Journal of Universal Computer Science, vol. 12, no. 7 (2006), 764-787
submitted: 28/1/06, accepted: 6/4/06, appeared: 28/7/06 © J.UCS

The Design of the YAP Compiler: An Optimizing Compiler for Logic Programming Languages

Anderson Faustino da Silva
(Federal University of Rio de Janeiro, Brazil
faustino@cos.ufrj.br)
Vitor Santos Costa
(Federal University of Rio de Janeiro, Brazil
vitor@cos.ufrj.br)

Abstract: Several techniques for implementing Prolog in a efficient manner have been devised since the original interpreter, many of them aimed at achieving more speed. There are two main approaches to efficient Prolog implementation: (1) compilers to bytecode and then interpreting it (emulators) or (2) compilers to native code. Emulators have smaller load/compilation time and are a good solution for their simplicity when speed is not a priority. Compilers are more complex than emulators, and the difference is much more acute if some form of code analysis is performed as part of the compilation, which impacts development time. Generation of low level code promises faster programs at the expense of using more resources during the compilation phase. In our work besides using an mixed execution mode, we design an optimizing compiler that using type feedback profiling, dynamic compilation and dynamic deoptimization for improving the performance of logic programming languages.

Key Words: Dynamic compilation, Just-in-Time compiler, compiler optimizations
Category: C.4, D.3.4

1 Introduction

Programming language designers have always searched for programming languages and features that ease the programming process and improve programmer productivity. One promising approach is logic programming. Logic programming languages provide programmers with powerful techniques for writing programs quickly and easily.

Prolog (PROgramming in LOGic) [Sterling and Shapiro 1986] is a logic programming language based on a subset of first order logic [Wikipedia 2006]. Some constructions of first order logic have been removed to allow competitive performance, and some extra-logical features have been added such as: flow control, input/output, and meta-programming. Prolog works by querying a database of facts and rules. The language is thus declarative in nature. You tell the system how to recognize the answer and the system searches for the answer. This makes Prolog both easier and harder than other languages. Easier because there is tremendous leverage obtained from the declarative approach, harder because some commonly used programming schemes are not supported.

Several techniques for implementing Prolog have been devised since the original interpreter [Colmerauer 1993]. There are two main approaches to efficient Prolog implementation: **(1)** compiling to bytecode and then interpreting it (emulators) [SICStus 2006, YAP 2006] or **(2)** compilers to native code. This second approach can be divided into two categories. One solution is for the compiler to generate machine code directly [Roy and Despain 1992, SICStus 2006, Marien 1993, Taylor 1991, Diaz and Codognet 2001]. An alternative is to generate code for a language, such as `C`, for which compilers are readily available [Morales et al 2004].

Each solution has its advantages and disadvantages. Generating low level code promises faster programs at the expense of using more resources during compilation. Emulators have smaller load/compilation time and are a good solution for their simplicity when speed is not a priority; executing the same Prolog code in different architectures boils down to recompiling the interpreter. Compilers are more complex than emulators, and the difference is much more acute if some form of code analysis is performed as part of the compilation, which impacts development time. In this work we propose a solution that combines advantages of the two approaches.

The different approaches are useful in different situations and for different parts of a program. The emulator approach can be very useful during development, and for non-performance bound portions of programs. On the other hand, some fragments of code should be optimized at all levels. In this case, the standard optimizations of imperative language compilers (such as: global register allocation [**?**], code motion [Knoop et al 1994a], instruction reordering [Goodman and Hsu 1988], and others [Muchnick 1997]) become important for improving performance in Prolog compilers.

Aggressive compiler optimizations can reduce the overhead impose by Prolog language features. However, the long compilation times introduced by optimizing compilers delay the programming environment's responses to changes in the program. Furthermore, optimization also conflicts with source-level debugging. Thus, programmers have to choose between abstraction and efficiency, and between responsive programming environments and efficiency.

This work shows how to reconcile these seemingly contradictory goals by using *dynamic optimization* performing optimizations lazily. Three techniques work together to achieve high performance and high responsiveness in Prolog programs:

1. **Type feedback** achieves high performance by allowing the compiler to compile only the executed path based on information extracted from the runtime system.

2. **Adaptive optimization** achieves high responsiveness without sacrificing

performance by using a emulator to interpret initial code while automatically compiling heavily used parts of the program with an optimizing compiler.

3. **Dynamic deoptimization** shields the programmer from the complexity of debugging optimized code by transparently recreating non-optimized code as needed. No matter whether a program is optimized or not, it can always be stopped, inspected, and single-stepped. Compared to previous approaches, deoptimization allows more debugging while placing fewer restrictions on the optimizations that can be performed.

In YAPc we propose precisely such an approach: we implemented a compilation from Prolog to native code, using dynamic compilation. Our starting point is develop a dynamic optimizing compiler, essentially an high-performance emulator-based system. Its system will be an evolution of the YAP emulator. This facilitates mixing emulated and native code. In our system we implement the techniques enumerated above, and we go beyond the WAM [Warren 1983] to improve the execution speed.

With better performance yet good interactive behavior, these techniques make high-performance possible both for pure logic languages and for application domains requiring higher ultimate performance.

Our immediate goal is to build an efficient, usable implementation of Prolog. However, we are not willing to compromise Prolog's language semantic and other expressive features. We want to preserve the illusion of the system directly executing the program as the programmer wrote it, with no user-visible optimizations. This constraint has several consequences:

– *The programmer must be free to edit any clause in the system.*

– *The programmer must be able to understand the execution of the program and any errors in the program solely in terms of the source code and the source language constructs.* This requirement on the debugging and monitoring interface to the system disallows any internal optimizations that would shatter the illusion of the implementation directly executing the source program as written. Programmers should be unaware of how their programs get interpreted or compiled.

– *The programmer should be isolated even from the mere fact that the programs are getting compiled at all.* No explicit commands to compile a clause or program should ever be given, even after programming changes. The programmer just runs the program. This illusion of hiding the compiler would break down if the programmer were distracted by mysterious pauses due to compilation, analysis, or optimization. Ideally any pauses incurred by the implementation of the system would be imperceptible, such as on the order of a fraction of a second when in interactive use. Longer running batch programs can be interrupted by longer pauses, as long as the total time of the program is not slowed so much that the programmer becomes aware of the pauses.

Within these constraints on the user-visible semantics of the system, our main objective is excellent run-time performance. We wish to make Prolog with similar powerful features competitive in performance with traditional imperative languages such as C and Pascal. If these performance goals are met, many programmers may be able to switch from traditional languages to logic languages and begin to reap the benefits afforded by logic programming.

