Using the SGI Pro64 Open Source Compiler Infra-Structure for Teaching and Research

José Nelson Amaral*, Christopher Barton, Andrew C. Macdonell, Matthew McNaughton
[1] Department of Computing Science
University of Alberta, Edmonton, CA
{http://www.cs.ualberta.ca}

Abstract—
Modern optimizing compilers are complex programs that require from tens to hundreds of people-years to be developed. Thus professors must use third-party compiler infra-structures to introduce students to compiler optimizations. Until recently only infra-structures developed at universities, research institutes, or by GNU were widely available for teaching. However, in May 2000, SGI made public the source code for Pro64, a highly optimized suite of compilers for the Intel Architecture 64 (IA-64) that is an evolution of the established MIPSPro suite of compilers. The use of a production-level compiler infra-structure for teaching is thus new. In this paper we report our experience using the Pro64 in a graduate compiler optimization class. We paired the study of the Pro64 with the use of IMPACT within Trimaran, and with performance studies conducted with the MIPSPro compilers. The students feedback indicate that they valued working with a state-of-the-art compiler infra-structure and studying open research topics for their class projects.

Keywords— **Compiler Optimization, Code Generation, IA-64, Pro64, Trimaran, IMPACT, NUE, Ski, SUIF**

I. Introduction

In this paper we describe our experience using a production level compiler infra-structure in a first-year graduate class focused on compiler design and optimization. By design, this class concentrates on the optimizations that take place either at the intermediate level representation or at the back end of the compiler. We assume that the front end — the traditional parsing and lexical analysis performed at the front end of the compiler — is given. However, even students with no background in compilers should be able to relate the text of a program source code with its intermediate representations and the transformations that take place at the middle and back-end of a compiler. Thus, in the first two weeks of the course we present a walk-through of the front-end: the transformation of a program source code into an abstract syntax tree, a three-address code representation, the formation of a control flow graph, and the data dependence graph of each basic block.

Various compilers are available for use in a graduate level compiler course. Well known examples include SUIF [wwwf], Impact/Trimaran [Lab], and gcc [wwwa]. We discuss these alternative compilers in Section VII. We decided to use the Pro64, an infra-structure made available as an open-source development project by SGI in May of 2000. This decision was based on the fact that the Pro64 evolved from the SGI MIPSPro suite of compilers, and thus it is a production level compiler that incorporates most of the well known optimization techniques implemented in compilers in the industry. Another strong motivation is the fact that the Pro64 targets the IA-64 architecture, a new architecture jointly developed by Hewlett-Packard and Intel that includes a number of features that have direct impact on the code generation process: parallelism expressed explicitly at the machine code level, organization of instructions into bundles, predication, control and data speculation, a register stack with a register stack engine, rotating registers, and hardware supported software pipelining. Last, the fact that the source code for the Pro64 is freely available for the scientific community and that its developers are inviting contributions to the infra-structure makes it a very attractive option for teaching and researching in an academic environment.

The very characteristics that recommend the choice of a production compiler as an infra-structure for the lab assignments in a graduate class — the exposition of the students to the structure of a production level tool, and the opportunities to explore many optimization techniques within a single framework — also lend to a fairly steep learning curve for the students. Our goal when relating our experience in this paper is to provide an approach to the daunting task of getting acquainted with such a complex piece of software, while creating opportunities for a more in depth analysis of some of the optimization techniques.

This paper is organized as follows. In Section II we describe the Pro64 suite of compilers and discuss some techniques used by the students to get acquainted with the infra-structure. Section III describes the assignments in which we use Trimaran for a comparison with the Pro64. In Section IV we describe the performance studies conducted using the MIPSPro compiler suite. Section V briefly describes the HP Native User Environment (NUE) and Ski simulation and emulation tools. Section VI describes the methodology and topics for our class projects. In Section VII we survey the tools and infra-structures used by other graduate courses covering compiler optimization.

*Research supported by grants 239013-01 and 240274-01 from the Natural Sciences and Engineering Council (NSERC), Canada.

