"Lean and Efficient Software: Whole-Program Optimization of Executables"

Project Summary Report #1
(Report Period: 9/25/2012 to 12/24/2012)

Date of Publication: January 3, 2013

Sponsored by Office of Naval Research (ONR)

Contract No. N00014-12-C-0521
Effective Date of Contract: 09/25/2012
Requisition/Purchase Request/Project No.
12PR10102-00 / NAVRIS: 1100136

Technical Monitor: Sukarno Mertoguno (Code: 311)
Contracting Officer: Casey Ross

Submitted by:

GRAMMATECH

Principal Investigator: Dr. David Melski
531 Esty Street
Ithaca, NY 14850-4201
(607) 273-7340 x. 123
melski@grammatech.com

Contributors:
Dr. David Cok
Dr. Alexey Loginov
Tom Johnson
Brian Alliet
Dr. Suan Yong

Financial Data Contact:
Krisztina Nagy
T: (607) 273-7340 x.117
F: (607) 273-8752
knagy@grammatech.com

Administrative Contact:
Derek Burrows
T: (607) 273-7340 x.113
F: (607) 273-8752
dburrows@grammatech.com

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
JAN 2013		**00-00-2012 to 00-00-2012**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Lean and Efficient Software:Whole-Program Optimization of Executables	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **GRAMMATECH,531 Esty Street,Ithaca,NY,14850**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**13**	

1 Financial Summary

Total Contract Amount (1 year)	$399,984.00
Costs Incurred During the Performance Period (09/25/2012-12/24/2012)	$36,253.83 (see the comments under section 4.1 below)
Costs incurred to date (to 12/31/2012)	$36,253.83
Estimated to complete	$363,730.17

2 Project Overview

Background:
Current requirements for critical and embedded infrastructures call for significant increases in both the performance and the energy efficiency of computer systems. Needed performance increases cannot be expected to come from Moore's Law, as the speed of a single processor core reached a practical limit at ~4GHz; recent performance advances in microprocessors have come from increasing the number of cores on a single chip. However, to take advantage of multiple cores, software must be highly parallelizable, which is rarely the case. Thus, hardware improvements alone will not provide the desired performance improvements and it is imperative to address software efficiency as well.

Existing software-engineering practices target primarily the productivity of software developers rather than the efficiency of the resulting software. As a result, modern software is rarely written entirely from scratch—rather it is assembled from a number of third-party or "home-grown" components and libraries. These components and libraries are developed to be generic to facilitate reuse by many different clients. Many components and libraries, themselves, integrate additional lower-level components and libraries. Many levels of library interfaces—where some libraries are dynamically linked and some are provided in binary form only—significantly limit opportunities for whole-program compiler optimization. As a result, modern software ends up bloated and inefficient. Code bloat slows application loading, reduces available memory, and makes software less robust and more vulnerable. At the same time, modular architecture, dynamic loading, and the absence of source code for commercial third-party components make it hopeless to expect existing tools (compilers and linkers) to excel at optimizing software at build time.

The opportunity:
The objective of this project is to investigate the feasibility of improving the performance, size, and robustness of binary executables by using static and dynamic binary program analysis techniques to perform whole-program optimization directly on compiled programs. The scope includes analyzing the effectiveness of techniques for specializing library subroutines, removing redundant argument checking and interface layers, eliminating dead code, and improving computational efficiency. The contractor expects the optimizations to be applied at or immediately prior to deployment of software, allowing them to tailor the optimized software to its target platform. Today, machine-code analysis and binary-rewriting

techniques have reached a sufficient maturity level to make whole-program, machine-code optimization feasible. These techniques open avenues for aggressive optimization that benefit from detailed knowledge of an application's composition and its environment.

Work items:

We expect to develop algorithms and heuristics to accomplish the goals stated above. We will embed our work in a prototype tool that will serve as our experimental and testing platform. Because "Lean and Efficient Software: Whole-Program Optimization of Executables" is a rather long title, we will refer to the project as *Layer Collapsing* and the prototype tool as ***Laci*** (for **LA**yer **C**ollapsing **I**nfrastructure).