Other goals are secondary to the constraint of a source-level execution model and the goal of rapid execution. In particular, run-time and compile-time space overheads are less of a concern than run-time speed. Modern computer platforms, especially workstations, are typically equipped with a large amount of physical main memory, and this amount is increasing at a rapid rate. We therefore are willing to use more space than would a straightforward implementation in order to meet our execution speed goals.

Of course, we do not only wish to implement Prolog efficiently, but also a larger class of logic languages. Fortunately, the techniques used are not specific to the Prolog language. They were applied in languages such as SELF, C++ and Java. In our work we demonstrated how this techniques can be used in logic languages.

The rest of this paper is organized as follows. The section 2 presents the brief description about Prolog language. The section 3 describes how some implementation beyond the WAM to achieve high performance. The section 4 presents our optimizing compiler. And the section 5 summarizes ours conclusions.

2 The Prolog Language

Prolog [Sterling and Shapiro 1986] is a general-purpose language based on the Horn clause subset of first-order predicate calculus [Wikipedia 2006]. Execution of a Prolog program is effectively an application of theorem proving by first-order resolution. Fundamental concepts are unification, tail recursion, and backtracking. Prolog is a rich collection of data structures in the language and human reasoning, and a powerful notation for encoding end-user applications. It has its logical and declarative aspects, compactness, and inherent modularity. And, it is used in many artificial intelligence programs and in computational linguistics, especially natural language processing. Its syntax and semantics are considered very simple and clear.

In a dynamically typed language such Prolog, variables may contain objects of any type at runtime. Hence, it must be possible to determine the type of an object at run-time by inspection. Terms can be represented as tagged words: a word contains a tag field and a value field. The tag field contains the type of the term, namely: atom, number, list, or structure. The value field is used for different purposes depending on the type. It can contain the value of integers, the address of unbound variables, compound terms (list or structures), and it ensures that each atom has a value different from all other atoms. Unbound variables are implemented as self-referential pointers, i.e., they point to themselves. When two variables are unified, one of them is modified to point to the other. Therefore it may be necessary to follow a chain of pointers to access a variable's value. This is called dereferencing the variable.

This language feature provide a more expressive mechanism for describing and manipulating data structures. Widespread use of this feature greatly increases the frequency of procedure calls or the number of branches over traditional programming styles using concrete data types. With dynamic data types, each manipulation is conceptually a procedure call that invokes the handler of one concrete type. A system with heavy use of dynamic data types imposes a significant overhead.

To eliminate the run-time cost of abstraction, implementations can expand the body of a called procedure in place of the procedure call; this technique is known as procedure integration or inlining [Zhao and Amaral 2003]. When an operation on an abstract data type is invoked, the compiler can expand the implementation of the operation for that abstract data type in-line, eliminating the procedure call. With aggressive use of inlining, the overhead of abstract data types can be virtually eliminated, removing a performance barrier that might discourage the use of an important program structuring tool. Besides, the several branches can be easier eliminated using the data type information.

3 Improving the Performance of Prolog Programs

Prolog implementations have made great progress in execution efficiency with the development of the WAM [Warren 1983, Ait-Kaci 1991]. However, to improve the execution speed it is necessary to go beyond the WAM. This section discusses how several implementations beyond the WAM to achieve higher performance. Basically, four approaches were proposed: (1) to reduce instruction granularity, (2) to exploit determinism, (3) to specialize unification, and (4) to use dataflow analysis.

The WAM is an elegant mapping of Prolog to a sequential machine. Most instructions specialize the general unification algorithm. Even so, these instructions can be quite complex, so that several optimizations are not possible.

One can reduce instruction granularity in three steps [Komatsu et al 1986, Tamura 1986]. The first step is to compile Prolog into WAM. In the second step the intermediate code is translated into a directed graph. Each WAM instruction becomes a subgraph containing simple operations such as case selection on tags, jumps, assignments, and dereferencing. This graph can be optimized through rewrite rules. Finally, in the third step the intermediate code is translated into some program which is sent to its high-quality optimizing compiler, or the system can generate machine code directly.

The majority of predicates written by human programmers are intended to give only one solution, ie, they are *deterministic*. However, too often they are compiled in an inefficient manner using indexing to choose the correct clause, when they are really just case statements. This is inefficient since backtracking requires saving the machine state and restoring it repeatedly.

Significant improvements over the WAM are obtained by avoiding backtracking deterministic predicates. The WAM itself has indices on only the first argument and saves all registers in choice points. Turk [Turk 1986] was one of the first work describing several optimizations that reduce the time necessary to restore machine state when backtracking.

SICStus Prolog [SICStus 2006, Nassen 2001] system reduces the overhead from backtracking support by creating choice points in two parts: first, it save only a small part of the machine state, postponing saving the remainder until the point in the clause where it can be determined that the head unification and simple tests succeeded.

Several systems have generalized the first argument indexing of the WAM. BIM_Prolog [Marien 1993] can index on any argument when given appropriate declarations. SEPIA [Meier et al 1989] incorporates heuristics to decide which predicate arguments are important for deterministic selection. It uses the first "indexable" argument of a predicate. If there are several possibilities it first uses the argument where it is more likely that fewer clauses will be selected.

We propose *type-feedback profiling* for YAPc. The key idea of type feedback is to extract type information from the runtime system and feed it back to the compiler. To obtain the type profile, the standard clause dispatch mechanism has to be extended to record the desired information, e.g., we keep a table of all predicates invoked and of *how*. Based on this profiling, the compiler can decided what to compile.

The WAM unification instructions (`get` and `unify`) are complex. They operate in two modes (read mode and write mode) depending on the type of the object being unified, they dereference their arguments, and they trail variable bindings. The profiling information is designed to keep sufficient context to be able to compile unification directly into simpler instructions.

Significant improvements over the WAM are possible for unification. Turk [Turk 1986] describes several optimizations related to compilation of unification, to reduce the overhead of explicitly maintaining a read/write mode bit and remove some superfluous dereferencing and tag checking. Marien [Marien 1988] describes a method to compile write mode unification that uses a minimal number of memory operations and avoids all superfluous dereferencing and tag checking. Van Roy [Roy 1989] introduces in this work a simplified notation and extending it for read mode unification, but the scheme suffers from a large code size expansion. The Aquarius system [Roy 1990] modifies this technique to limit the code size expansion at a slight execution time cost. Meier [Meier 1990] has developed a technique that generalizes Marien's idea for both read and write mode and achieves a linear code size, also with a slight execution time cost.

Beer [Beer 1998] suggested the use of a simplified representation of Prolog variables for which binding is much faster. His work introduces several new tags for this representation, which he calls *uninitialized variables*, and keeps track of them at run-time. He shows that both dereferencing and trailing are reduced significantly.

To reduce the overhead of unification our system applies several optimizations to unification. We use two version of unify function, one in read mode and other in write mode.

