Abstract Interpretation and Low-Level Code Optimization *

Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85715

Abstract

Abstract interpretation is widely accepted as a natural framework for semantics-based analysis of program properties. However, most formulations of abstract interpretation are in terms of high-level semantic entities that do not adequately address the needs of low-level optimizations. In this paper we discuss the role of abstract interpretation in low-level compiler optimizations, examine some of its limitations, and consider ways in which they might be addressed.

1 Introduction

The process of compilation, by which executable code is generated from a source program, can be thought of as a series of transformations and translations through a succession of languages, starting at the source language and ending at the target language. In this picture, we can distinguish between two kinds of transformations: *translations*, which take a program in a language and produce a program in a different (usually "lower-level") language; and *optimizations*, which transform a program in a language to another program in the same language. As an example, a compiler that we have implemented for a logic programming language called Janus [38] works by translating the input programs into C, then invoking a C compiler to generate executable code. In this system, we can identify the following language levels: (1) the source language; (2) the Janus virtual machine language; (3) C; (4) the intermediate representation(s) within the C compiler; and (5) the target machine language. In principle, optimizing transformations can be applied at each of these five language levels; our current implementation applies optimizations at levels 2 (the Janus virtual machine), 4 (the intermediate representation(s) of the C compiler) and 5 (the target machine code), the last two within the C compiler. In each case, the optimizations can be seen as program transformations at a particular language level. A fundamental requirement of the compilation process is that it should be "semantics-preserving" in the sense that the "meaning," or behavior, of the executable code should conform to what the semantics of the source program says it should be. For this to happen, it is necessary in general that both translations and optimizations should be semantics-preserving in this sense. Since our primary focus is on optimizations rather than translations, we will assume here that our translations satisfy this requirement, and focus our attention on optimizations.

It is very often the case that an optimization is not universally applicable. In other words, in order to ensure that an optimization does not alter the observable behavior of a program in unacceptable ways, we have to ensure that certain preconditions particular to that optimization are satisfied. As an example, consider register allocation in a C compiler: the value of a variable can be kept in a register only if certain conditions regarding aliasing are fulfilled. In general, this means that it may be necessary to examine a program and extract some information about its behavior, which can then be used for optimization purposes. Further, in order to verify that the properties so inferred describe all possible runtime behaviors of a program, it is necessary to be able to relate the analyses to the semantics of the language in a precise way.

Semantics-based techniques such as abstract interpretation [23, 24, 25] provide a natural framework for such program analyses. The general idea is to rely on the formal semantics of a program to specify all of its possible computational behaviors, and to derive finitely-computable descriptions of such behaviors by systematically approximating the operational behavior of the

*This work was supported in part by the National Science Foundation under grant CCR-9123520.

Benchmark	Execution Time (μsecs)			Heap Usage (words)		
	no-opt	opt	no-opt/opt	no-opt	opt	no-opt/opt
`aquad`	55467	20569	2.67	30884	544	0.018
`bessel`	12577	12364	1.02	689	452	0.656
`binomial`	6055	5720	1.06	1208	6	0.005
`chebyshev`	32234	8500	3.79	30002	6	0.0002
`e`	13713	9832	1.39	6005	6	0.001
`fib`	13839	4711	2.94	6389	5	0.001
`log`	44967	35432	1.27	28870	6	0.0002
`mandelbrot`	102517	23942	4.28	69533	654	0.009
`muldiv`	16621	12705	1.31	5	5	1.000
`nrev`	8525	8018	1.06	10507	10507	1.000
`pi`	25565	12144	2.10	20007	6	0.0003
`sum`	6503	1694	3.84	5	5	1.000
`tak`	18043	5340	3.38	7121	5	0.001
Geometric Mean :			2.02			

Table 1: Performance improvements due to low-level optimizations (`jc` on a Sparcstation-IPC)

program. The correctness of an analysis can then be derived from the mathematical relationships between the actual computational domain of the program and the domain of descriptions manipulated by the analysis, and between the actual operations executed by the program and the approximations to those operations used during the analysis.

Optimizing program transformations can be viewed at many levels, corresponding to the different levels of languages encountered during compilation. At a high level, for example, we have transformations such as finite differencing [55, 66], recursion removal (i.e., transformation of recursive programs to tail recursive form) [5, 29], deforestation [19, 73], transformations for parallelization and vectorization (see, for example, [3, 12]), as well as various transformations described by Bacon *et al.* [6]. At the level of "intermediate code" we have machine-independent low-level optimizations such as induction variable elimination [1], closure representation optimization in functional languages [46, 47], and dereferencing optimizations in logic programming languages [68, 71]. At a lower level still we have machine-dependent transformations such as register allocation [9, 15, 20] and instruction scheduling [36, 59]. Conceptually, we can divide these various optimizations into two classes: *high level optimizations*, which correspond roughly to optimizations that can be expressed in terms of transformations on the source program (or its abstract syntax tree); and *low-level optimizations*, which involve constructs and objects that are not visible at the source level, and which therefore cannot be so expressed (this classification is not absolute, of course: whether or not an optimization is to considered "low-level" depends, among other things, on the language being considered: for example, in a language with explicit constructs for iteration, the implementation of a tail recursive procedure in terms of iteration could be considered as a high-level optimization; in a language without source-level iterative constructs, however, this would be a low-level optimization).

