INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Faster Run-time Specialized Code using Data Specialization

Julia L. Lawall and Gilles Muller

N°3833

Decembre 1999

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--3833--FR+ENG

Faster Run-time Specialized Code using Data Specialization

Julia L. Lawall and Gilles Muller

Thème 2 — Génie logiciel
et calcul symbolique
Projet COMPOSE

Rapport de recherche n° 3833 — Decembre 1999 — 22 pages

Abstract: Run-time specialization is a technique that optimizes a program based on run-time information. In this context, specialization time must be constrained, limiting the possibility to further optimize the specialized code. We present a low-cost methodology for improving the code generated by a run-time specializer. This result is acheived by combining run-time specialization with another form of automatic specialization, data specialization. We show how to use our approach to implement compaction of run-time specialized code in the framework of the Tempo specializer for C programs. We find that the compaction optimization can improve the performance of the specialized code by up to a factor of 4, while adding only about 10% to the cost of run-time specialization on most of our examples.

Key-words: run-time code specialization, data specialization, program staging

(Résumé : tsvp)

Supported in part by NSF Grant EIA-9806718.

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

Amélioration d'un code spécialisé à l'exécution par spécialisation de données

Résumé : La spécialisation de programmes à l'exécution est une technique d'optimisation qui permet d'exploiter des informations connues uniquement au moment de l'exécution. Dans ce contexte, le temps de spécialisation doit être borné pour ne pas pénaliser l'exécution du programme. Cette contrainte limite les possibilités d'optimisations additionnelles au moment de la génération du code. Cet article présente une méthode permettant d'optimiser le code généré par spécialisation à l'exécution, sans pour autant introduire de surcoût significatif en temp d'exécution. Ce résultat est obtenu en associant une autre technique de spécialisation, appelée spécialisation de données, à la spécialisation à l'exécution. Nous appliquons cette technique pour effectuer une compaction du code généré par le spécialiseur à l'exécution de Tempo. Nos résultats montrent que sur nos exemples, la compaction du code permet d'améliorer la performance des programmes générés d'un facteur allant jusqu'à 4, avec une dégradation du temps de spécialisation limitée à 10%.

Mots-clé : spécialisation à l'exécution, spécialisation de données, programmation étagée

1 Introduction

Partial evaluation is a form of program specialization that optimizes a program with respect to supplementary information about its inputs. A partial evaluator evaluates constructs that depend only on such information (*static* constructs), and rebuilds the remaining *dynamic* constructs to form a specialized program. Recently, attention has turned to partial evaluation at run time [4, 8, 14]. A critical issue for run-time specialization is the need to produce efficient executable code at a minimal run-time cost. In this paper, we present a low-cost methodology for improving the performance of run-time specialized code.

One approach to limiting the cost of run-time specialization is to assemble the specialized program from precompiled fragments of the source program, known as *templates*. Templates contain *holes* representing static subterms whose values are determined during run-time specialization. At run time, the specializer evaluates the static constructs, and copies the compiled templates corresponding to the dynamic constructs into an output buffer. Holes are filled at this time. Variants of this approach have been taken by several run-time specializers, such as Tempo [4, 19], DyC [8, 9], and Cyclone [10].

For many applications, the cost of template-based run-time code generation has been shown to be amortized in only a few invocations of the specialized program [9, 19]. Nevertheless, because templates are compiled before specialization, when there is only approximate knowledge of how they will be assembled, the compiled templates are not highly optimized. The cost of further optimizing the specialized program at run time has to be carefully controlled. Thus, in many cases this approach to run-time specialization gives less speedup than specialization at compile time, where the specialized program can be compiled using an optimizing compiler [13, 19].

One problem with the template-based approach is that templates are written to the output buffer as they are selected. This strategy implies that if we perform optimizations that modify the size or location of previously-emitted templates, we must recopy code that has already been written to the output buffer. Copying code involves substantial memory access, and may require updating branch offsets. These operations would substantially increase the cost of specialization. The goal of this paper is to allow templates for the entire specialized program to be selected and analyzed before any templates are written into the output buffer, thus allowing optimizations on the chosen templates without code copying.

Our approach

In this paper, we propose to divide the run-time specializer into two phases: a first phase that evaluates the static constructs and selects templates, and a second phase that emits the code for the selected templates. This organization allows us to perform global analysis of the selected templates in the first phase without introducing the cost of copying code. Nevertheless, this approach potentially introduces significant extra cost to pass information about the selected templates from the first phase to the second phase. In this paper, we show that data specialization [3, 15] of a dedicated run-time specializer generated by Tempo

automatically produces an efficient two-phase run-time specializer. We then show how to exploit the two-phase approach to implement inter-template optimizations.

As an example of the kinds of optimizations that can be implemented using our approach, we consider compaction of the specialized code. Tempo's run-time specializer begins the specialization of each function by allocating a buffer to hold the specialized code. Because the size of the specialized function is not known at this point, the size of this buffer is chosen arbitrarily. When the sizes of the specialized functions vary widely, this approach leads to considerable fragmentation. In our approach, we calculate the size of the specialized code in the first phase, and allocate buffer space of the correct size in the second phase. This leads to more compact code. In our experiments, compaction improves the performance of the specialized program by up to a factor of 4.