The specific work items are listed below:

1. The contractor will investigate techniques for specializing libraries and third-party components—i.e., techniques for deriving custom versions of libraries and components that are optimized for use in a specific context.
   1.1. The contractor will evaluate program-slicing and program-specialization technology developed independently at the referenced university.
   1.2. The contractor will investigate techniques for recovering intermediate program representation (IR) required for slicing and specialization techniques. The contractor will focus on the following tasks:
      1.2.1. Using static binary analyses for IR recovery.
      1.2.2. Using hybrid static and dynamic binary analyses for IR recovery.
      1.2.3. Studying trade-offs between the two approaches.
      1.2.4. Identifying the approach to be implemented in a prototype tool.
2. The contractor will attempt to implement a prototype optimization tool. This objective can be subdivided into the following subtasks:
   2.1. Implement IR-recovery mechanisms.
   2.2. Extend and improve the implementation of the slicing or specialization technology transferred from the university.
   2.3. Investigate the tradeoff between improved performance through specialization and the resulting increase in executable size.
   2.4. Investigate options for handling dynamically linked components and libraries.
3. The contractor will investigate techniques for further optimization of executables and for collapsing library interface layers. The contractor will consider:
   3.1. Selective inlining of library functions.
   3.2. Specialization of executables to the target platform.
   As time and resources permit, the contractor will attempt to implement these additional techniques in the prototype optimization tool.
4. The contractor will evaluate the prototype optimization tools implemented or received from the university experimentally. The contractor will use synthetic benchmarks, as well as real-world open-source software for the evaluation.
5. The contractor will maintain project documentation and produce comprehensive progress reports and a detailed final report.

3 Staffing

The following personnel are participating in this project.

Dr. David Melski is the GrammaTech Principal Investigator.

Dr. Alexey Loginov is the key architect of the binary analysis infrastructure**.**

Dr. David Cok will be responsible for program management, infrastructure and the user-facing aspects of the resulting tool**.** He is also the PI for GrammaTech's effort on the DARPA Rapid project; that project is producing some key underlying technology that will be used by the Layer Collapsing project.

Dr. Suan Yong is a senior scientist having detailed knowledge of the binary analysis infrastructure and algorithms.

Brian Alliet is the principal implementation engineer.

Tom Johnson is the resident expert on the API for editing the Intermediate Representation of an analyzed binary. He will be consulted regarding the current state and designs for improvement of this API.

David Ciarletta will contribute (beginning 1/8/13) to infrastructure development and measuring overall algorithm and tool robustness.

4 Accomplishments during the reporting period

4.1 Planned level of effort

The principal goals for the first three months of the project were to plan the details of the project work, to assess the applicability of existing tools and algorithms, and to perform some feasibility experiments. Consequently the initial level of effort has been low for the first quarter, while the effort was primarily planning and assessment. Substantial holiday time in November and December and the fact that the relevant engineers were not immediately available because they were winding down other projects also contributed to the slow start. Starting in January, we have 1.5 engineers dedicated to implementation and testing the implementation, based on the initial evaluations and plans. Consequently, the rate of work, measured in both hours of effort and implementation progress, will ramp up very significantly in January.

4.2 Planning

Planning occupied a significant fraction of the initial effort. Our tasks fall into these categories:

- Assessment of current state of technical capability and implementation infrastructure. The technical work in the first quarter primarily fell into this category.
- Planning the engineering staffing for the project
- Implementing the necessary infrastructure (testing, performance evaluation, needed support software, bug and issue tracking, …)

- Researching the specific technical tasks as outlined in the SOW

The result of the planning discussions is shown in the milestone table (section 6). In addition, we selected appropriate engineering staff and started their efforts on the project. [One was not immediately available because of other project commitments, but has now begun work (as of December); a second is beginning work on the project on January 8.]

4.3 Assessment of GrammaTech infrastructure

GrammaTech has an existing infrastructure for analyzing and manipulating a binary executable. It consists of several cooperating pieces:

- The foundation of the project is provided by the CodeSurfer/x86 analysis engine for Intel x86 machine code. This engine analyzes a raw executable, producing an intermediate representation (IR). The CodeSurfer/x86 IR has been enhanced by continued GrammaTech investment and is the basis for many tools and contract activities. For example, CodeSurfer/x86 underlies GrammaTech's CodeSonar flaw-finding tool for binary executables and libraries. CodeSurfer also is a reverse-engineering tool for binaries.

  GrammaTech's analysis engine for C programs is sound and precise (at least for ANSI-C-compliant programs). However, analyzing a raw binary is considerably more difficult. In fact, the problem of disassembling a binary is known to be *undecidable*. Examples of deficiencies in the IR obtained from pure binaries include incorrect data sizes, missed external symbols, and unknown indirect jump targets. (Most problems stem from difficulties disambiguating data from code and pointers from scalar data.)