II. THE PRO64 OPEN SOURCE INFRA-STRUCTURE

The Pro64 is a suite of optimizing compiler tools for Linux systems running on Intel IA-64 processors. The Pro64 provides compilers for the languages C, C++, and Fortran90/95, and also supports OpenMP. It conforms to the IA-64 Linux Application Binary Interface (ABI) and Application Programming Interface (API) standards. It is open to all researchers and developers in the community. Because the Pro64 was released before IA-64 processors were broadly available, it is important to notice that the code generated by the compiler can be executed under the HP Native User Environment (NUE) (see Section V). The intermediate representation for the Pro64 is WHIRL, which provides five levels of representation. As the compilation progresses, the code is lowered through these levels. Most optimization algorithms are tied to a specific level in the representation. [1] The use of a common intermediate representation allows the integration of compilers for multiple languages that generate code for multiple architectures. The Pro64 itself is an evolution of the SGI MIPSPro suite of compilers.

The Pro64 performs Inter-Procedural Analysis (IPA) and optimizations that include alias analysis, array section, code layout analysis, and fully integrated optimizations such as inlining, cloning, dead function and variable elimination and constant propagation. The IPA is transparent to the user and provides information to the loop nest optimizer, the main optimizer, and the code generator.

At the loop level, most of the well known transformations, such as loop fission, loop fusion, loop unroll and jam, loop interchange, loop peeling, loop tiling, and vector data prefetching are performed [WMC96]. These transformations are applied based on a unified cost model and are integrated with software pipelining.

All traditional global optimizations are implemented using a static single assignment (SSA) form of the code [CCK+97]. The SSA representation is extended in the Pro64 to accommodate unique features of the IA-64. Some of these extensions include the representation of aliases and indirect memory operations, the integration of partial redundancy elimination, and support for speculative code motion.

One aspect of the Pro64 that makes it suitable for the teaching of compiler optimization techniques is the very rich set of switches available in the compiler. These switches allow the user to turn on/off various classes of optimizations such as loop nest optimization, software pipelining, etc. These switches also allow the user to control the behavior of some optimizations. For instance a switch allows the user to specify how aggressive the tail duplication should be in the formation of hyperblocks. Moreover, an extensive set of internal debugging flags can be used to communicate the state of the compiler, among other things, at various stages of the compilation process. These flags allow detailed studies of specific optimizations.

Our approach to use the Pro64 compiler for teaching was (1) to have the students experimenting with various optimizations on SPEC benchmarks[2]; (2) to study a portion of the code for a well known optimization — we studied software pipelining — comparing with another compiler for a similar architecture; and (3) to assign each team of two students a project centered on an open research problem and ask the students to either solve the problem or propose a strategy for obtaining a solution.

Because the Pro64 was developed in an industry setting and only later made available to the research community, it lacks a cadre of people who had a significant hand in creating the code-base, and that can teach new people its inner workings.[3] Thus students working with Pro64 must use a careful and disciplined reading of code and published papers to learn how the compiler is constructed. Fortunately the Pro64 code is well written and well documented. Nevertheless it does require a great amount of time to read and understand.

Students learned to make effective use of simple tools commonly available on Unix systems, such as `find`, `xargs`, and `grep` to read the source code and find portions relevant to the problem under study. When reading code, frequently the student will come across a symbol that seems important to the function at hand, and will want to know what it means. One can search the entire source tree using the above-named tools, save the output for later use in a file named after the search pattern used, and scan it for relevant hits. One can learn about a symbol both by seeing where else and how often it is called, and by reading its definition. For instance, if a function is called from many places it is likely to be a general utility function.

III. A WARM-UP WITH TRIMARAN

Although the combination of Pro64 with NUE forms a strong platform for compiler research, it does not provide an easy-to-use environment for a first exercise on advanced compiler research. Thus for our first laboratory exercises we used Trimaran [Lab]. The advantage of Trimaran is that it allows the user to visualize data dependence graphs, control flow graphs, and to generate static and dynamic statistics of

[1]Most of the description of the Pro64 presented in this section is based on the discussions that Amaral had with Jim Dehnert and Guang Gao for the preparation of the tutorial presented at PACT00 [GADT00]. A large number of individuals contributed to organize the information available in that tutorial and partly transcribed here.

