Dynamic Runtime Optimization

Thomas Kistler

Technical Report 96-54

Department of Information and Computer Science
University of California at Irvine
Irvine, CA 92697-3425

20th November 1996

Dynamic Runtime Optimization

Thomas Kistler

Department of Information and Computer Science
University of California at Irvine
Irvine, CA 92697-3425

Abstract. In the past few years, code optimization has become a major field of research. Many efforts have been undertaken to find new sophisticated algorithms that fully exploit the computing power of today's advanced microprocessors. Most of these algorithms do very well in statically linked, monolithic software systems, but perform perceptibly worse in extensible systems. The modular structure of these systems imposes a natural barrier for intermodular compile-time optimizations.
In this paper we discuss a different approach in which optimization is no longer performed at compile-time, but is delayed until runtime. Reoptimized module versions are generated on-the-fly while the system is running, replacing earlier less optimized versions.
In the first part of this paper we argue that dynamic runtime reoptimization will play an important role in future software systems and discuss the requirements for a modular, extensible operating system to support dynamic runtime optimization. In the second part we give an overview of promising intermodular and profile-guided reoptimizations. We also measure the characteristics of a modular, extensible operating system in order to estimate the potential of such optimizations.

1 Introduction

Generating high quality code is a challenge that has not just come into fashion recently. The first computer programs, written by hand in assembly code, had to be optimized to fit into the limited storage space and to complete their tasks within an acceptable period of time.

With the advent of high level programming languages and compilers that automatically translate user programs into machine code, this kind of "hand tweaking" of code more or less fell into oblivion. The primary goal of high level languages was to simplify program development and to free the programmer from writing machine dependent code. Generating optimized code was not of primary concern. Although the underlying hardware was seldom fully exploited, the generated instruction sequences were of fairly high quality. Besides, time-critical sections could still be written in assembler.

Only recently, with the introduction of more sophisticated processor architectures, the introduction of RISC computers, and the general availability of cheap hardware resources, have optimization techniques experienced a revival. Appropriate use of processor features, such as caches, pipelines, multiple instruction units, and register windows have lead to an increase in speed by an order of magnitude. Since the use of processor features cannot usually be influenced by the programmer, many efforts have been spent to increase the quality of the generated code by enriching the compiler with optimization techniques.

Except for optimizations that directly operate on instruction sequences (e.g. peephole optimization), most of today's algorithms are based on both semantic information that is collected statically at compile-time and precise knowledge about the underlying hardware architecture. Naturally, the more information that is available about a program, the better the results will be. Local optimizations, operating on basic blocks only, perform perceptibly worse than global optimization techniques that are based on modular dataflow analysis. Some strategies have even been implemented to optimize code patterns across compilation-unit boundaries — so called *intermodular optimizations*. Intermodular optimizations however are hardly feasible in modular systems. Key concepts like information hiding, data abstraction, or component reusability require hiding implementation details. Since the compiler cannot usually see what the programmer cannot see either, modularity can generally not be brought into line with the requirements of intermodular optimizations. These optimizations depend on global implementation knowledge. Only when a fixed number of modules is linked together to form a monolithic self-contained application (in which case global information is available to the compiler) can these systems profit from intermodular optimizations.

Today's optimization techniques have some other major drawbacks. First, they are all based on static program analysis and do not take into account the system's or even the program's dynamic behavior. Optimizations are applied uniformly to each section in the program, even though it is well known that only small portions of a program account for most of the execution-time [Knu70, Ing71].

Second, we are at the moment witnessing far-reaching changes in the field of software architecture. Applications of the next generation are very likely to come as a set of small software components instead of one large monolithic application. Such components, often called "applets", are loaded dynamically on demand and can be put together arbitrarily, forming new applications which are tailored to the specific needs of a user. Furthermore, new components can be downloaded and linked into the system at any time, immediately extending the functionality of an application. Although these systems have uncontested advantages, their highly dynamic nature imposes a natural barrier to interprocedural and intermodular optimizations. Since neither the end-user-configuration nor the components' interaction schemes are known to the optimizer at compile-time, code-optimization is limited to *intramodular* techniques.

Beside changes in the software architecture of future applications, we are also seeing major transitions in software distribution. Nowadays, applications are written to suit exactly one specific operating system and one specific processor type, forcing software vendors to maintain and distribute several different versions of their applications. However, in the near future, software components will be distributed in a portable format and will be able to run both on multiple operating systems and on multiple processor architectures. The components will either be interpreted at runtime, or machine code will be generated from the portable object file on-the-fly at load-time. Consequently, static optimization cannot be performed anymore during compilation of the source code since the compiler lacks information about both the target operating system and the target architecture.