  In some projects we have used a source-code assist (named DVT – Disassembly Validation Tool). DVT uses source code to assist in the interpretation of the corresponding executable. However, for Laci's deployment scenario, as for the DARPA project referenced below, source code is not generally available. We will hone some of our techniques with the assist of DVT, but we will continually work on improving the IR created from pure binaries. While the undecidability of disassembly prevents us from achieving perfect disassembly on all input binaries, we will establish criteria that allow us to maximize the benefits to the IR for programs that rely on common programming idioms and compiler optimizations.

- CodeSurfer/x86 includes an API for manipulating and rewriting the IR. This capability has been the basis of tools that manipulate the IR and then produce an output executable with new properties; examples are obfuscating the output executable or partitioning between software and firmware (an FPGA).

  One useful mode of rewriting is the *null transform*. This transform analyzes an executable and then writes out a new version of the executable without affecting the executable's safe behaviors. The result will not be the same as the original—the code and data may be moved to new effective addresses and individual code or data blocks may be reordered—but assuming that the program had no unsafe operations (and that the IR was constructed correctly) the original and the transformed versions will exhibit the same behavior. The null transform does not accomplish the optimization

desired by the project, but allows validating the IR construction and essential rewriting infrastructure.

We will be using the rewriting API for Laci. At the high-level three capabilities are required: (i) the ability to analyze the IR to identify the interactions within an executable, (ii) the ability to transform portions of the IR, and (iii) the ability to produce an executable out of the (transformed) IR. The rewriting API provides the second and third capability. However, it does need embellishment and correction to be applied to the purpose of this contract. For example, the API provides partial support for deletion: while individual CFG nodes (corresponding to machine-code instructions) can be deleted, complete CFGs (corresponding to program functions) cannot be deleted at this time. We expect to fix this technical deficiency shortly. Additionally, the API provides no facility for reflecting changes to the CFG in the IR computed from the CFG (such as program-dependence edges). Efficient incremental updates to the IR are not computationally feasible for some components of the IR, such as program-dependence edges (these imply a degree of transitive computation known not to be efficiently maintainable). We expect to rely on Laci's transformations being independent enough to be performed without requiring updating the IR in between transformations. We will evaluate this assumption for each implemented transformation.

- A higher-level client of the rewriting API is a module called *model reduction*, which includes code for computing *compressed CFGs*. Given a CFG and a subset of its nodes designated as interesting nodes, a compressed CFG is a graph in which uninteresting nodes are removed and control-flow edges are preserved between the interesting nodes. We evaluated whether this module could serve as a basis for introducing a new rewriting API primitive. A primitive of this form would improve the performance of UW's specialization slicing. The module appears suitable for providing an efficient primitive but the module's contract with its clients needs to be elaborated: should the primitive always remove exactly the specified collection of nodes (e.g., even if the resulting CFG may be disconnected), thus shifting the burden of the safety of the transformation to its clients, or should it shrink or expand the set of nodes to be deleted in order to satisfy a meaningful validity criterion? If we opt to allow modifying the set of nodes to be deleted, what should such a validity criterion encode? In initial discussions, we elaborated several potential criteria and found the space of criteria to be fuzzy and requiring heuristics.

- GrammaTech is contributing to a DARPA project that also requires analysis of binary executables without the help of source code. As part of the DARPA project we have exercised and extended our techniques to determine how different parts of the executable interact and which portions depend on which other portions.

  In the case of the DARPA project however, the goal is to extract working sub-components that are reusable in new applications. Laci's goal is easier in one respect and harder in another. It is easier in that the subject executable is being rewritten in place. Thus portions of the executable that are inscrutable can simply be left as is—as too complicated to optimize. Furthermore, there is no need to be able to reuse the

result of Laci in a new application; the result is simply reused as a stand-alone application.

On the other hand, the DARPA project allowed for some degree of failure. Laci must be sound in the sense that any transformation it does apply must be known (or at least very likely) to be correct. Thus there is a greater demand on the rigor of the program analyses. The combination of the need for soundness and the ability to analyze complete executables places additional scalability demands on Laci.

In summary, improvements to IR construction will be required to enhance the soundness and the scalability of IR construction. The rewriting API provides a solid foundation for Laci prototyping, although we anticipate making improvements and extensions in the course of the project.

4.4 Evaluation of existing algorithms and software

One of the early tasks (Task 1.1 above) is to assess the executable slicing work at the University of Wisconsin and its relevance to Layer Collapsing. The brief summary of that evaluation is that the algorithms have merit, but the implementation itself will need rework to fit in with the rest of the GrammaTech infrastructure.