There are two reasons why low-level optimizations are important. The first is that they are beyond the reach of the user. The point is that when faced with a compiler that does not do much in the way of high-level optimizations, the determined user can, in principle, carry out the transformations manually where necessary in order to obtain code with good performance. With a compiler that does not perform low-level optimizations, however, there is little that even the most determined of users can do. In particular, this implies that in the absence of low-level optimizations, even carefully crafted programs written by skilled programmers will incur performance penalties over which they have little control. The second reason such optimizations are important is that they can produce substantial performance improvements. As an example of this, Table 1 gives some performance numbers for `jc`, an implementation of a dynamically typed logic programming language [38]. The `jc` compiler currently performs only low-level optimizations: call forwarding [27], which is a form of jump redirection at the intermediate code level; a simple form of inter-procedural register allocation for output value placement [7]; and representation optimization (i.e., using unboxed values where possible) for numerical values [8]. As Table 1 indicates, for the benchmarks tested these optimizations more than double the speed of the programs on the average, and also lead to significant improvements in heap memory usage.[1] The speed of the resulting code is competitive with that of optimized C code written in a "natural" imperative style: on the benchmarks shown, the Janus programs—which are dynamically typed and use dataflow synchronization between producers and consumers—is, on the average, only 13% slower than C code compiled with `gcc2 -O2`, about 25% faster than C compiled with `cc -O2`, and 6% faster than C compiled with `cc -O4`. This indicates that low-level optimizations can be a valuable source of performance improvements.

The appeal of semantics-based program manipulation techniques is that they allow us to reason formally about the manipulations themselves, and certify with some confidence that such manipulations will not cause "bad things" to happen. This paper considers the applicability and relevance of semantics-based program analysis techniques such as abstract interpretation in the context of low-level code optimization. Specifically, we argue that "semantic mismatches" between the kinds of information typically produced by semantics-based analyses and the kinds of information needed by low-level optimizations limit the utility of such formally defensible analyses for these optimizations. Specifically, we consider two kinds of semantic mismatch: in Section 2 we consider the level at which the "concrete semantics" is considered; and in Section 3 we consider the problem of estimating runtime execution frequencies and costs.

2 Low-Level Semantics and Abstract Interpretation

It is not difficult to see that while the kinds of information provided by abstract interpretation (or other semantics-based analyses) are perhaps necessary for code optimization, they are by no means sufficient. Part of the problem is that the "concrete" semantics on which abstract interpretations are typically based are, from the standpoint of low-level code optimization, not concrete enough. They usually have little to say about the registers and bit vectors and pointers and other such low-level entities that are actually manipulated during program execution. Indeed, the concrete semantics usually encountered can themselves be seen as abstractions of lower-level characterizations of program behavior, where some or all of the information about machine-level entities has been abstracted away. The problem, of course, is that usually we think of the process of abstraction as forgetting about "irrelevant" aspects of the behavior of a program, while in this case it is precisely the most relevant aspects of the program's behavior that are being forgotten.

The problem can be addressed by abstract interpretation based on a low-level semantics. While this does not seem different from any other sort of abstract interpretation at a conceptual level, the practical details can become messy. As an example, it is very likely simpler and more convenient to manipulate a high-level representation of a program, such as an abstract syntax tree, for such analyses, since the number of different kinds of objects and operations that have to be dealt with for such representations is relatively small. However, it is not clear that a high level program representation can encode low-level information in a reasonable way without (implicit or explicit) assumptions about the behavior of the code generator. This, in turn, implies that such analyses, while simple to implement initially, are potentially fragile.[2]

An example of this situation arises in the context of dereference chain length analysis in Prolog systems. In general, variable-variable unifications during the execution of a Prolog program can cause pointer chains to be set up, and these need to be dereferenced before the value of a variable can be accessed. Dereferencing arbitrary-length (tagged) pointer chains is a fairly expensive operation, so static analyses to infer the lengths of dereference chains can be very helpful in improving program performance—in particular when they allow dereference operations to be omitted entirely [51, 69, 71]. However, high-level semantics for Prolog typically do not have much to say about low-level aspects such as pointer-chain lengths: for example, when two variables are unified, such semantics say nothing about how the pointers are oriented. Because of this, dereference chain length analyses that manipulate high-level representations of programs—such as those of Van Roy [71] and Taylor [68, 69]—must either limit their precision by refusing to handle any situation where the high-level semantics is not unambiguous, or expose themselves to potential fragilities by making assumptions about the code generator. Closure analysis in the Orbit compiler for Scheme [46, 47] provides another example of the use of a high level representation

[1] These numbers do not include the effects of tail call optimization, though strictly speaking that is a low-level optimization in our context. If the effects of tail call optimization are included, the speed improvement is by a factor of about 3.4.

