A COMPILER FOR CREATING EVOLUTIONARY SOFTWARE AND APPLICATION EXPERIENCE

Brian K. Schmidt
Monica S. Lam

Technical Report No.: CSL-TR-99-782

April 1999

This project is, in part, supported by funding from an NSF Young Investigator Award and by Sun Microsystems, Inc.

A Compiler for Creating Evolutionary Software and Application Experience

Brian K. Schmidt and Monica S. Lam

Technical Report No.: CSL-TR-99-782

April 1999

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
William Gates Computer Science Building, 4A-408
Stanford, CA 94305-9040
<pubs@shasta.stanford.edu>

Abstract

Recent studies have shown that significant amounts of value repetition occur in modern applications. Due to global initialized data, immediate values, address calculations, redundancy in external input, etc.; the same value is used at the same program point as much as 80% of the time. Naturally, attention has begun to focus on how compilers and specialized hardware can take advantage of this *value locality.* Unfortunately, there is significant overhead associated with dynamically recognizing predictable values and optimizing for them; and all too, this cost dramatically outweighs the benefits

There are various levels at which value locality can be observed and used for optimization, ranging from register value re-use to function memoization. We are concerned with predictability of program variable values across multiple runs of a given program. In this paper we present a complete system that automatically translates ordinary sequential programs into *evolutionary software*, software that evolves to improve its performance using execution information from previous runs. This concept can have a significant impact on software engineering, as it can be used to replace the manual performance tuning phase in the application development life cycle. Not only does it alleviate the developer from a tedious and error-prone task, but it also has the important side effect of keeping applications free from obscure hand optimizations which muddle the code and make it difficult to maintain or port. This concept can also be used to produce efficient applications where static performance tuning is not adequate.

Our system automatically identifies targets for program specializations and instruments the code to gather high-level profiling information. Upon completion, the program automatically re-compiles itself when the new profile information suggests that it is profitable. The programmer is completely unaware of this process, as the software tailors itself to its environment. We have demonstrated the utility of our system by using it to optimize graphics applications that are built upon a general-purpose graphics library. While much of this work is based on well-established techniques, this is the first practical system which takes advantage of predictability in a way such that the overhead does not overwhelm the benefit.

Key Words & Phrases: compiler, optimization, profiling, program specialization, evolutionary, value redundancy, value prediction.

Copyright © 1999

Brian K. Schmidt and Monica S. Lam

1 Introduction

Despite recent advances in compiler technology, manual performance tuning remains an important phase in today's software development cycle. This hand optimization is tedious and time-consuming, and it often has the unfortunate side effect of obscuring the code, thus making the software more difficult to debug, maintain, and port to other platforms. This problem is even more critical today in the face of a trend toward increased use of general-purpose software libraries and common software architectures to improve programming productivity. To support a wide range of uses such libraries must be general in nature and provide a modular interface. All too frequently, however, the cost of providing this generality is reduced performance. While the need for tuning the code to adapt to one's program is great, the cost of nonconformity to standard interfaces can be greater.

One of the most common forms of performance tuning is *specialization*. Specialization is the technique of generating alternate paths through the code that are optimized for certain conditions. Since value locality is so common, this can be a particularly profitable optimization. There are several stages to specialization: determining the set of special cases, generating the specialized code, managing the specialized versions, and choosing the right version to execute. A software developer may know the important cases on which to specialize *a priori*, or may use profiling to determine the common cases that warrant specialization. Unfortunately, every specialization makes the code more complex and harder to maintain, and the management of different versions can become quite cumbersome. The programmer is often faced with the difficult trade-off decision of when to specialize. Also, program specialization is static in nature; programmers create the specialized versions according to the expected common cases, possibly derived from observing some sample input programs. However, the behavior of the code may be highly dynamic and may not have a set of common cases.