[2]We were forced to use MIPSPro, the predecessor of the Pro64 for this assignment because we did not have an IA-64 machine available for the students during this first edition of the class.

[3]This is in contrast with compilers developed in academic settings, such as the SUIF at Stanford and the IMPACT at Illinois. Some of the creators of those compilers went on to be professors in other institutions and thus are their natural disseminators.

the code generated using a cycle level simulator. Moreover, because the Play-Doh architecture was used in the early studies to define the IA-64 architecture, comparing the code generated for these two architectures the students can learn about the code generation process used in Impact and in the IA-64 [SR00].

Trimaran is a test-bed which is comprised of three major components. First is the HPL Play-Doh architecture from Hewlett-Packard laboratories [KSR00]. This software is used to create descriptions of machines, which can then be used by the simulator. The second component of Trimaran is Elcor, also from Hewlett-Packard Laboratories. Elcor is an intermediate representation which is used by the compiler when performing back-end optimizations [AKR98]. The third component of Trimaran is Impact, a front-end compiler developed at the University of Illinois [ACM+98, CMC+91]. Many ideas integrated in Trimaran and in the Explicit Parallel Instruction Computer (EPIC) concept were originated within the Cydra 5 architecture developed at Cydrome [DHB89, DT93, RYYT89]. Figure 1 illustrates our interpretation of the history of the development of Trimaran. The three components of Trimaran combine to form a machine simulator package that allows the user to customize a machine, compile code for it, and simulate the code execution on the customized machine. Elcor, the intermediate representation of Trimaran, has a textual representation called Rebel. Rebel supports the representation of control flow and data dependences, and it allows the implementation of optimizations for multiple programming languages [AKR98].

Krishna Palem's research group (ReaCT-ILP) produced an easy-to-use GUI interface to integrate the components of Trimaran [wwwc]. Using this interface students can enable and disable optimizations in the compiler, examine the code generated (including control flow graphs and data dependences), run the code generated in the Play-Doh cycle-level simulator, and collect and display statistics of the various types of instructions executed, as well as identify hot regions of the program. Students may then either change optimizations in the compiler or change the architecture specification and examine the effects of these changes in the simulation statistics.

One drawback of using a simulator-based infra-structure such as Trimaran, is that we are limited to small programs that can be analyzed with reasonable effort and that can run under the simulator within reasonable time. Nonetheless, the students benefited from the use of Trimaran in the first portion of the class. A brief description of two of our assignments will best illustrate our use of Trimaran as a teaching tool:

Local Register Allocation Soon after the conventional techniques for local register allocation — liveness analysis, interference graphs, and graph coloring [PBT89, CK91, Cha82, CH90] — were presented in class, the students were asked to examine the Impact source code for register allocation and to write a brief description of the algorithm used for register allocation in that compiler, including the generation of spill code.

Software Pipelining After the students were introduced to software pipelining and rotating registers in class [DT93, DHB89, BRRS92] they were given a simple piece of code (an inner product computation) and were asked to compile the code both in Trimaran and in the Pro64, with software pipeline enabled and disabled. They were then asked to hand analyze the four versions of the code to estimate the number of cycles that each version is expected to execute. They were also asked to compare and find differences between the code generated for the Play-Doh architecture specification by Impact and the code for the IA-64 generated by the Pro64. Although time consuming, the detailed study of a small piece of code gave the students great insight into both how software pipelining works, and the differences between the two architectures. Finally the students were asked to run the code on the Play-Doh simulator and under NUE to compare the results from their analysis with the simulator results.

Trimaran also allows researchers to add their own code transformations and optimizations. Although a suitable tool for architecture research — it generates code for a configurable architecture — Trimaran does not yet generate code for the IA-64 and thus its use for the study of compiler optimizations for that architecture is limited. Moreover, our goal from the start of the class was to use a production-level compiler.