[2] In our Janus system, for example, we found that an optimization to eliminate unncessary dereference operations, based on an analysis that used the abstract syntax tree of the program similar to analyses of Van Roy [71] and Taylor [68, 69], led to incorrect code being generated when the mechanism for dealing with suspensions changed. It turned out that as an ill-advised "convenience hack" the analysis made implicit assumptions about whether or not the code generator would return output values in registers. These assumptions were rendered invalid when the code generator was modified to handle suspensions differently, but the analysis phase did not know about this.

$$\begin{tikzcd}[column sep=huge, row sep=huge]
\text{Source Program } P \arrow[r, "concrete\ semantics"] \arrow[d, "\text{transformation } \tau"'] & M \text{ “Concrete” Meaning} \arrow[d, "\text{abstraction } \alpha"] \\
\text{“Approximate” Program } P' \arrow[r, "concrete\ semantics"] & M' \text{ “Abstract” Meaning}
\end{tikzcd}$$

Figure 1: Program Analysis using Abstract Compilation

for analyzing low-level aspects of a program behavior: in this case, decisions about the low-level representation of closures are based on the structure of the abstract syntax tree for the program.

It may be possible to get around this problem in some cases by "lifting" implementation-level aspects of a program to the source level and then treating the analysis and optimization problems as high-level issues. This approach is taken in &-Prolog [40], a parallel Prolog system, which extends the source language to allow various lower-level parallelization and synchronization issues to be addressed at the source level. Another example of such an approach can be seen in exposing low-level representational aspects of data, such as whether they are boxed or unboxed, at the source level, and formulating representation optimizations in terms of source-level program transformations [49, 57]. However, it may not always be possible to capture low-level optimizations by lifting them to the source level in this way (for example, it is not clear how the implementation of aggregate updates in a single-assignment language via compiler-introduced destructive updates (see, for example, [35, 37, 42, 53]) could be expressed at the source level).

The alternative is to use a lower level representation, e.g., a sequence of intermediate code instructions. This has the advantage that the appropriate low-level details have been made explicit and can be reasoned about without having to resort to assumptions about the behavior of other parts of the compiler. This is conceptually cleaner and more defensible than the previous approach. However, there are two important practical problems that arise with this approach. First, the number of operations that have to be accounted for is likely to be considerably larger in a low-level representation than in a high-level representation. Second, relationships between objects, e.g., whether or not two objects overlap in memory, may be harder to reconstruct by examining a sequence of low-level operations.

Because of the large number of different operations that might be encountered in a low-level representation of a program, and the comparatively larger size of such a representation, one might expect a low level abstract interpretation to be considerably slower than a high level one. This problem can be alleviated to some extent by a technique that, with tongue firmly in cheek, we call "abstract compilation." The idea is the following: to reduce the cost of program analysis, instead of repeatedly traversing an internal representation of the program P being analyzed, we partially evaluate an abstract interpreter to with respect to P so as to produce a program P' which, when executed, yields the result of analyzing the original program P [30, 41]. In practice, for any particular analyses that we wish to implement in a compiler, we will know enough about the corresponding abstract interpreters that instead of invoking a general purpose partial evaluator on such an interpreter and the input program P, we can simply make a single pass over P and produce P' (indeed, we initially thought of this in terms of program transformation rather than partial evaluation): this is illustrated in Figure 1. The idea is similar to the notion of "need expressions" proposed by Maurer [52] in the context of strictness analysis. McNerney also uses a similar approach for an abstract interpretation to verify the correctness of low-level compiler optimizations [50].

At first glance it might appear that such an approach is practical only in languages, such as Prolog and Lisp, where it is easy to create program fragments "on the fly" and execute them. For languages such as C, for example, the traditional model for generating executable code for a program would most likely incur much too much I/O overhead, in writing out a program (or executable code) into a file and then reading it back in, to make this worthwhile. However, recent work in dynamic code generation for such languages [34, 45] indicates that the runtime overhead associated with creating and executing code for such languages at runtime can be made small enough to make such an approach practical. The success of dynamic code generation in the SELF system [16] also suggests that the "abstract compilation" approach may be practically usable in general.

The second problem referred to above is that relationships between objects that may be relatively straightforward to detect at a high level may be much harder to rediscover in a lower level analysis. For example, a value that is easily identifiable as a list or a tree at a high level may be visible only as a jumble of pointers during a low level analysis, making it much more complicated to rediscover relationships between its components (e.g., compare high-level type inference as in [2] with comparable low-level analyses as in [18, 33]). On the other hand, not all structural relationships between objects may be amenable to high-level analysis, e.g., sharing relationships between objects may depend on specific implementation decisions that are invisible at a high level [54]. We have found that combining high- and low-level analyses works well for this [38]. The idea is to first carry out a high-level analysis and annotate the high-level representation of the program with this information. When this is translated to a lower-level representation (e.g., from an abstract syntax tree to a sequence of intermediate code instructions), the high-level properties are also translated into low-level terms alongside, and the low-level representation annotated appropriately. Subsequent low-level optimizations can then use the low-level information in a straightforward way.

3 Cost Models and Code Optimization

A fundamental problem in low-level code optimization is that abstract interpretation can tell us only whether a particular optimization is permissible: it has nothing to say about whether or not it is desirable in a particular context. For example, we may discover, as a result of alias analysis, that a variable may be kept in a register over the course of a computation without affecting the result. It may turn out, however, that this is not a worthwhile thing to do because it precludes the use of that register to hold another, more frequently accessed, variable. The kinds of information typically obtained from abstract interpretation provide little guidance on the latter point.