A particularly attractive solution to achieving specialization is to create self-adaptive software, code which can automatically tune itself in an evolutionary fashion as it is run [4]. The idea is that every time a program is run, it is automatically profiled, and the profile data is used to generate an optimized version in its place. Technically speaking, this concept is not too different from the well-established idea of profile-driven program optimizations. However, it has a significant advantage from a usage perspective. It is fully automatic, requiring no intervention from the programmer. In fact, the programmer does not even know that the binary is automatically being updated. This is important because past experience suggests that profile-driven optimizations are seldom employed in practice due to the lack of tools to make good use of the profile data. The programmer is often left to the task of poring over the profile data, looking for optimization opportunities. In addition, this paradigm allows the code to continually evolve and track the actual usage, instead of having just a single round of improvement. We term this *evolutionary software*.

There are many challenging difficulties in the design of such a system: how to reduce the profiling cost so that the overhead is negligible and acceptable, deciding what information to gather, how to gain enough performance to justify the system, and how to manage the specializations. This paper presents the first end-to-end system that automatically compiles a program into self-adaptive code, and we demonstrate its utility for real applications. Our system consists of the following:

- **Identifying candidates for specialization.** Unlike typical partial evaluators which attempt to derive as much statically known values from a given set of inputs as possible, our compiler identifies what it wishes to use for specialization by analyzing the opportunities for optimization.

- **Targeted high-level profiling.** Instead of the usual approach of measuring the commonly executed paths using standard profiling techniques, our compiler uses a high-level targeted profiling scheme. It inserts code into the program to record high-level information that has been determined to be useful for optimization. The run-time overhead is negligible as only a small amount of profile data are collected in each run.
- **Run-time environment for specialized codes.** The self-adaptive software uses specialized codes whenever possible, collects the profile data, and automatically invokes the compiler to create new specialized code upon completion of the execution. The program automatically manages the cache of specialized instances for future runs.

This system can be applied at different stages of software development and execution. A software developer can use the system to specialize the software before packaging the code for shipment. If determining how the software should be optimized is highly sensitive to how it is used dynamically, then this concept can be applied at execution time. The adaptation may be user-specific, application-specific, or site-specific. Users may wish to generate optimized code based on their personal use so that they can achieve high performance without conflicting with others. Libraries that are developed with our tool can be shipped with a smaller footprint and then automatically adapt to the usage patterns at a given site, allowing behavior of multiple users to be aggregated for the benefit of others. In the case where the distributed software is proprietary, it is necessary for the specialization to be applied to the binary code. Previous research results on enhanced binary representations that provide enough information for such optimizations [12] and tools for manipulating binaries [8] are applicable here.

We are advocating a paradigm where the software is compiled, executed and re-compiled to be further optimized. For programs such as operating systems and network kernels, we cannot afford to bring the system down to install the specialized code, and it is necessary that we dynamically link in the new code. If the advantage of the specialization is limited, then trade-offs between the compilation time and the execution time become significant, and various techniques created in research on dynamic code generation are applicable.

In the next section, we describe the real-life experience in the development of a graphics library for rasterization to provide a concrete example of the difficulties in performance tuning. In Section 3, we present an overview of the system, its overall design and the algorithms used. Section 4 describes our experience in using our system to optimize two applications using a rasterization library that has not undergone manual performance optimization.

2 An Example — A Graphics Library

The Stanford Computer Graphics Laboratory has developed a large graphics library, which serves as an excellent example of a case where evolutionary software could have an enormous impact. This library is named FruGL and is comprised of roughly 15,000 lines of C code which implement a subset of the standard OpenGL library interface. It is used heavily by the graphics lab for research, experimentation, evaluation, and development. FruGL is intended for use by performance-critical applications, and so it has been highly optimized. Currently, it is capable of rendering on the order of one million triangles per second, with a goal of rendering 10 million triangles per second. Since rasterization is a central process to any graphics library, particular attention has been paid to optimizing the rasterizer.

FruGL's rasterization routine is responsible for rendering all the polygons in a given scene, and it represents a major performance bottleneck for the library. This routine iterates over a typically large set of polygons and performs the calculations necessary to render the pixels they circumscribe. Each polygon may be rendered in a large number of modes based on the current scene state, e.g. whether or not Z-buffering is being employed. However, even the most complex scenes typically exercise less than ten modes. The decision as to which mode to use is performed by a large numbers of conditionals in the inner rasterization loop. This is extremely inefficient, since the predicates are based on variables which are read-only in the loop body. Hence, the conditionals could be hoisted out of the loop and performed once, allowing the loop to be scheduled much more efficiently. However, the number of conditions that are possible is so large (over 16,000) that generating all the possible combinations statically would result in massive code blow-up. This is of particular concern for low resource environments such as a network computer in which there will only be enough storage to cache a few specialized versions of the code.

