Specialization for Applications Using Shared Libraries

Ping Zhu Siau-Cheng Khoo
Department of Computer Science, National University of Singapore
zhuping/khoosc@comp.nus.edu.sg

Abstract

Shared libraries have been prevalently deployed in many systems and application domains in the last decade. Their ubiquity depends largely on their allowance for *sharing*; ie., a single copy of the library is used across multiple applications running on a single system. Unfortunately, traditional partial evaluation does not consider the sharing issue. Specifically, libraries are treated as if they are statically linked, and sharing is not preserved during the process of creating and running specialized libraries, especially across different specialized applications. In this paper, we propose a methodology for *run-time specialization* that aims to *maximize* sharing during the whole specialization process. Specifically, we advocate a stand-alone specialization of shared libraries (independent of their clients), and propose a specialization mechanism which enables sharing of run-time specialized library code both within a specialized application and across multiple specialized applications. Our proposal includes a novel *static transformation* that constructs a generic specialization library/component, aiming to eliminate code duplication arising at compile-time, as well as a novel *run-time specialization* that eliminates code duplication occurring at run-time.

Categories and Subject Descriptors D.1.2 [*Automatic Programming*]: Program transformation; D.2.m [*Miscellaneous*]: Reusable software; D.3.4 [*Processors*]: Run-time environments

General Terms Languages

Keywords Run-time specialization, Shared Libraries

1. Introduction

A library is a collection of subroutines (functions, methods or procedures) that can be reused to develop various applications. Libraries are commonly categorized into two types *static libraries* and *shared libraries*, according to the way they are linked with applications. In the last decade, there has been a proliferation in deployment of shared libraries in implementations of essential services in many systems and application domains. Their ubiquity depends largely on their allowance for *sharing*; ie.,

- At compile-time, there is one single copy of a shared library in the disk. Applications that use a shared library contain only references to the library. The binary of a shared library is not included in the binary of the compiled application at link-time. Thus the size of the executables is considerably smaller than that of the executables obtained from using static libraries;
- At run-time, all executables of applications refer to the same executable copy of the library. Consequently, maximal reuse of shared libraries is achieved.

Clearly, *sharing* enables reduction in both disk and memory usages. Furthermore, it facilitates bug fixing for shared libraries. Specifically, since only one copy of the shared library is maintained, all applications that use the library immediately benefit from the fix, without having to be rebuilt.

Traditional specialization techniques, such as partial evaluation, have been designed for specializing applications using *static libraries*. When dealing with applications that use shared libraries, the technique is oblivious to sharing of these shared libraries.

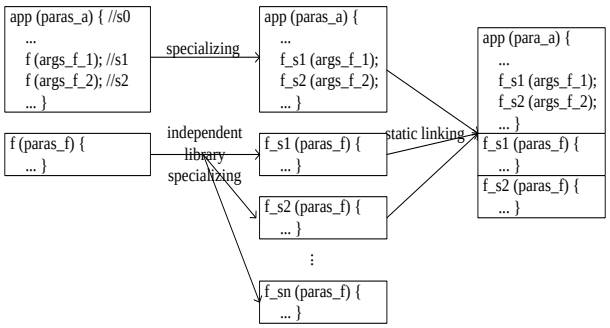

Figure 1. The typical specialization process of traditional specialization framework

Consider a typical specialization process, as shown in Figures 1. Two specialization contexts `s1` and `s2` are derived at library `f`'s two call sites respectively, when specializing an application *app* with respect to its initial binding-time information `s0`. The library `f`, on the other hand, is specialized independently, i.e. free from the specialization contexts established only within any specific applications. Different specialized libraries, `f_s1` and `f_s2` are generated with respect to `s1` and `s2`. In the latter stage of the specialization process, specialized libraries `f_s1` and `f_s2` are statically linked to the specialized application to produce a stand-alone executable. This forces extra copies of specialized libraries to be created, nullifying sharing across multiple specialized applications. Furthermore, in the case of run-time specialization – as we will elaborate later, the code of a specialized library may also be *replicated in multiple uses* within the specialized application.