Last but not least, computing power has soared dramatically during the past few years. Not only has the performance of new processor architectures increased from one processor to the next, but the time between generations has shortened as well, due to new innovative design and manufacturing techniques. This development diminishes the effects of optimizations that can be achieved. Applications optimized for one specific processor type cannot necessarily take full advantage of the new features of its successor models.

For all of the above reasons, we propose a new system architecture that delays program optimization until runtime. Optimization either takes place in the background while the system is running, or on explicit request of the programmer. Runtime optimization manages to combine the benefits of modular concepts and intermodular optimizations. It can utilize more information about user behavior or the target architecture than static optimizations and thus can achieve superior results for most optimizations.

2 A New System Architecture

We are currently implementing a system that performs optimization at runtime rather than at compile-time. By moving the optimization stage from compile-time to runtime we manage to eliminate all of the previously mentioned disadvantages of static optimizations and even improve on them with new techniques. Our system consists mainly of four parts that are shown in Fig. 1.

The *compiler* generates an object-file from the source code. In order to deal with all aspects of portability, each object file contains a portable intermediate representation of the program rather than native machine instructions. Except for constant folding, which is target-invariant and can be implemented in a straightforward manner, the compiler performs no code enhancements. Doing so might obstruct the potential for future dynamic runtime optimizations.

As soon as a module needs to be loaded, the *dynamic code-generating loader* transforms the intermediate representation into a native instruction sequence and executes the module body. Since many of the optimization algorithms have a great runtime complexity, optimizations are forgone at load-time in favor of fast module loading and short user response-time. This appears to be a good choice since the overall code-quality generated by our loader is quite good. In most cases, its code can compete with current non-optimizing compilers.

Once modules are loaded, the *adaptive profiler* starts collecting information about the system's runtime behavior. Its primary goal is to provide all the necessary background information upon which to base optimization decisions. It first monitors information on the level of procedures. In this context, the profiler takes a close look at call-frequencies, call-sequences, and call-durations. Furthermore, information about parameters and their dynamic types can be very valuable for certain optimizations. As soon as the measured results stabilize, monitoring is narrowed down to statement-blocks or even single statements in order to facilitate optimizations for long-running instruction sequences. It primarily keeps watch on variable-counts, execution-frequencies of loops and on how often conditional paths are executed.

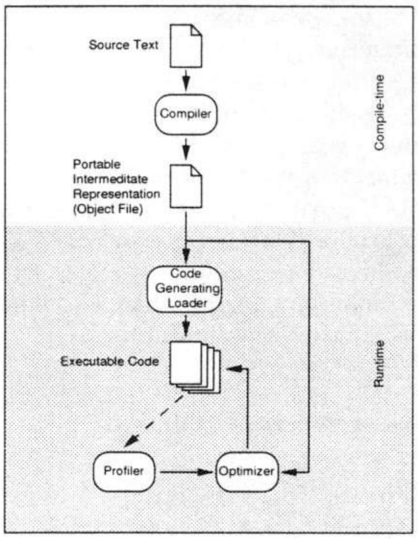

Fig. 1. System Architecture

The *dynamic optimizer* periodically recompiles modules that consume most of the execution-time in the *background*. The key idea is to profit from idle time (we have measured an average idle time of more than 90 percent in our interactive operating system) in order to perform optimizations that would take up a lot of user-time if performed on explicit demand. The optimizer operates on the intermediate program representation utilizing information accumulated by the profiler. Basing the optimizations on a profiler has one big advantage. If the profiler manages to pinpoint all the time-consuming parts (in general 5% of the code accounts for more than 50% of execution-time [Knu70, Ing71]), the optimizer is capable of spending more time on increasing the code quality of the sections that account for most of the execution-time. That way, most time is spent on highly optimizing a few important sequences rather than applying optimizations uniformly to each section of the program. Less frequently executed sections are optimized sparsely and no optimization is performed on barely executed sections. The challenging issue is to find the best balance between the number of blocks to optimize and the resulting speed-up. Still, in some cases, optimizations can take substantial time. To assume that the computing power is sufficient to recompile the whole system in one step, without impact on response-time, is rather optimistic, at least nowadays. Timings taken from existing compilers [Bra95] have shown that applying optimizations to a program takes at least 5 times as long as compiling the program. Therefore it may be preferable to perform optimizations gradually and to periodically reoptimize program parts.