To address the rasterization performance issue, the developers of FruGL decided to create certain specialized versions of the rasterizer. They hand-picked a set of 777 rasterization modes that were believed to be the most likely to occur in practice. Specializations for these cases were then generated manually. This approach was problematic for a number of reasons. First, since FruGL applications only exercised a small number of modes in practice, a great deal of effort and storage was wasted on unused code. Second, when the specialization choices made by the designers did not reflect actual use, application performance suffered needlessly. Finally, maintaining 777 versions of the rasterization routine has proven to be difficult and error-prone. If this process were automated, only the needed specializations would ever be created, and the developers would only need to concern themselves with maintaining the general version.

3 A System for Evolutionary Software

As a realization of the evolutionary software paradigm, we have developed a simple system using the SUIF compiler infrastructure. We first use the augmented SUIF compiler to analyze the programs for potential targets of specializations. The compiler generates code to choose the specialized versions when available and record new potential targets for specializations otherwise. It also includes code to measure the effectiveness of the specialized code. After each run, the code automatically invokes the compiler again to re-compile the special cases that are deemed to be worthwhile. In the following, we describe each of the major stages of the system for supporting evolutionary software: identifying specialization candidates, profiling, and generating and maintaining self-adaptive code.

3.1 Identifying Specialization Candidates

The process of identifying specialization candidates can potentially be very complex and challenging. However, given that the evolutionary software paradigm is new, we have decided to keep this simple and only automate techniques that have previously been used by hand and proven useful. With our system in place, it is relatively easy to experiment with different heuristics and to extend the capability of this component of the system when we have a better understanding of the nature and characteristics of evolutionary software.

One of the most basic forms of specialization used in practice is to replace a parameter in a program with a fixed value. For example, every programmer has probably used a "debug" flag in his or her program. If performance is not a consideration, the debug flag would have been a parameter that the software developer can supply according to needs. But instead, the debug flag is usually made a constant in a program, and a software developer must re-compile the program whenever the usage changes. A self-adaptive system, on the other hand, can automatically keep track of the two versions of the software, and provide the developer with both speed and simplicity.

While having to re-compile a program to change the debug flag is a nuisance, it does not affect the end user at all since it is permanently disabled when the code is in production mode. However, the debug flag is symptomatic of a more serious problem. Consider the example of a simulator. For generality, it is desirable to make the simulator take as many different settings as possible. However, each allowed choice in the simulator incurs additional overheads. Suppose we wish to simulate the cache subsystem; hardwiring the cache sizes and cache line sizes into the program can convert many general multiplication and division operations into simple shift and add operations.

This observation leads to a simple, yet useful strategy. In the first step, we wish to identify variables that are used to determine the mode of the execution of a program or of a procedure. We look for all the variables that are written only once in the entire program, the scope of specialization ranges from when all the parameters are defined to the end of the program execution. We also look for variables that are invariant within a procedure, by employing a simple flow-insensitive pass through the procedure body to determine those variables that are not modified[1]. The scope of the specialization is the procedure itself. This is especially important for general-purpose libraries which often utilize large numbers of settings to determine their mode of operation.

In the second step, we determine if specializing the code on these variables can be advantageous. We consider the frequencies of operations involving the variables, giving heavier weight to operations nested within loops. We also estimate the advantage of replacing the variables with fixed values. The set of operations we target include:

- **Conditions in determining the control flow.** Knowing the values of the constants can simplify control flow, enable dead code elimination, which in turn can make many more optimizations such as instruction scheduling and register allocation much more effective. Note that since conditions can only have two possible values, the compiler can statically generate two versions of the same loop and insert a test to be performed at run time to determine the version to execute. However, if there are many loop invariant conditions in the program, exhaustively generating all versions can be prohibitively expensive.
- **Expensive calculations such as multiplications and divisions.** Multiplication can be eliminated if the input data are 0. A scenario where this may be interesting is an inner loop in graphics code to apply the same (small) transformation matrix which may contain many zeros to a large data set. Divisions by powers of two can be easily replaced with shift operations.
- **Function pointers.** If the inner loops make function calls through the same function pointer, we can specialize the code for commonly supplied function pointer values.