In this paper, we describe a novel framework for specializing applications which use *shared libraries*, with the intention to *preserve sharing* of shared libraries, and *minimize footprints* of specialized shared libraries during the execution of specialized applications. The footprint of a specialized library is a specialized code

PEPM'08, January 7–8, 2008, San Francisco, California, USA.

produced by executing the specialized library with respect to the concrete specialization values which are available at run-time.

The first step to ensure that specialization preserves sharing is to enable independent specialization of shared libraries. This has been addressed by existing specialization techniques, such as *specialization module* proposed by Le Meur et al in [14, 8], *specialization class* proposed by Consel et al in [10, 19], *component specialization* introduced by Bobeff and Noyé in [3, 4] and *profitability-oriented component specialization* advocated by the authors earlier in [21]. Profitability-oriented component specialization aims at discovering all possible use contexts of a library, the existence of which ensure profitable specialization. A specialization is considered profitable if it can eliminate some conditional tests during its process. Consider the function `power` defined in Figure 2. It computes the base `b` to the power `e`.

```
int power (int b, int e) {
  int z ;
  if (e == 0)
    return 1 ;
  else {
    z = b * power(b, e - 1) ;
    return z ;
  }
}
```

Figure 2. Function `power`

The complete set of specialization scenarios derived for function `power` in profitability-oriented component specialization which enables us to prepare the function `power` for all possible cases of specialization, are listed in Figure 3[1]

$$\begin{array}{ll} \mathtt{p_{ss1}}: & bt_b = \mathtt{S} \wedge bt_e = \mathtt{S} \\ \mathtt{p_{ss2}}: & bt_b = \mathtt{D} \wedge bt_e = \mathtt{S} \\ \mathtt{p_{ss3}}: & bt_b = \mathtt{D} \wedge bt_e = \mathtt{D} \end{array}$$

Figure 3. Specialization scenarios derived for function `power` in PCS

Note that:

- `S` and `D` are binding-time constants representing *static* and *dynamic* values respectively;
- bt_b and bt_e are binding-time variables pertaining to the function parameters `b` and `e` respectively.

Independent specialization of a library generates various specialized libraries aiming for being used under different use contexts. As we employ run-time specialization [9] in our specialization process, every specialized library is constructed from a template file and a run-time specializer. Different specialized libraries may share identical templates, leading once again to code (template) duplication. Figure 4 presents two template files, which are constructed by Tempo and belong to the specialized libraries `power_pss2` and `power_pss3` respectively. For simplicity of presentation, some statements in the template file are omitted. The duplicated template code is underlined in the two template files.

To eliminate such duplication, we propose a novel *static transformation* which converts a source library into a *generic specialization component* – `GSC` for short. A `GSC` contains a *global template repository* sharable among different *local* run-time specializers. This enables the `GSC` to minimize code duplication at compile-time.

```
/** Template file of power_pss2 **/

extern int tmp_power (int b) {
/** Beginning of t0
 int z ;
/** End of t0
 if (N0){
/** Beginning of t1
  return 1 ;
/** End of t1
 else {
/** Beginning of t2
  z = b * ((int(*)(int))(&CH0))(b) ;
  return z ;
/** End of t2
/** Beginning of t3
}
/** End of t3
```

```
/** Template file of power_pss3 **/

extern int tmp_power (int b , int e) {
 int z ;
 if (e == 0){
  return 1 ;
 else {
  z = b * ((int(*)(int, int))(&CH0))(b , e - 1) ;
  return z ; }
}
```

Figure 4. Two template files adapted from Tempo

We employ run-time specialization (instead of compile-time specialization) in order to deal with the intricacy associated with maintaining dynamic linkage of specialized libraries. Specifically, it enables *minimal library footprints* to be created during execution of specialized applications, a desired feature found in many executions of shared libraries. To facilitate understanding this technical detail, we give an overview of traditional run-time specialization below.

