INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Experiments in Program Compilation by Interpreter Specialization

Scott Thibault, Laurent Bercot, Charles Consel, Renaud Marlet, Gilles Muller,

Julia Lawall

N° 3588

Décembre 1998

THÈME 2

Rapport de recherche

ISSN 0249-6399

Experiments in Program Compilation by Interpreter Specialization

Scott Thibault, Laurent Bercot, Charles Consel, Renaud Marlet, Gilles Muller, Julia Lawall

Thème 2 — Génie logiciel
et calcul symbolique
Projet COMPOSE

Rapport de recherche n 3588 — Décembre 1998 — 20 pages

Abstract: Interpretation and run-time compilation techniques are becoming increasingly important due to the need to support heterogeneous architectures, evolving programming languages, and dynamically downloaded code. Although interpreters are easy to write and maintain, they are inefficient. On the other hand, run-time compilation provides efficient execution, but is costly to implement. One way to get the best of both approaches is to apply program specialization to an interpreter in order to generate an efficient implementation automatically.

Recent advances in program specialization technology have resulted in important improvements in the performance of specialized interpreters. This paper presents and assesses experimental results for the application of program specialization to both bytecode and structured-language interpreters. The results show that for general-purpose bytecode, program specialization can yield speedups of up to a factor of four, while specializing certain structured-language interpreters can yield performance equivalent to code compiled by a general-purpose compiler.

Key-words: Specialization, interpreters, Java, Caml, domain-specific languages

(Résumé : tsvp)

This research is supported in part by France Telecom/CNET

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Compilation de programmes par spécialisation d'interprètes

Résumé : Les techniques d'interprétation et de compilation à la volée sont d'un intérêt croissant en raison de l'hétérogénéité des architectures matérielles, de l'évolution des langages de programmation et du besoin de chargement dynamique de code. L'avantage majeur des interprètes réside dans leur facilité de conception et de maintenance. Cependant, ils sont relativement inefficaces. Par ailleurs, si les compilateurs à la volée permettent une exécution efficace, ils sont longs et complexes à implémenter. Dans cet article, nous nous intéressons à une approche différente consistant à utiliser la spécialisation de programmes afin de générer automatiquement un compilateur à partir d'un interprète.

Les progrès récents de la technologie de spécialisation de programmes ont entraîné une augmentation de la performance des interprètes spécialisés. Nous présentons ici des résultats expérimentaux sur la spécialisation d'interprètes de code intermédiaire (bytecode) et d'interprètes pour langages structurés. Nos résultats montrent que sur des interprètes de code intermédiaire, la spécialisation apporte un gain en vitesse d'exécution allant jusqu'à 4. Sur des interprètes pour langages structurés, le code spécialisé peut être aussi performant qu'un programme compilé par un compilateur traditionnel.

Mots-clé : spécialisation, interprètes, Java, Caml, langages dédiés

1 Introduction

Modern computing environments are characterized by heterogeneous architectures, evolving programming languages, and dynamically loaded code. Traditional compilers, which perform complex, machine-specific optimizations, are not well-suited to such environments. These problems have stimulated renewed interest in interpretation as a realistic language implementation technique. Interpreters provide portability, ease of modification, and rapid deployment of dynamically-loaded code. Nevertheless, interpretation carries a significant performance penalty. One solution is to generate compilers from interpreters (*e.g.,* [19, 34]). Various techniques have been proposed to achieve this goal:

Run-time code generation. Run-time code generation languages, such as VCODE [11] and 'C [12], provide a high-level notation in which to write programs that generate executable code at run time. This approach has been successfully used in networks for generating highly efficient filters [13]. Similarly, an interpreter written in a language that supports run-time code generation can be modified to generate code rather than executing it. Nevertheless, the approach is error-prone since little or no correctness verification of the code generation process is performed.

Ad-hoc bytecode interpreter optimization. Piumarta and Riccardi have proposed to improve the performance of bytecode programs by selective inlining, *i.e.,* replacing each bytecode by the corresponding fragment of the compiled code of the interpreter [31]. This approach is effective, obtaining up to 70% of the performance of C, and safe, because the translator is specified in terms of the original interpreter. The main limitation of this technique is that no run-time optimization can be performed. For instance, this approach prevents the replacement of ordinary bytecode instructions by `_quick` bytecode instructions. The use of `_quick` bytecode instructions is known to be one of the main sources of efficiency of the Java virtual machine [14].

Directive-driven specialization. Specialization permits the optimization of a program by precomputing expressions that depend only on early known inputs. Using the directive-driven specialization approach [3], the programmer has to instrument the original interpreter with annotations that help drive the specializer. While this approach automates the process of code generation, the correctness of specialization depends on the annotations. Thus, it is still error prone.