Since the system knows exactly which modules are loaded at runtime and how they interact with each other, it is not restricted to local optimizations. Indeed, performing intermodular code improvements on loaded applications is made possible only now by the introduction of runtime code optimization.

As soon as a set of optimizations has been applied to a module, the system *replaces* the running instance with the optimized module version. This process requires updating intermodular dependencies.

3 The Importance of the Intermediate Representation

An important aspect that has not been discussed thus far is the choice of the intermediate representation. As we have seen earlier, it is used both for generating a first native version of a module and later on for supplying semantic information to the optimizer in order to achieve high code quality. Consequently the representation must hold enough information to perform the optimizations desired. It must also facilitate on-the-fly compilation that is fast enough to compete with loading of compiled code and that generates native code of high quality. Last but not least, the intermediate representation should not contain any machine dependent information in order to be highly portable.

For these reasons we base our system on slim binaries [Fra94, FK96]. Slim binaries are a form of intermediate representation that does not contain any object code at all, but a portable description of a module's contents. This makes these files completely independent of the eventual target machine. Slim binaries are based on an encoding of abstract syntax trees rather than on virtual instruction sequences as employed in the Java virtual machine representation [LYJ96]. Using a predictive compression algorithm to encode recurring sub-expressions, slim binaries facilitate storing and decoding programs efficiently, both in terms of space and time. Moreover, object code generation can be carried out on-the-fly and takes about as much time as loading traditional object files [FK96].

A tree-based encoding is different from a virtual machine representation in that it cannot be interpreted easily. Program interpretation was essential for efficient linking and loading of object files in operating systems with limited memory capacity and computing resources. However, with the recent increase of computing power and the advent of on-the-fly compilers, this argument loses force.

Tree-based encodings on the other hand have many valuable advantages over low-level representations. First of all, they are completely independent of any target machine, whereas chances of a virtual machine representation not suiting a specific processor type are reasonable. Second and even more important, the abstract syntax-tree preserves all the semantic program information which is essential for effective optimization algorithms. It keeps all the control-flow and maintains the notion of basic blocks. Tree-based encodings also contain all the information necessary for debugging tools that operate on the language level rather than on the instruction level.

In order for low-level representations such as byte-codes to achieve comparable optimization results, a tree or control-flow graph must be reconstructed before performing optimizations. In the past, object files have therefore been instrumented with hints about block boundaries [Han74]. Even worse, the absence of type information in such low-level representations prevents a complete set of optimizations like polymorphic inline caches [HCU91] or runtime type feedback [HU94].

4 Promising Intermodular Optimizations

In this section, we will present optimizations that can profit specifically from the execution profiles and semantic information available at runtime. Generally this applies to all optimizations that are based on heuristics. They tend to achieve much better results in our experimental system than on any statically compiled system since they now are based on exact, measured execution profiles rather than on imprecise assumptions and since they now can be applied to the whole system rather than to single procedures. Optimizations that only rely on local information will not be discussed here because they are fundamental in every system (e.g. copy propagation, common subexpression elimination, peephole optimizations, strength reduction, instruction scheduling).

4.1 Intermodular Inlining

Perhaps one of the most auspicious optimizations is *intermodular inlining*. The basic idea of inlining is to replace a call to a procedure by the body of that procedure. This not only avoids the costs of calling and returning from a procedure (i.e. copying parameters to registers or on the stack, and allocating and disposing of the parameter-passing area) but also increases the potential for successive optimizations (e.g. common subexpression elimination) since it allows the inlined body to be improved within the context of the caller. In general, separately compiled systems are restricted to inlining procedures within module boundaries. Only by neglecting extensibility and portability can inlining be performed across compilation units. This case is called *intermodular inlining*.

In our architecture, in which optimization is performed while the system is running, there are no restrictions. In addition, the idea can be taken even one step further by inlining calls to *procedure variables* if a procedure is bound to only one variable at runtime. The same idea holds for method calls, given that the method is never overridden at runtime. For object oriented systems, even hardcoding method calls without actually inlining the calls results in noticeable speed-ups as no method lookups are required as a result.