1.1 Background of Run-time Specialization

Partial evaluation is commonly categorized into two types *compile-time specialization* and *run-time specialization*, according to when concrete specialization values are available. For run-time specialization, these concrete specialization values are available until run-time. Specialized libraries produced by run-time specialization normally exist in *generating extension* form. A *generating extension*, denoted as $\mathtt{f}^{\mathtt{ge}}_{\mathtt{ss}}$, is a code produced by specializing a library `f` with respect to its binding-time parameter information `ss` using run-time specialization.

There are two notable run-time specialization systems for imperative languages, namely Tempo [9, 15, 1] and DyC [12, 11]. Generating extensions produced by these run-time specialization systems are commonly comprised of two parts:

- A *template file*. It contains several *source templates*, each of which is (partially-)*dynamic* program fragment parameterized by *static* expressions. Each template is delimited by labels with artificial gotos to make sure the templates will be considered in isolation by the compiler. The template file is compiled into a binary. With the help of a template compiler, the information about the size and location of each compiled source code (in the

[1] Note that only *profitable* specialization scenarios and totally-dynamic scenario are included.

literature it is termed as *object template*) is extracted at compile-time.

- A *run-time specializer*. It not only represents the computations that are completely decided by the *static* parameters, but also contains operations to manipulate object templates. The operations include selecting templates, filling static values in the template (which is termed as *template instantiation*), and dumping instantiated templates through executing *memory copy* instructions.

The compiled template file and run-time specializer are linked together to create a generating extension of a library. At run-time, a footprint is firstly created by executing the generation extension with respect to concrete values of static inputs. More specifically, a memory block is dynamically allocated to store all the templates that are selected, dumped and instantiated by the run-time specializer. Every template captured by the template file is a candidate for dumping. The reason for dumping templates is that instantiated templates can be different from their original ones, because the former replace holes in the latter by static values. In the second phase at run-time, the footprint is executed further with respect to the concrete values of remained *dynamic* parameters to produce final results.

1.2 Code Duplication Arising at Run-time

Consider again the template file of `power_pss2` depicted in Figure 4. There are four templates detected by Tempo, as delimited by comments. Suppose `power_pss2` is called with static input 2. The memory block dynamically allocated by traditional run-time specialization for the footprint $\texttt{power}_2^{\texttt{fp}}$ comprises the following sequence of templates:

$$[\texttt{t}_0; \texttt{t}_2^1; \texttt{t}_0; \texttt{t}_2^0; \texttt{t}_0; \texttt{t}_1; \texttt{t}_3; \texttt{t}_3; \texttt{t}_3]$$

where $\texttt{t}_2^1$ and $\texttt{t}_2^0$ are two templates instantiated from original template $\texttt{t}_2$ within which the static expression is filled with 1 and 0 respectively. We notice that templates $\texttt{t}_0$ and $\texttt{t}_3$ stored by the memory block allocated for $\texttt{power}_2^{\texttt{fp}}$ are identical to the original templates stored by the memory block allocated for the template file of `power_pss2`. This example illustrates that the *dumping all templates* strategy adopted by traditional run-time specialization is oblivious to code duplication. Specifically,

1. Templates $\texttt{t}_0$ and $\texttt{t}_3$ appear multiple times in $\texttt{power}_2^{\texttt{fp}}$.
2. Suppose `power_pss2` is called with static input 3 elsewhere. Another memory block dynamically allocated for the footprint $\texttt{power}_3^{\texttt{fp}}$ comprises the following sequence of templates:

   $$[\texttt{t}_0; \texttt{t}_2^2; \texttt{t}_0; \texttt{t}_2^1; \texttt{t}_0; \texttt{t}_2^0; \texttt{t}_0; \texttt{t}_1; \texttt{t}_3; \texttt{t}_3; \texttt{t}_3; \texttt{t}_3]$$

   Here, $\texttt{t}_2^2$, $\texttt{t}_2^1$ and $\texttt{t}_2^0$ are three templates instantiated from original template $\texttt{t}_2$ within which the static expression is filled with 2, 1 and 0 respectively. We notice that templates $\texttt{t}_0$ and $\texttt{t}_3$ are duplicated in both $\texttt{power}_2^{\texttt{fp}}$ and $\texttt{power}_3^{\texttt{fp}}$.