Automatic specialization. Automatic specialization [19, 5] replaces the manual annotations of directive-driven specialization by automatically inferred annotations based only on a description of the known inputs. This approach to generating a compiler is more reliable, because it requires little or no modification of the interpreter. While compiling programs by specializing interpreters has been extensively explored, these studies have been done in the context of language subsets, using restricted implementation languages (*e.g.,* pure functional languages) [8, 18, 20].

Recently, major advances in automatic program specialization have been achieved that allow the design of program specializers for industrial languages such as C. Examples of such specializers are C-Mix [1], developed at DIKU, and Tempo, developed at IRISA. In this paper, we use Tempo for all our specializations. Tempo specializes C programs both at compile time and run time [9]. It has been successfully used for operating systems as well as scientific programming.

The ability to specialize C programs opens up many new opportunities for compiling programs from interpreters. This paper presents new results in compiling programs by specializing interpreters for *realistic* languages ranging from Objective Caml to a language for active network protocols, PLAN-P [36]. We report very promising performance obtained for the compiled programs.

In this paper we evaluate the results of the following recent advances in program specialization in terms of the performance of specialized interpreters.

Off-line and on-line compilation. Tempo enables programs to be specialized both at compile time and run time. Consequently, by specializing interpreters, not only can programs be compiled off-line but they can also be compiled efficiently on-line, thus achieving *Just-in-time* compilation.

Realistic languages. Because Tempo can process real-sized programs, interpreters for realistic languages can be treated. We have specialized interpreters for abstract machines (O'Caml, JVM, Berkeley Packet Filter) and interpreters for high-level languages such as a language for graphic device drivers, GAL [38], and PLAN-P.

When the interpreter is written in an efficient language (*i.e.,* C), specializing the interpreter with respect to a program generates an efficient implementation. Concretely, we show that for languages dedicated to a domain such as GAL and PLAN-P, the performance of compiled programs is similar to and sometimes better than the performance of equivalent compiled programs written in a general-purpose language. Additionally, we show that specializing bytecode interpreters for general-purpose languages can achieve speed increases of up to a factor of four.

These new results clearly demonstrate that program specialization can be a key tool in language development and implementation for an important emerging class of languages, domain-specific languages and bytecodes for portable/mobile applications.

The rest of the paper is organized as follows. Section 2 gives a short overview of the Tempo specializer. Section 3 presents experiments in specializing bytecode interpreters. Section 4 describes results of the specialization of structured code interpreters. We conclude in Section 5 by assessing the perspectives offered by interpreter specialization.

2 A Specializer for C programs: Tempo

Tempo is an off-line specializer for C programs[1] [6]. An off-line specializer is divided into two phases: analysis and specialization. The input to the analysis phase consists of a program and a description of which inputs will be known during specialization. Based on this information, the analysis phase produces an annotated program, indicating how each program construct should be transformed during specialization. Because C is an imperative language including pointers, the analysis phase performs alias, side-effect, and dependency analyses. The accuracy of these analyses is targeted towards keeping track of known values across procedures, data structures, and pointers [16, 17]. Following the analysis phase, the specialization phase generates a specialized program based on the annotated program and the values of the known inputs.

Tempo can uniformly perform compile-time and run-time specialization [7]. Although compile-time is traditional, run-time specialization opens up new opportunities, since programs can now be specialized with respect to values not known until run time. In the run-time case, Tempo generates a dedicated run-time specializer, and binary templates that represent the building blocks of all possible specialized programs. At run time, the specializer performs the computations that rely on the actual input values, selects binary templates and instantiates them with computed values [9, 29]. In the context of interpreters, this functionality makes it possible to remove the interpretation layer at run time, thus achieving a form of Just-In-Time compilation.

Tempo has been successfully used for a variety of applications such as operating systems (Sun Remote Procedure Call — RPC [27, 25], Chorus Inter-Process Communication — IPC [39]) and scientific programs (*e.g.,* convolution filters, FFT [29]). These applications have demonstrated two key features of Tempo: (1) it can process realistic programs that have not carefully been crafted for specialization (2) it can generate highly-optimized programs (*e.g.,* the specialized layer of the Sun RPC runs 3.5 times faster than the original one).

3 Bytecode Interpreters

We first examine the specialization of bytecode (flat, linearized code) interpreters. Typically, bytecode instructions correspond closely to machine instructions without being tied to a particular architecture. Thus, bytecode is often used in the context of a virtual machine. Because bytecode interpreters provide both dynamic program loading and heterogeneity, they are increasingly used in operating systems and embedded systems. Having the ability to generate efficient compiled code for various platforms from an existing interpreter is thus a promising technique. We investigate the specialization of three bytecode interpreters: the Java Virtual Machine, Objective Caml, and the Berkeley Packet Filter. We first examine issues common to most bytecode interpreters, and then consider the interpreters individually.

[1] Tempo is publicly available at `http://www.irisa.fr/compose/tempo`.