In most compilers, inline decisions are based on size heuristics since it is extremely difficult to estimate how inlining affects runtime performance. No simple rule can be given as to which procedures to inline. For example, while the number of instructions that can be saved is easy to precalculate, the effect of modified code locality on cache performance is very hard to predict. Only with the availability of an integrated profiler can the system further refine these decisions. Issues to take into account are call frequencies, call-durations, or even the results of previous inline steps.

Instead of actually copying the call bodies into the call site, *procedure cloning* [IBM94] and *partial evaluation* [Sur93, Jon93] are related techniques that generate multiple versions of the procedure body, each customized for a specific set of callers. These optimizations are usually preferred to inlining for larger procedures that are called frequently with multiple, but partially fixed parameter lists. These parameter lists are then hardcoded into the corresponding procedure clones in order to benefit from further optimizations. Since dynamic type information is also present at runtime in our system, even dynamic types can be hardcoded into call-chains and thus speed up message sends dramatically in object oriented systems. As for inlining, we expect procedure cloning and partial evaluation to do perceptibly better when applied to the whole system.

4.2 Intermodular Register Allocation

A second optimization from which we expect improved speed-ups is *intermodular register allocation*. Recent RISC machines have been built with small and simple instruction sets and fast instructions that operate on registers only. The few provided load and store instructions are, however, an order of magnitude slower because they access main memory. Hence chip manufacturers rely on smart compilers that keep as much data as possible in the large provided register sets. Unfortunately non-optimizing compilers use only a small subset of these registers, wasting a lot of computing power. Even popular register allocation schemes, like the ones developed by Chaitin and Chow [Cha82, CH84], fail to yield satisfactory results for systems applying separate compilation. The main problem is that register allocation has to be applied separately to every single module instead of to the whole system. This results in different registers being assigned to the same global variable or the same register being assigned to different local variables. Only late code optimization can avoid these problems.

In our experimental system, registers are assigned at link-time over the entire system, hence the same set of registers can be assigned to procedures that are not alive at the same time. In addition when one procedure calls another, the optimizer can make sure that they use different registers. The same principle holds for parameters. No particular parameter passing mechanism is enforced. Rather parameters are assigned to disjunctive register sets for fast argument passing, and in order to avoid saving and restoring data around procedure calls. In some cases it might even be worth storing constants in particular registers.

In our runtime optimizer, all allocation strategies are based on an intermodular call graph as proposed by Wall [Wal86] and on execution profiles. The latter are particularly useful in estimating which registers to spill in case of shortcomings.

4.3 Intermodular Code Elimination and Code Motion

A completely different group of optimizations deals with intermodular code elimination and code motion. The idea is to either remove code in case it cannot be reached or to relocate it to program parts that are less frequently executed. Well known optimizations are *loop invariant code motion* — which tries to move loop-invariant code outside of loops — and *partial dead code elimination* or *partial redundancy elimination* — that tries to move statements into conditional paths.

Just as with register allocation and inlining, good algorithms have been around for single procedures and single modules, but only poor solutions have been presented so far for intermodular analysis. We hope to improve this situation by the introduction of dynamic runtime optimization.

4.4 Cache Optimizations

Today's new processor designs use fast on-chip caches and somewhat slower off-chip second level caches in order to improve system performance. In the near future, these caches will become dramatically faster than main memory. Even today, it typically takes ten times longer to retrieve data from main memory than from a cache. It is therefore very important to avoid cache misses whenever possible. With the availability of on-the-fly compilers and runtime optimizers *cache optimizations* can now try to meet these requirements.

Basically there are three groups of possible cache optimizations: data cache optimizations, instruction cache optimizations, and optimizations that take into account cache parameters but don't have the explicit goal of improving cache performance. *Data cache optimizations* on the one hand are easy to implement and achieve good results. The idea is to improve the temporal data locality by grouping variables that are repeatedly used in the same period of time. Spatial locality can also be improved and the size of the working set decreased by re-ordering and compacting records. Cache blocking is a further, but much more complicated technique to improve data cache behavior, especially for numerical programs [WL91]. All of the above data cache optimizations have in common that they are based either on statically weighted access computations or on execution profiles.