In this paper we propose a novel *run-time specialization* to address code duplication problem arising at run-time. Specifically, we adopt a different strategy for *template dumping* that minimizes the footprint of specialized shared libraries in the specialized applications by reducing the number of duplicated object templates created in the dynamically allocated memory block. With this new strategy, we propose a run-time specialization mechanism to manage the new structure of the footprint.

1.3 Contributions

We summarize the technical contributions of this paper as follows: We propose a methodology for *run-time specialization* that aims at *maximizing* sharing during the entire specialization process. Specifically,

- We introduce a novel *static transformation* to construct a *generic specialization component* for a shared library, aiming at eliminating code duplication at compile-time.
- We devise a novel *run-time specialization* that eliminates code duplication at run-time, thus minimizing the footprints of specialized shared libraries during application executions.

The rest of this paper is outlined as follows: Section 2 introduces the core language and notational conventions used in this paper. In Section 3 we elaborate our proposals to address the code duplication problem arising at compile-time and run-time, named `GSC` *construction* and a novel run-time specialization to executing a `GSC`. The algorithms to implement our proposals is elaborated in Section 4. A complete picture of the novel specialization framework for specializing application using shared libraries is sketched out in Section 5. Before concluding, we survey related work in the areas of component specialization and run-time specialization in Section 6.

2. Languages and Notations

In this paper we choose the implementation model of *shared libraries* to be C function definitions. The terms *shared libraries* and *functions* are thus used interchangeably. The core language is a subset of C language. Its abstract syntax is defined in Figure 5.

v	$\in$	**Var**	Variables
f, g	$\in$	**FName**	Function names
n	$\in$	$\mathcal{N}$	Numerals
b_{op}	$\in$	**BOp**	Binary operators
	::=	$+ \mid - \mid * \mid / \mid == \mid != \mid < \mid > \mid >= \mid <= \mid \&\& \mid \mid\mid$	
e	$\in$	**Exp**	Expressions
	::=	$n \mid v \mid f\ (e_1, \ldots, e_n) \mid e_1\ b_{op}\ e_2$	
s	$\in$	**Stat**	Statements
	::=	$s_1; s_2 \mid \texttt{while}\ e\ s \mid \texttt{return}\ e \mid v = e \mid \texttt{if}\ e\ s_1\ \texttt{else}\ s_2$	
$decl$	$\in$	**Decl**	Declarations
	::=	$\texttt{int}\ v$	
fd	$\in$	**FDef**	Function definitions
	::=	$\texttt{int}\ f\ (decl^+)\ \{decl^*\ ;\ s\}$	

Figure 5. Syntax of the core language - A C subset

The core language excludes features such as pointers, compound data structures, global variables, etc. The evaluation strategy for function calls is restricted to call-by-value and every function definition must return a value.

For ease and clarity of presentation, we define several notations and explain them as follows:

- $\texttt{f}^{\texttt{gsc}}$: A `GSC` of a library `f`
- $\texttt{f}_{\texttt{ss}}^{\texttt{aa}}$: An action annotated code of the library `f` constructed with respect to a specialization scenario `ss`
- $\texttt{f}_{\texttt{ss}}^{\texttt{ge}}$: A generating extension of the library `f` constructed with respect to a specialization scenario `ss`
- $\texttt{f}_{\texttt{ss}}^{\texttt{rts}}$: A run-time specializer of the library `f` constructed with respect to a specialization scenario `ss`
- $\texttt{f}_{\texttt{val}_\texttt{s}}^{\texttt{fp}}$: A footprint of the library `f` constructed by executing its corresponding generating extension with respect to the values of static variables $\texttt{val}_\texttt{s}$

3. Our Approach

In this section, we introduce our proposal to construct a *generic specialization component* (GSC for short) for a shared library. We also describe the run-time properties of a GSC introduced by the novel run-time specialization.