3.1 Specialization of bytecode interpreters

All bytecode interpreters have a similar structure, as illustrated by the fragment of a bytecode interpreter displayed in Figure 1-a. The inputs to a bytecode interpreter are typically the bytecode program and a stack. Specialization should fully eliminate the dispatch depending on the bytecode program, producing a specialized program that only manipulates the stack. The specialized program is thus essentially a concatenation of the implementations of the program instructions. So that the dispatch on the program can be fully eliminated by specialization, the fundamental problem is to ensure that the program counter's value is statically known at every program point in the interpreter.

```
Val execVM(Prog pg, Stack sp)
{
  Index pc = 0;
  while(TRUE)
  {
    switch(pg[pc])
    {
      RETURN:
        return sp[0];
      ...
      IFEQ:
        if(sp[0] == 0)
          pc += get_target(pc);
        else
          pc += NEXT;
      ...
    }
  }
```

a: Original interpreter

```
Val execVM(Prog pg, Stack sp, Index pc)
{
  while(TRUE)
  {
    switch(pg[pc])
    {
      RETURN:
        return sp[0];
      ...

      IFEQ:
        if(sp[0] == 0)
          return execVM(pg, sp, pc+get_target(pc));
        else
          return execVM(pg, sp, pc+NEXT);
      ...
    }
  }
```

b: Interpreter based on recursive call

Figure 1: Fragment of bytecode interpreter (known constructs are underlined)

When there is a conditional branch in the bytecode, the choice of which branch to take can depend on the values of the inputs to the bytecode program, which are not known during specialization. Thus, the specializer cannot determine whether the program counter will be assigned the address of the true branch or the false branch and its value cannot be statically known at the point after the conditional. This situation is illustrated by the interpretation of the `IFEQ` instruction in Figure 1-a. Since the value of the program counter is unknown after the conditional, it cannot be known for the next iteration, and thus, all references to the program counter within the loop are considered to be unknown, and no specialization occurs.

Within the branches of the `if`-statement implementing a conditional branch instruction, the value of the program counter is still known. It does not become unknown until after the `if`-statement, when the analysis merges the two possibilities. Thus one approach to solve the problem is for the specializer to systematically duplicate all of the code that is executed after every `if`-statement (including the subsequent iterations of the loop) within the two branches of the `if`-statement. This approach is known as *continuation-passing style specialization* [1, 4]. While this approach avoids the need to manually alter the source program, unconstrained continuation-passing style specialization can lead to code explosion.

The approach we take instead is to manually duplicate the continuation of an `if`-statement in the branches of the `if`-statement only when the duplication is necessary to allow the program counter to remain known. Code explicitly following the `if`-statement can be simply copied into the two branches. Subsequent iterations of the loop are modeled by first extracting the loop into a separate procedure, and then making a recursive call. Finally, each branch ends with a return, so that the value of the program counter does not affect the rest of the while loop. The result of this translation is illustrated in Figure 1-b.

3.2 The Java Virtual Machine (JVM)

Java bytecode is a perfect target for specialization: its execution relies on a bytecode interpreter; alternatively, native code can be run using on-line (JIT) or off-line compilers. For our specialization experiment, we have targeted the Harissa system [26]. Harissa is a flexible environment for Java execution that permits mixing both compiled and interpreted code. Harissa's compiler (Hac) is one of the most efficient compilers for Java, while the interpreter (Hi) is slightly faster than the Sun JDK 1.0.2 interpreter. Hi is a 1000-line hand-optimized C program and thus demonstrates the power of Tempo's analyses and transformations.

3.2.1 Applying Specialization

In addition to bytecode inlining, Java bytecode interpreters offer several opportunities for specialization that are specific to dynamic loading of classes. As suggested by Sun [14], certain instructions can be optimized by rewriting them into others (prefixed by `_quick`) which take advantage of information available at run time. This situation typically occurs when instructions refer to an element of the constant pool and need to dynamically resolve its entry. Subsequent invocations of the same instructions need not resolve the entry again. Interestingly, in terms of program specialization, a quick instruction is simply a specialized version of the generic instruction: it is specialized with respect to the constant pool and a specific entry.

3.2.2 Performance

We evaluate the performance of the specialized Java bytecode interpreter using Caffeine 3.0 benchmarks [35]. Each Caffeine micro-benchmark tests one feature of the Java machine, and produces numbers, in CaffeineMarks (higher is faster), that allow one to compare heterogeneous architectures and Java implementations directly. Among them, we consider three tests (Loop, Sieve, Float) that are included in the "embedded" test suite. The other tests are not relevant for this experiment since they measure the efficiency of features of the JVM such as graphics or memory allocation that are not related to compilation.