Traditionally, global analysis of logic programs is used to derive information to improve program execution. Both type and control information can be derived and used to increase speed and reduce code size. The analysis algorithms studied so far are all instances of a general method called abstract interpretation [Cousot 1992]. The idea is to execute the program over a simpler domain. If a small set of conditions are satisfied, this execution terminates and its results provide a correct approximation of information about the original program.

Warren [Warren et al 1988] was the first to study the practicality of global dataflow analysis in logic programming. He described two dataflow analyzers: (1) MA3, the MCC And-parallel Analyzer and Annotator, and (2) Ms, an experimental analysis scheme developed for SBProlog. MA3 derives aliasing and ground types and keeps track of the structure of compound terms, while Ms derives ground and nonvariable types. His paper demonstrate that both dataflow analyzers are effective in deriving types and do not increase compilation time by too much. Marien [Marien and Demoen 1989], Van Roy [Roy 1990], and more recently Morales [Morales et al 2004] also obtain similar results.

Analysis results in both code size reduction and speed increase. However, the the effects of analysis on code size and speed are fundamentally different. Derived types allow both tests and the code that handles other types to be removed. Tests are usually fast. The code to handle all possible outcomes of the tests can be very large. Besides, deriving types that have a logical meaning is not sufficient. Performance increases significantly when the analysis is able to derive types that have only an operational meaning, such as dereference (reference chains), trailing, and aliasing-related types (uninitialized variables).

Unfortunately, global analysis has not scaled well to larger, very complex, Prolog applications. Such applications can be rather hard to understand statically. Type-feedback profiling seems rather useful in the context as it guides the selection of executable paths from the execution dynamics. YAPc will compile only relevant paths, and interpret the remaining code. YAPc will rely on the signicant body experience of the global analysis community to derive which information is most usefui.

4 The Design of the Optimizing Compiler

Based on the idea of using the type feedback information, we have developed a new optimizing compiler for Prolog. The compiler's design goals were twofold:

– **High compilation speed**. Since the runtime system uses dynamic compilation, the compiler should be as fast as possible in order to minimize compilation pauses. We avoided costly global analyzes and tried to restrict ourselves to algorithms whose time complexity is at most linear in the size of the source program.

– **High performance for large programs**. The second design goal was to surpass the performance of the previous compiler's logic programs, and especially to provide good performance for large applications. In addition, performance should be reasonably stable: minor changes in the source program should not significantly affect performance.

Dynamically-typed logic programming languages historically have run much slower than traditional statically-typed programming languages. This performance gap is attributable largely to the lack of representation-level type information in the dynamically-typed languages. This representation-level information about an variable is embodied in the cell in a cell-based system.

If the compiler could infer the cell at compile-time, it could eliminate much of the run-time overhead associated with dynamic typing. In a dynamically-typed language, the compiler must insert extra run-time type-checking code around type-safe primitives. If the compiler could infer the type of the arguments to the type-checking primitive, then it could perform the type checks at compile-time rather than run-time.

In logic programming language, the compiler must insert extra run-time code to implement dynamic binding of data type to target variable based on the run-time argument type of an predicate. If the compiler could infer the type of an argument, then it could determine the data type at compile-time instead of run-time, and it can replace the extra run-time code by code's executable path only. This code would subsequently be amenable to further optimizations such as copy propagation [Bacon et al 1994], sparse conditional constant propagation [Wegman and Zadeck 1991], dead code elimination [Knoop et al 1994b], code motion [Knoop et al 1994a], loop versioning [Bacon et al 1994] and others. We implemented these techniques in our optimizing compiler.

Clearly, the run-time performance of dynamically-typed logic programs could be dramatically improved if the compiler could infer representation-level type information in the form of cells. On the surface, this would seem to imply that statically-typed languages, with lots of type information available to the compiler, would have a huge advantage in performance over their dynamically-typed counterparts.

In other languages, the type of a variable specifies the representation or implementation of the contents of the variable. This static information corresponds to knowing the exact class of the contents of the variable and hence supports the optimizations described above that reduce the gap between dynamically-typed languages and statically-typed languages.

The lack of static representation-level type information limits the run-time performance of logic languages. Consequently, our compilation techniques will strive to infer this missing representation-level type information, so that the compiler can perform optimizations to eliminate the overhead of dynamic typing and predicate orientation. Once these optimizations have been performed, the task of compiling a dynamically-typed logic program reduces to the task of compiling a traditional statically-typed program.

To optimize the Prolog program the compiler must prove that the argument of a predicate has a single type, i.e., that the argument value is monomorphic. In general, however, Prolog arguments are polymorphic: it may denote values of different types at different times, and the same source code works fine for all these types.

In most cases, however, the compiler's task is not so easy: all predicate's arguments really own potentially polymorphic data type. Nevertheless, the compiler can frequently optimize even polymorphic data types. The compiler includes the technique type prediction that can transform some kinds of polymorphic argument into monomorphic argument.

Since identifying and creating monomorphic sections of code can be fairly time consuming, the Prolog compiler seeks to conserve its efforts. In particular, the compiler attempts to compile only those parts of the Prolog program that are actually executed. The compiler only performs optimizations on demand, exploiting system's dynamic compilation architecture. Additionally, many cases that could arise in principle but rarely arise in practice, such as integer overflows or illegally-typed arguments to primitives, are never actually compiled by the Prolog compiler, thus saving a lot of compile time and compiled code space and allowing better optimization of the parts of programs that are executed.

Figure 1 shows an overview of the optimizing compiler. The compiler is divided into front-end and back-end. The front-end first realizes the parsing, that only represent the WAM code into an abstract syntaxe tree (AST) reducing the instruction granularity. After parsing, the compiler transforms the AST into a SSA-based control-flow graph [Cytron et al 1991, Click and Paleczny 1995]. Using the type-feedback profiler the control-flow graph is minimized and some SSA-based optimizations are applied. When the front-end finishes its execution, the back-end initiates. It transforms the SSA-based representation into a low level representation, and it applies some optimizations. Like the front-end, the back-end uses type-feedback profiler to guide this phase. Following this step, the back-end allocates registers and finally it generates native code. When the compiler finishes the compilation process, the native code is installed into runtime system and is scheduled for execution.

The bulk of the compiler effort lies in between the two halves of a traditional compiler. This "middle half" of the compiler performs the representation-level type analysis and region selection that bridges the semantic gap between the high-level polymorphic program input to the compiler and the lower-level monomorphic version of the program suitable for the optimizations performed by a traditional compiler back-end.