For example, in the FruGL graphics library, the main rasterization routine accepts a set of read-only variables as input to the procedure. These variables are used as flags to determine the outcome of a large set of conditionals within a loop body. Since the loop could be scheduled much more effectively if these conditionals were hoisted out of it, our system identifies them as potential candidates for specialization.

1. We also check to ensure that parameters do not have their addresses taken. This avoids the problem of performing alias analysis but also potentially reduces optimization opportunities. We plan to incorporate alias analysis in the future.

3.2 Targeted, High-Level Profiling

For each targeted variable, the compiler inserts checks into the code to determine if the values of variables of an execution provide any opportunities for optimization. For example, if the goal is to convert divisions into shift operations, the compiler would consider finding inputs that are powers of two a success. It is not necessary to record the values of the inputs if knowledge of their value has no value in improving performance. In this way, we can keep the profile data we collect small.

In addition to gathering information on the program variables, our system also monitors the effectiveness of specialization by automatically inserting performance instrumentation code on the specialized code regions. The knowledge of the benefits of each specialization allows better cost-benefit analysis in the management of specialized versions.

At the end of each execution, the program automatically outputs a list of newly-encountered values as well as high-level profiling information on the performance of specializations that were exercised during the run.

In the case of the FruGL library, our system inserted code to profile the values of the flags passed into the main rasterization routine. Since the flags represented binary values, our system was able to insert code to record their values using a simple bitmask, thereby significantly reducing the storage and execution time required to maintain the information. In addition, code was inserted to time the general-purpose routine, as well as specialized versions which may be created later on. The profiler also monitors itself to ensure that it is not inducing too high a cost. If so, it scales back the amount of data it collects, restricting itself to the set of values that is most profitable for optimization.

3.3 Code Generation and Evolution

Once our optimization tool has identified, marked, and profiled potential sources of optimization, the program is ready to run, and code evolution begins. Each time execution terminates, the program analyzes the profiling information and identifies any expensive, commonly occurring cases that have not been specialized. Then, it automatically initiates compilation on any newly-identified cases for specialization. In this way, the code evolves over time to adapt to particular usage patterns.

For each code region that may be specialized, the compiler uses a dispatch table to direct execution to the correct specialization (or general code) based on those values. In the case where a specialized version has not been found, the program executes the general version, and the instrumentation code records the encountered values as a potential specialization target. Otherwise, the specialized version is run, and the instrumentation code records the frequency of the use of the specialized code as well as timing information. To minimize the cost of this dispatch mechanism, we use a hash of the run-time value of the data being monitored to index into a table which contains the full value and the location of the specialized code. An optimized routine is used to compare the values and jump to the correct code. Our system restricts the size of the value that can be profiled to ensure the dispatch overhead is negligible.

At the end of the run, a specialization is created for each new invariant value by cloning the function in which it is utilized and asserting its run-time value statically. The compiler propagates the constant values, removes dead code and invokes the standard optimizations on the specialized code.

Using this information, the compiler is able to automatically enumerate all the useful cases to specialize, as well as to identify the expected range of values for run-time constants. Further, only the cases that are actually encountered will ever be analyzed in this fashion. This is an extremely general and robust technique which incurs very little overhead. This analysis is used to direct future code generation decisions.

The program also automatically manages the cache of specialized instances for future runs. Due to storage limitations or other concerns, it is often impractical to cache all the specializations that are created. Thus, over time certain specializations may need to be discarded. The statistics gathered by the high-level profiling are useful for making these decisions. Currently, we use a simple LRU replacement algorithm to make space available. We can augment this scheme with cost-benefit trade-offs, taking into consideration the size and performance improvement of the specialized version over the general case. In this way software can evolve into highly optimized, space-efficient code in a manner which is completely transparent to the end user.