The rest of this paper is organized as follows. We first describe Tempo's approach to run-time specialization in more detail in Section 2. Section 3 defines data specialization, describes our use of data specialization to optimize run-time specialized code, and illustrates the technique using an implementation of code compaction. In Section 4, we consider an alternate interpreter-based approach. Section 5 assesses the performance of the compaction optimization. We find that the specialization time is almost identical to that of the original specializer produced by Tempo, and that compaction significantly improves the performance of the specialized code. Finally, we describe related work in Section 6, and conclude in Section 7.

2 Run-Time Specialization in Tempo

Tempo's approach to run-time specialization has been designed with the goals of minimal run-time overhead for specialization, low development cost, and efficient specialized code. To reduce the run-time overhead, much of the work of run-time specialization is performed at compile time. In particular, the compile-time phase performs binding-time analysis, which identifies static and dynamic expressions, and then constructs templates and a dedicated *run-time specializer* based on this information. At compile time, templates are converted to object code using a standard compiler (`gcc`). The use of `gcc` eliminates the need to develop a new code generator and enhances portability. Because compiler optimizations can be applied, the code within the templates is efficient. The run-time phase simply invokes the run-time specializer on the static values. The run-time specializer evaluates the static constructs and uses the templates to build the specialized program. To limit the run-time overhead, Tempo performs no further optimization of the templates at run time.

2.1 Implementation

We now describe Tempo's implementation of run-time specialization in more detail using an example, the interpreter for arithmetic expressions shown in Figure 1. Suppose we specialize the interpreter with respect to the expression argument `e` (static), leaving the environment argument `r` unknown (dynamic). Figure 1 illustrates the result of binding-time

analysis according to this information. Specialization of the interpreter eliminates tests on the program structure; indeed, interpreters are a classical application area for specialization [11].

```
int interpret(struct Exp *e, struct Env *r) {
  switch (e->type) {
  case NUM : return e->num.value;
  case ID  : return lookup(r,e->id.name);
  case SUM : return interpret(e->sum.e1,r) + interpret(e->sum.e2,r);
  }
}
```

Figure 1: An interpreter of arithmetic expressions. Static constructs are shown in courier font, while dynamic constructs are shown in boldface.

Based on the results of binding-time analysis, a template is constructed for each dynamic block. In the case of the interpreter example, there are five templates, one representing the beginning of the function, one representing the end of the function, and one for each of the `switch` branches. The C code corresponding to the templates for the `NUM` and `ID` cases are shown in Figure 2. The addresses `&H0` and `&H1` represent place-holders (*holes*) that will be filled in during specialization with static values.

Template for `NUM` case (`tmp2`):

```
return (int)(&H0);
```

Template for `ID` case (`tmp3`):

```
return lookup(r, &H1);
```

Figure 2: Templates `tmp2` and `tmp3` for `interpret`

The run-time specializer contains a code-generation function for each function of the source program. The code-generation function for `interpret` is shown in Figure 3.[1] Each code-generation function has the structure of the corresponding source function, modified as follows. The parameters are the static parameters of the source function. A buffer is allocated for the specialized code, using the function `rts_alloc_code`. Each dynamic block is replaced by an invocation of the macro `DUMP_TEMPLATE`. `DUMP_TEMPLATE` copies the template indicated by the second argument into the output buffer, at the position indicated by `code_ptr`, and increments `code_ptr` according to the size of the template, as indicated by

[1] NB: To simplify the presentation, we have eliminated some of the arguments of the code-generation macros. The description provided here is sufficient to present the techniques of the paper, but is not sufficient to implement a run-time specializer in practice. A more complete description is available elsewhere [19].

the third argument. Once a template is written into the output buffer, its holes can be filled. Figure 3 illustrates the use of `PATCH_HOLE` to fill a hole with the value of a static expression, and the use of `PATCH_CALL_HOLE` to patch a function call instruction with the address of the specialized definition. In each case, the first argument is the value to patch, and the second argument is a constant indicating the offset of the hole in the current template. There are similar macros to treat calls to external functions and to correct branch offsets. Each code-generation function returns a pointer to the beginning of the generated code, as stored in the variable `spec_ptr`.

```
void *rts_interpret(struct Exp *e) {
  char *code_ptr = rts_alloc_code(MAX_FN_SIZE);
  char *spec_ptr = code_ptr;

  DUMP_TEMPLATE(code_ptr, tmp1, 8);
  switch (e->type) {
  case NUM :
    DUMP_TEMPLATE(code_ptr, tmp2, 12);
    PATCH_HOLE(e->exp.num.value, 0);
    break;
  case ID :
    DUMP_TEMPLATE(code_ptr, tmp3, 28);
    PATCH_CALL_HOLE(rts_lookup(e->id.name), 8);
    break;
  case SUM : ...
  }
  DUMP_TEMPLATE(code_ptr, tmp5, 12);
  return (void *)spec_ptr;
}
```

Figure 3: Simplified code-generation function for `interpret`

2.2 Memory allocation issues

This approach to run-time specialization is quite simple and is effective in practice [19, 24]. Nevertheless, some issues remain. Let us consider in more detail the allocation of the buffer in which to hold the specialized code for each function.