Instruction cache optimizations, on the other hand, are very hard to implement and not at all feasible at compile-time. Only if a time-critical section consists of a small number of procedures that call each other can we ensure that the bodies are located in separate cache lines. If, on the other hand, a time-critical section spans over many procedures, as is normally the case for object oriented systems, it is hardly possible to avoid line conflicts. Therefore, we adapt existing optimizations to take into account cache parameters (e.g. size, line size, degree of associativity) instead of writing new ones with the sole goal of improving instruction cache performance. Consider loop unrolling as an example, where the primary goal is to remove unnecessary control structures and to overlap the execution of several loop iterations. Unfortunately, if the unrolled loop results in more cache misses, the runtime performance may even be decreased by this optimization. Hence cache characteristics should be considered in this kind of code improvement.

5 Optimization Potential for an Extensible Modular Operating System

In order to quantify the potential achievements of the presented optimizations, we have collected statistics about the Oberon System [WG89] for Power Macintosh computers. The Oberon System is an interactive, extensible operating system. It not only includes representative applications like a graphical user interface, a native and portable compiler, a text- and a graphics-editor but also smoothly integrates Internet services that will be essential parts of future operating systems (e.g. e-mail, WWW-browser, ftp, news, telnet, gopher). The included native compiler translates programs written in the programming language Oberon [Wir88] to PowerPC 601 machine instructions. Its code quality can be roughly compared to the quality of non-optimizing C compilers.

Table 1 gives a general overview of the system. There are two interesting points to be observed. First, there are more calls to external procedures than to local ones. This might be surprising at first sight but actually reflects the achievement of two key goals of modular systems: abstraction and reuse. Hence, our claim that intermodular optimizations are very important to achieve good speed-ups can be reemphasized at this point.

Number of modules	229
Number of procedures	6660
Number of external procedure calls	20299
Number of local procedure calls	17217
Number of indirect procedure calls	3411
Number of references to external variables	7756
Number of references to local variables	43540

Table 1. General Static System Information

Second, unlike in dynamically typed object oriented systems, there are very few indirect calls in our system. As a result, many code improvements that could be implemented to accelerate message-calls like polymorphic inline caches [HCU91] or runtime type feedback [HU94] would have very little effect on our system.

Table 2 and Table 3 show average counts of various properties per procedure. The numbers given in Table 2 have been collected during compilation of all modules. Thus they do not take into account the fact that some procedures are executed more frequently than others.

Procedure size (bytes)	422.14
Number of statements per procedure	17.10
Number of external calls per procedure	3.05
Number of local calls per procedure	2.59
Number of indirect calls per procedure	0.51
Number of references to external variables per procedure	1.16
Number of references to local variables per procedure	6.54
Number of unused registers per procedure (on PowerPC)	15.82

Table 2. Static Procedure Information

Weighting the values with the effective execution-time spent in a procedure during an average working session yields slightly different values (Table 3). The most interesting result here is that almost 50% of the available registers are unused during execution-time (the PowerPC architecture defines 32 user-level, general-purpose registers). Consequently there is an enormous potential for register related optimizations. Not only would it be worthwhile to keep local variables in registers since they are accessed ten times more often than external variables, but we could also keep local variables and parameters of two successively called procedures in disjunctive register sets to avoid expensive parameter-passing and register-saving. In comparison to statically optimized code, speed-ups in the range of 10-25% have been reported for similar register allocation schemes [Wal86].

Number of external calls per procedure	4.79
Number of local calls per procedure	3.25
Number of indirect calls per procedure	1.64
Number of references to external variables per procedure	1.02
Number of references to local variables per procedure	10.55
Number of unused registers per procedure (on PowerPC)	11.82

Table 3. Dynamic Procedure Information

We have also examined the distribution of procedure sizes which is given in

Fig. 2. In general, procedures in our system are small. Half of all procedures are smaller than 240 bytes which corresponds roughly to 60 instructions or 10 Oberon source-code statements. The great potential for inlining is evident. Even when considering only to inline leaf-procedures (i.e. procedures that do not call any other procedure) still every tenth procedure would be a potential candidate.