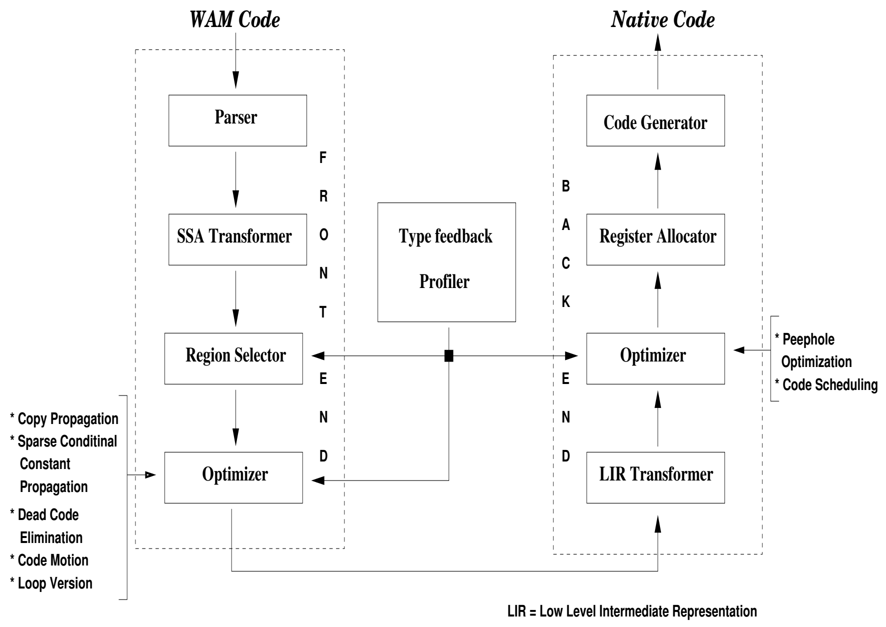

Figure 1: The Optimizing Compiler's Architecture

4.1 Dynamic Compilation

Our system uses dynamic compilation [Auslander et al 1996] not only to take advantage of type feedback but also to determine which parts of an application should be optimized in the first place. Figure 2 shows an overview of the dynamic compilation process in our system.

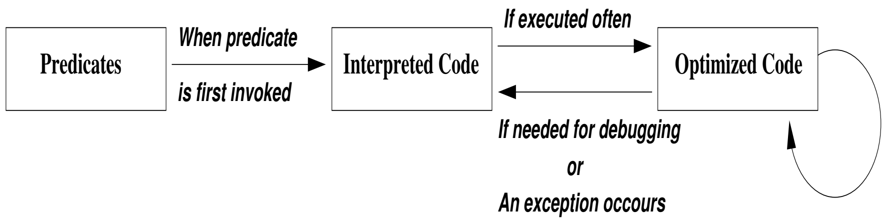

Figure 2: The Dynamic Compilation Process

When a source predicate is invoked for the first time, it is interpreted by the emulator. If it is executed often, it is compiled and optimized using type feedback. Having a very fast compiler is essential to reduce compile pauses in an interactive system using dynamic compilation. Sometimes, an optimized predicate is reoptimized to take advantage of additional type information or to adapt it to changes in the program's type profile.

This system has to discover opportunities for compilation without programmer intervention. In particular, it has to decide:

- *When to compile*: how long to wait for type information to accumulate,
- *What to compile*: which compiled code would benefit most from the additional type information, and
- *Which executable path*: selecting only executable path deserving compilation.

The following subsections discuss each of these questions. The solutions presented here all employ simple heuristics, but nevertheless work well.

4.2 When to Compile

The system uses counters to detect compilation candidates. Each interpreted predicate has its own counter. In its prologue code, the predicate increments the counter and compares it to a limit. If the counter exceeds the limit, the compilation system is invoked to decide which (if any) predicate should be compiled. If the predicate overflowing its counter, it is not compiled, and its counter is reset to zero.

Originally, counters were envisioned as a first step, to be used only until a better solution was found. However, the trigger mechanism ("when") is much less important for good compilation results than the selection mechanism ("what"). Since the simple counter-based approach has worked well [da Silva 2005], we did not extensively investigate other mechanisms. However, there are some interesting questions relating to invocation counter:

– *Ideally, the system would compile those predicates where the optimization cost is smaller than the benefits that accrue over future invocations of the optimized predicate.* Of course, the system does not know how often an predicate will be executed in the future, but an rate-based measure also ignores the past: an predicate that executes less often than the minimum execution rate will never trigger an compilation, even if it is executed several times.

– *The invocation limit should not be a constant, rather, it should depend on the particular predicate.* What the counters are really trying to measure is how much execution time is wasted by running interpreted code. Thus, an predicate that would benefit much from optimization should count faster (or have a lower limit) than an predicate that would not benefit much from optimization. Of course, it may be hard to estimate the performance impact of optimization on an particular predicate.

– *How should life times be adapted when executing on a faster (or slower) machine?* Suppose that the original life parameter was 10 seconds, but that the system now executes on a new machine that is twice as fast. Should the life parameter be changed, and if so, how? One could view that faster machine as a system where real time runs half as fast, and thus reduce the life to 5 seconds. However, one could also argue that the invocation rate limit is absolute: if an predicate executes less than n times per second, it is not worth optimizing.

– *Similarly, should the life time be measured in real time, CPU time, or some machine-specific unit.* Intuitively, using real time seems wrong, since interference from other tasks or from the user would influence compilation. Using CPU time has its own problems, too: for example, if most of the time is spent in garbage collection or compilation, the life time is effectively shortened since interpreted predicates get less time to execute and increase their invocation counters. On the other hand, this effect may be desirable: if not much time is spent in compiled Prolog code, optimizing that code will not increase performance by much.

4.3 What to Compile

When a counter overflows, the compilation system is invoked to decide which predicate to compile (if any). A simple strategy would be to always compile the predicate whose counter overflowed, since it obviously was invoked often. However, this strategy would not always work well. For example, suppose that the predicate overflowing its counter just returns a constant. Optimizing this clause would not gain much; rather, this clause should be inlined into its caller. In general, *to find a good candidate for compilation, we need to walk up the call chain and inspect the callers of the clause triggering the compilation.*

4.3.1 Overview of the Compilation Process

The figure 3 shows an overview of the compilation process. Starting with the predicate that overflowed its counter, the compilation system walks up the stack to find a "good" candidate for compilation. Once an predicate is found, the compiler is invoked to optimize it, and the interpreted version is discarded. If no predicates are found the interpreter continues the execution. During the optimizing compilation, the compiler marks the restart point (i.e., the point where execution will be resumed) and computes the contents of all live registers at that point. If this computation is successful, the optimized predicate replaces the corresponding interpreted predicate on the stack, possibly replacing several interpreted activation records with a single optimized activation record. Then, if the newly optimized predicate isn't at the top of the stack, compilation continues with then newly optimized clause's callee. In this way, the system optimizes an entire call chain from the top predicate down to the current execution point.

