© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Static and Dynamic Program Compilation by Interpreter Specialization

SCOTT THIBAULT, [*] CHARLES CONSEL thibault@gmvhdl.com, consel@irisa.fr
COMPOSE group, IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France

JULIA L. LAWALL [**] jll@cs.bu.edu
Computer Science Deptartment, Boston University, 111 Cummington St., Boston, MA 02215, USA

RENAUD MARLET, GILLES MULLER {marlet,muller}@irisa.fr
COMPOSE group, IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract. Interpretation and run-time compilation techniques are increasingly important because they can support heterogeneous architectures, evolving programming languages, and dynamically-loaded code. Interpretation is simple to implement, but yields poor performance. Run-time compilation yields better performance, but is costly to implement. One way to preserve simplicity but obtain good performance is to apply program specialization to an interpreter in order to generate an efficient implementation of the program automatically. Such specialization can be carried out at both compile time and run time.

Recent advances in program-specialization technology have significantly improved the performance of specialized interpreters. This paper presents and assesses experiments applying program specialization to both bytecode and structured-language interpreters. The results show that for some general-purpose bytecode languages, specialization of an interpreter can yield speedups of up to a factor of four, while specializing certain structured-language interpreters can yield performance comparable to that of an implementation in a general-purpose language, compiled using an optimizing compiler.

Keywords: partial evaluation, compilation, compiler design, Just-In-Time compilation, run-time code generation, domain-specific languages, bytecode languages.

1. Introduction

Modern computing environments are characterized by heterogeneous architectures, evolving programming languages, and dynamically-loaded code. Domain-specific languages, for example, may evolve rapidly. Similarly, mobile computing applications typically require that dynamically-loaded code be kept at a high enough level to allow portability and verification. Traditional compilers, which perform complex, machine-specific optimizations and have a high development cost, 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, inter-

[*] Author's current address: Green Mountain Computing Systems, Inc. So. Burlington, VT., USA
[**] Supported in part by NSF Grant EIA-9806718.

pretation carries a significant performance penalty. To eliminate this performance penalty, one solution is to systematically transform an interpreter into a code generator, *i.e.* a compiler [20, 34]. Various techniques have been proposed to achieve this goal:

Run-time code generation. The run-time code generation language 'C provides a high-level notation, based on C, in which to write programs that generate executable code at run time. An interpreter written in C can thus be translated into 'C and slightly modified to generate code rather than executing it. Nevertheless, this approach is error-prone, since little or no verification of the code generation process is performed. Other run-time code generation languages, such as MetaML [36], Modal-ML [41], and Cyclone [16], provide type systems that ensure the correctness of the code generation process. Directly using such a language, however, still implies that the code-generating program is written by hand.[1]

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 [32]. 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 an inliner has to be explicitly constructed from each interpreter.

Directive-driven specialization. Specialization optimizes a program by precomputing expressions that depend only on early-known inputs. In particular, specialization of an interpreter with respect to a statically-known program replaces each term of the program with its (possibly also specialized) implementation, thus acting as a compiler. Using the directive-driven specialization approach [3], the programmer instruments 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 [7, 20] 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. Specialization can also give better results, *e.g.*, if the analysis provides multiple, context-sensitive annotations of individual source-program constructs.

Compilation by automatic specialization of an interpreter has a long history [4, 6, 10, 19, 20, 21]. Program specialization has been shown to automatically process the static semantics of the languages, including scope resolution, storage calculation, and type checking. Such experiments, however, have been done in the context of languages subsets, using restricted implementation languages, such as a functional subset of Scheme. Coding an interpreter in such a high-level language prevents lower-level optimizations from being expressed in the interpreter, and thus limits the performance of the compiled code.

When an interpreter is written in an efficient language, such as C, specializing the interpreter with respect to a program can generate an efficient implementation of the program. Recent advances in automatic program specialization have allowed the development of program specializers for C, including C-Mix [1], developed at DIKU, and Tempo [8], developed at IRISA. Unlike C-Mix and earlier specializers for functional languages, Tempo enables programs to be specialized both at compile time and at run time. Run-time specialization is necessary when the specialization values are not available until run time, as is the case for dynamically-loaded code. Specialization of an interpreter to the source program at run time achieves *Just-In-Time* compilation.