The code-generation function `rts_interpret`, defined in Figure 3, invokes another code-generation function `rts_lookup`. Because both functions generate code and because `rts_interpret` may generate more code after the call to `rts_lookup`, each function needs its own output buffer. As shown in Figure 3, we solve the problem by reserving all of the

space needed for each specialized function at once, at the beginning of the corresponding code-generation function, using a call to `rts_alloc_code`.

This allocation strategy raises both safety and efficiency concerns. Because run-time specialization performs loop unrolling, it is not possible to determine in advance the size of a specialized function. Thus, safety requires that each use of `DUMP_TEMPLATE` verify that the size of the template does not exceed the remaining space allocated to the specialized function. If the template is too large, the run-time specializer has to allocate a new buffer, and either generate a `goto` statement, or copy the code already generated for the current function into the new buffer. The latter option is implemented in Tempo. The use of a `goto` statement can hurt the performance of the specialized program, while code copying can substantially add to the run-time specialization cost. A partial solution, taken by Tempo, is simply to reduce the likelihood of exceeding the buffer size by allocating a very large buffer for each specialized function. In our experiments, however, a program composed of such widely-scattered function definitions does not effectively use the instruction cache.

These issues can be resolved by determining the exact size of each specialized procedure before beginning code generation. We use this analysis as an example in presenting our data-specialization-based optimization technique.

3 Inter-Template Optimization, Using Data Specialization

Data specialization is a program-staging technique. In particular, data specialization separates a program into two phases: a *loader* phase and a *reader* phase [3, 15]. The loader phase evaluates the static constructs, and creates a *cache* containing the values of the static expressions that occur in a dynamic context. The reader phase evaluates the dynamic constructs, referring to the cache of specialized data for the values of the static subexpressions. Automatic data specialization has been implemented in Tempo, and has been shown to give significant performance improvements both by itself and in conjunction with Tempo's compile-time specialization [3].

Suppose we apply data specialization to the run-time specializer of `interpret`, where we indicate that the function that allocates space in the specialized code buffer, `rts_alloc_code`, gives a dynamic result. The loader and reader corresponding to the code-generation function for `interpret` (Figure 3) are shown in Figure 4. The locations of the selected templates are static and thus stored in the cache by the loader. On the other hand, the code pointer is dynamic, so the occurrences of `DUMP_TEMPLATE` are all in the reader. Thus, data specialization produces a run-time specializer in which all of the templates are selected before any are written into the code buffer. We have achieved automatically the division of the run-time specializer into phases that will allow us to efficiently implement inter-template optimizations.

We now describe how to exploit the staging provided by data specialization to implement inter-template optimization, using code compaction as an example. Our optimization

```
void **rts_interpret_loader(struct Exp *e, void **Cache) {
  *Cache = tmp1;
  switch (*((int *)(Cache + 1)) = e->type) {
  case NUM :
    *((int *)(Cache + 2)) = tmp2;
    *((int *)(Cache + 3)) = e->exp.num.value;
    Cache = Cache + 4;
    break;
  case ID : ...
  case SUM : ...
  }
  *Cache = tmp5;
  return Cache + 1;
}

void **rts_interpret_reader(void **Cache) {
  char *code_ptr = rts_alloc_code(MAX_FN_SIZE);
  char *spec_ptr = code_ptr;

  DUMP_TEMPLATE(code_ptr, *Cache, 8);
  switch (*((int *)(Cache + 1))) {
  case NUM :
    DUMP_TEMPLATE(code_ptr, *((int *)(Cache + 2)), 12);
    PATCH_HOLE(*((int *)(Cache + 3)), 0);
    Cache = Cache + 4;
    break;
  case ID : ...
  case SUM : ...
  }
  DUMP_TEMPLATE(code_ptr, *Cache, 12);
  cache_return = Cache + 1;
  return (void *)spec_ptr;
}
```

Figure 4: Data specialization of the code-generation function for `interpret`. Use of the cache is shown in italics.

strategy is to perform some analyses in the loader phase, and then use the results to emit optimized code in the reader phase. In particular, information about each template determined at compile time and made explicit in the run-time specializer can be collected at specialization time, where it is known which templates will be generated and in what order.

Because these decisions are made based on the static inputs to the source program, such analyses are placed by the data specializer in the loader. We then transmit the results of the analysis to the reader, which contains the code generation operations, thus allowing these operations to optimize the generated code based on this information.

To implement code compaction, we need to accumulate the size of the selected templates in the loader. Conveniently, the template size is already an argument of `DUMP_TEMPLATE`. Thus, we simply introduce a local variable `fnsize` in each code generation function, and modify `DUMP_TEMPLATE` to additionally increment `fnsize` according to the size of the chosen template. The calculation of `fnsize` is completely static, and is thus placed in the loader.

Now, we would like to use the calculated function size as the argument to `rts_alloc_code`. If we simply replace `rts_alloc_code(MAX_FN_SIZE)` by `rts_alloc_code(fnsize)`, however, the data specializer will cache the current value of `fnsize`, which is 0, in the loader and use this value when `rts_alloc_code` is called in the reader. Instead, at the beginning of the reader, we want to use the value of `fnsize` that is calculated at the end of the loader.