If the interpreted predicate cannot be replaced on the stack, they are left to finish their current activations, but subsequent invocations will use the new, optimized predicate. The main effect of failing to replace the interpreted predicate is that additional recompilations may occur if the interpreted code continues to execute for a while. For example, if the optimized predicate contains an loop but cannot be placed on the stack immediately, the optimization system may later try to replace just the loop body with optimized code.

4.3.2 Selecting the Predicate to be Compiled

The system selects the predicate to be compiled by examining several metrics. For any clause **c**, the following values are defined:

- **p**.*size* is the size of **p**'s instructions.
- **p**.*count* is the number of times **p** was invoked.
- **p**.*sends* is the number of calls directly made from **p**.

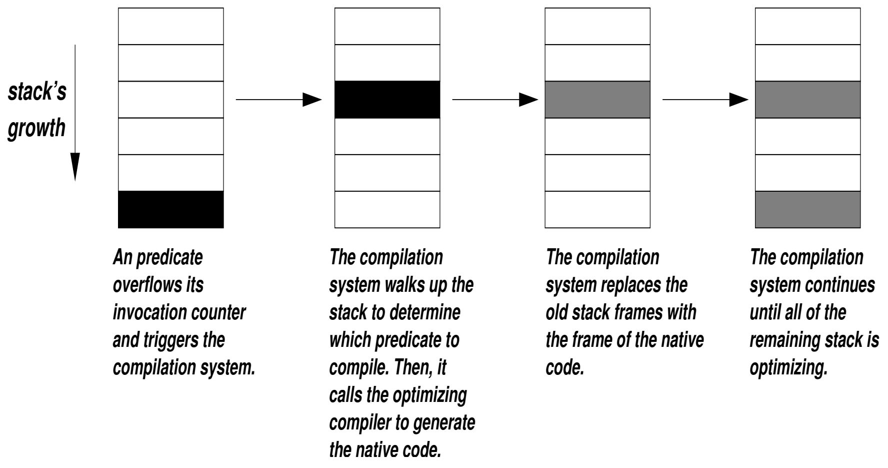
stack's growth
An predicate overflows its invocation counter and triggers the compilation system.
The compilation system walks up the stack to determine which predicate to compile. Then, it calls the optimizing compiler to generate the native code.
The compilation system replaces the old stack frames with the frame of the native code.
The compilation system continues until all of the remaining stack is optimizing.

Figure 3: The Compilation Process

– **p**.*version* records how many times **p** has been recompiled.

The search for a recompilee can be outlined as follows. Let `trip` be the predicate tripping its counter, and `recompilee` be the current candidate for compilation.

1. Start with `recompilee` = `trip`.
2. If `recompilee` has predicate in it body, choose the predicate's lexically enclosing predicate if it meets the conditions described above. This rule eliminates predicates by inlining the predicate used into the predicate's home. If inlining succeeds, the predicate can usually be optimized away completely.
3. Otherwise, choose `recompilee's` caller if it meets the conditions below. This rule will walk up the stack until encountering an predicate that is either too large or does not appear to cause many calls to be executed.
4. Repeat steps 2 and 3 until `recompilee` doesn't change anymore.

Whenever the compilation system considers a new `recompilee` **p**, it will only accept the new `recompilee` if it meets both of the following conditions:

– **p**.*count* $>$ *MinInvocations* and **p**.*version* $<$ *MaxVersion*. The first condition ensures that the predicate has been executed enough times to consider its type information. The second prevents endless compilation of the same predicate.

– $\mathbf{p}.\mathit{sends} > \mathit{MinSends}$ or $\mathbf{p}.\mathit{size} < \mathit{SizeLimit}$ or **p** is unoptimized. The first condition accepts predicates executing many calls, and the other two accept predicates that are likely to be combined with the callee through inlining.

The assumptions underlying these rules are that frequently executed predicates are worth optimizing, and that inlining small predicates will lead to faster execution. Although the rules are simple, they appear to work well in finding the "hot spots" of applications.

The rules used by the compilation system for finding a "good" compilation candidate in many aspects mirror the rules used by the compiler for choosing "good" inlining and executable path opportunities. For example, the rule skipping "tiny" predicates has an equivalent rule in the compiler that causes "tiny" predicates to be inlined. Ideally, the compilation system should consult the compiler before every decision to walk upwards on the stack (i.e., towards a caller) to make sure the compiler would inline that send. However, such a system is probably unrealistic: to make its inlining decisions, the compiler needs much more context, such as the overall size of the caller when combined with other inlining candidates. Therefore, compilation decisions in such a system would be expensive, and this approach was therefore rejected. However, the compilation system and the compiler do share a common structure in the our system. Essentially, the compiler's criteria for walking up the stack are a subset of the compiler's criteria for inlining.

After a compilation, the system also checks to see if compilation was effective, i.e., if it actually improved the code. If the previous and new compiled clauses have exactly the same non-inlined calls, compilation did not really gain anything, and thus the new clause is marked so it won't be considered for future compilations.

4.4 Which Executable Path

Programs often contain rarely or never executed paths. Compiling them can cause adverse effects that reduce the effectiveness of code generated. For example, on optimizing compiler can spend a lot of time applying aggressive optimizations in rarely executed code. The problem is that the times we implicitly assume procedures are the units for compilation. Procedure boundaries have been a convenient way to partition the process of compilation, but it are not necessarily a desirable unit to perform optimizations. If we can eliminate from the compilation target those portions that are rarely or never executed, we can focus the optimization efforts only on non-rare paths and this would make the optimization process both faster and more effective.

In a dynamic compilation system, this technique is especially useful. First, dynamic compilers can take advantage of runtime profile information from currently executing code and use this information for the region selection process.

Second, they are very sensitive to the compilation overhead, and this technique can significantly improve the total compilation time and code size. Third, they can avoid generating code outside the selected regions until the code is actually executed at runtime.

In our work, we design a region-based compilation technique. We no longer treat procedures as the unit of compilation, as in traditional procedure-based compilation, and select only those portions that are identified as non-rare paths. The term region refers to a new compilation unit, which results of the exclusion of all rarely executed portions of these procedures.

The key components for the our approach are region selection, and partial inlining. For region selection, we employ type feedback to identify executables paths. This process and inlining can affect each other, in the sense that inlining exposes another target for region selection, and the region selection process in turn conserves the inlining budget. Thus the inlining process can be performed for parts of a procedure, not for the entire body of the procedure.

For constructing the native code for one predicate, the optimizing compiler only performs inlining of the procedure's executable paths. Note that, besides the compiler performs inlining of the predicates used into the predicate's home, it also performs inlining of functions that performing unification,and dereferencing.