In this paper, we show that interpreter specialization (both at compile time and at run time), using Tempo, can generate efficient compiled code for realistic languages ranging from Objective Caml, a byte-coded general-purpose language, to PLAN-P, a domain-specific language for implementing active-network protocols. We find that specializing a bytecode interpreter can increase performance by up to a factor of 4, and that specializing an interpreter for a high-level domain-specific language can increase performance by up to a factor of 85. In the case of domain-specific languages, we find that interpreter specialization yields performance comparable to that of an equivalent program written in a general-purpose language and compiled with an optimizing compiler. The results of our experiments clearly demonstrate that program specialization can be a key tool in the development and efficient implementation of new languages.

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, while Section 4 presents experiments in specializing 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 [8]. 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. These analyses keep track of known values across procedures, data structures, and pointers [17, 18]. 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 [9]. The implementation of run-time specialization is divided into compile-time and run-time phases. At compile time, Tempo generates both a dedicated run-time specializer and binary-code 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 templates, and instantiates these templates with computed values [11, 31]. Because this approach to run-time specialization has low overhead, it is suitable for implementing run-time compilation.

Tempo has been successfully used for a variety of applications such as operating systems (Sun Remote Procedure Call — RPC [26, 27], Chorus Inter-Process Communication — IPC [40]), and scientific programs (*e.g.,* convolution filters [35], FFT [31]). These applications have demonstrated two key features of Tempo: (1) it can process realistic programs that have not been carefully crafted for specialization (2) it can generate an efficient implementation from generic software (*e.g.,* the specialized layer of the Sun RPC runs 3.5 times faster than the original one). In this paper, we show that Tempo is an effective tool for specializing interpreters for both low-level bytecode languages and high-level domain-specific languages.

3. Bytecode Interpreters

We first examine the specialization of bytecode (flat, linearized code) interpreters. Typically, bytecode instructions correspond closely to machine instructions, but without being tied to a particular architecture. Thus, bytecode is often used in the context of a virtual machine. Because bytecode interpreters both provide dynamic program loading and allow 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 (JVM), Objective Caml (O'Caml), and the Berkeley Packet Filter (BPF). We first examine issues common to most bytecode interpreters, and then consider the interpreters individually.

3.1. Specialization of bytecode interpreters

Most 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 on the bytecode instructions, producing a specialized program that only manipulates the stack. The specialized program is thus essentially a concatenation of the implementations of the program instructions. To achieve this effect, we need to ensure that the program counter's value is statically known at every program point in the interpreter.

After executing most instructions, the program counter is simply set to the next instruction. In this case, the value of the program counter depends only on the structure of the interpreted program, and is thus statically known. When there is a conditional branch in the bytecode, however, the choice of whether to branch or continue with the next instruction can depend on the values of the inputs to

```
Val execVM(Prog pg, Stack sp)
{
  Index pc = 0;
  while(TRUE)
  {
    switch(pg[pc])
    {
      ADD: {
        int v1 = POP(sp);
        int v2 = POP(sp);
        PUSH(sp,v1+v2);
        pc += NEXT(pg[pc]);
        break;
      }
      ...
      IFZERO:
        if(POP(sp) == 0)
         pc += JMP_OFFSET(pg[pc]);
        else
         pc += NEXT(pg[pc]);
        break;
      ...
    }
 }
```