IV. Measuring Performance with the MIPSPro Compiler Suite

Although we had access to the Pro64 source code and to the NUE/Ski simulation environment, in this edition of the class we did not have access to a machine equipped with a IA-64 processor. As an alternative, we had an assignment in which each student in the class was assigned a floating point and an integer benchmark from the SPEC2000 benchmark suite. The students were required to use the performance measuring tools available in SGI systems (SpeedShop) to obtain a detailed profile of the benchmark execution and identify the portions of the benchmark where the code spent most execution time [Inc99]. These measurements were performed in a machine equipped with a MIPS R10K processor, and used the MIPSPro compiler. Each student was then asked to analyze the portion of the code where most of the execution time was spent, and to select a compiler optimization that the student expected to produce changes in the program runtime.

The goal of this assignment is to give the student an op-

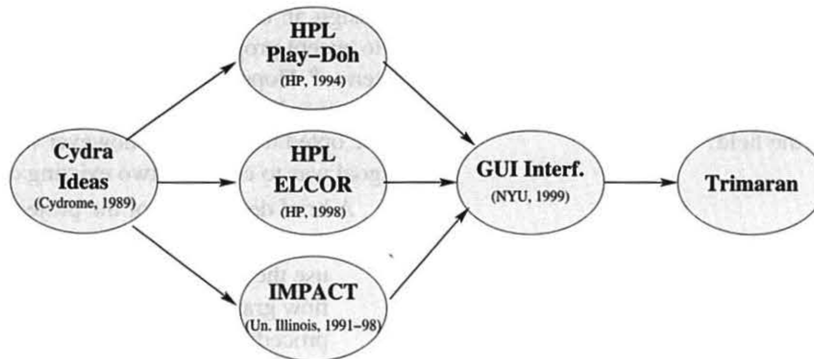

Fig. 1: Evolution of Trimaran

portunity to learn to use the performance measuring tools to study the behavior of a program, to study a compiler optimization of their choice, and to be acquainted with some of the benchmarks in the SPEC2000 suite. Students were then asked to present, in class, an overview of the benchmarks that they studied, the structure of the portion of code where these benchmarks spent considerable amounts of time, the optimization that they studied, and their explanation for why that optimization did or did not produced the execution time differences that they expected.

Students remarked that this assignment consumed a lot of their time. An in-class discussion revealed that by and large they focused on finding an optimization that would produce a substantial change in the program runtime. This was a deviance from the original assignment goal. Understanding why turning off a given optimization, that on first examination appears to be relevant for a program performance, does not produce measurable execution time changes is more informative and less time consuming than searching for an optimization that does make a difference.

V. Using a Compiler Without a Processor: The HP-NUE Environment

Studying compiler optimization on a compiler infrastructure that generates code for a processor that is not yet available is a challenge. However we were not the only ones facing this challenge. As a matter of fact, currently all processor manufacturers are required to deliver an optimizing compiler when the first processors of a new architecture become available. As a consequence, practitioners in the industry are often generating code optimization for processors that are not yet available for testing. Thus students should be trained to work with estimations and indirect measurements of performance. In our case we had the Ski IA-64 Simulator and the Native User Environment (NUE) from HP [Co.]. Ski simulates the IA-64 architecture as a functional simulator of the IA-64. It allows a compiler developer to debug the code generator by following the code generated in the simulator environment. However because it is a functional simulator, it does not allow for a good approximation of the execution time of the code generated. Moreover, Ski simulates a sequential execution of the program, and thus, it cannot reveal potential hazards created by the explicit parallelism inserted by the compiler in the IA-64 assembly. It will, on the other hand, provide statistics on the number of instructions of each type executed by a program. Such statistics are valuable feedback for compiler optimization [Co.00].

NUE is an integrated cross-compiler and emulation tool for a Linux operating system running on an IA-64 processor. It allows the testing of applications developed for IA-64 Linux systems on an IA-32 computer. Often Ski is used in conjunction with NUE both for the development of applications and for compiler development. Although these tools were not suitable for the performance studies that we carried out on MIPSPro/MIPS R10K systems, they are helpful to verify that modifications to the compiler still produce correct code. NUE was also used in class assignments that were designed to make the students more familiar with special features of the IA-64 such as rotating registers, predicate registers, and instruction bundling.