A GSC $\mathtt{f}^{\mathtt{gsc}}$ is constructed from a library f. Formally, given a specialization scenario ss synthesized by $\mathtt{f}^{\mathtt{gsc}}$, concrete values $\mathtt{val}_\mathtt{s}$ and $\mathtt{val}_\mathtt{d}$ for static and dynamic inputs to f respectively conforming to ss, the following holds:

$$\begin{array}{rcl} \mathtt{f}^{\mathtt{gsc}}(\mathtt{ss}) & = & \mathtt{f}^{\mathtt{ge}}_{\mathtt{ss}} \\ \text{where } \mathtt{f}^{\mathtt{ge}}_{\mathtt{ss}}(\mathtt{val}_\mathtt{s}, \mathtt{val}_\mathtt{d}) & = & \mathtt{f}^{\mathtt{fp}}_{\mathtt{val}_\mathtt{s}}(\mathtt{val}_\mathtt{d}) \\ & = & \mathtt{f}(\mathtt{val}_\mathtt{s}, \mathtt{val}_\mathtt{d}) \\ \mathtt{f}^{\mathtt{fp}}_{\mathtt{val}_\mathtt{s}} & = & \mathtt{f}^{\mathtt{rts}}_{\mathtt{ss}}(\mathtt{val}_\mathtt{s}) \end{array}$$

$\mathtt{f}^{\mathtt{gsc}}$ can thus be deemed as a *meta* generating extension because $\mathtt{f}^{\mathtt{gsc}}(\mathtt{ss})$ returns a generating extension for the specialization scenario ss.

3.1 Reducing Code Duplication at Compile-Time: Building a Template Repository

As indicated in Section 1, the main source of code duplication arising at compile-time is related to duplication of identical templates scattered among different template files possessed by different generating extensions. In [21] we conducted a preliminary investigation to address this code duplication problem at the action-annotated code level: We constructed a web of action annotated codes in which *sharable action-annotated code fragments* are identified and shared by *all* action-annotated codes generated with respect to different specialization scenarios. Essentially, *sharable action-annotated code fragments* are identical action-annotated codes created required by different specialization scenarios.

Action annotations attached to program constructs are clear instructions dictating the construction of specialized codes, which have been supported by several popular partial evaluators such as Schism [5, 6] and Tempo [7]. The action domain $\mathbf{AC_{val}}$ used in this paper comprises four values ev, rd, rb, and id, representing the actions *evaluation, reduce, rebuild* and *identity* respectively.

We enhance our earlier work on eliminating code duplication in a web of action-annotated codes by constructing a GSC, aiming at eliminating duplicated *templates* at compile-time. We introduce a novel static transformation performed over a set of action-annotated codes, detecting *sharable templates* by looking up each action-annotated statement. Sharable templates are derived from identical action-annotated statements. Thereafter, instead of creating different template files separately for different specialization scenarios, we build a *global template repository* which captures all distinct templates possessed by the different template files.

Figure 7 presents an example of a template repository derived from three action annotated variants of function power depicted in Figure 6. For simplicity of presentation, we omit those action annotations which can be inferred easily. For example, if an expression or a statement is annotated by ev (or id), the action annotations of all the nested program constructs are also ev (or id). The algorithm to derive the template is presented in Section 4

We leverage the traditional two-part structure of generating extension in constructing the GSC. A GSC $\mathtt{f}^{\mathtt{gsc}}$ is composed of a set of *local run-time specializers* and a *global template repository*; the latter is shared by all local run-time specializers. Each *local* run-time specializer pertains to specialization of the library with respect to a distinct specialization scenario.