In the case of the FruGL library, 14 flags are monitored, and they are compacted into a single 16-bit integer for use as both the hash and the value for comparison. The bitmask is created and hashed into the table, where it is compared with previously-generated specializations. If one is found, the table includes the address of the appropriate function, which is invoked with any arguments passed to the original routine. Otherwise, the code falls through to the original, general-purpose routine.

4 Sample Graphics Applications

To evaluate the potential gains from our tool, we used it to fully automate the optimization process of the rasterizer in the FruGL graphics library described in Section 2. The FruGL rasterizer is decomposed into separate setup and rendering functions, and based on the large numbers of loop-invariant conditionals found in the procedure bodies, our tool selected both functions for specialization. We used two applications built on top of FruGL to exercise the rasterizer.

The first application, `viewer`, creates a model of a cube and texture maps parts of a Gothic cathedral scene onto each of the faces. The cube may be manipulated by user input, and new scenes are dynamically rendered in response. The second application, `armitest`, creates a complex model of an armadillo action figure and renders it to the screen. Controls are provided to manipulate the model and adjust various settings, such as texture mapping, depth testing, number of polygons, etc. We tested `viewer` and `armitest` on a 134MHz SGI Indy with an R4600 microprocessor. We also ran `armitest` on a 167MHz Sun UltraSPARC processor and on an Intel Pentium Pro machine

Using the above applications, we conducted some preliminary experiments to evaluate the performance gains from using our tool to optimize the above rasterization process. In each experiment we iterated through a number of the possible mode settings for the rasterizer. After the first execution with a particular mode, a specialized rasterizer was automatically created and used in the next run. We gathered timing measurements for the general and specialized versions of the two rasterization procedures for each mode tested. It turns out that both procedures exhibited similar gains. Therefore we discuss the measurements for the primary rasterization procedure only. More detailed experimental results will be presented in the full paper.

In order for our system to be an effective optimization tool, the overhead it introduces by collecting profile information and performing dynamic dispatch must not exceed the performance gains from specialization. Over the course of each run, we measured this cost and found it to be

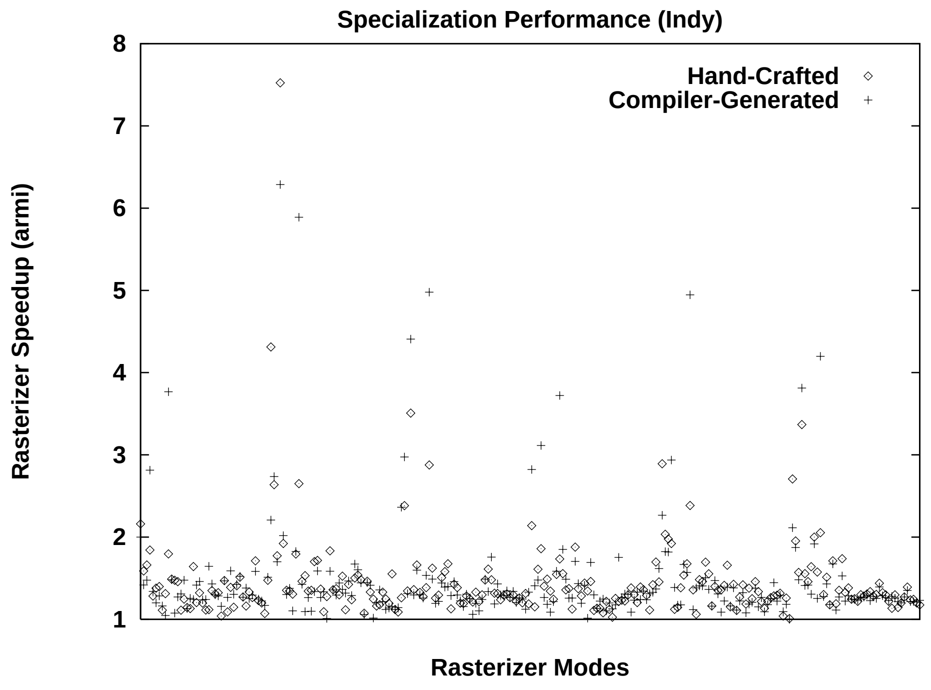

Figure 1 Speedup of individual specializations over the general rasterization routine on an SGI Indy with an R4600 microprocessor. Each point represents the speedup of the rasterization procedure execution time for a particular rasterizer mode setting under the `armitest` application.

roughly 5–7 microseconds for each instance in which a run-time invariant value was recorded and dispatch performed. This is negligible compared to the execution times of the general and specialized rasterization procedures, which were typically hundreds of microseconds.

