“Lean and Efficient Software: Whole-Program Optimization of Executables”

Project Summary Report #1
(Report Period: 6/30/2014 to 9/30/2014)

Date of Publication: Oct 10, 2014

Sponsored by Office of Naval Research (ONR)

Contract No. N00014-14-C-0037
Effective Date of Contract: 06/30/2014

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

Submitted by:

Principal Investigator: Thomas Johnson
531 Esty Street
Ithaca, NY 14850-4201
(607) 273-7340 x. 134
tjohnson@grammatech.com

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
10 OCT 2014		**30-06-2014 to 30-09-2014**

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, Inc,531 Esty Street,Ithaca,NY,14850-4201**	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	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**7**	

1 Financial Summary

Contract Effective Date	06/30/2014
Contract End Date	06/30/2016
Reporting Period	06/30/2014 – 09/30/2014
Total Contract Amount	$602,165
Incurred Costs this Period	$85,702
Incurred Costs to Date	$85,702
Est. Cost to Completion	$516,463

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:

Our objective in this project is to substantially improve 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: specializing library subroutines, removing redundant argument checking and interface layers, eliminating dead code, and improving computational efficiency. In particular, we will apply specialization and partial evaluation technology, integrating the new technology with the techniques developed during the previous contract effort. We expect the optimizations to be applied at or

immediately prior to deployment of software, giving our tool an opportunity 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. Thus, we believe there is now a great opportunity to design tools that will revolutionize the software development industry.

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 for the base contract period are listed below:

1. **Investigate specialization opportunities.** The contractor will design and implement limit studies that will help focus the search for fruitful applications of partial evaluation and set goals for attainable improvements.
2. **Transfer UW technology.** The contractor will transfer program-specialization or partial-evaluation technology from the University of Wisconsin and integrate it into the contractor's tool chain.
3. **Improve and extend UW technology.** The contractor will improve the robustness and scalability of the transferred technology, and complete partially implemented components and functionality.
4. **Improve and extend IR construction and rewriting.** The contractor will improve intermediate-representation construction and rewriting infrastructure as needed to demonstrate functionality on the primary test subjects.
5. **Develop and maintain test infrastructure.** The contractor will create an extensive suite of test applications, and will maintain and extend it as necessary. The contractor will also implement validation and measurement functionality that will enable tracking the robustness and benefits of program transformations.
6. **Investigate security implications.** As time permits, the contractor will study the effect of different instruction-generation mechanisms, such as peephole superoptimization, on security. As time permits, the contractor will also study whether polyvariant specialization enables (i) the creation of finer security-relevant models of program behavior and (ii) more accurate or efficient enforcement of security policies. If earlier tasks that are essential in completing a functional prototype require more effort, we propose to shift this task to the option period, with the possible adjustments of lower effort on either or both of the first two option-period tasks.

7. **Produce deliverables and attend required meetings.** The contractor will produce technical documentation in the form of reports and a working software prototype. The contractor will attend meetings requested by the program monitor.

3 Accomplishments during the reporting period

This report covers the first three months of the base contract period. Given the length of calendar time that elapsed (1 year) between the completion of the previous contract funding design and development of LACI and the start of the current contract, a substantial level of effort was required to "dust off" the original prototype, ramp up management and planning activities, and refresh the expertise of our development team on the capabilities of the existing technology.

However, some initial technical progress has been made during this period as well. We:

- Reviewed the technical design for LACI and developed a plan for enhancing the rewriting capabilities to be more robust in the face of disassembly ambiguities.
- Engaged in initial discussions with our colleagues at the University of Wisconsin (UW) to assess progress on UW's specialization slicing and partial evaluation technologies.
- Brought in-house the prototype for UW's specialization slicing to connect with LACI and evaluate its capabilities.
- Began converting LACI's implementation to leverage the more robust rewriting mechanism.
- Began adding support to LACI to handle 64-bit executables.

The following sections provide details on these accomplishments.

3.1 Making Rewriting More Robust

During the first phase of this project, we invested substantial effort in performing rewriting correctly. Our approach was to focus on improving LACI's IR recovery (provided by CodeSurfer/SWYX) to eliminate disassembly errors. This enabled us sufficient robustness to handle a substantial number of executables, including the entire coreutils utility suite. However, it seems clear that IR recovery can never be fully general. Larger programs inevitably contain some characteristic for which IR recovery fails, leading to LACI producing a rewritten program that contains potentially fatal errors.

We decided to take a step back at the beginning of this second phase to re-assess our approach. The bottom-line conclusion is that, while it's certainly beneficial to have the IR recovery possible, LACI's rewriting process must account for the potential that the IR recovery contains one or more disassembly errors.

The REINS[1] system introduced a new style of rewriting that allows for disassembly ambiguity. Under a related DARPA-funded SBIR, "Automatic Detection and Patching of Vulnerabilities in Embedded Systems" (W31P4Q-14-C-0083), GrammaTech is drawing inspiration from the REINS approach to develop ADAPT, a verifiable rewriting platform. Like

REINS, ADAPT offers correct rewriting even when disassembly is ambiguous. Unlike REINS, ADAPT goes further to ensure preservation of intended program semantics and incorporates a verification framework to prove correctness of both the original semantics and added security policies.