a: Original interpreter

```
Val execVM(Prog pg, Stack sp, Index pc)
{
  while(TRUE)
  {
    switch(pg[pc])
    {
      ADD: {
        int v1 = POP(sp);
        int v2 = POP(sp);
        PUSH(sp,v1+v2);
        pc += NEXT(pg[pc]);
        break;
      }
      ...

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

b: Rewritten interpreter

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

the bytecode program, which are not known during specialization. This situation is illustrated by the interpretation of the `IFZERO` instruction in Figure 1-a, which tests whether the value at the top of the stack is zero, and sets the program counter accordingly. Because the value of the condition is unknown, it is impossible to know whether the program counter will be assigned the address of the destination or the next instruction, and thus its value cannot be statically known after the conditional branch instruction. Subsequently, all references to the program counter are considered to be unknown, and no specialization occurs.

To solve the problem, we use the following observation about the `if`-statement used to implement a conditional branch instruction. Within the branches of this `if`-statement the value of the program counter is still known. It is not until after the `if`-statement that the program counter becomes unknown, when the analysis merges the two possibilities because of the unknown test. Thus, one approach that avoids the need to merge the two possible values of the program counter is for the specializer to systematically duplicate the specialization of all of the code that is executed after every `if`-statement (including the subsequent iterations of the interpretation loop) within each branch of the `if`-statement. This approach is known as *continuation-passing style specialization* [1, 5]. Nevertheless, while continuation-passing style specialization avoids the need to manually alter the source program, it can lead to code explosion.

To avoid unnecessary code growth, we instead modify the source program to simulate continuation-passing style specialization only when needed. In particular, we manually duplicate the continuation of an `if`-statement in its branches only when the duplication is necessary to allow the program counter to remain known. The main difficulty is to make the entire continuation, including subsequent iterations of the interpretation loop, explicit in the interpreter. To this end, we extract the interpretation loop into a separate procedure, parameterized by the program counter. Now at any point we can continue interpretation either implicitly by reaching the end of the loop, or explicitly by making a recursive call.

Concretely, the program is manually rewritten by first extracting the interpretation loop into a separate procedure, as described above. Then, each `if`-statement implementing a bytecode conditional branch is rewritten as follows. First, code explicitly following the `if`-statement, if any, is simply copied into the branches. Next, a recursive call is added to the end of each branch, to model the subsequent iterations of the interpretation loop. The result of each such call is then immediately returned, using a `return`-statement. This use of a `return`-statement reflects the fact that the recursive call performs the entire remaining computation. It also has a beneficial effect on the binding times. Because the analyses of Tempo do not propagate values over a `return`-statement, the program counter is not considered unknown after the transformed `if`-statement. The result of this transformation is illustrated in Figure 1-b. Note that the applicability of this transformation relies on the fact that the interpreter is structured, and is independent of the structure of the interpreted program.

We apply this rewriting to each of the bytecode interpreters considered. In the case of the O'Caml and BPF interpreters, we perform other language-specific modifications to improve the performance of the specialized code, as described below. While these transformations are only necessary if the interpreter is intended for specialization, the changes are systematic, localized, and do not compromise the readability and the maintainability of the interpreter.

3.2. The Java Virtual Machine (JVM)

Java bytecode is a perfect target for specialization: it is designed to be executed on a bytecode interpreter. Nevertheless, hand-written run-time (JIT) or off-line compilers are often used to improve performance. For our specialization experiment, we target the Harissa system [28]. Harissa is a flexible environment for Java execution that permits mixing both compiled and interpreted code. Harissa's compiler (Hac) generates among the most efficient code for Java programs [29], while the interpreter (Hi) is slightly faster than the Sun JDK 1.0.2 interpreter. Hi is a 1000-line hand-optimized C program.

Applying Specialization The Java language is designed to support dynamically-loaded bytecode. In this context, an efficient and effective compiler that can be invoked at run time (*i.e.* a Just-In-Time compiler) is essential. Run-time specialization of a bytecode interpreter with respect to the dynamically-loaded code provides such compilation, by replacing each bytecode instruction by its native implementation.