VI. The Class Projects

An important component of our teaching/learning methodology is to assign a research topic for groups of two students. [4] These research topics are then investigated in the form of a class project that runs in parallel with the other activities in the class. First we describe the methodology that we use to manage the course projects, and then we briefly de-

[4] We assigned research topics to students rather than ask the students to come up with a topic. Our reasoning is that in the first weeks of a graduate course, the students have very little knowledge or background on compiler research. Thus the instructor is in a much better position to pick topics that are relevant and that might lead to a thesis topic later.

scribe the project assignments in the first edition of the class. The class projects have two main goals: (1) to provide a focus for the students to learn, in detail, one aspect of the compiler infra-structure; (2) to expose the students to an open research problem that is relevant in the field.

A. Project Methodology

At the end of the second week of class, each student is assigned to a team, and each team is assigned a research topic and a short reading list. The *project specification* provided by the instructor contains a high-level description of a research topic and a first approximation for a problem statement. Two weeks later there is a *project interview* in which the students present a clear and concise problem statement, demonstrate their understanding of the problem, indicate how they intend to attack the problem, and propose a plan of activities for the project. During this interview the instructor has the opportunity to correct misconceptions, advise against overly ambitious proposals, and provide pointers to additional resources.

After the project interview the students should be in a position to write a *project proposal* in which they describe the project in more detail and spell out their plan of activities, including intermediate milestones and any roadblocks that they foresee. Giving feedback on this proposal enables the instructor to once more provide guidance to the students. At the middle of the term the students submit a *project progress report*. To avoid creating undue burden to the students, this progress report is very informal and brief — in our class it was an email message. Nonetheless, because of the many pressures for time from the other activities in the class and from other classes, this report is an important mechanism to make sure that the students focus on the project early on. Again the instructor can advise at this point. Because we work on open research problems, the progress report is an opportunity to refocus some projects.[5] Finally in the last week of classes each group of students makes a *project presentation* where they describe to the class what they learned with the project. Two weeks after the presentations, teams submit a *final project report*. Besides these formal mechanisms to follow a project, at any class meeting a student can be asked to give a *water cooler report* of the project, *i.e.* a 3-minute description of progress made.

B. Project Topics

There is a tradeoff on choosing a topic for a project in such a class: one might ask the students to re-implement a known optimization in the compiler and expect to see performance measurements at the end of the term. Alternately one might assign an unsolved open research problem and be prepared to accept proposals for solving the problem at the end of the term. [6] Hopefully the study of some of these problems lead to either Master or Ph.D. level research topics. By and large we opted for the latter, however we did have a project whose goal was to compare two existing optimization techniques.

A brief description of the projects in our class follows:

- To implement partial function inlining. The idea is to use the frequency of execution of the edges of a control flow graph of a procedure and select a *hot* portion of the procedure to be inlined, thus reducing the number of function calls executed while keeping the code bloating to an acceptable level.
- To investigate the effectiveness of the rotating register mechanism in software pipeline.
- To measure stride regularity in the traversal of pointer-based data structures. This project is a continuation of the research on prefetching induction pointers implemented earlier in the MIPSPro [SAG+01].
- To compare aggressive tail duplication in the algorithm for hyperblock formation, as proposed by Scott Mahlke [Mah96] with a more conservative approach that results in less tail duplication.

The main goals of the projects — to afford the students an opportunity for in-depth learning of an optimization technique in the compiler infra-structure and the investigation of an open research problem — were accomplished. The partial inlining idea is now under pursuit as a research project.

VII. Other Compiler Optimization Teaching Experiences

To put our approach of using a production level compiler infra-structure to teach compiler optimization at the graduate level in perspective, in this section we briefly describe the compiler infra-structures and teaching approach in similar graduate courses taught at other universities. Most of the instructors surveyed are well know. Although we had some private communications with some of them, most of the information related here is from the course descriptions at the course web-sites. In Table I we list the websites for the compiler, simulator, and virtual machine infra-structures refered to in the paper.