To solve this dilemma, we modify the run-time specializer to use an array `fns`, containing an entry for each specialized function. This array is used to communicate information from the end of the loader to the beginning of the reader. In the case of code compaction, `fns` contains the size of each specialized function. We declare to Tempo that the elements of `fns` are to be considered dynamic, so that the data specializer places references to these values in the reader.[2] Nevertheless, we can initialize an element of `fns` to a static value, by hiding the operation in a call to an external function (`update_fns`).[3] If we do not declare the side-effect to Tempo, this external function call is considered static, and placed in the loader by the data specializer. Not declaring the side effect violates the assumptions on which the correctness of the binding-time analysis of Tempo are based, but achieves the desired effect.

The compacting code-generation function for `interpret`, including the modified definition `DUMP_TEMPLATE` is shown in Figure 5. To avoid showing the definition of `DUMP_TEMPLATE`, we have renamed the original definition `ORIGINAL_DUMP_TEMPLATE`. New code is shown in italics. The result of applying data specialization to this run-time specializer is shown in Figure 6. This definition is similar to the result of data specialization of the original run-time specializer, as shown in Figure 4, except for the computation of the function size and the use of the `fns` array. Overall, the modifications are localized and easily automatable.

To carry out run-time specialization using the staged run-time specializer, we first apply the loader to the original (static) arguments and the cache, and then apply the reader to the cache. Figure 7 shows the entry point of the run-time specializer for `interpret`, assuming that `rts_interpret` is indeed the "main" function of the original run-time specializer.

[2] The binding-time analysis of Tempo treats arrays *monovariantly*, meaning that once the array contains any dynamic element, all elements are considered dynamic.

[3] After the generation of the loader and reader using data specialization, the call to `update_fns` can be replaced by a macro.

```
#define DUMP_TEMPLATE(code_ptr, template, size) {\
  ORIGINAL_DUMP_TEMPLATE(code_ptr, template, size) \
  fnsize += size; }

/* stores the size of each specialized function */
int fns[MAX_FNS_SIZE];

/* used to generate the unique integer identifier for
   each specialized function */
int cur;

void *rts_interpret(struct Exp *e) {
  int id = cur++;
  char *code_ptr = rts_alloc_code(fns[id]);
  char *spec_ptr = code_ptr;
  int fnsize = 0;

  DUMP_TEMPLATE(code_ptr, tmp1, 8);
  ...
  DUMP_TEMPLATE(code_ptr, tmp5, 12);

  update_fns(id,fnsize);  /* implements fns[id] = fnsize; */

  return (void *)spec_ptr;
}
```

Figure 5: Code-generation function for `interpret` modified to implement compaction. New code is shown in italics.

```
/* stores the size of each specialized function */
int fns[MAX_FNS_SIZE];

/* used to generate the unique integer identifier for each
   specialized function */
int cur;

void *rts_interpret_loader(struct Exp *e, void **Cache) {
  int id;
  int fnsize = 0;

  *((int *)Cache) = id = cur++;

  /* as before, with the calculation of fnsize added */
  *(Cache + 1) = tmp1;
  fnsize += 8;
  ...
  *Cache = tmp5;
  fnsize += 12;

  update_fns(id,fnsize);
  return Cache + 1;
}

void *rts_interpret_reader(void **Cache) {
  char *code_ptr = rts_alloc_code(fns[*((int *)Cache)]);
  char *spec_ptr = code_ptr;

  ORIGINAL_DUMP_TEMPLATE(code_ptr, *(Cache + 1), 8); /* as before */
  ...
  ORIGINAL_DUMP_TEMPLATE(code_ptr, *Cache, 12);

  return (void *)spec_ptr;
}
```

Figure 6: Data specialization of the modified run-time specializer for `interpret`

```
void *rts_interpret(struct Exp *e) {
  void *Cache[MAX_CACHE_SIZE];
  cur = 0;
  rts_interpret_loader(e,Cache);
  return rts_interpret_reader(Cache);
}
```

Figure 7: Entry point of the data specialized run-time specializer

4 Inter-Template Optimization, Using Interpretation

We have seen how to use data specialization to stage a run-time specializer into a phase that selects templates, followed by a phase that emits the corresponding code. While this approach is largely automatic and uses existing technology, it does require running the specializer twice at compile time — once to generate the run-time specializer, and a second time to stage it. Another approach is to modify the run-time specializer to created a record of the selected templates and the hole values, and then manually write an interpreter that generates the specialized code based on this record, which can be viewed as a sequence of code-generation instructions. While the interpreter need only be written once, the implementation of optimizations in this framework can be complex, and the entire implementation may need to be reorganized to accommodate new optimizations.

The interpreter approach allows any number of optimization phases to be inserted between the generation of the code generation instructions by the run-time specializer and the outputting of specialized code by the interpreter. To implement code compaction, we could add a phase that for each specialized function sums the size arguments of the `DUMP_TEMPLATE` instructions, allocates a buffer of the correct size, and then updates the `code_ptr` argument of each `DUMP_TEMPLATE` instruction accordingly. Because the interpreter is independent of the source program, we can implement more drastic modifications, such as rearranging or replacing templates. These operations can be difficult to express in the data specialization framework, because the dynamic code of each code-generation macro is explicitly encoded in the reader.