Beyond simply replacing instructions by their implementation, specialization of Java bytecode can optimize the generated code with respect to constants explicit in the bytecode instructions. In particular, dynamically-loaded Java code depends on the constant pool of the environment into which it is loaded. Specializing the dynamically-loaded code to the constant pool eliminates references to this information during execution. Indeed, this functionality was anticipated by the designers of the Java bytecode language [14]. When a bytecode instruction first refers to an element of the constant pool, it must resolve the entry. Subsequent invocations of the same instruction, however, need not resolve the entry again. Thus, the virtual machine can replace such an instruction by a "quick" instruction, which does not perform the resolution step. Essentially, specialization of the interpreter with respect to the dynamically-loaded code automatically generates "quick" instructions.

Performance We evaluate the performance of the specialized Java bytecode interpreter using the Caffeine 3.0 benchmarks. 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[2].

Table 1. Results of the Caffeine 3.0 Java benchmark (in CaffeineMarks, higher is faster)

	JDK 1.0.2	JDK 1.1.6	Hi	Hi (no quick inst)	Kaffe	Run-time spec. of Hi	Hac	Compile-time spec. of Hi
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

The 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), as well as the compile-time and run-time specializations of the Hi interpreter. The results are shown in Table 1. The first four columns compare the performance of the interpreters. Due to the many manual optimizations implemented by Sun, the JDK 1.1.6 interpreter is about twice as fast as the older JDK 1.0.2 interpreter. As expected, Hi performs better than JDK 1.0.2. To illustrate the impact of the "quick" instructions, we also test the Hi interpreter with the "quick" instructions disabled. With this modification, its performance declines by a factor of up to 10.

By specializing Hi with "quick" instructions disabled, we get an average speedup of 32 for compile-time specialization and 21 for run-time specialization.[3] This speedup is mainly derived from the elimination of the program counter and by the specialization of generic instructions into instructions having the functionality of "quick" instructions. The last four columns of Table 1 compare the performance of run-time and compile-time specialized code with the code generated by hand-optimized JIT and off-line compilers, respectively. The optimized JIT kaffe produces code that is up to 4 times faster than that produced by run-time specialization, while the optimized off-line compiler Hac produces code that is up to 10 times faster than that produced by compile-time specialization. We elaborate on the reasons for this gap in Section 3.5. Nevertheless, the specialized code is up to four times faster than the Hi interpreter implementing the "quick" instructions.

3.3. The Objective Caml Abstract Machine (O'Caml)

Our second bytecode interpreter is the O'Caml abstract machine. The O'Caml bytecode language is significantly different from that of the JVM because it is the target of a functional language. For example, the O'Caml bytecode interpreter 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 [32]. We obtain performance comparable to their results. However, unlike selective inlining, specialization is a general tool that can be applied to a larger class of applications.

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 language supports higher-order functions, the program counter is not statically known. When a closure that is the value of an arbitrary expression is applied, 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 individually, 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. This modification is crucial for obtaining significant benefit from specialization of an interpreter for a language with higher-order functions.

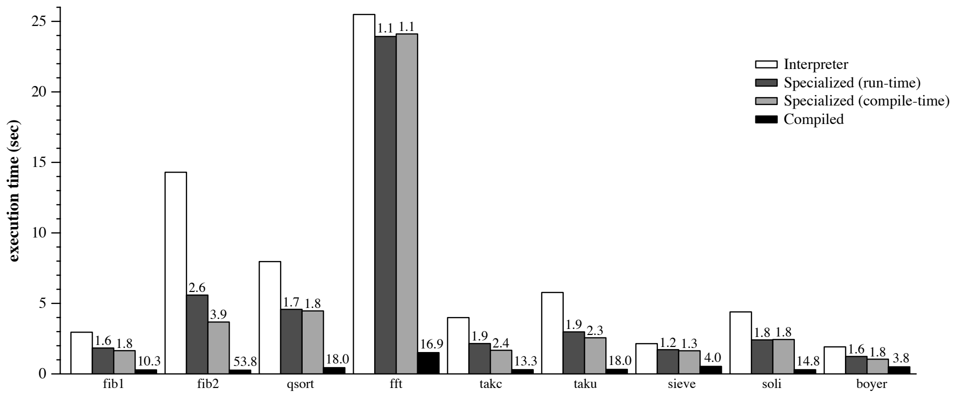

Figure 2. Results on O'Caml benchmark suite (execution times and speedups)

Performance To measure the performance of the specialized interpreter, we used a standard O'Caml benchmark suite. The programs in the benchmark suite range in size from 20 lines (`takc` and `taku`) to 900 lines (`boyer`). Figure 2 compares the performance of these benchmarks when interpreted by the standard optimizing interpreter, when compiled by specializing this interpreter (at compile time and at run time), and when compiled using the standard native-code compiler. The heights of the bars represent the relative run times. The number on each bar indicates the speedup as compared to 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. We use `fib2` assess the specific benefits of specialization in Section 3.5. 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 [32]. It is not surprising that the specialized version is not significantly faster than the inlining approach because the O'Caml bytecode language 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 implementation of each instruction (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, and are loaded into the kernel where they are traditionally interpreted at the expense of high computational cost [25]. The BPF [24] is considered as a reference implementation for many optimization techniques [13, 30].

Table 2. BPF benchmarks (time in seconds)

	Interpreter	CT Spec.	CT Speedup	RT Spec.	RT Speedup
Pentium	3.32	0.98	3.40	1.95	1.70
Sparc (original)	2.62	0.68	3.89	1.57	1.67
Sparc (modified)	—	0.40	6.56	1.31	2.01