The Stanford University Intermediate Format (SUIF) compiler is used in a number of courses covering compiler optimization [wwwf]. A feature that makes SUIF an appealing research tool is the modularity of its design. Each stage of the compiler is written as a separate module. This modularity al-

[5] In this first edition of the class, when the progress reports were turned in we knew that the IA-64 hardware would not be available for performance measurements before the end of the term. Thus we adapted some of the projects for limited experimentation under NUE/Ski.

[6] In our class, although the students were told early that they were being assigned open research problems, they assumed that they were expected to solve the problems and to present performance measurements at the end of the term. Thus the re-evaluation of the goals at the progress report time was very important.

Infra-Structures	Location
Pro64	oss.sgi.com/projects/Pro64/
SUIF	suif.stanford.edu/suif/suif2
Trimaran	www.trimaran.org
HPF	www.crpc.rice.edu/HPFF/
Gnu gcc	gcc.gnu.org
IMPACT	www.crhc.uiuc.edu/IMPACT/index.html
Sablevm	www.sablevm.org
Jalopeno	www.research.ibm.com/jalapeno/
NUE	www.software.hp.com/LIA64
SimpleScalar	www.cs.wisc.edu/ mscalar/simplescalar.html

TABLE I: Websites for compilers, simulators, and virtual machine websites.

lows for new passes to be easily integrated into the compiler. A student can create his/her own compiler by combining different stand alone programs that communicate using files and invoking different passes using the SUIF driver program that operates on intermediate representations in memory. Typically, passes are applied one after the other, however for efficiency SUIF also allows passes to be pipelined to enable the compilation of large programs. Moreover, the intermediate representation of SUIF is extensible and allows the creation of new IR objects. SUIF also includes extensions for object-oriented programming and a browser tool, *sbrowser*, for examining the internal representations. Simple-SUIF is a simplified version of the SUIF compiler that can be more suitable for use in compiler courses and that allows students to develop their own compiler passes.

Todd Mowry (Carnegie Mellon University) uses the SUIF compiler and its internal representation as the basis for his assignments. Besides traditional optimization techniques, his course discusses static single assignment, software pipelining, and memory hierarchy optimizations. Like our course, he has class discussions based on research papers on topics such as pointer analysis, profiling techniques, and dynamic optimizations. For the assignments, he asks that the students implement an optimization pass in SUIF. Typical assignments include writing an analyzer that calculates reaching definitions, or implementing a dead code elimination pass. Similar to ours, his course has a project. The research topics for the projects often center on open research problems such as scalable pointer analysis, profile-driven prefetching in pointer-based structures, optimizing for access locality, and branch prediction.

Ken Kennedy (Rice University) focuses on the back-end of compilers. Different from ours, his course includes topics related to vectorization and parallelization of programs. The focus of these techniques is to uncover parallelism that can be exploited in multiprocessor systems. Topics presented by students include dependence analysis, code transformation, list scheduling, and inter-procedural analysis. The course also covers fine and coarse grained parallel code generation, parallelism detection, and compilation for high performance languages such as C, Verilog, Fortran 90, and HPF [wwwb].

Rajiv Gupta's course (University of Arizona) targets several architectures including superscalar, VLIW, and EPIC style processors. Besides typical back-end issues such as control and data speculation, branch prediction, load/store disambiguation, dynamic and static instruction scheduling, and software pipelining, he covers power issues related to processors and caches, as well as compiling for multimedia instruction sets. The tools used for the course project include Trimaran, the SUIF compiler, the SimpleScalar simulator [wwwe], and FastSim simulators [SL98].