	JDK 1.0.2	JDK 1.1.6	Hi	Hi (no quick inst)	Kaffe	Run-time spec.	Hac	Compile-time spec.
Sieve	90	191	127	16	479	242	1590	398
Loop	85	155	122	11	1119	302	4780	496
Float	103	219	126	16	1110	-	1980	454

Table 1: Results of the Caffeine 3.0 Java benchmark (in CaffeineMarks)

Tests were performed on a Sun Ultra-1/170Mhz by comparing three interpreters (JDK 1.0.2, JDK 1.1.6, Hi), a public domain JIT compiler (Kaffe), and the Harissa compiler (Hac). The results are shown in Table 1. Due to many manual optimizations implemented by Sun, the JDK 1.1.6 interpreter is about twice as fast as the older JDK 1.0.2 version. As expected, Hi performs better than JDK 1.0.2. Interestingly, disabling `_quick` instructions within Hi slows down the interpreter by a factor of ten. By specializing Hi with `_quick` instructions disabled, we get an average speed-up of 32 for compile-time specialization and 21 for run-time specialization. This speedup includes automatic specialization of program counter elimination and of the generic instructions whose result is equivalent to `_quick` instructions. On the other hand, optimized classic and JIT compilers are still much faster by a factor of 2 to 10. We elaborate on the reasons for this gap in section 3.5. Nevertheless, the specialized code is up to four times faster than the interpreted code.

3.3 The Objective Caml Abstract Machine

Our second bytecode interpreter is the Objective Caml (O'Caml) abstract machine. The O'Caml bytecode is significantly different than the JVM in that it is the target of a functional language. For example, the O'Caml bytecode implements closures to handle higher-order functions.

In PLDI '98, Piumarta and Riccardi used the same O'Caml bytecode interpreter to demonstrate how selective inlining can optimize direct threaded code [31]. We obtain performance comparable to the results obtained with their inlining technique. However, unlike selective inlining, specialization is a general tool that can be applied to a larger class of applications. For example, selective inlining would not work with the JVM quick instructions, whereas using Tempo we get the functionality of quick instructions without having to implement them explicitly.

3.3.1 Applying Specialization

As for bytecode interpreters in general, the goal of specializing the O'Caml bytecode interpreter is to eliminate instruction decoding and dispatch. However, because the O'Caml bytecode supports higher-order functions, the program counter is not exactly statically known.

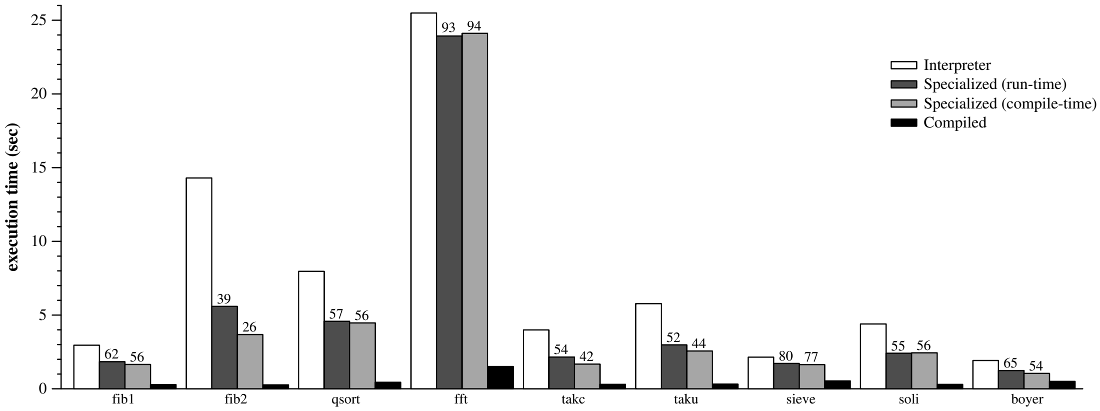

Figure 2: Results on O'Caml benchmark suite

When a called function is the value of an arbitrary expression, it is not possible to determine the address of the entry point of the called function based on the bytecode program alone. Nevertheless, although the number of different closures that can be created during the execution of a program is potentially unbounded, the set of code fragments associated with these closures is bounded by the number of closure-creating instructions (`closure` and `closurerec`) in the bytecode program. Thus we simply specialize the interpreter with respect to all of the possible code fragments, and store the specialized code in a table, indexed by the address of each unspecialized fragment. At run-time, a function call is implemented by using the code pointer of the invoked closure to extract the specialized definition from this table.

3.3.2 Performance

To measure the performance of the specialized interpreter, we used a standard O'Caml benchmark suite.[2] The results of these benchmarks are shown in Figure 2 for four versions: interpreted using the standard optimized interpreter, compiled by specializing the interpreter at run time, compiled by specializing the interpreter at compile time, and compiled using the standard native code compiler. The heights of the bars represent the relative run times. The numbers on the bars for the specialized interpreters indicate the run time as a percentage of the run time of the interpreted code. In the graph, fib1 represents the recursive fib in the standard benchmarks and fib2 represents an iterative version not in the original benchmark suite. All measurements were taken on a Sun Ultra-1/170Mhz.