```
/* 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

Applying Specialization As for the other bytecode languages, specializing the BPF interpreter eliminates instruction dispatch. Additionally, we take advantage of the fact that the interpreter will be specialized, by manually coding optimizations into the interpreter. Figure 3-a shows the original interpreter code for a packet load instruction on a big-endian machine. 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 test might make the original interpreter slower, it results in faster specialized programs because the condition of the added `if`-statement is known statically and evaluated at specialization time.

Performance Table 2 shows the execution time for a simple 10-instruction filter program applied to 5000 packets. Results are given for the PC Pentium-Pro 200

Mhz and for a Sun Ultra-1/200Mhz, 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 includes a test that one might not implement for ordinary interpretation.

3.5. Discussion

In this section, we have presented 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 the limited speedup: bytecode languages often already contain specialized instructions, and there are other optimizations such as stack elimination that are typically performed by compilers but that are not achieved by specialization.

Both the JVM and O'Caml bytecode languages include many specialized instructions to improve performance. For example, both bytecode languages include an instruction of the form `LOAD n`, which loads the n^{th} item on the stack. However, for small values of n, the bytecode languages also include 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 generic instructions.

Additional speedups obtained by 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 compile-time specialization. We performed two main transformations, by hand, on a compile-time specialized version of the `fib2` program. Since some of the looping within the interpreter is implemented using recursive calls, as described in Section 3.1, the specialized code is a set of recursive functions. So that the state of the interpreter is accessible to all of these functions, it must be maintained in global variables. As a result, the compiler does not perform register allocation or any optimizations on these variables. Thus, we first inline the specialized functions into a single function, and declare the state variables locally within this function. Second, we convert stack elements to local variables. This transformation permits register allocation and eliminates many memory references. After applying both transformations, the resulting program is almost identical to the program generated by the native O'Caml compiler. The only significant remaining difference 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

An emerging trend in software development consists of designing languages specific to a particular domain. This approach is actively studied in both academia [12] and industry [2, 22, 23]. These languages, called *Domain-Specific Languages* (DSLs), consist of notations, abstractions, and values that are specific to a particular family of problems. Associated with a DSL is a set of properties that can be statically verified, ensuring the safety of a DSL program. Both languages studied in this section are DSLs: PLAN-P is a language aimed at developing network application protocols and GAL is a language for specifying video card device drivers.

4.1. Specialization of DSL interpreters

In assessing the benefits of compilation via program specialization for high-level languages, it is important to differentiate between general-purpose languages (GPLs) and DSLs. The design of a GPL is typically stable, and, if the language is an industry standard, high-quality compilers are often available for many different platforms. Compilation by specialization of an interpreter does not generally achieve the performance of such compilers. Nevertheless, because DSLs may have a limited audience or evolve frequently, it is often not feasible to develop optimizing compilers for them.

Furthermore, DSLs do offer some features that allow compilation via specialization to generate efficient code. First, a DSL often simply amounts to a glue language whose interpreter mainly invokes domain-specific building blocks. These building blocks may be coded in a GPL, and thus can be compiled using a traditional, highly optimizing compiler. In this case, eliminating the interpretive overhead by specialization yields code comparable to a highly optimized GPL implementation. Second, the properties associated with a DSL can allow optimizations in the interpreter that are not possible for a GPL. For example, a Java execution environment must implement run-time array-bounds checks to ensure safe memory access. If the restricted nature of a DSL prevents the programmer from expressing out-of-bounds array accesses, the interpreter need not perform such run-time tests, and yet can still provide the same level of safety as Java. When code is compiled by interpreter specialization, this optimization is naturally implemented in the compiled code. For these reasons, specialization of the PLAN-P and GAL interpreters produces code that is comparable to that generated by a GPL compiler on equivalent programs.

Specialization of an interpreter of a high-level language gives a more dramatic performance improvement than the specialization of an interpreter for a bytecode language, because a high-level language contains more complex syntactic constructs that can be processed in more complicated ways at compile time (*e.g.* type checking, scope resolution, etc.). We find that DSL programs compiled using program specialization can run between 10 and 85 times faster than interpreted programs.