The extra flexibility of the interpretive approach comes at a performance penalty. There are three sources of inefficiency:

- Fragmentation in the code-generation buffer. The buffer of code-generation instructions used by the interpreter is allocated in the same way that the buffer for the specialized code of each function is allocated by the original run-time specializer. In contrast, new cached values are simply added to the end of the data specialization cache, as indicated by the `Cache` pointer, leaving no empty space. Note though that, the buffer of code-generation instructions is only used during specialization, and thus its structure has no impact on the performance of the specialized program.

- The replacement of constants and local variables by memory references. The code-generation macros each have some arguments that are constants, *e.g.* the template size argument to `DUMP_TEMPLATE`. The data specializer considers that these values are not worth storing in the cache, and simply places them explicitly in the reader. In the interpreter approach, all of the arguments of the code-generation macros must be stored in the code-generation instructions, including the constant arguments. Each code generation function uses some local variables that we have omitted from our presentation of run-time specialization for simplicity. These variables are similarly converted to memory references in the interpreter approach.
- The replacement of straight-line code by branches. Typically, a block of the original run-time specializer consists of a use of the macro `DUMP_TEMPLATE` followed by code to patch holes. The interpreter, on the other hand, consists of a loop containing a `switch` statement with a case for each instruction. Thus, every instruction involves a jump.

The benchmarks in Section 5 show that these problems cause stalls due to machine cache behavior and branch mispredictions. In our experiments, specialization using the interpreter approach is up to 70% slower than specialization of the original run-time specializer produced by Tempo.

5 Benchmarks

We now assess the performance of the compaction optimization using two benchmarks: an interpreter for the PLAN-P language [25, 26], and an image-manipulation program [22]. PLAN-P is a domain-specific language for implementing active network protocols. The image-manipulation program applies a convolution filter to blur an image. This program was originally written in Java, and translated to C using the Java bytecode-to-C compiler Harissa [17]. The PLAN-P interpreter is over 7000 lines of C code, while the image-manipulation program is approximately 4000 lines. The PLAN-P interpreter was shown to benefit significantly from run-time specialization. Nevertheless, some fine-tuning of the size of the buffer allocated for each specialized function was needed to obtain the best performance. The image-manipulation program was found to benefit significantly from compile-time specialization, but no improvement was obtained by run-time specialization in some cases. Neither program was developed specifically to test the approach presented in this paper.

5.1 Methodology

Experiments were carried out using a 167 MHz Sun Ultra1 running Solaris 2.7, having 128 MB of main memory.

The run-time specializers were compiled using `gcc` version 2.8.1 with the options `-O3 -mcpu=ultrasparc`. The templates for the image-filtering application were compiled using `gcc` version 2.8.1 with the options `-O2 -mcpu=ultrasparc -fno-delayed-branch`

	PLAN-P		blur	
input size	39 lines	161 lines	3 × 3 filter	7 × 7 filter
# functions	604	3009	39	199
spec. code size (bytes)	19872	98040	3552	17152
max fn. size (bytes)	260	884	1232	6512
ave. fn. size (bytes)	32	32	91	86
DS cache size (bytes)	18108	87012	1688	8616
max # interp entries/fn.	31	55	124	644
# templates	1466	1466	72	72

Table 1: Size of the experiments.

`-fno-thread-jumps`.[4] Because of a bug in more recent versions of `gcc`, the templates for the PLAN-P interpreter were compiled using `gcc` version 2.6.3 with the options `-O -fno-thread-jumps -fno-delayed-branch`.

Specialization times were measured using the ultrasparc system counters, using the *Perfmon* tool.[5] Execution of the specialized image-manipulation program was measured using `gethrtime`, because of its long duration. Execution of the specialized PLAN-P interpreter was measured using the ultrasparc system counters.

5.2 Performance evaluation

Table 1 presents various aspects of the size of the experiments. We specialize the PLAN-P interpreter with respect to two programs, one 39 lines (`plearn.pp`), and one 161 lines (`spym.pp`). We specialize the image-manipulation program with respect to two blurring convolution filters, a 3 × 3 filter and a 7 × 7 filter. On these programs, compaction improves the performance of the specialized program by a factor of more than 2 for the PLAN-P interpreter, and more than 4 for the image-manipulation program. Note that the performance improvement comes from the improved layout of the specialized code in memory, rather than from an improvement in the process of allocating this memory itself. Memory allocation is performed at specialization time, and thus its cost has no impact on the performance of the specialized code.

Compaction is most useful when the specialized program consists of many functions of widely varying size. In this situation, the memory allocation strategy described in Section 2.2 suggests that a non-compacting specializer should conservatively allocate a large buffer for every specialized function, producing code that is widely scattered in memory. Specializing the PLAN-P interpreter generates a large number of specialized functions. The largest

[4] Inlining, as provided by `-O3`, is not interesting for the templates. The options `-fno-delayed-branch -fno-thread-jumps` suppress optimizations that make it difficult to extract the compiled templates from the object file.

[5] URL: `http://www.cps.msu.edu/~enbody/perfmon.html`

function is up to over 27 times the average function size. The image-manipulation program is overall a smaller example than the PLAN-P interpreter, but exhibits an even more extreme range of function sizes; for a 7×7 filter the size of the largest function is over 75 times the average function size.