The emulator's complex code (figure 5) for executing the simple Prolog program (figure 4) that calculating an number of Fibonacci's serie, can be reduced to few hardware instructions. The optimizing compiler can reduce this control-flow graph as "what" compile has already decided and the type feedback informations are available. In our system this situation always occurs, because when the program is initially interpreted the type feedback repository is constructed. Before the compilation process to, the runtime system has already chose "what" compile. This way, that runtime system invokes the optimizing compiler, it can be provided with fresh informations. The optimizing compiler performs two steps for reducing and optimizing the control-flow graph. In the first step, the graph is reduced into the bold part. After this, the compiler performs some aggressive optimizations, resulting in highly optimized code.

4.5 Adding Type Feedback to the Runtime System

The main implementation problems of languages that supporting some form of late binding arise from the paucity of information statically available at compile time. That is, the exact meaning of some operations cannot be determined statically but is dependent on dynamic (i.e., runtime) information. Therefore, it is hard to optimize these late-bound operations statically, based on the program text alone.

There are two approaches to solving this problem. The first one, dynamic compilation [Plezbert and Cytron 1997] moves compilation to runtime where

```
fib ( 1, 0 ).
fib ( 2, 1 ).
fib ( X, Y ) :–
      X > 2,
      X1 is X – 1,
      fib (X1, Y1 ),
      X2 is X – 2,
      fib ( X2, Y2 ),
      Y is Y1 + Y2.
```

Figure 4: Fibonacci Prolog Code

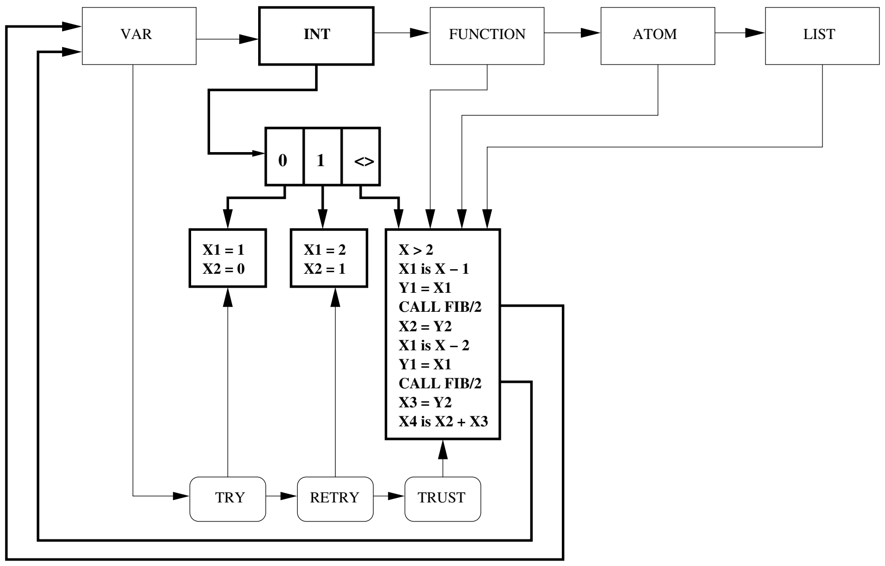

Figure 5: Control-flow Graph for fib(10,X)

additional information is available and can be used to better optimize the late bound operations. This is the approach taken by this work and several previous systems, SELF compiler [Chamblers 1992] [Holzle 1994], and Java compilers [MicroSystems 2003, Paleczny et al 2001, Ciernick et al 2000, Suganuma 2000].

Different of SELF compiler, our work initially interprets the code and only optimizes later, after it has become clear that the code is used often. It approach is the same of the Java compilers, but these not perform type feedback-based optimizations. Our approach makes it possible to generate better code than "eager" systems because the compiler has even more information available.

If it is not possible to move compilation to runtime, one can use the second, more conventional approach of moving the additional runtime information to the compiler. Typically, the information is collected in a separate run and written to a file, and the compiler is re-invoked using the additional information to generate the final optimized program.

Type feedback [Holzle and Ungar 1994, Agesen and Holzle 1995] works with either one of these approaches. For now, we will concentrate on the first approach. The key idea of type feedback is to extract type information from the runtime system and feed it back to the compiler. To obtain the type profile, the standard predicate dispatch mechanism has to be extended in some way to record the desired information, e.g., by keeping a table of receiver types per call site. Type feedback does not require techniques such as dynamic compilation or adaptive recompilation. If anything, these techniques make it harder to optimize programs: using dynamic compilation in an interactive system places high demands on compilation speed and space efficiency. For these reasons, our implementation of type feedback has to cope with incomplete information (i.e., partial type profiles and inexact invocation counts) and must refrain from performing some optimizations to achieve good compilation speed.

This technique is often added to a conventional batch-style compilation system. In such an ambicious system (figure 6), optimization would proceed in three phases. First, the executable is instrumented to record receiver types. Then, the application is run with one or more test inputs that are representative of the expected inputs for production use. Finally, the collected type and profiling information is fed back to the compiler to produce the final optimized code. In our dynamic system (figure 7), this process is divided into two phases. Initially, the code is interpreted and the interpreter collects runtime informations. The final phase is similar that using for conventional batch-style compilation system.

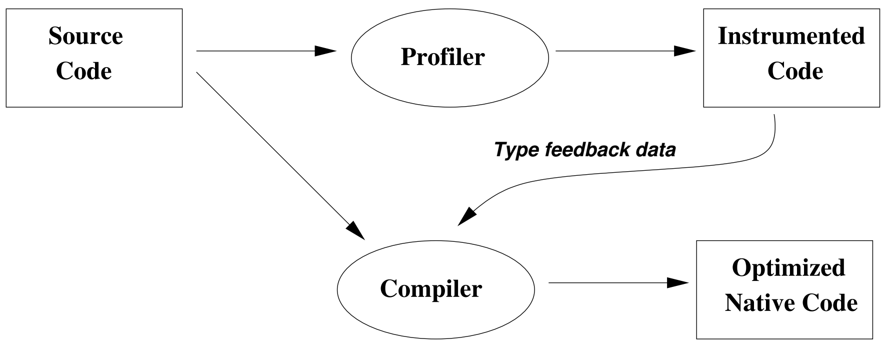

Figure 6: Type Feedback in a Statically Compiled System

Having obtained the program's type profile, this information is then fed back

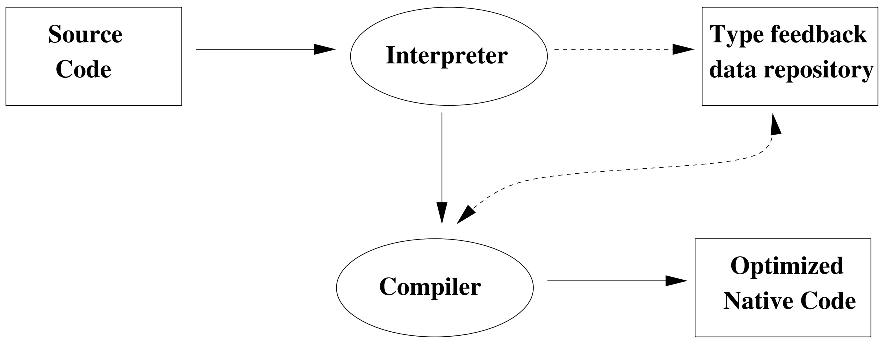

Figure 7: Type Feedback in our Dynamic System

into the compiler so it can optimize dynamically-dispatched calls by predicting likely receiver types and selecting executable paths for these types.