A key feature of both REINS and ADAPT is the way they structure the rewritten executable. The original code and data are preserved verbatim in the rewritten executable. However, only the data is actually used. The original code is retained as a conservative guard against programs that store data intermixed with code (or even read their own code as data). The rewritten code is added to the executable in a separate address range. The rewritten code is structured such that data and code references all refer to the original code and data. Thus the rewritten code "thinks" it is still executing in the context of the original executable. This requires that function pointers, return addresses, and indirect jump targets be translated on-the-fly at runtime from the original code space to the rewritten space.

This approach does incur overhead. The need to perform translation of code addresses dynamically will necessarily add cost to each indirect control-flow transfer. However, this cost comes with the promise of higher confidence in the robustness of the rewriting process. For LACI, we believe this will provide a fair balance. And LACI's optimizations should recover the overhead incurred by optimizing latent inefficiencies present in the transformed executable.

We began this quarter to convert LACI to leverage ADAPT's new rewriting infrastructure. This required some enhancement to the CodeSurfer/SWYX infrastructure in order to support the conservative notion of disassembly that ADAPT's rewriting requires. This component has been completed. However, an outstanding task is to rework the reassembly and relinking toolchain to support retaining the original code and data sections verbatim in the generated executable. We expect this to be completed in the next reporting period.

3.2 Evaluation of UW Technology

During Phase 1, we reviewed UW's executable slicing technology. At that time, we had deemed the technology not yet ready to incorporate into LACI. During the first quarter of the current contract, we reviewed the status of this technology with UW. In addition, we discussed new work that UW is developing on partial evaluation and synthesizing instructions from logical QFBV formulae. These latter two capabilities are still not ready to transition (though, they may be more solid by the end of November); however, we have decided to begin transitioning the executable slicing.

We've brought the code in house and have begun the process of connecting it to LACI. Some holes will need to be filled in, however. The basic framework for using the technology on LACI will be to construct an executable slice backward from all points in the subject program that trigger externally visible output (whether that be text printed to the display, data written to a file, or setting the program's return status.) So it will be necessary to implement

an analysis that constructs that set of points. We have started on this, but have not completed it yet. We plan to continue work on it in the next month.

3.3 Improved IR Recovery Infrastructure

A portion of the quarter was invested in improving the basic IR recovery capabilities that LACI builds on. Because of the calendar delay in starting the second phase of the project, LACI had suffered a level of "bit rot" due to changes in the underlying CodeSurfer/SWYX technology. While much of the effort involved simple cleanup work, we also decided to tackle two meatier technical problems: support for 64-bit code and more general-purpose handling of jump tables.

The original LACI prototype supported only 32-bit software on the x86 platform. Given that modern computer systems have by and large shifted to 64-bit software, we believe it's important that LACI transition to 64-bit software to remain relevant for modern systems. At the time when the Phase 2 contract started, much of the CodeSurfer/SWYX infrastructure underlying LACI had already been extended to support 64-bit software; however, some components that LACI depends on were still 32-bit only. During the first quarter of this project, we completed implementation of these remaining components. There is remaining work to do to extend the LACI transforms to work for 64-bit software. We plan to tackle this in the coming months.

The original handling of jump tables in CodeSurfer/SWYX was implemented using ad hoc pattern matching. While effective for common patterns, the technique can be misled by slightly abnormal code structure. A more principled approach is to symbolically evaluate the code leading up to the use of a jump table to determine the location and bounds of the jump table and, consequently, the set of targets reachable at the indirect control-flow transfer instruction that uses it. This is natural to do with CodeSurfer/SWYX's ability to represent the semantic behavior of a sequence of instructions as a QFBV (Quantifier-Free Bit-Vector) logical formula. We implemented this symbolic evaluation this month and initial testing demonstrates that it functions quite well. This improvement will help LACI by providing more robust understanding and handling of indirect transfer instructions.

4 Goals for the next reporting period

In the next reporting period we expect to complete the following:

- Complete conversion to the ADAPT rewriting technique.
- Implement the necessary connectivity to exercise UW's specialization slicing code in LACI's context.
- Review UW's progress on partial evaluation and instruction synthesis.
- Continue to improve support and robustness for both 32-bit and 64-bit software.

5 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 Mtg		9/4/2014	9/4/2014
Transition Specialization Slicing	7/2014	12/2014	
Robustness & Reliability of IR & Rewriting	7/2014	12/2014	
First Quarterly Report		9/30/2014	
Transition Partial Evaluation and Instruction Synthesis	12/2014	5/2015	
Second Quarterly Report		12/30/2014	
Third Quarterly Report		3/30/2014	
Evaluation	4/2015	6/2015	
Final Report		6/30/2014	

6 Issues requiring Government attention

None.

Reference List

1. Wartell, R., Mohan, V., Hamlen, K., and Lin, Z., *Securing untrusted code via compiler-agnostic binary rewriting.* In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*. 2012. pp. 299-308.