The main overhead of the data specialization approach is the size of the cache. In our experiments, the cache is quite large, approaching the size of the specialized code. Because the elements of the cache are mostly accessed sequentially, however, this structure has good locality. Furthermore, preliminary experiments show that it should be possible to almost halve the size of the data specialization cache by improving the caching strategy of Tempo.

The last two lines of Table 1 describe the memory requirements of the interpretive approach: the maximum number of code generation instructions per function and the size of the table indicating the most recent address at which a copy of each template was emitted. Each code generation instruction uses 24 bytes. The size of the table of templates is determined by the number of templates in the program, which is fixed at compile time.

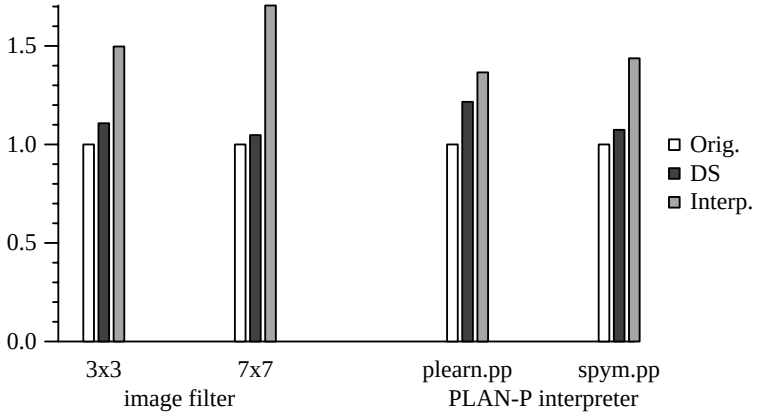

Figure 8: Specialization time using the data-specialization and interpreter approach, as compared to the original run-time specializer

Figure 8 compares the specialization time using the data-specialization-based approach (DS) and the interpretive approach (Interp.) to that of using the original run-time specializer (Orig.). We include both user and system time. The size of the function buffers used by the original run-time specializer (*i.e.*, the value of `MAX_FN_SIZE`) has little effect on the specialization time. Thus, for each program, we use the smallest size that is sufficient for both test cases.

The data-specialization approach has little overhead with respect to the original specializer. For specialization of the PLAN-P interpreter with respect to the program `plearn.pp`, the data specialization approach is 22% slower than the original specializer. In other cases, the data specialization approach is less than 10% slower. The object code of the data-specialized run-time specializer for the PLAN-P interpreter is approximately twice as large

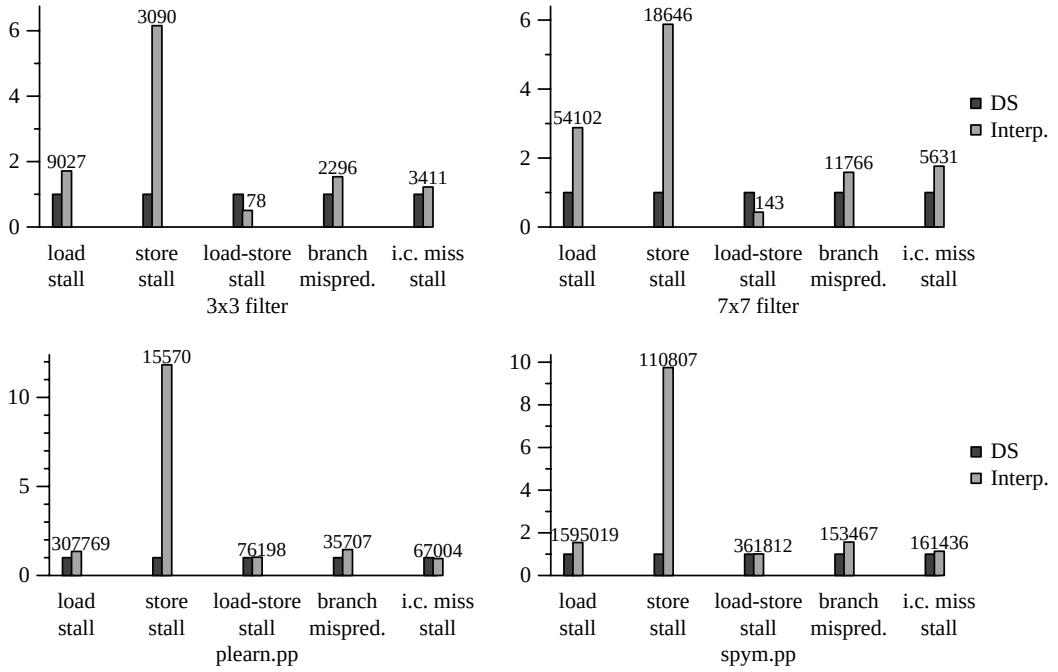

Figure 9: CPU stalls of the interpreter approach, as compared to the data-specialization approach. The height of the bar for data specialization is always 1. The numbers on top of the bars for the interpreter indicate the absolute number of stalls.

as the object code of the original run-time specializer. Consequently, there are significantly more page faults during execution to load this code. These page faults seem to have a significant impact on the performance of the specialization of `plearn.pp`. Reducing the number of values that are cached by the loader, as described above, should also reduce the code size, and thus the page fault overhead.