The first application we tested was `viewer`. The maximum speedup due to specialization over the general case was 3.72 times, but this application exhibited a wide range of performance improvements across the rasterizer settings we tried. This suggests that it would be advantageous to use the cost-benefit ratio in determining the specializations to save in the cache.

The second application we tested was `armitest`. Speedups of rasterization procedure execution times due to specialization over the general case for an SGI Indy are presented in Figure 1. From the graph it is clear that performance improvements were relatively homogeneous with typical gains in the 20% to 30% range, as well as larger speedups of 3 to 7. Figure 2 presents the UltraSPARC results. Again, typical speedups are in the 20% to 30% range, with larger gains up to 11 times speedup. These results are quite respectable, particularly for graphics applications which tend to exercise the rasterizer heavily. Also, note that in both cases the compiler-generated specializations perform equally well as the hand-crafted ones.

The primary optimization that results from the specialization of the rasterization routines is the elimination of conditionals with invariant predicates. Many of today’s latest processors, including the Pentium Pro machine, have sophisticated branch prediction hardware. It is of particular interest to know if the branch prediction hardware would obviate the need for such optimizations. Thus, we have chosen a Pentium Pro machine to run our `armitest` application. Speedups of rasterization procedure execution times due to specialization over the general case are presented in Figure 3. With speedups in the 1.5 to 2 range and a maximum of over 14, the graph clearly shows that specialization is important even for processors with aggressive branch prediction hardware.

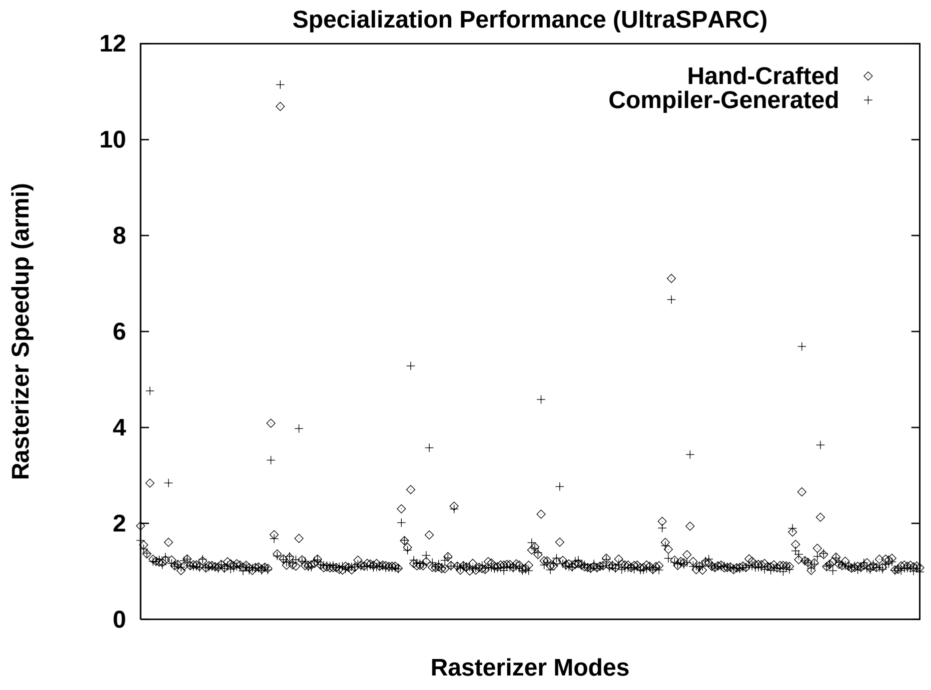

Figure 2 Speedup of individual specializations over the general rasterization routine on a 167MHz Sun UltraSPARC microprocessor. Each point represents the speedup of the rasterization procedure execution time for a particular rasterizer mode setting under the `armitest` application.