In all of these benchmarks, run-time specialization achieves results that are equivalent to or slightly better than the results reported for the selective inlining technique [31]. It is not

[2] Available at `ftp://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/benchmarks/objcaml.tar.gz`

surprising that the specialized version is not significantly faster than the inlining approach because the O'Caml bytecode is already optimized with many specialized instructions. Thus, the ability to evaluate some of the instruction calculations is not needed and specialization only selects and inlines the instructions (dispatch elimination).

3.4 The Berkeley Packet Filter (BPF)

A packet filter is a piece of code that is used to identify network packets belonging to a given application. Packet filters are written using a dedicated bytecode language. They are loaded into the kernel where they are traditionally interpreted at the expense of high computational cost [24]. The Berkeley Packet Filter [23] is considered as a reference implementation for many optimization techniques [13, 28].

3.4.1 Applying Specialization

```
/* Load 32 bit value */
case BPF_LD|BPF_W|BPF_ABS:
  k = pc->k;
  if (k + sizeof(int32) > buflen)
    return 0;

  A=((u_int32)*((u_char *)p+k+0)<<24|
     (u_int32)*((u_char *)p+k+1)<<16|
     (u_int32)*((u_char *)p+k+2)<<8|
     (u_int32)*((u_char *)p+k+3)<<0);
  continue;
```

a: Original interpreter

```
/* Load 32 bit value */
case BPF_LD|BPF_W|BPF_ABS:
  k = pc->k;
  if (k + sizeof(int32) > buflen)
    return 0;

  /* p is always aligned. */
  if (((p+k)&0x3)==0)
    A=*((u_int32 *)(p+k));
  else
    A=((u_int32)*((u_char *)p+k+0)<<24|
       (u_int32)*((u_char *)p+k+1)<<16|
       (u_int32)*((u_char *)p+k+2)<<8|
       (u_int32)*((u_char *)p+k+3)<<0);
  continue;
```

b: Modified for specialization

Figure 3: Fragment of BPF interpreter

As for the other bytecode languages, specializing the BPF interpreter eliminates instruction dispatch. Here, we also take advantage of the fact that the interpreter will be specialized by coding optimizations in the interpreter. Figure 3-a shows the original interpreter code for a packet load instructions. This instruction loads the 32 bit value stored at a fixed offset from the beginning of the packet. Due to alignment requirements on the SPARC, these load instructions access the values one byte at a time, in case the address is not aligned. Figure 3-b shows an implementation of these instructions that is intended to be specialized. This version chooses between two implementations for each instruction. If the address is aligned the value is loaded all at once, otherwise the value is loaded one byte at a time. While this would make the original interpreter slower, it results in faster specialized programs because

	Interpreter	CT Spec.	CT Speedup	RT Spec.	RT Speedup
Pentium	3.32	0.98	3.40	1.95	1.7
Sparc (original)	2.62	0.68	3.89	1.57	1.67
Sparc (modified)		0.40	6.56	1.31	2.01

Table 2: BPF benchmarks

the condition of the added `if` statement is known statically and evaluated at specialization time.

3.4.2 Performance

Table 2 shows the execution time for a simple filter program applied to 5000 packets. Results are given for the Pentium and for the Sparc, with and without the optimization for aligned loads. For the Sparc version with alignment optimization, the speedups are relative to the original interpreter, since the modified interpreter is slowed down by an optimization that one would never implement for ordinary interpretation.

3.5 Discussion

Section 3 presents performance results for both run-time and compile-time specialization of three different kinds of bytecode interpreters. While the speedup obtained by specialization is significant, it does not compete with results obtained with hand written off-line or run-time compilers. There are two main reasons for this difference: bytecodes often already contain specialized instructions, and compilers typically perform stack elimination.

Both the JVM and O'Caml bytecode include many specialized instructions to improve performance. For example, both bytecodes include an instruction of the form `LOAD n`, which loads the n^{th} item on the stack. However, for small values of n, the bytecodes also define instructions with a fixed value for n, *i.e.*, `LOAD_0, LOAD_1, LOAD_2`, etc. In the JVM there are, additionally, the *quick* instructions which are specialized versions of the generic instructions.

Additional speedups using compilers are due to transformations like stack elimination. To determine the cause of the difference between the specialized O'Caml bytecode interpreter and the native compiler, we have measured the effects of various transformations on the results of a compile-time specialization. We performed three main transformations, by hand, on a compile-time specialized version of the fib2 program. Since the interpreters are written with recursive calls, as described in Section 3, the specialized code is a set of recursive functions. Thus, the first transformation is to merge the specialized functions into a single function. The second transformation is to make the interpreter's state variables local to the resulting function. Since normally the specialized code is a set of functions, the interpreter's state variables must be made global to be shared by all the functions. As a result the compiler does not perform register allocation or any optimizations on these variables. Finally, stack elements are converted into local variables. These permits register allocation and eliminates many memory references.