4.2. PLAN-P

The PLAN-P language allows the programmer to define network protocols that manipulate packets associated with a specific application [37, 38]. 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 was originally based on PLAN, a Programming Language for Active Networks [15], which is dedicated to network diagnostics. Nevertheless, the semantics of PLAN-P is significantly different in order to treat a larger scope of applications, such as the adaptation of distributed applications and services [38]. 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 yield the same throughput as equivalent hand-crafted C versions [37].

Performance We assess 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 the source of each packet in order to determine the LAN to which a host is connected, so that packets for the host are only repeated on its LAN. We implement the learning bridge using a hash table to record the source address and the LAN of each received packet. The implementation in the PLAN-P language is 40 lines long.

We have done two types of benchmarks: (1) micro-benchmarks that measure the pure computation time of the learning bridge, without including the run-time system, (2) a real benchmark that measures the throughput of the system. The micro-benchmarks 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.

Table 3. Ethernet bridge micro-benchmark (time per packet, in micro-seconds)

	Embedded C	PLAN-P specd. at run time	Java compiled with Hac	PLAN-P interpreted
Sun Ultra-1	3	14	19	1218
PC/Pentium Pro	4	182	18	440

The micro-benchmarks measure the time spent to treat a single packet on a PC Pentium-Pro 200 Mhz and a Sun Ultra-1/170Mhz (see Table 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. On the PC, the run-time specialized code is much less efficient, mainly because Tempo is currently 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 is implemented.

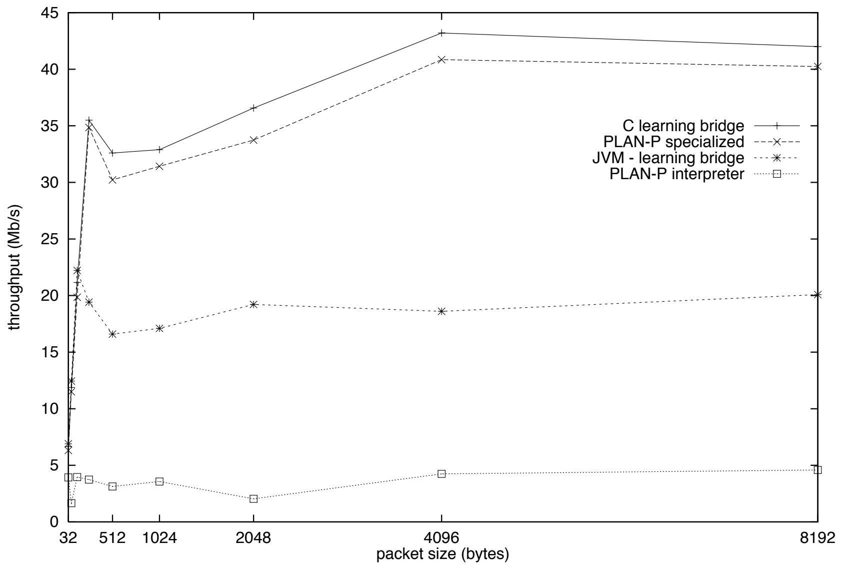

Figure 4. TCP bandwidth of the Ethernet learning Bridge

To assess the impact of specialization on the performance of the entire system, we considered a network consisting of 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. As shown in Figure 4, the PLAN-P interpreter has the lowest throughput. The Java version interpreted using the Sun JDK 1.0.2 has greater throughput than the PLAN-P interpreter, but still has considerable interpretive overhead. Since specialization of the 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% greater than the throughput of the code automatically produced by specialization.

4.3. GAL

GAL is a language for the specification of graphic adaptors for the purpose of generating device drivers [39]. 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 using an interpreter simply to minimize the implementation time. 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.

The GAL language is a glue language, connecting generic building blocks. These building blocks can also be specialized to remove the interpretation of their parameters.

Performance The GAL interpreter has been developed for the publicly available XFree86 X11 server. 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 specialized code 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 on a PC Pentium-Pro 200 Mhz 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 percentage column gives the percentage of the performance obtained as compared to the performance of the hand-coded driver in C. As is clearly shown by these results, there is no loss in performance due to the use of GAL, and yet GAL provides an easier and more reliable method to develop device drivers.