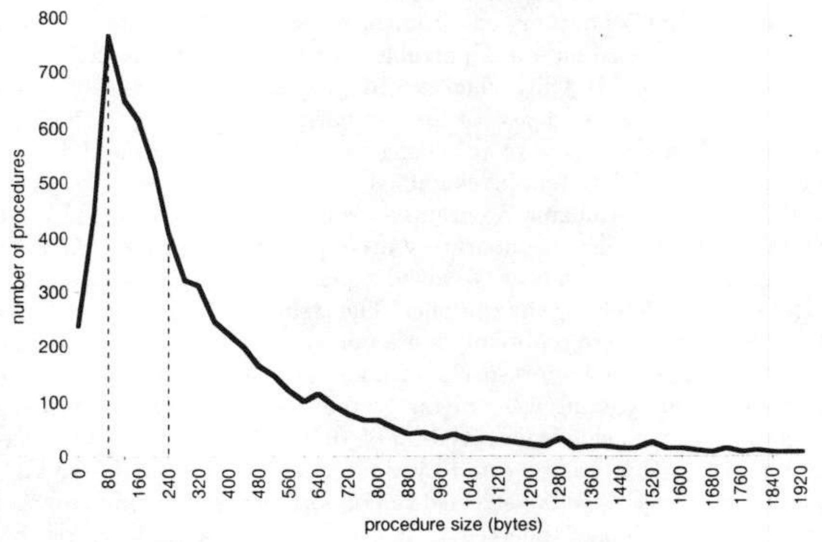

Fig. 2. Procedure Size Distribution

The results that we have measured during test runs were very promising.In some cases, when inlining frequently-executed procedures, a speed-up factor of 9 could be achieved. In addition, small procedures have a second positive impact on inlining. If the optimizer inlined every second call, the overall code size would only grow by about 25%.

6 Related Work

Pioneering research in program reoptimization was done by Knuth [Knu70], Ingalls [Ing71], and Jasik [Jas71]. Their work can be classified as *iterative optimization* and involves a feedback loop between the system and the programmer. A profiler monitors the execution of a program and creates data upon which the programmer bases further optimizations. The program is then run again to obtain updated profiles. The disadvantages of this approach are obvious. Not only does the inclusion of the programmer in the compile-load-run cycle make the results dependent on his knowledge, but also input values may vary from run to run, diminishing previous optimizations.

A group at the University of Washington examined value-specific data-dependent optimizations [KEH93], where code is optimized at runtime around particular input values. This strategy can best be compared to partial evaluation that is applied at runtime. Unfortunately the programmer is still involved in that he/she has to explicitly identify the bottlenecks in the application. This is done by marking sections using templates or fragments in the source code.

Removing the programmer from the feedback loop was one of the main ideas in Hansen's [Han74] automated optimization system. For speed considerations, his system was based on a non-portable intermediate representation that could be interpreted directly. Only when exceeding a certain runtime threshold was the representation translated *just-in-time* to native code and optimized for speed. He reported that the new reoptimizing FORTRAN-IV system did better than any single compiled system he examined.

Rather than optimizing programs at runtime, the Titan/Mahler project at Digital Western Research Laboratory attempted to perform optimizations only at the time of linking. In order to avoid expensive loading, program analysis was performed completely by the compiler. The results were then integrated into the object files which were represented in a portable register transfer language. The results confirmed the importance of intermodular optimizations. Although inlining or procedure cloning was not integrated, the built-in intermodular register allocation scheme achieved a speed-up of 10-25% [Wal86]. Another speed-up in the range of 5-10% was reported by the implementation of intermodular code motion. In addition [SW93] stressed the importance of dynamic profiles and described that previously collected variable-use profiles almost invariably achieve better results than compiler estimates.

In order to make object oriented languages competitive with traditional languages, eliminating the overhead of dynamic dispatching at runtime is probably the most important and the most promising optimization. Several techniques have been presented in the last few years to achieve this goal, such as class prediction [Höl94] and iterative class analysis [CU90] (both of them have been integrated into the SELF-system [US87]). Interprocedural class analysis, a related technique, was proposed by Grove [Gro95]. However it is still an open question whether traditional statically-typed languages can profit from these techniques.

7 Conclusion

Traditional static optimization algorithms suffer from two important deficiencies. First, they cannot be applied to portable code, and second, intermodular optimizations can only be performed on statically linked monolithic applications, thus thwarting extensibility and reducing the applicability for modular systems.

Since extensibility and portability will play an important role in future software systems, we are currently implementing a system architecture in which program optimization is performed in the background while the system is running. Unlike earlier proposals, optimizations are not applied uniformly to each section of the program but rather take into account the program's dynamic behavior and are only applied to the parts that account for most of the execution-time. The system utilizes object files that are based on a tree-based intermediate representation, and is guided by an adaptive profiler. A tree based encoding has many advantages over most commonly used low-level intermediate representations.

Finally, we have presented some examples of intermodular optimization techniques and illustrated their enormous potential for modern extensible operating systems.