One might feel that this is not, after all, such an important issue because the primary technical problem in program analysis and optimization is to ensure that "bad things" do not happen, i.e., an optimization does not cause a program to behave incorrectly. It is undeniably true that correctness is fundamentally more important than performance, and that we should always choose to compute a correct result—perhaps slowly—rather than an incorrect result quickly. It can be argued, however, that identifying "bad things happening" with semantic incorrectness takes too narrow a view of the situation. Given two computations that both produce the same correct solution to a problem, we would probably choose the one that is faster, or uses less memory, or is better according to some appropriate measure of performance. In such a setting, if the performance of a program is adversely affected by the poor decisions of an optimizer, one can certainly argue that "bad things" have happened.

As an example of a perfectly plausible optimization where inadequate attention to low-level details can lead to a performance degradation, consider subprogram inlining (which is conceptually very similar to the "unfolding" transformation of Burstall and Darlington [13]). The main motivation behind this transformation, where a call to a subroutine is replaced by (an appropriate instance of) the body of the called subroutine, is to reduce program execution time by eliminating the overhead associated with calling the subroutine and eventually returning from it. Davidson and Holler have shown, however, that register usage can be adversely affected by inlining: first, the number of registers that have to be saved and restored at a subroutine call may increase after inlining; and second, register allocation decisions may change as a result of inlining, causing some frequently accessed variables to be stored in memory [26]. This can cause the inlined program to actually run slower than the program without inlining. Cooper *et al.* report a similar experience—though for different reasons—with subprogram inlining in Fortran [22]. Richardson [62] describes a somewhat different form of "bad things happening" in the context of this transformation: individual functions may grow enormously in size as a result of inlining (even though the overall growth of the size of the entire program may be relatively modest), leading to greatly increased time and space requirements during compilation and optimization, and in the worst case causing compilation to fail due to inadequate memory.

Another example of this phenomenon can be seen in stack allocation of closures in functional languages [46, 47]. The idea is that while closures need to be heap allocated in general, with enough information about the lifetime of a closure in a program it may be possible to avoid this and allocate it on the stack instead (for a discussion of various low-level considerations for stack vs. heap allocation, see [4]). Unless care is exercized, however, this can lead to an increase in the memory requirements of a program because dead variables in stack-allocated closures are nevertheless traversed by the garbage collector [17]. In extreme cases, this can cause a program to fail at runtime due to insufficient memory availability.

The final example of potentially-pessimizing optimizations we consider is tabulation (also known as memoization), where calls to a function or procedure, and the corresponding return values, are noted in a table [10]. The idea is that by consulting this table, subsequent calls may be able to reuse a previously computed value and thereby avoid having to actually execute the called function. An oft-cited example of the benefits of tabulation is the naive exponential-time Fibonacci function, which runs in linear time with tabulation. However, if functions are tabulated without careful consideration of the relative costs and benefits of tabulation, the cost of table manipulation can overwhelm any benefits that accrue from it. As an example, in an experiment with tabulation using Ackermann's function, we found that the computation generated so many entries in the table that even though table lookups incurred a great many successful "hits," the cost of table management led to an overall slowdown in the program. The large number of table entries also led to a significant increase in the memory requirements of the program, raising again the specter of runtime failure due to insufficient memory.

These examples illustrate two points: first, without careful attention to low-level details, even apparently plausible optimizations can result in an overall degradation in program performance; and second, such performance degradations should be taken seriously as a "bad thing." In the worst case they can lead to execution failure in correctly written programs, and this is no better than an incorrectly performed optimization. A fundamental motivation behind program analysis frameworks such as abstract interpretation is to give such analyses a solid foundation on the mathematical semantics of programming languages and thereby allow us to reason formally about properties such as correctness. This, in turn, is driven by the desire to ensure that any transformations that are performed do not change the behavior of a program in undesirable ways. This suggests the need for reasonable cost models that are able to account for low-level aspects of program execution in sufficient detail that optimizations guided by them can reasonably be expected to not "goof up" too badly (Dean and Chambers [28] discuss the use of such cost models to guide the subprogram inlining optimization discussed above).

Note that the need for low-level cost models does not go away if we "lift" low-level operations to the source level, as is done for boxing and unboxing operations using representation types [57]. For example, Henglein and Jørgensen's notion of formally optimal boxing [39] does not take into account machine level costs (or execution frequencies). Because of this, it may happen that a program that is compiled to formally optimal form may be slower, at runtime, than one that is not optimal in this sense, but which uses a low level cost model and execution frequency information to guide the placement of boxing and unboxing operations (e.g., see [56]).

Unfortunately, the construction of reasonable low-level cost models seems nontrivial for a number of reasons. First, it seems quite difficult to predict the "concrete" cost of a program, e.g., in terms of the number of machine cycles it takes to execute the program on a particular input, because even if we choose to ignore the characteristics and behavior of the underlying operating system, we would have to account for machine-level aspects of execution, such as cache behavior, in considerable detail. One possibility might be to abstract away from such "really low-level" and more or less unpredictable aspects and use some kind of abstract machine description that nevertheless models some of the more important aspects of an implementation. Such abstract cost models have been used successfully, for example, for data representation optimizations [65], for improving data locality [14, 78], and register allocation (see, for example, [9, 15, 20]).