Table 4. XBench results with GAL

Server	lines/s	percent	rectangles/s	percent
S3 XAA (standard server)	189,000	-	203,000	-
S3 AM (interpretation eliminated)	150,000	79	169,000	83
S3 PE (completely specialized)	191,000	101	205,000	101

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 perform low-level optimizations such as stack elimination.
- 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.
- Compilation by interpreter specialization gives good performance for DSLs. Since it is not practical to manually develop a traditional optimizing compiler for domain-specific languages that rapidly evolve, specialization provides a needed alternative.

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 phones, smartcards, active networks, etc.), the combination of domain-specific languages, interpreters, and specialization offers an appealing solution for the design and implementation of run-time environments.

Availability The systems described in this paper are available at the following URLs:

- Tempo is available at `http://www.irisa.fr/compose/tempo`
- Harissa is available at `http://www.irisa.fr/compose/harissa`
- The Caffeine benchmarks CaffeineMark 3.0 are available from Pendragon Software at
`http://www.webfayre.com/pendragon/cm3/index.html`
- The O'Caml benchmark suite is available at
`ftp://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/benchmarks/objcaml.tar.gz`
- GAL is available at `http://www.irisa.fr/compose/gal`
- The XFree86 X11 server is available at `http://www.xfree86.org`
- A prototype of the PLAN-P run-time system is available at
`http://www.irisa.fr/compose/plan-p`

Acknowledgments We would like to thank Luke Hornof, Ulrik Pagh Schultz, and the anonymous referees for helpful comments on a draft of this paper.

Notes

1. Cyclone can also be used in the back end of the Tempo specializer described below [16].
2. The other tests are not relevant for this experiment since they primarily measure the efficiency of features of the JVM that are not related to compilation, such as graphics or memory allocation.
3. Speedup is calculated by dividing the execution time of the unspecialized code by the execution time of the specialized code. Here, and subsequently in the paper, the time for specialization is not included in the performance measurements.

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. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, PA, May 1996. ACM SIGPLAN Notices, 31(5).
4. A. Bondorf and J. Palsberg. Generating action compilers by partial evaluation. *Journal of Functional Programming*, 6(2):269–298, 1996.
5. 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.
6. C. Consel and O. Danvy. Static and dynamic semantics processing. In *Conference Record of the Eighteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, Orlando, FL, USA, January 1991. ACM Press.
7. 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.
8. 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.
9. 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.
10. 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.
11. C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996. ACM Press.
12. *Conference on Domain Specific Languages*, Santa Barbara, CA, October 1997. Usenix.

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. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, volume 34(1) of *ACM SIGPLAN Notices*, pages 86–93. ACM, June 1998.
16. L. Hornof and T. Jim. Certifying compilation and run-time code generation. *Higher-Order and Symbolic Computation*, 12(4):337–376, December 1999.
17. L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 63–73, Amsterdam, The Netherlands, June 1997. ACM Press.
18. 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.
19. 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.
20. N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.
21. S.C. 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).
22. D. Ladd and C. Ramming. Two application languages in software production. In *USENIX Symposium on Very High Level Languages*, New Mexico, October 1994.
23. 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.
24. 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.
25. 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.
26. G. Muller, R. Marlet, and E.N. Volanschi. Accurate program analyses for successful specialization of legacy system software. *Theoretical Computer Science*, 248(1–2), 2000. To appear in TCS Volume 248/1-2.
27. 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.
28. 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.
29. G. Muller and U. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, pages 44–51, March 1999.
30. 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.
31. 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.

32. I. Piumarta and F. Riccardi. Optimizing directed threaded code by selective inlining. In PLDI'98 [33], pages 291–300.
33. *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, Montreal, Canada, 17–19 June 1998.
34. D.A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn and Bacon, Inc., 1986.
35. U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 367–390, Lisbon, Portugal, June 1999.
36. W. Taha, W. Benaissa, and T. Sheard. Multi-stage programming: Axiomatization and type safety. In *Automata, Languages and Programming, 25th International Colloquium (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 918–929, Aalborg, Denmark, July 1998.
37. 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.
38. S. Thibault, J. Marant, and G. Muller. Adapting distributed applications using extensible networks. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 234–243, Austin, Texas, May 1999. IEEE Computer Society Press.
39. S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: from design to implementation – application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May–June 1999.
40. E. N. Volanschi. *An Automatic Approach to Specializing System Components.* PhD thesis, Université de Rennes I, February 1998.
41. P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and Modal-ML. In PLDI'98 [33], pages 224–235.