In addition to the essential topics for compiler optimization, Laurie Hendren (McGill University) also covers compilation for object oriented languages. Although her course includes optimizations for both C and Java, the course projects focus on Java. The main tool for the Java projects is the Soot framework. Soot is a compiler analysis tool that is used to analyze and modify Java class files by using a Java intermediate representation, *Jimple* [GH01]. The 2001 version of the course included the use of SableVM, a portable Java Virtual Machine developed by Hendren's research group [wwwd]. Examples of project topics include: decompiling, obfuscation, space optimization, and optimizations for numerical computing (in the Soot framework); and a profiling framework and generational garbage collection (in SableVM).

Tarek Abdelrahman (University of Toronto) uses the Simple-SUIF compiler to study the usual optimization topics. Two of the class assignments require each student to write a compiler pass: (1) to create the control flow graph, and (2) to implement a generic data flow analysis problem solver. A third assignment is similar in scope with a class project and requires each student to implement several optimizations in Simple-SUIF including: loop invariant code motion, induction variable elimination, copy propagation,

Professor	Course Website
Tarek Abdelrahman	Not Available
José Nelson Amaral	www.cs.ualberta.ca/ amaral/courses/680/index.html
Guang R. Gao	www.capsl.udel.edu/courses/cpeg421/2001/
Rajiv Gupta	www.cs.arizona.edu/ gupta/teaching/620/
Laurie J. Hendren	www.sable.mcgill.ca/ hendren/621/main.html
Todd C. Mowry	www.cs.cmu.edu/ tcm
Ken Kennedy	www.cs.rice.edu/ ken/comp515/
Michael D. Smith	www.eecs.harvard.edu/cs253/
Mary Lou Soffa	www.cs.pitt.edu/ soffa

TABLE II: Professors and Websites of Graduate Compiler Courses Mentioned in this Paper

and two other optimizations of the student's own choosing (for graduate students only).

Michael D. Smith (Harvard University) also uses SUIF for homework and class projects. Homework includes writing a pass for dead code elimination, and project topics in the 2000 version of the class included static single assignment, inlining, debugger support, common subexpression elimination, loop-invariant code motion, and code instrumentation and profiling.

Mary Lou Soffa (University of Pittsburgh) focuses on topics related to compiler front-ends. The course also covers an examination of run-time environments, including parameter passing and garbage collection. She also covers traditional optimization techniques such as data flow analysis and local and global optimizations.

Guang R. Gao course at the University of Delaware concentrates on the tradeoffs between software and hardware. This class has a high concentration on register allocation, software pipelining, and loop level optimization.

For space constraints, we must leave out many courses and universities, but the sample above has enough variation to provide information about the varied approaches to teaching compiler optimization and to the alternative infra-structures available to offer hands-on experience to students in such a class. We observe a balance between the foundational topics such as control and data flow and advanced research topics in all the courses examined. Most of the compiler courses examined use compiler infra-structures developed in academia, with few special projects using production-level compilers.

VIII. Final Remarks

Although challenging both for instructors and students, the use of a production-level compiler infra-structure, such as the Pro64, in a graduate class provides the opportunity to tackle state-of-the-art research problems. The students also welcomed the opportunity to analyze the organization of such a mature compiler and to learn from the experience of studying such a complex piece of software. The experience of working with this infra-structure is very similar to the situation that they will encounter when joining a compiler development group in industry. Moreover they will be acquainted with an infra-structure that can be re-targeted, with reasonable effort, to produce code for different processors/architectures. When they change a single aspect of the compiler they are able to measure the effect of such change along with all the standard optimizations already implemented in the production-level compiler.

IX. Acknowledgements

This course could not have been organized around the use of the Pro64 without the help of many friends, including Guang R. Gao, Hongbo Yang, and Alban Douillet at the University of Delaware. James Dehnert and Sun Chan (formerly with SGI), were kind enough to provide advice throughout the class. Shin-Ming Liu is continuously advising us on the continuation of the partial inlining work, now as a research project. Richard Shapiro guided our study of the implementation of hyperblocks in Pro64. We also would like to thank John Anvik, Nathan Bullock, Ling Zhao, and Peng Zhao for the many discussions throughout the class.

References

[ACM+98] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W.-M. H. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *25th International Symposium on Computer Architecture*, pages 227–237, Toronto, Canada, July 1998.