However, even with simplifications to the machine model to make it tractable, we may need estimates of execution frequencies for different parts of a program to give an estimate of its cost: this is crucial for optimizations where a reduction in cost in one part of a program may be traded for a possible increase in cost in another part. Where current systems use execution frequency estimates, however, they very often tend to rely on fairly simple-minded heuristics based on the static loop nesting structure of the program. This can lead to estimates that are quite imprecise. As an example, a common heuristic used for register allocation in compilers is to assume that each loop is executed some fixed number of times, usually between 3 and 10 (see, for example, [9, 15, 20, 58, 70]). Wall's studies indicate, however, that the profiles of basic block execution frequency and procedure call frequency obtained using this technique can be surprisingly poor, being, in many cases, not much better than random profiles [75]. As users, we have experienced this problem in the context of our Janus compiler [38], which translates programs to C and invokes `gcc`: our lack of explicit control over register allocation in the C compiler,[3] combined with its often imperfect execution frequency estimates, occasionally lead to the unexpected situation where transformations at the Janus virtual machine level that one would reasonably expect to yield speed improvements actually produced slowdowns in overall execution speed. As a concrete example, in a benchmark program to evaluate Chebyshev polynomials, when we turned off garbage collection—expecting an improvement in execution speed because of a reduction in the number of explicit overflow checks on the heap pointer—we found that the change in the number and distribution of static references to the heap pointer led to changes in the register allocation decisions in the C compiler that resulted in an overall slowdown of about 50%.

The problem is not entirely that static analysis problems such as the estimation of execution frequencies and costs are not amenable to formal methods. Early work on these problems includes that of Cohen and Zuckerman, who consider cost analysis of Algol-60 programs [21]; Wegbreit, whose pioneering work on cost analysis of Lisp programs addressed the treatment of recursion [76]; and those of Ramshaw [60] and Wegbreit [77], who discuss the formal verification of cost specifications. Since then, the question of cost analysis has been investigated by a number of researchers: see, for example, [11, 32, 44, 48, 61, 63, 64, 67, 72, 74]. Many of these use semantics-based methods: for example, Rosendahl [63] uses abstract interpretation for cost analysis, and Wadler [72] uses projection analysis. Despite this fact, the use of formally defensible semantics-based techniques for the estimation of execution frequencies or program costs does not seem very common in actual compilers. This could possibly be due to a perception that such techniques are interesting research tools but too expensive to be part of a compiler. Another reason may be that the information obtained from such analyses, which are typically propositions of the form *"on an input of length N the function `f` requires (at most) $0.5N^2 + 1.5N + 1$ computational steps"*, are not directly amenable to low-level code optimization applications, which would prefer to have more absolute information of the form *"variable `x` is accessed 23000 times"*.

Some recent work on dynamic control of task creation in parallel systems [31, 43] suggests how cost estimates based on semantics-based methods might be incorporated into compilers. In essence, the idea in [31, 43] is to use polyvariant specialization at a low-level to construct different versions for each procedure: one version handles inputs that are large enough to justify the overheads associated with the creation of parallel tasks, and another handles inputs that are small enough that sequential execution is preferable. At runtime, the appropriate version of a function is selected dynamically by comparing the size of the input arguments with a system-dependent "threshold size" for that function that is determined at compile time. In principle, one could imagine using a similar approach for other low-level optimizations as well: generate code for different versions of a program fragment to account for different various optimization scenarios, and choose the one that is appropriate in any particular context, if necessary dynamically. Chambers [16] refers to this kind of application of polyvariant specialization to arbitrary pieces of a program (rather than being limited to, say, functions or procedures) as *splitting*. A straightforward implementation of this idea seems impractical because of the almost certain explosion in code size it would incur. Moreover, interactions between different low-level decisions in different versions would have to be taken into account. It would be interesting to see whether such problems could be addressed well enough to make it practical to incorporate semantics-based methods for execution frequency and cost analysis into compilers.

4 Summary

Compiler optimizations can be divided into two broad classes: high-level optimizations, which correspond to transformations expressible in terms of source-level constructs; and low-level optimizations, which are not so expressible. While abstract interpretation is widely accepted as a natural framework for semantics-based program analyses, we have found that in many cases, such analyses are not quite suitable for low-level optimizations. There are two main reasons for this. The first is that there is often a "semantic mismatch" between the kinds of information abstract interpretations provide, and the kinds of information a compiler wants for its low-level optimizations: abstract interpretations are typically formulated in high-level program semantics, while for low-level optimization we need information about machine-level entities like registers, pointers in memory, etc. The second reason is that in order to carry out a low-level optimization, in general it is not enough to know that the optimization is permissible: we need to know also that it is desirable. Determining whether a particular optimization is desirable in a particular context requires low-level cost models, as well as knowledge about execution frequencies. While there has been a considerable body of work on semantics-based methods for execution cost analysis of programs, these techniques do not seem to be used very much within actual compilers, which tend to use simple and potentially imprecise heuristics. Again, this is

[3] While `gcc` version 2 provides extensions that provide some degree of user control over hardware register allocation, we do not use them at this time for portability reasons.

due in part to a semantic mismatch: semantics-based cost analyses typically yield cost functions (or execution frequency functions) that are expressed in terms of input size, while for optimization purposes it is easier to work with absolute values for execution frequencies and costs.