After applying these three transformations, the resulting program is almost identical to the program generated by the native O'Caml compiler. The only significant difference remaining is due to the fact that the O'Caml compiler uses Unix signals to implement signals, whereas the bytecode interpreter performs frequent checks to poll for pending signals.

4 Structured Code Interpreters

Interpreters for structured code are very different from interpreters for bytecode. Most importantly, structured languages are higher level in that they offer syntactic constructs and declarations. In terms of compilation, and thus specialization, this translates into more actions that can be performed at compile time than in the case of bytecode programs. As a result, compiled programs can run 10 to 100 times faster than interpreted programs.

The motivation in developing interpreters for structured languages is not to compete with compilers for general-purpose languages (*e.g.,* gcc). Indeed, general-purpose languages are stable in their design, and, if they become industry standards, high-quality compilers are available for many different platforms. In this context, aiming at compiling code by specialization raises little interest.

Beyond general-purpose languages, an emerging trend consists of developing languages specific to a particular domain (or family of problems). This approach is actively studied in both academia [10] and industry [2, 21, 22]. These languages, called *Domain-Specific Languages* (DSL), consist of notations, abstractions, and values that are specific to the kind of problems they are targeted for. DSLs require prototyping in the development phase, and extensibility to address future needs. In fact, these are well-known features offered by a language implementation based on an interpreter.

Both languages studied below are domain-specific languages: PLAN-P is a language aimed at developing network application protocols and GAL is a language for specifying video card device drivers. As shown by the performance of the programs compiled by specialization, DSLs can compete with equivalent programs hand-written in general-purpose languages, and sometimes even run faster. This efficiency is due to the fact that a DSL often amounts to a glue language: it combines building blocks. This interpretation layer can be systematically removed by specialization. Furthermore, the building blocks, written in a general-purpose language, can be efficiently compiled using a traditional hand-crafted compiler. Finally, because a DSL is restricted, it may rely on operations that are simpler and faster than those used for a high-level general-purpose language. This last observation makes it possible for specialized interpreters to be faster than native code produced by a compiler for a general-purpose language such as Java.

4.1 PLAN-P

PLAN-P[3] allows the programmer to define protocols that manipulate packets associated with a specific application [36, 37]. Because the network is a shared resource, each router needs to verify that downloaded PLAN-P programs satisfy its safety and security constraints. Furthermore, a network is often heterogeneous. Thus, to facilitate verification and allow portability, PLAN-P programs are downloaded as source code. Because new applications may be deployed on the network at any time, PLAN-P programs must be downloaded and checked dynamically. In this context, traditional off-line compilation would be too time-consuming. Thus, PLAN-P can either be interpreted or compiled using a JIT.

The PLAN-P language is originally based on PLAN, a Programming Language for Active Networks [15], which is dedicated to network diagnostics. While PLAN-P retains most of the SML-like syntax of PLAN, the semantics is significantly different in order to treat a larger scope of applications such as the adaptation of distributed applications and services [37]. While PLAN is interpreted, our PLAN-P interpreter is specialized at run-time using Tempo, thus achieving the same functionality as a JIT. Our previous experiments have shown that PLAN-P protocols can be as efficient as the equivalent hand-crafted C version.

Performance

We evaluate the performance of PLAN-P on a performance-demanding application, a learning bridge. A bridge is a network node that is connected between multiple LANs to form one logical LAN. A learning bridge keeps track of where packets come from in order to determine which LAN a host is connected to, so that packets for that host are only repeated on the LAN it is on. The learning algorithm uses a hash table to record the source address and the LAN of received packets, thus learning which host is on which LAN rather than repeating packets on all LANs.

We have done two types of benchmarks: (i) micro-benchmarks that measure the pure computation time without including the run-time system, (ii) a real benchmark that measures the throughput of the system. The interest of micro-benchmarks is to evaluate the comparative performance of specialization *vs* compilation, while real benchmarks measure the impact of specialization on the real system taking into account input/output, cache accesses, etc.

	Embedded C	PLAN-P specialized at run time	Java compiled with hac	PLAN-P interpreted
Sun Ultra-1 170Mhz	3	14	19	1218
PC/Pentium Pro 200 Mhz	4	182	18	440

Table 3: Ethernet bridge micro-benchmark (times in micro-seconds)

[3] A prototype of the PLAN-P run-time system is available at `http://www.irisa.fr/compose/plan-p`.