Specialization using the interpreter approach is up to 70% slower than specialization using the original specializer. In Section 4, we observed that the interpreter approach performs more memory references than the data-specialization approach, with poorer locality, and that the interpreter approach performs more branches. To assess the impact of these features, Figure 9 compares the number of CPU stalls in user mode due to caches misses, load-store dependences, and branch misprediction. These figures clearly show that the data specialization approach has significantly better cache behavior than the interpreter approach. For the image-filtering example, the interpreter approach suffers from significantly more branch mispredictions as well.

Finally, we consider the performance of the specialized programs. Figure 10 shows the speedup produced by specializing the image-manipulation program to a 3×3 blurring filter

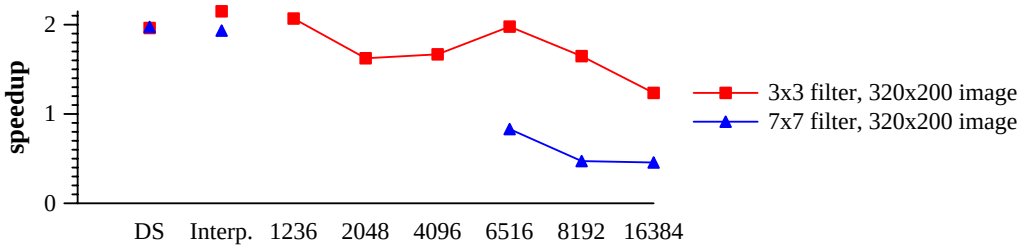

Figure 10: Speedup obtained by specializing the image-manipulation program (buffer sizes in bytes)

and a 7×7 blurring filter. In each case, we apply the filter to a 320×200 pixel image. The data-specialization-based and interpreter-based approaches both produce compact programs, and specialization to either filter size produces a speedup of about 2 over the unspecialized program. The remaining speedups are for Tempo's original implementation of run-time specialization with the indicated buffer size for each specialized function. We test the minimum possible buffer sizes, and various larger sizes that are powers of two. Using Tempo with a relatively small buffer size, the speedup varies between 1.5 and 2 for a 3×3 filter. For a 7×7 filter, however, the smallest buffer sizes are not large enough for the largest specialized function. For this filter size, the result of specializing using Tempo is always slower than the original program. The compacted code is over four times faster than the code produced by Tempo using an estimated function size of 8K or 16K bytes.

Figure 11 shows the speedup produced by specializing the PLAN-P interpreter with respect to the program `plearn.pp`.[6] Again, the data points to the right of the interpreter approach are for Tempo, with various function buffer sizes. Here specialization always produces a speedup. Nevertheless, the compacted specialized program is over twice as fast as the specialized program generated by Tempo using a function buffer size of 2K bytes. Analysis of the number of stalls due to instruction cache misses show that these steadily increase as more space is allocated for each function.

6 Related work

We briefly review the history of data specialization, compare our approach to another optimization of run-time specialization in Tempo, and compare our approach to another template-based run-time specializer performing run-time optimizations.

Data specialization

Manual data specialization is a fairly common program optimization. For example, the implementation of the Fast Fourier Transform available from `netlib` consists of an initialization

[6] For technical reasons, it was not possible to meaningfully test the `spym.pp` program.

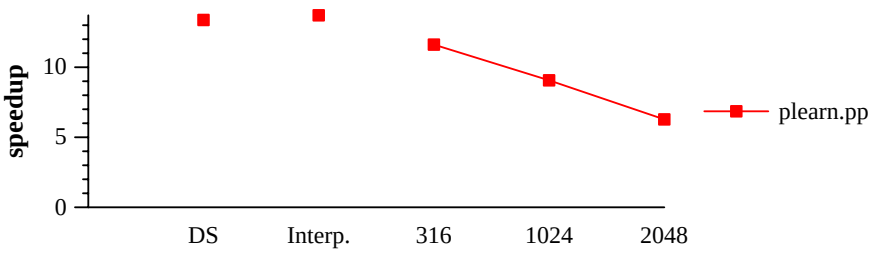

Figure 11: Speedup obtained by specializing the PLAN-P interpreter to `plearn.pp` (buffer sizes in bytes)

procedure that stores the needed sine and cosine values in an array followed by a procedure that computes the Fast Fourier Transform using these stored values [18]. Similarly, the implementation of `jpeg` compression available in the Spec benchmark suite precomputes values used in converting between color formats [6].

Automatic data specialization was first developed by Barzdins and Bulyonkov [2], and described and extended by Malmkjær [15]. These approaches are more complex than the implementation of data specialization used here. In particular, they perform memoization of the data specialization cache. Memoization reduces the size of the cache, but it is not obvious how to implement it efficiently enough for use in the context of run-time specialization.

Knoblock and Ruf implement data specialization for a subset of C and investigate its use in an interactive graphics application [12]. They consider several techniques to limit the size of the data specialization cache, including the use of static-single-assignment (SSA) form to confine the positions at which static values are cached to join points. This technique could be useful in our approach. They also observe that data specialization has very low overhead.

Chirokoff *et al.* compare the benefits of program and data specialization, and show that it can be useful to combine these techniques [3]. Our implementation of data specialization in Tempo builds on that of Chirokoff.