[AKR98] S. Aditya, V. Kathail, and B. R. Rau. Elcor's machine description system: Version 3.0. Technical Report HPL-98-128, Hewlett-Packard Laboratories, Palo Alto, CA, July 1998.

[BRRS92] P. P. Tirumalai B. R. Rau, M. Lee and M. S. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pages 283–29, San Francisco, CA, June 1992.

[CCK+97] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proc. of SIGPLAN 97 Conference on Program-*

ming Language Deisng and Implementation, pages 273–286, May 1997.

[CH90] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[Cha82] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, June 1982.

[CK91] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pages 192–203, Toronto, Canada, June 1991.

[CMC+91] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Walter, and W.-M. H. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *18th International Symposium on Computer Architecture*, pages 266–275, Toronto, Canada, May 1991.

[Co.] Hewlett-Packard Co. ia-64 linux developer tools. http://www.software.hp.com/LIA64.

[Co.00] Hewlett-Packard Co. *Ski IA-64 Simulator Reference Manual*, rev. 1.01 edition, April 2000.

[DHB89] J. C. Dehnert, P. H.-T. Hsu, and J. P. Bratt. Overlapped loop support in the cydra 5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 26–38, Boston, MA, April 1989.

[DT93] J. C. Dehnert and R. A. Towle. Compiling for cydra 5. *The Journal of Supercomputing*, (7):181–227, 1993.

[GADT00] G. R. Gao, J. N. Amaral, J. Dehnert, and R. Towle. The SGI pro64 compiler infrastructure: A tutorial. Tutorial presented at the International Conference on Parallel Architecture and Compilation Techniques (PACT2000), October 2000.

[GH01] E. Gagnon and L. Hendren. SableVM: A research framework for the efficient execution of java bytecode. In *Java Virtual Machine Research and Technology Symposium*, Monterey, CA, April 2001.

[Inc99] Silicon Graphics Inc. SpeedShop user's guide. Technical Report 007-3311-006, Mountain View, CA, 1999. available at http://techpubs.sgi.com/library.

[KSR00] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL-PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett Packard, Palo Alto, CA, FEb. 2000.

[Lab] ReaCT-ILP Laboratory. Trimaran: An infrastructure for research in instruction-level parallelism. http://www.trimaran.org.

[Mah96] S. A. Mahlke. *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[PBT89] K. Kennedy P. Briggs, K. D. Cooper and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pages 275 – 284, Portland, OR, June 1989.

[RYYT89] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The cydra 5 departmental supercomputer - design philosophies, decisions, and trade-offs. *IEEE Computer*, pages 12–35, January 1989.

[SAG+01] A. Stoutchinin, J. N. Amaral, G. R. Gao, S. Jain J. Dehnert, and A. Douillet. Speculative pointer prefetching of induction pointers. In Reinhard Wilhelm, editor, *Compiler Construction 2001 — European Joint Conferences on Theory and Practice of Software*, Lecture Notes in Computer Science, pages 289–303, Genova, Italy, April 2001. Springer-Verlang.

[SL98] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *8th international conference on Architectural support for programming languages and operating systems*, pages 283–294, San Jose, CA, October 1998.

[SR00] M. S. Schlansker and B. R. Rau. EPIC: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, Feb 2000.

[WMC96] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture*, pages 274–286, Paris, France, December 1996.

[wwwa] GNU compiler collection. http://gcc.gnu.org.

[wwwb] High performance fortran (HPF). http://www.crpc.rice.edu/HPFF/.

[wwwc] Real time compilation technology and instruction level parallelism. http://www.cs.nyu.edu/react-ilp/. New York University.

[wwwd] SableVM: A bytecode interpreter. http://www.sablevm.org.

[wwwe] The simplescalar architectural research tool set, version 2.0. http://www.cs.wisc.edu/ mscalar/simplescalar.html. University of Wisconsin.

[wwwf] The SUIF 2 compiler system. http://suif.stanford.edu/suif/suif2.