A fairly obvious solution to the first problem is to use a low-level concrete semantics that makes explicit the entities that are of interest in the context of low-level optimizations. The main pragmatic problem here is that low-level program representations tend to be considerably larger than high-level representations, making analyses more expensive. A possible solution is to reduce the overhead associated with interpreting a program over an abstract domain by using some form of "abstract compilation," i.e., by executing (an appropriately modified form of) the low-level representation of the program instead of interpreting its components. There is the additional issue that program properties that are relatively easily inferrable at a high-level may be obscured in a lower-level analysis, but this can be handled by initially analyzing the program at a high level, then translating the high-level program properties into low-level terms during the translation of the program into a lower-level language.

The second problem can be addressed, at least in principle, via polyvariant specialization at the low-level. This idea has been applied to controlling dynamic task creation in parallel systems, and appears to work reasonably well. However, a significant problem that has to be addressed when applying this to low-level code optimization is that of controlling code growth.

The appeal of semantics-based program manipulation techniques is that they allow us to reason formally about the manipulations themselves, and certify with some confidence that such manipulations will not cause "bad things" to happen. Much of the current practice of low-level optimizations seems guided by simple heuristics rather than careful semantic treatment. Because of this, it is not clear that much can be said about whether or not "bad things"can happen: an indeed, we sometimes do encounter situations where apparently plausible "improvements" to a program can lead to a degradation in its performance. This is undesirable, but if semantics-based techniques can be adapted for low-level optimizations it may be possible to reduce or eliminate such anomalous situations in the future.

Acknowledgements

Numerous valuable discussions with Manuel Hermenegildo are gratefully acknowledged.

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] A. Aiken, E. L. Wimmers, and T. K. Lakshman, "Soft Typing with Conditional Types", *Proc. 21st. ACM Symposium on Principles of Programming Languages*, Portland, Oregon, Jan. 1994, pp. 163–173.

[3] R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form", *ACM Transactions on Programming Languages and Systems* vol. 9 no. 4, Oct. 1987, pp. 491–542.

[4] A. W. Appel and Z. Shao, "An Empirical and Analytical Study of Stack vs. Heap Cost for Languages with Closures", Research Report CS-TR-450-94, Dept. of Computer Science, Princeton University, March 1994.

[5] J. Arsac and Y. Kodratoff, "Some Techniques for Recursion Removal from Recursive Functions", *ACM Transactions on Programming Languages and Systems* vol. 4 no. 2, Apr. 1982, pp. 295–322.

[6] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler Transformations for High-Performance Computing", *Computing Surveys* vol. 26 no. 4, Dec. 1994, pp. 345–420.

[7] P. A. Bigot, D. Gudeman, and S. K. Debray, "Output Value Placement in Moded Logic Programs", *Proc. Eleventh Int. Conf. on Logic Programming*, June 1994, pp. 175–189. MIT Press.

[8] P. A. Bigot and S. K. Debray, "A Simple Approach to Supporting Untagged Objects in Dynamically Typed Languages", Draft Report, Dept. of Computer Science, University of Arizona, Tucson, Nov. 1994.

[9] D. Bernstein, M. C. Golumbic, Y. Mansour, R. Y. Pinter, D. Q. Goldin, H. Krawczyk, and I. Nahshon, "Spill Code Minimization Techniques for Optimizing Compilers", *Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, June 1989, pp. 258–263.

[10] R. S. Bird, "Tabulation Techniques for Recursive Programs", *Computing Surveys* vol. 12 no. 4, Dec. 1980, pp. 403–417.

[11] B. Bjerner and S. Holmström, "A Compositional Approach to Time Analysis of First Order Lazy Functional Programs", *Proc. ACM Conference on*

Functional Programming Languages and Computer Architecture, 1989, pp. 157–165.

[12] F. Bueno, M. García de la Banda and M. Hermenegildo, "Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization, *Proc. International Symposium on Logic Programming*, Nov. 1994, pp. 320–336. MIT Press.

[13] R. M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs", *Journal of the ACM* vol. 24 no. 1, Jan. 1977, pp. 44-67.

[14] S. Carr, K. S. McKinley, and C.-W. Tseng, "Compiler Optimizations for Improving Data Locality", *Proc. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, Nov. 1994, pp. 252–262. SIGPLAN Notices vol. 29 no. 11.

[15] G. J. Chaitin, "Register Allocation via Graph Coloring", *Proc. 1982 ACM Conference on Compiler Construction*, Boston, June 1982, pp. 98–104.

[16] C. Chambers, *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*, Ph.D. Dissertation, Stanford University, 1992.

[17] D. R. Chase, "Safety Considerations for Storage Allocation Optimizations", *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, June 1988, pp. 1–10.

[18] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of Pointers and Structures", *Proc. ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 296–310.

[19] W.-N. Chin, "Safe Fusion of Functional Expressions", *Proc. ACM Conference on Lisp and Functional Programming*, San Francisco, June 1992, pp. 11–20.

[20] F. C. Chow and J. L. Hennessy, "The Priority-Based Coloring Approach to Register Allocation", *ACM Transactions on Programming Languages and Systems* vol. 12 no. 4, Oct. 1990, pp. 501–536.

[21] J. Cohen and C. Zuckerman, "Two Languages for Estimating Program Efficiency", *Communications of the ACM* vol. 17 no. 6, June 1974, pp. 301–308.

[22] K. D. Cooper, M. W. Hall, and L. Torczon, "Unexpected Side Effects of Inline Substitution", *ACM Letters on Programming Languages and Systems* vol. 1 no. 1, March 1992, pp. 22–32.