The relevant work by Prof. Thomas Reps's team at the University of Wisconsin builds on CodeSurfer/x86 to create slices from executables that can, themselves, be packaged as new, working executables containing only the instructions from the original program that were relevant to the slice. Through our ongoing collaborative relationship with Wisconsin, we were able to access the code repository containing the prototype slicing utility that is the result of this work. The prototype contains the following pieces:

- Three different slicing-based algorithms for constructing new software that performs just the computation in a slice. These algorithms operate at a high abstraction level and make use of the CodeSurfer/x86 API, directing it to perform the actual construction of an executable slice.

- Abstraction code for creating some higher-level primitives for interacting with CodeSurfer's general-purpose API.

- Higher-level rewriting primitives that aren't directly provided by CodeSurfer's API.

Each of these pieces was written in STk (an extension of the Scheme programming language). CodeSurfer/x86 provides two (mostly) comparable API's, one in STk and one in C. The STk version of the API allows much quicker prototyping, as well as interactive experimentation. However, the language takes a bit of familiarity in order to get the best results. It is very easy to naively write code with performance problems.

Our assessment of the current state of the code suggests that it would benefit from cleanup by a more experienced developer, in order to eliminate some common STk errors and take advantage of more of the efficient primitives provided by the language. The current code appears to have scalability issues that would be easily resolved in such a cleanup.

The three algorithms for directing the construction of an executable slice appear to provide relevant tools for this project. It's possible that we may be able to use them as part of a suite of transformations that perform the layer collapsing. The primary question in regards to the applicability of these algorithms is their scalability and performance. Preliminary experiments at UW-Madison indicated that the algorithms may not be suitable for realistic executables, although our examination of the code suggests a number of readily-applicable optimizations.

The abstraction code extends the basic API that CodeSurfer/x86 provides. Its primary goal is to provide some higher-level primitives for commonly performed queries against the CodeSurfer/x86 API. This part of the code would likely benefit the most from rewriting by an experienced STk developer.

The third piece addresses a missing part of CodeSurfer/x86's rewriting API. In particular, the rewriting API does not currently provide good primitives for deleting IR components (a critical step in creating a trimmed-down version of a program). It is possible to effect deletion of various entities in the IR, but the mechanisms are not natural. Thus the Wisconsin team felt the need to create its own deletion API.

In studying this piece, we believe the best course of action would not be to adopt the deletion routines directly, but rather use them as inspiration to add new deletion API routines directly to CodeSurfer/x86 (implemented "under the hood" in C/C++.)

In summary, we believe that the first two components could be reused for this project - albeit with some rewriting. We plan to draw from the third component for new rewriting API infrastructure implemented internally in CodeSurfer/x86. Upon completion of some essential cleanup and optimization, we will be able to evaluate the scalability and precision of specialization slicing on more realistic executables.

4.5 Initial experiments and prototyping

In December, we began applying the existing infrastructure to the Layer Collapsing task. These experiments assessed the existing state of the infrastructure and provided some early data on the amount of implementation work to be expected in the remainder of the project. The following list summarizes the key activities and accomplishments.

- Updated the *pretty printer* (the module responsible for producing the assembly that is later assembled into object code with the tool `nasm`) to support the "full" output mode, which emits the entire compilation unit. Previously it only supported the "partial" output mode where the client chose only certain symbols to output.

- Added stronger type checking to the `ast-create` function that synthesizes new instructions and data. Previously, one could create invalid ASTs, causing subsequent crashes in code that traversed the ASTs. This problem was discovered when testing rewriting examples in the older portions of the manual. The TSL representation of the Intel x86 ASTs has changed slightly since the creation of the examples, but `ast-create` continued to accept invalid ASTs.

- Upon making the adjustments enabled by stronger type-checking of `ast-create`, we validated all rewriting examples in the manual (fixing a few minor issues along the way).

- Performed the *null transform* (which emits the input code and data, although usually with blocks of code and data reordered) on various binaries. After fixing a few minor issues, we found the null transform to work well with the assist of DVT technology to produce reliable IR.

- Attempted the null transform without the use of DVT. We discovered several issues stemming from imperfect disassembly. An illustrative example is the following instruction: `lea ebx, [eax+<number>]`.