8 Acknowledgments

Michael Franz and Martin Burtscher provided many helpful comments on an earlier version of this paper. Martin Burtscher also developed a prototype algorithm for intermodular inlining upon which some of the presented statistics are based. More information about the Oberon project and research related topics at the University of California at Irvine can be found on the World Wide Web at http://www.ics.uci.edu/~oberon.

References

[Bra95] M. M. Brandis; *Optimizing Compilers for Structured Programming Languages*; (Doctoral Dissertation) Eidgenössische Technische Hochschule Zürich; 1995

[Cha82] G. J. Chaitin; Register Allocation & Spilling via Graph Coloring; In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pp 98–105. Published as *SIGPLAN Notices 17(6)*; June 1982

[CH84] F. C. Chow, J. L. Hennessy; Register Allocation by Priority-Based Coloring; In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pp 222–232. Published as *SIGPLAN Notices 19(6)*; June 1984

[CU90] C. Chambers, D. Ungar; Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs; In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp 150–164. Published as *SIGPLAN Notices 25(6)*; June 1990

[Fra94] M. Franz; *Code-Generation On-the-Fly: A Key to Portable Software*; (Doctoral Dissertation) Verlag der Fachvereine, Zürich; 1994

[FK96] M. Franz, Th. Kistler; *Slim Binaries*; Technical Report 96-24, Department of Information and Computer Science, UC Irvine; 1996

[Gro95] D. Grove; The Impact of Interprocedural Class Analysis on Optimizations; *CASCOM '95 Proceedings*; November 1995

[Han74] G. J. Hansen; *Adaptive Systems for the Dynamic Run-Time Optimization of Programs*; (Ph.D. Dissertation) Department of Computer Science, Carnegie-Mellon University; 1974

[HCU91] U. Hölzle, C. Chambers, D. Ungar; Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches; In *ECOOP '91 Conference Proceedings*. Published as *Springer Verlag Lecture Notes in Computer Science 512*; 1991

[Höl94] U. Hölzle; *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*; (Ph.D. Dissertation) Department of Computer Science. Stanford University; 1994

[HU94] U. Hölzle, D. Ungar; *Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback*; In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp 326–336. Published as *SIGPLAN Notices 29(6)*; June 1994

[IBM94] IBM; *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*; IBM Order Number SA23-2737-00

[Ing71] D. Ingalls; The Execution Time Profile as a Programming Tool; In *Design and Optimization of Compilers*, pp 107–128, Prentice-Hall; 1971

[Jas71] S. Jasik; Monitoring Execution on the CDC 6000's; In *Design and Optimization of Compilers*, pp 129–136, Prentice-Hall; 1971

[Jon93] N. Jones; Special Issue on Partial Evaluation; *Journal of Functional Programming 3(4)*; 1993

[KEH93] D. Keppel, S. J. Eggers, R. R. Henry; *Evaluating Runtime-Compiled Value-Specific Optimizations*; Technical Report 93-11-02, Department of Computer Science and Engineering, University of Washington; 1993

[Knu70] D. E. Knuth; *An Empirical Study of FORTRAN Programs*; IBM Report RC 3276; 1970

[LYJ96] T. Lindholm, F. Yellin, B. Joy, K. Walrath; *The Java Virtual Machine Specification*; Addison-Wesley; 1996

[Sur93] R. Surati; *A Parallelizing Compiler Based on Partial Evaluation*; Technical Report AITR-1377, Artificial Intelligence Laboratory, Massachusetts Institute of Technology; 1993

[SW93] A. Srivastava, D. W. Wall; A Practical System for Intermodule Code Optimization at Link-Time; *Journal of Programming Languages*; 1993

[US87] David Ungar, R. B. Smith; SELF: The Power of Simplicity; In *OOPSLA '87 Conference Proceedings*, pp 227–241. Published as *SIGPLAN Notices 22(12)*; December 1987

[Wal86] D. W. Wall; Global Register Allocation at Link Time; In *Proceedings of SIGPLAN '86 Symposium on Compiler Construction*, pp 264–275; July 1986

[WG89] N. Wirth, J. Gutknecht; The Oberon System; *In Software-Practice and Experience 19(9)*, pp 857–893; September 1989

[Wir88] N. Wirth; The Programming Language Oberon; In *Software-Practice and Experience 18(7)*, pp 671–690; July 1988

[WL91] M. Wolf, M. Lam; A Data Locality Optimization Algorithm; In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp 30–44, Published as *SIGPLAN Notices 26(6)*; June 1991