Our approach of dynamic compilation has the same advantage of static compilation. Both have complete information since optimization starts after at program execution. However, a dynamic compilation system has a significant advantage because it can dynamically adapt to changes in the program's behavior.

4.6 More Details about Fibonacci's Code

As said previously, when the runtime system invokes the optimizing compiler, it is provided with fresh informations. These informations will indicate that the third predicate should be compiled and optimized, because for one execution of Fib(10,X) the third predicate is the more frequent invoked one.

The complexity of the emulator's code and the high quality of the optimizing compiler' code can be seen in only one instruction. For executing the instruction **X1 is X - 1** of the reduced graph (bold part in figure 5), the code shown in figure 8 would be executed by emulator. However, the optimizing compiler will reduce this code into few instructions. The profiling gotten by the emulator indicates that the first argument is a integer number, with this information the compiler reduces this code in the code of figure 9.

After these steps, the compiler again reduces the generated code. As the content of **X** was identified as being an integer number, no more exist the necessity of creating an integer term. Also no more exist the necessity of the macro **Bind()**, for the cases where **X1** indicates an unbound variable. When this situation occurs, the compiler is able to generate the code of figure 10.

```
OUT is Exp() {
    if checkIfExpIsGround()
      Error()
    else
      Ti = Eval(Exp);
    unify(Ti, OUT);
}
```

```
Term Eval(Term Exp) {
    switch_on_type(Eval) {
      case List:
         return Eval(Head(Exp));
      case Number:
         return Exp;
      case Atom:
         return EvalAtom(Exp);
      case Function:
          if (Arity(Exp) == 1 )
            return UnaryEval(Exp);
          else
            return BinaryEval(Exp);
      else Error();
    }
}
```

```
Term BinaryEval(Term Exp) {
    T1 = Arg( 1, Exp );
    T2 = Arg( 2, Exp );
     . . .
}
```

Figure 8: Emulator' code for *X1 is X - 1*

```
if ( IsInt( X ) )
    Tint = MkTerm( IntFromTerm(X) –  1 );
else
    Goto_Emulator();
if ( IsVar ( X1 ) )
    BIND ( X1, Tint );
else
    if ( IsAtom ( X1 ) )
        if ( X != X1 )
            Falha();
```

Figure 9: Optimizing code for *X1 is X - 1*

```
Tint = X – 1
X1 = Tint
```

Figure 10: Highly Optimizing code for *X1 is X - 1*

5 Conclusions

Late binding is a problem for traditional systems because often the source code does not have enough information to generate the best compiled code. By delaying optimization until the necessary information is available, a compiler can generate better code. Delaying optimization has the additional benefit of reducing compilation pauses by confining costly optimization to the time-critical parts of the program.

Late binding also allow us to reconcile global optimizations with source-level debugging. The debugging information provided by the runtime system only needs to support reading the source-level program state at points where the program may be interrupted, not at every instruction boundary. Debugging requests that go beyond reading the current state are handled by transparently deoptimizing compiled code and performing the request on the interpreted code. Thus, the programmer is not forced to chose between speed and source-level debugging: programs can be debugged at any time.

Type feedback and dynamic compilation improve both runtime performance and interactive behavior. We believe that these techniques can be used to execute logic programs efficiently. Dynamic compilation is often regarded as complicated and hard to implement. We hope that our work shows that dynamic compilation can actually make the implementor's life easier. Once the underlying mechanisms are in place, new functionality based on dynamic compilation can be added relatively easily. For example, both the source-level debugging system and the handling of exceptional situations could be implemented with relatively little effort because the system already supported dynamic compilation. We believe that dynamic compilation can be an attractive choice for interactive development environments and runtime systems that using an emulator.

Several approaches has been implemented to support fully compiled execution of Prolog, and they have achieved good performance, but only for small applications. Unfortunately, similar results were not always forthcoming for logic programs well. Furthermore, runtime performance was achieved at the expense of considerable compiler complexity and compilation speeds that were too slow for interactive use.

This work has presented a new system that simultaneously improves execution and compilation speed. We go beyond the WAM for improving Prolog performance not only using the techniques described in section 3, but by using dynamic compilation too.

The progress made in implementing YAPc will encourage others to find even better solutions to the implementation challenges posed by logic languages, and to other systems also heavily using late binding. Our work will also contribute to make logic programming environments more popular, as in the past, such systems have often suffered from performance problems that have limited their acceptance.

References

[Agesen and Holzle 1995] Agesen, O., Holzle, U.: "Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages"; Proc. of the Conference on Object-Oriented Languages, 1995, 91-107.

[Ait-Kaci 1991] Ait-Kaci, H.: "Warren's Abstract Machine: An Tutorial Reconstruction"; MIT Press, 1991.

[Auslander et al 1996] Auslander, J., Philipose, M., Chambers, C., et al: "Fast, Effective Dynamic Compilation"; Proc. of the Conference on Programming Language Design and Implementation, 1996, 149-159.

[Bacon et al 1994] Bacon, D. F., Graham, S. L., Sharp, O.J.: "Compiler Transformations for High-Performance Computing"; ACM Computing Surveys, 26 4(1994), 345-420.

[Beer 1998] Beer, J.: "The Occur-Check Problem Revisited"; Journal of Logic Programming, 5, 3(1998), 243-261.

[Chamblers 1992] Chamblers, C.: "The Design and Implementation os SELF Compiler: an Optimizing Compiler for Object Programming Languages"; PhD thesis, Stanford University.

[Ciernick et al 2000] Cierniak, M., Lueh, G. Y., Sitchmoth, J. M.: "Practicing JUDO: Java Under Dynamic Optimizations"; Proc. of the Conference on Programming Language Design and Implementation, 2000, 13-26.

[Click and Paleczny 1995] Click, C., Paleczny, M.: "A Simple Graph-Based Intermeadiate Representation"; Proc. of the Workshop on Intermediate Representations, 1995, 35-49.

[Colmerauer 1993] Colmerauer, A.: "The Birth of Prolog"; Proc. of the Second History of Programming Languages Conference, 1993, 37-52.

[Cousot 1992] Cousot, P., Cousot, R.: "Abstract Interpretation and Applicaton to Logic Programs"; Journal of Logic Programming, 13, 2(1992), 103-179.