[23] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Apporoximation of Fixpoints", *Proc. Fourth ACM Symposium on Principles of Programming Languages*, 1977, pp. 238-252.

[24] P. Cousot, and R. Cousot, "Systematic Design of Program Analysis Frameworks", Proc. Sixth ACM Symposium on Principles of Programming Languages, 1979, pp. 269-282.

[25] P. Cousot, "Semantic Foundations of Program Analysis", in *Program Flow Analysis: Theory and Applications*, eds. S. S. Muchnick and N. D. Jones, Prentice-Hall, 1981.

[26] J. W. Davidson and A. M. Holler, "Subprogram Inlining: A Study of its Effects on Program Execution Time", *IEEE Transactions on Software Engineering* vol. 18 no. 2, Feb. 1992, pp. 89–102.

[27] K. De Bosschere, S. K. Debray, D. Gudeman, and S. Kannan, "Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages", *Proc. 21st. ACM Symposium on Principles of Programming Languages*, Portland, Oregon, Jan. 1994, pp. 409–420.

[28] J. Dean and C. Chambers, "Towards Better Inlining Decisions using Inlining Trials", *Proc. 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994, pp. 273–282.

[29] S. K. Debray, "Optimizing Almost-Tail-Recursive Prolog Programs", *Proc. Functional Programming Languages and Computer Architecture*, Nancy, France, Sept. 1985.

[30] S. K. Debray and D. S. Warren, "Automatic Mode Inferencing for Logic Programs", *J. Logic Programming* vol. 5 no. 3, Sept. 1988, pp. 207-229.

[31] S. K. Debray, N. Lin and M. Hermenegildo, "Task Granularity Analysis in Logic Programs," *Proc. ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990, pp. 174–188.

[32] S. K. Debray and N.-W. Lin, "Cost Analysis of Logic Programs", *ACM Transactions on Programming Languages and Systems*, vol. 15 no. 5, Nov. 1993, pp. 826–875.

[33] A. Deutsch, "On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher Order Functional Specifications", *Proc. 17th ACM Symposium on Principles of Programming Languages*, Jan. 1990, pp. 157–168.

[34] D. R. Engler and T. A. Proebsting, "DCG: An Efficient, Retargetable Dynamic Code Generation System", *Proc. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, Nov. 1994, pp. 263–271. SIGPLAN Notices vol. 29 no. 11.

[35] I. Foster and W. Winsborough, "Copy Avoidance through Compile-Time Analysis and Local Reuse", *Proc. 1991 International Symposium on Logic Programming*, San Diego, Nov. 1991, pp. 455–469. MIT Press, Cambridge.

[36] P. B. Gibbons and S. S. Muchnick, "Efficient Instruction Scheduling for a Pipelined Architecture", *Proc. ACM SIGPLAN '86 Conference on Compiler Construction*, June 1986, pp. 11–16.

[37] K. Gopinath and J. Hennessy, "Copy Elimination in Functional Languages", *Proc. Sixteenth ACM Symposium on Principles of Programming Languages*, Austin, TX, Jan. 1989, pp. 303–314.

[38] D. Gudeman, K. De Bosschere, and S.K. Debray, "`jc`: An Efficient and Portable Sequential Implementation of Janus", *Proc. Joint Int. Conf. and Symp. on Logic Programming*, Nov. 1992, pp. 399–413. MIT Press.

[39] F. Henglein and J. Jørgensen, "Formally Optimal Boxing", *Proc. 21st. ACM Symposium on Principles of Programming Languages*, Portland, OR, Jan. 1994, pp. 213–226.

[40] M. Hermenegildo and K. Greene, "The &-Prolog System: Exploiting Independent And-Parallelism", *New Generation Computing* vol. 9 nos. 3–4, 1991, pp. 233–257.

[41] M. Hermenegildo, R. Warren and S. K. Debray, "Global Flow Analysis as a Practical Compilation Tool", *Journal of Logic Programming*, vol. 13 no. 4, Aug. 1992.

[42] P. Hudak and A. Bloss, "The Aggregate Update Problem in Functional Languages", *Proc. Twelfth ACM Symposium on Principles of Programming Languages*, 1985, pp. 300–314.

[43] L. Huelsbergen, J. R. Larus, and A. Aiken, "Using Run-Time List Sizes to Guide Parallel Thread Creation", *Proc. ACM Conference on Lisp and Functional Programming*, June 1994, pp. 79–90.

[44] S. Kaplan, "Algorithmic Complexity of Logic Programs", *Proc. Fifth International Conference on Logic Programming*, Seattle, 1988, pp. 780–793. MIT Press.

[45] D. Keppel, S. J. Eggers, and R. R. Henry, "A Case for Runtime Code Generation", Technical Report 91-11-04, Department of Computer Science, University of Washington, 1991.

[46] D. Krantz, *ORBIT: An Optimizing Compiler for Scheme*, Ph.D. Dissertation, Yale University, 1988. (Also available as Technical Report YALEU/DCS/RR-632, Dept. of Computer Science, Yale University, Feb. 1988.)

[47] D. Krantz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams, "ORBIT: An optimizing Compiler for Scheme", *Proc. SIGPLAN '86 Symposium on Compiler Construction*, pp. 219–233.

[48] D. Le Métayer, "ACE: An Automatic Complexity Evaluator", *ACM Transactions on Programming Languages and Systems* vol. 10 no. 2, April 1988, pp. 248–266.

