino implements greedy branch alignment [2], an optimization that causes most conditional branches executed to be not taken, thus allowing for improved instruction fetch bandwidth for superscalars and other machines where discontinuous fetches can have a negative impact on performance. Camino also implements code alignment heuristics that mimic those of several versions of GCC, e.g., certain branch targets are placed on 8- or 16-byte boundaries to improve fetch bandwidth.

One unique optimization implemented by Camino is pat-

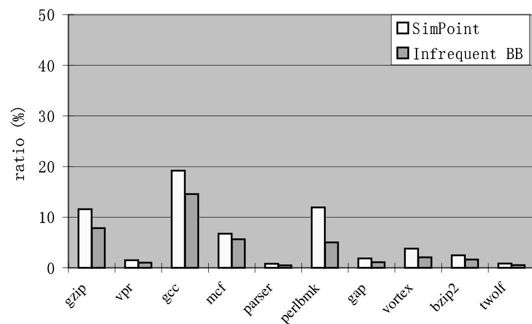

Figure 3: Execution time ratio of simpoints to the whole benchmark

tern history table partitioning [6]. This technique controls the placement of conditional branches by judicious insertion of no-op instructions such that similarly behaving conditional branches hash to a pre-defined partition of the pattern history table in the branch predictor. This greatly reduces the likelihood that branches with differing behavior will cause destructive interference in the pattern history table, and thus improves branch prediction accuracy.

We are currently coding a data-flow analysis engine for Camino. The first client for this engine will be the instrumentation code itself; we will use liveness analysis to determine exactly how much context must be saved by the instrumentation code without violating program semantics.

5.1. Greedy Branch Alignment

The greedy branch alignment algorithm transforms a procedure such that basic blocks incident on frequently executed edges are placed contiguously. The required input is the source code and edge profiles.

CFG edges are ordered by frequency, according to the profile data, to begin the optimization. Using this ordering, the most frequent execution paths of a procedure are placed in memory contiguously. Accordingly, certain frequent execution paths may not be contiguously laid out in memory due to the priority of a more frequent execution path. After the arrangement phase is accomplished, conditional branch senses (e.g. "less than" vs. "greater than") are fixed up such that the control flow is semantically correct.

If a basic block does not end in a conditional branch, e.g., a return to the caller of the procedure, no inversion of the conditional branch sense is possible. Changing the conditional branch sense implies that the taken path is more frequent than the not-taken path, which is impossible if the not-taken path is as or more frequent as the the taken path.

5.2. Pattern History Table Partitioning

Pattern history table (PHT) partitioning transforms the memory locations of conditional branches using no-op instructions to reduce destructive aliasing. Like the greedy branch alignment, PHT partitioning requires edges profiles to accomplish the feedback optimization.

Almost all modern microprocessors use a PHT to predict the outcomes of conditional branches by tracking the tendency of a branch to be taken or not taken given a branch address and histories of branch outcomes. PHTs have relatively few entries compared with the number of branches and branch histories. Thus, mispredictions caused by conflicts in PHTs are inevitable. Many PHT-based branch predictors are indexed using some of the lower bits of the address of the branch to be predicted. By manipulating these bits, we can direct branches to different areas in the PHT.

PHT partitioning exploits the nature of conditional branches. Many conditional branches are biased either to be taken or not taken. If we group the conditional branches such that most of the taken biased branches alias with similar behaving branches and same the for not taken, less destructive inference will occur. Camino implements two forms of PHT partitioning, bi-modal which uses two partitions as described previously. Four-way PHT partitioning uses four partitions employing 2 dimensions, taken or not-taken bias, and strength of the bias.

Careful placement of no-op instructions is neccessary for PHT partitioning so that decreased performance is avoided. A procedure's basic blocks are grouped into regions that can be separated by no-op instructions by PHT partitioning. There are two types of positions that end regions: 1) a basic block does not end with a conditional branch; and 2) basic block ends with a highly biased conditional branch, which, according to the profile data, is executed many times. No-op instructions after a jump instruction will not be executed, and therefore are a perfect spot for the needed no-op instructions. Execution of no-op instructions on the not-taken path of a conditional branch, which is highly biased to be taken, will rarely occur. The requirement that the branch is executed many times in the profile is two-fold: 1) excludes transforming of the uncommon case, which may be detrimental to performance; and 2) affirms the profile data reflects actual behavior of conditional branch.

The next phase of the PHT partitioning is to move the conditional branches to memory locations such that the maximum number of branches are assigned to the proper partition with respect to the expected frequency of the conditional branches based on the training run. Linearly iterating over the regions, the PHT partitioning algorithm determines the effect of inserting each possible number of one-byte no-op instructions from 0 through $2^k - 1$ on assigning branches to their proper partitions, where k is the number of bits from the branch's memory location needed to best trade-off between improved branch prediction accuracy and reduced instruction cache locality. For each proposed number of no-ops to insert, the algorithm recomputes the addresses of every branch instruction length in the region, taking into account changes in branch assigned to their proper pattern history table partition, weighted by frequency.

6. Conclusions and Future Work

Instrumentation tools are helpful in understanding program behavior on real systems. This paper introduces our instrumentation and profiling tool, Camino. It supports basic block count and edge count profiling, interprocedural path profiling, and basic block vector profiling for SimPoint-like phase classification. Implementation of new profiling or instrumentation is very easy because of the clear internal representation of Camino. Camino is currently used as a test bed for low-level performance, power or energy optimizations. We succesfully use Camino in charactering program power behavior with low error rates using a SimPoint-like phase classification method. The code optimizations supported by comino improve instruction fetch bandwidth and branch prediction accuracy. Camino is a work-in-progress; however, we anticipate making a public release of the infrastructure sometime in Spring of 2006. The path and edge profiling mechanisms used by Camino are somewhat inefficient. In future work we plan to use data-flow analysis to optimize each instrumentation site so that it will have minimal impact on performance. We are actively pursuing the SimPoint-style work on physical measurement with a goal of understanding the effect of power and energy optimizations at a fine-grained level. We are also exloring ways of using path profiling to improve branch predictor accuracy. Supporting the x86 architecture allows us to explore a variety of microarchitectures without portability issues; thus, we plan to investigate

path-based optimizations on the various Intel and AMD microarchitectures.

References

[1] B. D. Bus, D. Chanet, B. D. Sutter, L. V. Put, and K. D. Bosschere. The design and implementation of fit: a flexible instrumentation toolkit. In *Proceedings of the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering(PASTE)*, pages 29–34, 2004.

[2] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[3] N. Gloy and M. D. Smith. Procedure placement using Temporal-Ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, September 1999.

[4] D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[5] C. Hu, D. A. Jiménez, and U. Kremer. Toward an evaluation infrastructure for power and energy optimizations. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005, Workshop 11), CD-ROM / Abstracts Proceedings*, April 2005.

[6] D. A. Jiménez. Code placement for improving dynamic branch prediction accuracy. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pages 107–116, June 2005.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[8] S. McFarling. Program optimization for instruction caches. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 183–191, 1988.

[9] V. J. Reddi, A. Settle, and D. A. Connors. Pin: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the Workshop on Computer Architecture Education*, June 2004.

[10] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[11] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205, November 1994.