[Cytron et al 1991] Cytron, R., Ferrante, J., et al.: "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph"; ACM Transactions on Programming Languages and Systems, 13, 2(1991), 451-490.

[da Silva 2005] da Silva, A., Costa, V. S.: "An Experimental Evaluation of Java JIT Technology"; Proc. of the Brazilian Symposium on Programming Languages, 2005, 25-40.

[Diaz and Codognet 2001] Diaz, D., Codognet, P.: "Design and Implementation of the GNU Prolog System"; Journal of Funcional and Logic Programming, 13, 4(2001), 451-490.

[Goodman and Hsu 1988] Goodman, J. R., Hsu, W. C.: "Code Scheduling and Register Allocation in Large Basic Blocks"; Proc. of the International Conference on Supercomputing, 1988, 442-452.

[Holzle 1994] Holzle, U.: "Adaptative Optimization for SELF: Reconciling High Performance with Exploratory Programming"; PhD Thesis, Stanford University.

[Holzle and Ungar 1994] Holze, U., Ungar, D.: "Optimizing Dunamically-Dispatched Calls with Runtime Type Feedback"; Proc. of the conference on Programming Language Design and Implementation, 1994, 326-336.

[Knoop et al 1994a] Knoop, J., Rthing, O., Steffen, B.: "Optimal code Motion: Theory and Practice"; ACM Transactions on Programming Language and Systems, 16, 4(1994), 1117-1155.

[Knoop et al 1994b] Knoop, J., Rthing, O., Steffen, B.: "Partial Dead Code Elimination"; Proc. of the Conference on Programming Language Design and Implementation, 1994, 147-158.

[Komatsu et al 1986] Komatsu, H., Tamura, N., et al: " An Optimizing Prolog Compiler"; Proc. of the Logic Programming'86, 1986, 104-115.

[Marien 1988] Marien, A.: “An Optimal Intermediate Code for Structure Creation in a WAM-based Prolog Implementation”; Technical Report T 1988:01, Katholicke Universiteit Leuven.

[Marien and Demoen 1989] Marien, A., Demoen, B.: “ On the Management of Choicepoint and Environment Frames in the WAM”; Proc. of the North American Conference on Logic Programming, 1998, 1030-1047.

[Marien 1993] Marien, A.: “Improving the Compilation of Prolog in the Framework of the Warren Abstract Machine”; PhD thesis, Katholicke Universiteit Leuven.

[Meier 1990] Meier, M.: “Compilation of Compound Terms in Prolog”; Proc. of the North American Conference on Logic Programming, 1990, 63-79.

[Meier et al 1989] Meier, M., Aggoun, A., et al: “SEPIA - an Extendible Prolog System”; Proc. of the World Computer Congress, 1989, 1127-1132.

[MicroSystems 2003] Sun MicroSystems.: “The Java HotSpot Virtual Machien”; Technical Report, Sun Developer Network Community, 2003.

[Morales et al 2004] Morales, J., Carro, M., Hermenegildo, M. V.: ”Improved Compilation of Prolog to C Using Moded Types and Determinism Information”; Proc. of the International Symposium of Practical Aspects of Declarative Languages, 2004, 86-103.

[Muchnick 1997] Muchnick, S. S.: “Advanced Compiler Design and Implementation”; Morgan Kaufmann, 1997.

[Nassen 2001] Nassen, H.: “Optimizing the SICStus Prolog Virtual Machine Instruction Set”; Technical Report T2000:01, Intelligent Systens Laboratory, Uppsala University, 2001.

[Paleczny et al 2001] Paleczny, M., Vich, C., Click, C.: “ The Java Hotspot Server Compiler”; Proc. of the Java Virtual Machine Research and Technology Symposium, 2001, 120-131.

[Plezbert and Cytron 1997] Plezbert, M. P., Cytron, R. K.: “Does Just-In-Time = Better Late Than Never?”; Proc. of the Symposium on Principles of Programming Language, 1997, 120-131.

[Poletto and Sarkar 1999] oletto99 Poletto, M., Sarkar, V.: “ Linear Scan Register Allocation”; ACM Transactions on Programming Languages and Systems, 21, 5(1999), 895-913.

[Roy 1989] Roy, P. V.: “An Intermediate Language to Support Prolog’s Unification”; Proc. of the North American Conference on Logic Programming, 1989, 1148-1164.

[Roy 1990] Roy, P. V.: “Can Logic Programming Execute as Fast as Imperative Programming?”; PhD Thesis.

[Roy and Despain 1992] Roy, P. V., Despain, A.: “High Performance Logic Programming with the Aquarius Prolog Compiler”; IEEE Computer Maganizem 39, 1(1992), 54-68.

[SICStus 2006] Swedish Institute of Computer Sience. http://www.sics.se/isl/sicstus, accessed in January 12, 2006.

[Sterling and Shapiro 1986] Sterling, L., Shapiro, E.: “ The Art of Prolog”, MIT Press, 1996.

[Suganuma 2000] Suganuma, T., et al.: “Overview of the IBM Java Justin-Time Compiler”; IBM System Journal, 39, 1(2000), 66-76.

[Tamura 1986] Tamura, N.: “Knowledge-Based Optimization in Prolog Compiler”; Proc. of the Computer Society Fall Joint Conference, 1986.

[Taylor 1991] Taylor, A.: “High-Performance Prolog Implementation”; PhD thesis, Basser Department of Computer Science, University of Sydney.

[Turk 1986] Turk, A. K.: “Compiler Optimizations for the WAM”; Proc. of the Internationcal Conference on Logic Programming, 1986, 657-662.

[Warren 1983] Warren, D.: “An Abstract Prolog Instruction Set”; Technical Report 390, SRI International Artificial Intelligente Center, 1983.

[Warren et al 1988] Warren, D., Hermenegildo, M., Debray, S. K.: "On the Practicality of Global Flow Analysis of Logic Programs"; Proc. of the International Conference and Symposium on Logic Programming, 1988,684-699.

[Wegman and Zadeck 1991] Wegman, M. N., Zadeck, F. K.: "Constant Propagation with Conditional Branches"; ACM Transactions on Programming Languages and Systems, 13, 2(1991) 181-210.

[Wikipedia 2006] First Order Predicate Calculus. http://en.wikipedia.org/wiki/First-order-predicate-calculus, accessed in January 12, 2006.

[YAP 2006] The YAP Prolog System. http://www.ncc.up.pt/vsc/Yap, accessed in January 12, 2006.

[Zhao and Amaral 2003] Zhao, P., Amaral, J. N.: "To Inline or Not to Inline? Enhanced Inlining Decisions"; Proc. of the Workshop on Languages and Compilers for Parallel Computing, 2003, 11-23.