Figure 8 highlights the differences between the traditional approach and our new approach in organizing the generating extensions constructed for the function power with respect to three specialization scenarios[2]. In the traditional approach, three generating extensions are constructed separately, each of which contains a pair of a run-time specializer and a template file. In our new approach, we construct three local run-time specializers rts_1', rts_2' and rts_3' each of which is responsible for encoding the static computations and manipulating templates stored in the sharable template repository. The run-time specializers constructed in our approach are different from those produced by traditional run-time specialization, since we adopt different run-time specialization mechanisms in manipulating templates, which will be elaborated in the following section.

It is worth noticing that a GSC is constructed with the aim to handle *all* uses of the specialized shared library, rather than a specific application. Although this implies that a GSC can contain

```
/** power^aa_pss1 **/

int power (int b^ev, int e^ev) {
 (int z)^ev ;
 if (e == 0)^ev
  (return 1)^ev ;
 else {
  (z = b * power(b, e - 1))^ev ;
  (return z)^ev ;
 }
}
```

```
/** power^aa_pss2 **/

int power (int b^id, int e^ev) {
 (int z)^id ;
 if (e == 0)^ev
  (return 1)^id ;
 else {
  (z = b^id * power(b^id, e^ev - 1)^rb)^rb ;
  (return z)^id ;
 }
}
```

```
/** power^aa_pss3 **/

int power (int b^id, int e^id) {
 (int z)^id ;
 if (e == 0)^id
  (return 1)^id ;
 else {
  (z = b * power(b, e - 1))^id ;
  (return z)^id ;
 }
}
```

Figure 6. Action-annotated codes constructed for function power w.r.t. three specialization scenarios

Label	Template
p_{t0}	int z ;
p_{t1}	z = b * power(b, h0) ;
p_{t2}	return z ;
p_{t3}	if (e ==0)
p_{t4}	return 1 ;
p_{t5}	z = b * power(b, e-1) ;

Figure 7. A global template repository constructed for three action-annotated codes of function power

[2] For reasonable comparison, the template files produced by traditional run-time specialization, such as Tempo, are represented in terms of the templates produced by our approach

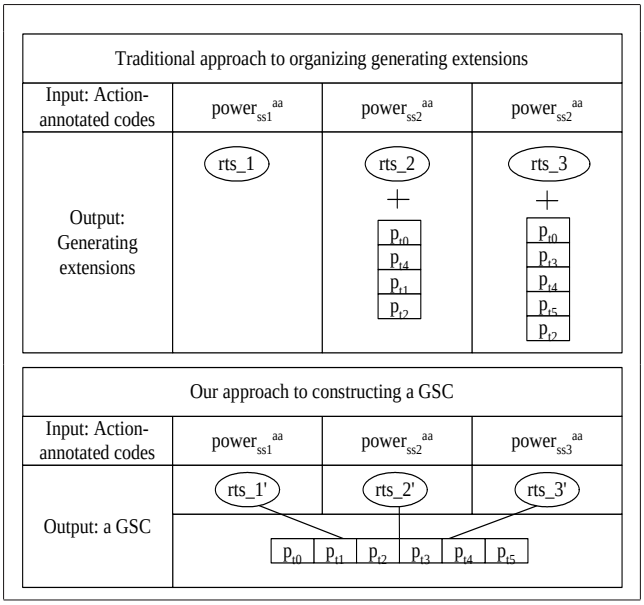

Figure 8. Two approaches to organizing generating extensions constructed for function `power` w.r.t. three specialization scenarios

several local run-time specializers, each of which is pertaining to specialization of a library with respect to a distinct specialization scenario, the size of the `GSC` is curbed by the fact that: (1) the number of such different specialization scenarios are limited only to those *profitable* ones, as explained in [21], and (2) the sharing of templates in the global template repository is enable, as explained above.

Our `GSC` construction not only maximizes sharing at compile-time, it also paves the way for maximizing sharing at run-time since `GSC` exists in shared library form and it is amenable to *sharing* at run-time.

3.2 A Novel Approach to Run-time Specialization

As introduced in Subsection 1.1, in the first phase at run-time, a footprint is created from the generation extension returned by $\mathtt{f}^{\mathtt{gsc}}(\mathtt{ss})$ with respect to concrete values of static input. The objective of our novel approach to run-time specialization is to minimize the footprints of specialized shared libraries during execution. We now describe how our approach achieves this objective.