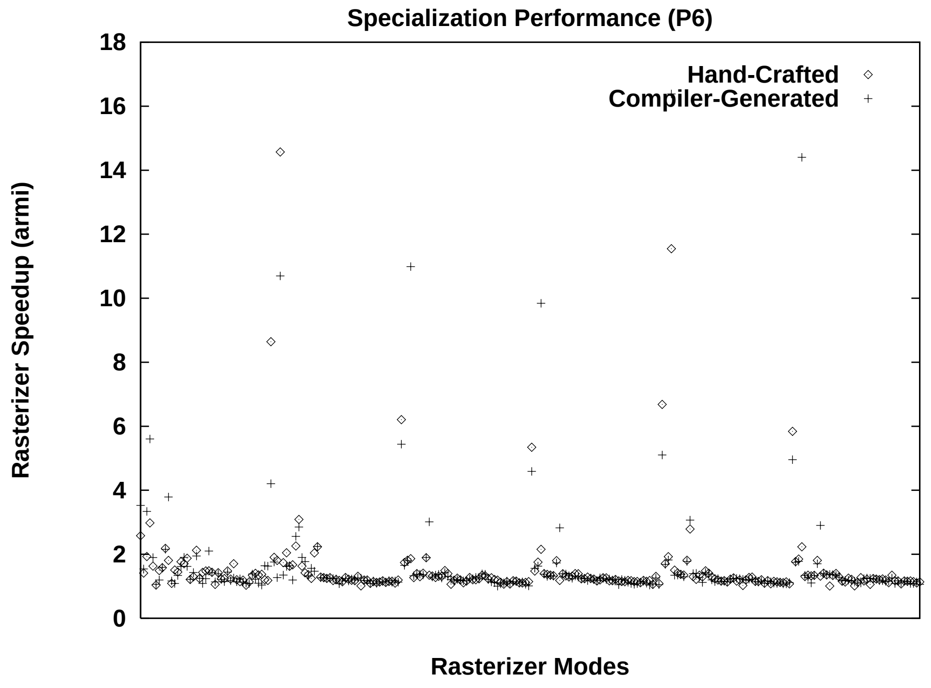

Figure 3 Speedup of individual specializations over the general rasterization routine on a Pentium Pro machine. Each point represents the speedup of the rasterization procedure execution time for a particular rasterizer mode setting under the `armitest` application.

Finally, what is perhaps our most important gain is not quantifiable. That is because it is a performance improvement in terms of software engineering effort on the part of programmers. By automating this optimization and tuning process, we have removed significant burdens of complexity from the developers of FruGL and other software packages. Thus, despite its simplicity, our tool can be used to great practical advantage today.

5 Related Work

There has been a great deal of previous work that uses profile-driven feedback in compiler optimizations; some of the recent examples include [3], [10], [13]. Instruction scheduling based on branch prediction is a well-established practice. Unlike most previous work which tends to rely on low-level profile information such as branch frequencies, our system identifies high-level information of interest and optimizes the code accordingly. In addition, our system automatically re-compiles itself when necessary without user intervention.

In the phase where we generate specialized codes, our compiler functions as a very simple-minded partial evaluator. We simply assert the variables as constants, and rely on conventional compiler techniques to perform the optimizations. Much work has been done on significantly more advanced forms of partial evaluation; for example, see the work of [6]. The work in this area generally requires the user to specify what variables should be considered as constant, and it attempts to translate the code to use the fixed values whenever possible. All the techniques developed in the partial evaluation area could be applied here, as our system becomes more aggressive in identifying possible cases of specialization. The focus of this work is not on partial evaluation techniques but on creating a completely automatic system that can assist the developers with their work at hand.