  It is generally impossible to know whether `<number>` refers to a symbol (e.g., it is the address of a global array) or a scalar (it is the offset into an array or a structure). This is one of the sources of the undecidability of disassembly. IDA Pro, which forms the basis of our disassembly generally assumes that `<number>` is a scalar (and that this instruction sets register `ebx` to point to the element or field at offset `<number>` of an object pointed to by register `eax`). Whenever `<number>` corresponds to a global symbol, we found disassembly to be incorrect. This can lead to the omission of the global symbol from the output, i.e., an unsound program transformation. Improving disassembly choices by means of advanced analyses, such as *value-set analysis*, will be an important part of this project.

- Developed two transformations to further test the complete infrastructure, evaluate the rewriting API, and gain experience with writing transformations:

  - The first transformation is unlikely to be important for Laci long-term but allowed us to stress-test some rewriting functionality. We implemented a *de-jump* transformation, which removes jump tables (accessed via instructions such as `jmp [eax*4+table]`), replacing them with a series of conditional branches (encoded by pairs of instructions such as `cmp eax,1; je table_1`). This tested the creation of instruction *hammock regions*, instruction-ast creation, the creation of instruction regions with multiple exit edges, and hammock region replacement.

  - The second transformation is the first example of a useful Laci transformation. We implemented a dead-code removal transformation, which removes entire functions that are unreachable from entry points of the executable. Starting at program entry points, we traverse the System-Dependence Graph (SDG—a data structure that represents the entire program), marking every Program-Dependence Graph (PDG—a data structure that represents a program function) that we encounter as reachable. Unreachable PDGs are then pruned out from IR before output. Initial evaluation of this simple technique is promising. After addressing some initial issues, we observed a 1-3% reduction in the size of several executables subjected to the transformation.

5 Goals for the next reporting period

In the next reporting period we expect to begin or complete the following (see the milestones table for dates):

- Complete the initial implementation of the null transform.
- Complete the evaluation of the UW technology.
- Design and implement the API for IR transformation.
- Continue the investigation and implementation of dead code removal.
- Begin the investigation of selective inlining.
- Put in place the infrastructure for testing the evolving prototype. The infrastructure will grow to be able to report performance of the optimization tool and the success rate of optimizations when using different prototype optimization techniques.
- Add synthetic and real-world benchmarks to the testing suite

6 Milestones

Interim results on multi-month tasks will be reported in the quarterly progress reports.

Milestone	Planned Start date	Planned Delivery/ Completion Date	Actual Delivery/ Completion Date
Kickoff meeting		As scheduled by Technical Monitor	
Evaluation of structure and code quality of UW technology (task 1.1)	10/2012	11/30/2012	11/30/2012
First Quarterly report (task 5)		1/3/2013	1/7/2013
Investigate and implement dead-code removal of entire functions(task 3)	12/2012	3/31/2013	
Implement a testable working prototype with the null-transform option (the foundation for tasks 2 and 4)	12/2013	2/28/2013	
Continuing task: Identify failures resulting from incorrect IR; correspondingly improve or repair the IR recovery techniques. (tasks 1.2 and	12/2012	9/24/2013, with all individual improvements noted in	

2.1)		quarterly reports	
Identify common coding idioms and compiler transformations that result in incorrect disassembly (task 2.1)	1/2012	2/15/2012	
Implement a testing infrastructure (task 2.3 and task 4)	1/2013	2/28/2012	
Design and implement the IR editing infrastructure (task 2).	1/2013	4/30/2013	
Evaluation of performance and precision of UW technology (task 1.1)	2/2013	3/31/2013	
Develop real-world and synthetic benchmarks to evaluate performance (task 4).	2/2013	9/24/2013, with interim progress each month	
Investigate disassembly improvements such as learning-based bottom-up disassembly and all-leads disassembly (task 2.1)	3/2013	5/31/2013	
Investigate selective inlining of library functions (task 3.1)	3/2013	7/31/2013	
Second quarterly report (task 5)		4/3/2013	
Investigate finding and deleting functionally dead code, possibly using slicing and specialization (task 2.2 and 3.2).	4/2013	8/31/2013	
Investigate specialization to target platforms or target environments (task 3.2)	4/2013	8/31/2013	
Implement aspects of the chosen disassembly extensions (task 2.1)	5/2013	8/31/2013	
Evaluate hybrid analyses as a complement to static analyses for recovering IR (Task 1.2)	5/2013	8/31/2013	
Third quarterly report (task 5)		7/3/2013	
Measure the performance tradeoff of various optimizations and evaluate the	7/2013	9/24/2013	

overall tool (task 2.3 and 4)			
Investigate options for handling DLLs (task 2.4)	8/2013	9/24/2013	
Final report (task 5)		10/24/2012 (contract end date)	

7 Issues requiring Government attention

There are no current issues.