In our approach, the loader computes all of the static values in order to select the templates. Shinjo has suggested, in private communication, that computing all of the static values before generating any specialized code could also allow more optimization of the templates at compile time, including optimizations that move holes between templates [23].

Incremental specialization

Incremental specialization is the process of specializing a program gradually, as information about the static inputs becomes available [5, 7]. Each phase generates an increasingly specialized specializer, with the goal of reducing the overall specialization time. Marlet, Consel, and Boinot have proposed to implement incremental run-time specialization by applying standard run-time specialization to the original run-time specializer to produce an incremental run-time specializer [16]. This process can be iterated to provide any number of levels of incremental specialization.

Our approach is also a form of incremental specialization. In particular, we specialize the run-time specializer with respect to the static inputs of the source program, leaving the parameters of the code-generation operations unknown. To specialize the run-time specializer, we use data specialization rather than run-time specialization. The low overhead of data specialization makes it more appropriate to a situation where the second stage (*i.e.* the reader for data specialization and the specialized specializer for run-time specialization) is used only once. Our approach is targeted towards improving the performance of the specialized code, rather than reducing specialization time. Thus, our approach critically relies on the ability to pass global information from one stage to the next. This functionality could also be useful in incremental specialization.

Optimizing run-time specialized code

DyC, developed by Grant *et al.*, also performs run-time program specialization [8, 9]. Like Tempo, DyC constructs the specialized code based on templates that are compiled before specialization, at compile time. Unlike Tempo, which emits an entire template at once, DyC generates specialized code one instruction at a time. This strategy facilitates run-time instruction-level optimizations such as strength reduction. Nevertheless, specialization with these optimizations is substantially more expensive than specialization using Tempo, or our approach.

A predecessor of DyC [1] separated the run-time specialization actions into *set-up code*, which generates a sequence of directives, and a *stitcher*, which follows these directives to generate the specialized code. This approach is quite similar to the interpreter approach, described in Section 4.

7 Conclusion and Future Work

Optimizing specialized code at run time must respect a delicate balance between performance improvement and specialization cost. We have shown that by separating the run-time specializer into two phases using data specialization, we can optimize run-time specialized code based on global analysis of the selected templates with little overhead.

Using our approach, we have shown how to implement compaction of run-time specialized code. The implementation requires only minor, local modifications of the run-time specializer. Despite the simplicity of the optimization, in our experiments we find that it can improve performance of the specialized program by up to a factor of 4. In future work, we plan to consider what other optimizations can be expressed naturally in the data specialization framework. In particular, we have begun to consider the problem of function inlining, taking into account the size of each function and register availability.

Acknowledgements

We would like to thank Olivier Danvy, Luke Hornof, Renaud Marlet, and Ulrik Pagh Schultz for helpful comments on this paper, and Charles Consel for suggesting the interpreter approach.

References

[1] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In PLDI'96 [20], pages 149–159.

[2] G. J. Barzdins and M. A. Bulyonkov. Mixed computation and translation: Linearisation and decomposition of compilers. Preprint 791 from Computing Centre of Sibirian division of USSR Academy of Sciences, p.32, Novosibirsk, 1988.

[3] S. Chirokoff, C. Consel, and R. Marlet. Combining program and data specialization. *Higher-Order and Symbolic Computation*, 12(4):309–335, December 1999.

[4] 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.

[5] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 44–46, Copenhagen, Denmark, June 1993. ACM Press. Invited paper.

[6] Standard Performance Evaluation Corporation. SPEC95.
URL: `http://www.specbench.org`.

[7] R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10:113–158, 1997.

[8] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. Technical Report UW-CSE-97-03-03, Department of Computer Science and Engineering University of Washington, May 1999. To appear in Theoretical Computer Science.

[9] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in DyC. In PLDI'99 [21], pages 293–304.

[10] L. Hornof and T. Jim. Certifying compilation and run-time code generation. *Higher-Order and Symbolic Computation*, 12(4):337–376, December 1999.

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

[12] T.B. Knoblock and E. Ruf. Data specialization. In PLDI'96 [20], pages 215–225. Also TR MSR-TR-96-04, Microsoft Research, February 1996.

[13] J.L. Lawall. Faster Fourier transforms via automatic program specialization. In J. Hatcliff, T.Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, volume 1706 of *Lecture Notes in Computer Science*, Copenhagen, Denmark, 1999. Springer-Verlag.

[14] P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI'96 [20], pages 137–148.

[15] K. Malmkjær. Program and data specialization: Principles, applications, and self-application. Master's thesis, DIKU University of Copenhagen, August 1989.

[16] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In PLDI'99 [21], pages 281–292.

[17] G. Muller and U. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, pages 44–51, March 1999.

[18] Netlib.
URL: `http://www.netlib.org/cgi-bin/netlibget.pl/bihar/{dcffti,dcfftf}.f`.

[19] 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.

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

[21] *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*, Atlanta, Georgia, USA, 1–4 May 1999.

[22] 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.

[23] Y. Shinjo. Personal communication. July 29, 1999.

[24] S. Thibault, L. Bercot, C. Consel, R. Marlet, G. Muller, and J. Lawall. Experiments in program compilation by interpreter specialization. Research Report 3588, INRIA, Rennes, France, December 1998.

[25] 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.

[26] 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.

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