Dynamic code generation [1], [2], [5], [7], [9] is also a form of program specialization. These systems do not determine what to specialize based on profitability, and many of them rely on the user to specify the subset of the variables for which a code segment should be tailored. Specialized code is generated at run time when the data are known. Here, the dynamic code generation time is of essence, as it must not negate the performance advantages of the specialized code. They are heavily constrained in the amount of effort that can be expended on compilation. As a result, fewer optimizations can be performed in the allotted time, and less efficient code is produced. In this case heavyweight optimizations are being traded away for low compiler overhead. We, on the other hand, propose a different trade-off. We are willing to expend more compiler effort to achieve higher performance. Each time an application is run its behavior is monitored, and this information is used to direct optimizations executed by the compiler before the next run. In this way the software evolves more slowly than in the dynamic code generation case, but it adapts well to its environment and results in highly efficient code.

6 Conclusions

This paper advocates the adoption of the evolutionary software paradigm as a means for eliminating manual performance tuning from the program development cycle. Currently, application developers are forced to perform optimizations, such as specialization, by hand. The unfortunate result of this situation is code that is often difficult to understand and maintain. Evolutionary software addresses this problem by placing the responsibility for tuning the software with the compiler. The compiler automatically determines the targets for specializations and inserts a small amount of code to collect high-level profile information. After each execution of the program, the compiler uses the generated statistics to replace the program with a more optimized version which takes advantage of run-time information. In this way the software evolves and adapts to individual environments and usage patterns, and the programmer is completely unaware of this process.

To illustrate the benefit of this approach, we have designed and implemented the first end-to-end system that automatically compiles a program into self-adaptive code. This tool automatically identifies opportunities for optimization and specializes programs based on run-time invariants that are discovered dynamically. We have demonstrated the practicality and usefulness of this tool by using it to optimize a large, complex graphics library. Although our tool is based on some well-established techniques, it makes a significant contribution because it represents the first practical system which makes effective use of value locality without being overwhelmed by the overhead associated with identifying predictable values and optimizing for them.

7 References

1. J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad, “Fast, Effective Dynamic Compilation,” *Proceedings of the Conference on Programming Language Design and Implementation*, May 1996, pp. 149–158.
2. C. Chambers, D. Ungar, and E. Lee, “An Efficient Implementation of SELF: A Dynamically-Typed Object-Oriented Language Based on Prototype,” *LISP and Symbolic Computation*, 4(3), July 1991, pp. 283–310.
3. R. Cohn and P. Lowney, “Hot Cold Optimization for Large Windows/NT Applications,” to appear in *IEEE Micro*, December 2, 1996.
4. A. DeHon and I. Eslick, “Computational Quasistatics,” *Transit Note 103,* MIT Transit Project, February 1994.
5. D. Engler and T. Proebsting, “DCG: An Efficient, Retargetable Dynamic Code Generation System,” *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994, pp. 263–272.
6. N. Jones, C. Gomard, and P. Sestoff, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall International Series in Computer Science, Prentice-Hall, 1993.
7. D. Keppel, S. Eggers, and R. Henry, “A Case for Run-Time Code Generation,” *Technical Report 91-11-04*, Department of Computer Science and Engineering, University of Washington, 1991.
8. J. Larus and E. Schnarr, “EEL: Machine-Independent Executable Editing,” *Proceedings of the Conference on Programming Language Design and Implementation*, June 1995, pp. 291–300.
9. P. Lee and M. Leone, “Optimizing ML with Run-Time Code Generation,” *Proceedings of the Conference on Programming Language Design and Implementation*, May 1996, pp. 137–148.
10. A. Samples, *Profile-Driven Compilation*, Ph.D. Thesis, U.C. Berkeley, April 1991.
11. R. Sites, A. Chernoff, A. Kirk, M. Marks, and C. Robinson, “Binary Translation,” Communications of the ACM, 36(2), February 1993, pp. 69–81.
12. R.Wahbe, S. Lucco, and S. Graham, “Adaptable Binary Programs,” *Technical Report CMU-CS-94-137*, School of Computer Science, Carnegie-Mellon University, April 1994.
13. D. Wall, “Predicting Program Behavior Using Real or Estimated Profiles,” *Proceedings of the Conference on Programming Language Design and Implementation*, June 1991, pp. 59-70.