[49] X. Leroy, "Unboxed objects and polymorphic typing", *Proc. 19th. ACM Symposium on Principles of Programming Languages*, Albuquerque, NM, Jan. 1992, pp. 177–188.

[50] T. S. McNerney, "Verifying the Correctness of Compiler Transformations on Basic Blocks using Abstract Interpretation", *Proc. Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, CT, June 1991, pp. 106–115.

[51] A. Mariën, G. Janssens, A. Mulkers, and M. Bruynooghe, "The Impact of Abstract Interpretation on Code Generation: an Experiment in Code Generation", *Proc. Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989. MIT Press.

[52] D. Maurer, "Strictness computation using special λ-expressions", in *Programs as Data Objects*, Oct. 1985, pp. 136–155. Springer Verlag LNCS vol. 217.

[53] A. Mulkers, W. Winsborough, and M. Bruynooghe, "Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs", *Proc. Seventh International Conference on Logic Programming*, Jerusalem, June 1990, pp. 747–762. MIT Press.

[54] A. Mulkers, W. Winsborough, and M. Bruynooghe, "Live-Structure Dataflow Analysis for Prolog", *ACM Transactions on Programming Languages and Systems* vol. 16 no. 2, March 1994, pp. 205–258.

[55] R. Paige and S. Koenig, "Finite Differencing of Computable Expressions", *ACM Transactions on Programming Languages and Systems* vol. 4 no. 3, July 1982, pp. 402–454.

[56] J. C. Peterson, "Untagged Data in Tagged Environments: Choosing Optimal Representations at Compile Time", *Proc. Functional Programming Languages and Computer Architecture*, London, Sept. 1989, pp. 89–99.

[57] S. Peyton Jones and J. Launchbury, "Unboxed values as first class citizens in a non-strict functional language", *Proc. Functional Programming Languages and Computer Architecture 1991*, pp. 636–666.

[58] M. L. Powell, "A Portable Optimizing Compiler for Modula-2", *Proc. SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984, pp. 310–318.

[59] T. A. Proebsting and C. N. Fischer, "Linear-time Optimal Code Scheduling for Delayed-Load Architectures", *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, June 1991, pp. 256–267.

[60] L. H. Ramshaw, *Formalizing the Analysis of Algorithms*, Ph.D. Thesis, Stanford University, 1979. (Also available as Report SL-79-5, Xerox Palo Alto Research Center, Palo Alto, California, 1979.)

[61] B. Reistad and D. Gifford, "Static Dependent Costs for Estimating Execution Time", *Proc. 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994, pp. 65–78.

[62] S. E. Richardson, *Evaluating Interprocedural Code Optimization Techniques*, Ph.D. Dissertation, Stanford University, 1991. (Also available as Technical Report CSL-TR-91-460, Computer Systems Laboratory, Stanford University, Feb. 1991.)

[63] M. Rosendahl, "Automatic Complexity Analysis", *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, 1989, pp. 144–156.

[64] D. Sands, "Complexity Analysis for a Lazy Higher-Order Language", *Proc. 3rd European Symposium on Programming*, May 1990, pp. 361–376. Springer-Verlag LNCS vol. 432.

[65] E. Schonberg, J. T. Schwartz, and M. Sharir, "An Automatic Technique for Selection of Data Representations in SETL Programs", *ACM Transactions on Programming Languages and Systems* vol. 3 no. 2, April 1981, pp. 126–143.

[66] M. Sharir, "Some Observations Concerning Formal Differentiation of Set Theoretic Expressions", *ACM Transactions on Programming Languages and Systems* vol. 4 no. 2, April 1982, pp. 196–225.

[67] J. Shultis, "On the Complexity of Higher-Order Programs", Technical Report CU-CS-288, University of Colorado, Feb. 1985.

[68] A. Taylor, "LIPS on a MIPS: Results from a Prolog Compiler for a RISC", *Proc. Seventh International Conference on Logic Programming*, Jerusalem, Israel, June 1990.

[69] A. Taylor, *High Performance Prolog Implementation*, Ph.D. thesis, University of Sidney, Australia, 1991.

[70] K. Thompson, "A New C Compiler", *Proc. Summer 1990 UKUUG Conference*, London, July 1990, pp. 41–51.

[71] P. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, 1990.

[72] P. Wadler, "Strictness Analysis Aids Time Analysis", *Proc. 15th. ACM Symposium on Principles of Programming Languages*, Jan. 1988, pp. 119–132.

[73] P. Wadler, "Deforestation: Transforming programs to eliminate trees", *Proc. European Symposium on Programming*, Nancy, France, March 1988, pp. 344–358. Springer-Verlag LNCS vol. 300.

[74] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison, "Accurate Static Estimators for Program Optimization", *Proc. ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994, pp. 85–96.

[75] D. W. Wall, "Predicting Program Behavior Using Real or Estimated Profiles", *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991, pp. 59–70.

[76] B. Wegbreit, "Mechanical Program Analysis", *Communications of the ACM* vol. 18 no. 9, Sept. 1975, pp. 528–539.

[77] B. Wegbreit, "Verifying Program Performance", *Journal of the ACM* vol. 23 no. 4, Oct. 1976, pp. 691–699.

[78] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm", *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991, pp. 30–44.