Micro-benchmark We first measure the time spent to treat a single packet on a PC Pentium-Pro and a Sun Ultra-1 (see Figure 3). On the Sun, while the run-time specialized PLAN-P bridge is 5 times slower than a hand-crafted embedded C version, it is 35% faster than a Java version compiled and optimized by Hac. Performance for the PC is worse: the speedup between the interpreter and the run-time specialized version is 87 for the Sun and 2.5 for the PC. The main reason for this difference is that the current version of Tempo is much less optimized for the Pentium than for the Sparc. In particular, for the Pentium, function inlining is not performed at run time. We expect to have the same level of performance for the PC as for the Sun when this optimization will be implemented.

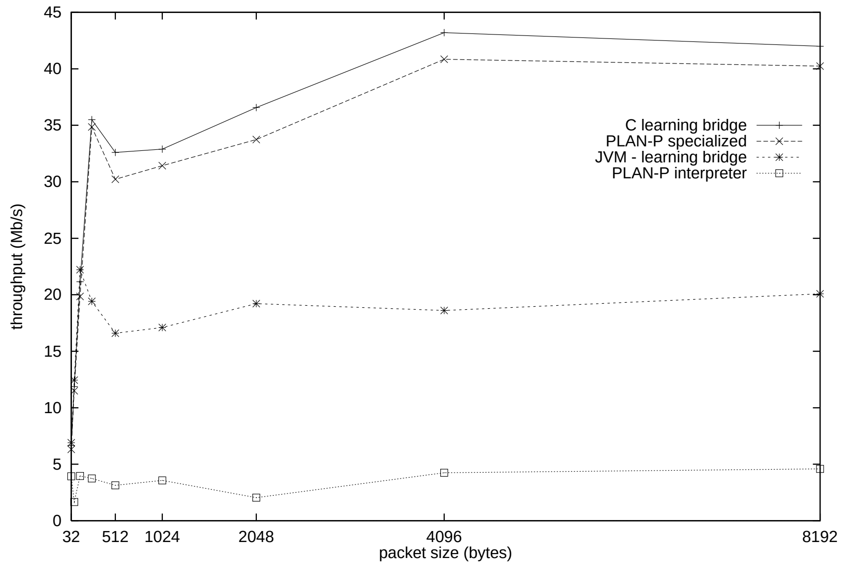

Figure 4: TCP bandwidth of the Ethernet learning Bridge

Real-benchmark The configuration used for the experiment was two hosts connected to a bridge via 100 Mbps Ethernet. Both the hosts and the bridge were Sun Ultra 1/170 Mhz. Throughput was measured using `ttcp` with packet sizes varying from 32 to 8192 bytes (see Figure 4).

The relative performance of the programs shown in Figure 4 is as one would expect. The PLAN-P interpreter has the lowest throughput. The Sun JDK has greater throughput than the PLAN-P interpreter, but still has considerable interpretation overhead. Since the specialized PLAN-P interpreter eliminates the interpretation layer, it achieves higher bandwidth than either source-code interpretation or bytecode interpretation. Finally, the throughput of the hand-written C code is only 4% larger than the throughput of the code automatically produced by specialization.

4.2 GAL

GAL is a language for the specification of graphic adaptors for the purpose of generating device drivers [38]. Using GAL allows the program to remain at a high level of abstraction, thus eliminating error-prone low-level code such as bit manipulation. GAL specifications are up to 10 times smaller than the corresponding C drivers. Additionally, the language allows specifications to be automatically checked for certain errors, such as the specification of registers that overlap.

GAL was implemented as a structured interpreter like PLAN-P, simply to minimize the implementation time. Using the interpreter also allows rapid driver development, since the compilation phases are eliminated. Once the specification is fully tested, however, it is desirable to generate compiled code. Since device drivers can be compiled off-line, compiled code can be generated by applying compile-time specialization to the GAL interpreter. Because the compiler is generated automatically from the interpreter, we are guaranteed that the functionality is preserved.

4.2.1 Performance

The GAL interpreter has been developed for the publicly available XFree86 X11 server [40]. The X server can be linked with the GAL interpreter or a driver generated by specializing the interpreter for a given GAL program. We evaluate the performance of the results of specialization using the standard XBench X server benchmarks. Although XBench reports several measures of performance, we are only concerned with the lines/second and rectangles/second measures, because these are the only operations that use the device driver.

Table 4 reports the XBench results obtained for three versions of an S3 device driver. The first server, S3 XAA, was built with the standard hand-coded C device driver included in the XFree86 distribution. The second server, S3 AM, was built using the GAL interpreter where the interpretation layer has been specialized and only the basic (unspecialized) building blocks remain. Finally, the S3 PE server was built using the GAL interpreter where both the interpretation layer and the building blocks have been specialized. The building blocks are also specialized because they are generic building blocks and the interpretation of their parameters can be removed by specialization. The percentage column gives the percent of the performance obtained as compared to the performance of the hand-coded driver in C. As clearly seen by these results, there is no loss in performance due to the use of GAL, and yet GAL provides a easier and more reliable method to develop device drivers.

Server	lines/s	percent
S3 XAA	189,000	-
S3 AM	150,000	79
S3 PE	191,000	101
Server	rectangles/s	percent
S3 XAA	203,000	-
S3 AM	169,000	83
S3 PE	205,000	101

Table 4: XBench results with GAL

5 Conclusion

Interpretation is reemerging as a significant programming-language implementation technique, both for portability and to enable rapid prototyping of evolving languages. Nevertheless, interpretation carries a significant performance penalty, when compared to traditional compilation. We have shown that specialization can help bridge this gap, generating compiled code safely and efficiently based on an interpreter. The experiments described in this paper show the following:

- It is now possible to specialize existing interpreters for real languages and achieve acceptable performance. Earlier work on specializing interpreters focused on toy languages implemented using functional languages.
- Although specialization of the Java interpreter achieves good speedup (4 times faster than the unmodified Hi and 22 times faster than Hi modified to eliminate the quick instructions), the performance is far from that produced by an optimizing compiler, because specialization does not eliminate the stack.
- In the case of O'Caml, we get same or better results than Piumarta and Riccardi as reported in PLDI'98. Moreover, specialization is a much more general technique.
- It is not practical to manually develop a traditional optimizing compiler for domain-specific languages that rapidly evolve. Specialization is particularly attractive in this context.

Our experiments show that program specialization is entering relative maturity. Thus we can expect that software engineers will soon have specializers, just as they now have parallelizers, that will help the design and prototyping of compilers. With the increasing need for dynamic code loading and heterogeneity support in many embedded systems (mobile phone, smartcards, active networks, etc.), domain-specific languages, interpreters, and specialization offer an appealing solution for the design and implementation of run-time environments.

References

[1] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[2] B.R.T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13, April 1995.

[3] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In PLDI'96 [32], pages 149–159.

[4] C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 496–519, Cambridge, MA, USA, August 1991. Springer-Verlag.

[5] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, January 1993. ACM Press.

[6] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), 1998.

[7] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.

[8] C. Consel and S.C. Khoo. Semantics-directed generation of a Prolog compiler. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the* 3^{rd} *International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 135–146, Passau, Germany, August 1991. Springer-Verlag.

[9] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In POPL96 [33], pages 145–156.

[10] *Conference on Domain Specific Languages*, Santa Barbara, CA, October 1997. Usenix.

[11] D.R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In PLDI'96 [32], pages 160–170.

[12] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In POPL96 [33], pages 131–144.

[13] D.R. Engler and M.F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 26–30, Stanford University, CA, August 1996. ACM Press.

[14] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996. ISBN 0-201-63451-1.

[15] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Network programming using PLAN. In *Workshop on Internet Programming Languages*, Chicago, May 1998.

[16] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In PEPM'97 [30], pages 63–73.

[17] L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 293–314, Paris, France, September 1997. Springer-Verlag.

[18] N.D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[19] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

[20] Siau Cheng Khoo and R. S. Sundaresh. Compiling inheritance using partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 211–222, New Haven, CT, USA, September 1991. ACM SIGPLAN Notices, 26(9).

[21] D. Ladd and C. Ramming. Two application languages in software production. In *USENIX Symposium on Very High Level Languages*, New Mexico, October 1994.

[22] D. A. Ladd and J. C. Ramming. Programming the Web: An application-oriented language for hypermedia service programming. In *Fourth International World Wide Web Conference*, Boston, Massachusetts, December 1995.

[23] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–269, San Diego, California, USA, January 1993. USENIX.

[24] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The Packet Filter: an Efficient Mechanism for User-level Network Code. In *The Proceedings of the 11th Symposium on Operating System Principles*, November 1987.

[25] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[26] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Portland (Oregon), USA, June 1997. Usenix.

[27] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In PEPM'97 [30], pages 116–125.

[28] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, Washington, October 1996.

[29] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.

[30] *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press.

[31] I. Piumarta and F. Riccardi. Optimizing directed threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 291–300, Montreal, Canada, 17–19 June 1998.

[32] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996. ACM SIGPLAN Notices, 31(5).

[33] *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, St. Petersburg Beach, FL, USA, January 1996. ACM Press.

[34] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[35] Pendragon Software. Caffeinemark 2.5. URL: http://www.webfayre.com/pendragon-/cm2/index.html, 1996.

[36] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, October 1998.

[37] S. Thibault, J. Marant, and G. Muller. Adapting distributed applications using extensible networks. Research Report 1200, IRISA, Rennes, France, August 1998.

[38] S. Thibault, R. Marlet, and C. Consel. A domain-specific language for video device drivers: from design to implementation. In DSL'97 [10], pages 11–26.

[39] E. N. Volanschi. *An Automatic Approach to Speclializing System Components.* PhD thesis, Université de Rennes I, February 1998.

[40] The XFree86 Project. http://www.xfree86.org/.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399