3.2.1 Dumping Fewer Templates

As indicated in Section 1, the issue of run-time code duplication mainly stems from the traditional strategy of *dumping all templates into the dynamically allocate memory block*. Under this strategy, even *totally dynamic templates*, which do **not** contain any embedded static holes and remain unchanged during instantiation, will be dumped, and possibly multiple times. Thus we can maximize sharing by choosing not to dump totally dynamic templates into the dynamically allocated memory block since they can also be located in the memory block allocated for the generating extension.

Consequently, in the first phase, only *hybrid* templates – those which contain at least one static hole to be filled by concrete values of *static* expressions – are dumped into a dynamically allocated memory block after being instantiated by filling concrete values into their holes. Operationally, dumping of hybrid templates is performed by dumping operations instrumented in the local run-time specializers of a `GSC`. On the other hand, totally dynamic templates need not be directed by dumping operations. By adopting this novel template dumping strategy, multiple occurrences of identical templates in the dynamically allocated memory block, as described in Subsection 1.2, can be replaced by references to the corresponding single copy of the templates residing in the global template repository of a `GSC`.

3.2.2 Connecting Templates

Figure 9 shows the layouts of memory blocks allocated for storing the footprints of `power` generated with respect to input 2 by the two approaches. $\mathtt{p}^1_{\mathtt{t1}}$ and $\mathtt{p}^0_{\mathtt{t1}}$ are two templates instantiated from original hybrid template $\mathtt{p}_{\mathtt{t1}}$ within which the static expression is filled with 1 and 0 respectively. The traditional approach allocates a memory block at run-time to store all the templates needed for the footprint. On the other hand, in our approach, the templates are split and kept in two separate memory blocks: (1) A dynamically allocated memory block keeps the instantiated hybrid templates which are instantiated and dumped from global template repository of $\mathtt{power}^{\mathtt{gsc}}$; (2) A memory block keeps the global template repository. Under this approach, the footprint is produced by linking templates from two separate memory blocks. Referring to Figure 9, the footprint is constructed by linking the hybrid templates in the dynamically allocated memory block and the shaded totally dynamic templates found in the template repository.

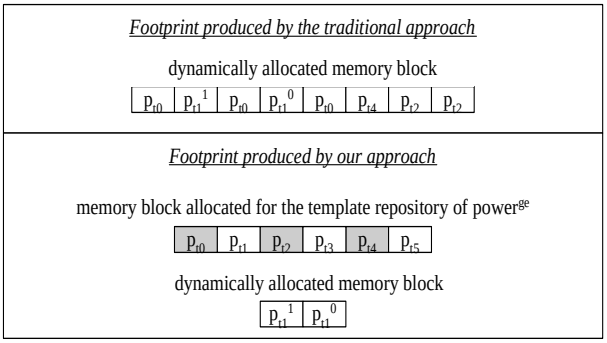

Figure 9. Layouts of memory blocks of the footprints for function `power` w.r.t. concrete value 2 generated by two approaches

As templates for a footprint are not in consecutive memory space, we need to connect them together so that execution of the footprint can proceed properly. A naive approach to connecting templates together is to associate each template with a goto instruction jumping to subsequent template. However we notice that when executing the footprint, even though instantiated hybrid templates residing in the dynamically allocated memory block only need to be executed once, totally dynamic templates residing in the global template repository may need to be executed multiple times. (For instance, if a totally dynamic template is nested within a while statement or the original function is a recursive function.) It is impossible to determine the unique subsequent template. Thus it is undesirable to associate with each template a goto instruction to connect them all together. We tackle this problem as follows:

1. During the first phase, besides creating the footprint, we build an *address table* simultaneously. This address table records a sequence of addresses of the object templates, depicting the program execution control flown among these templates during execution of footprint. The length of the address table is thus equal to the number of templates accessed during the first phrase.
2. We add two types of operations for the purpose of passing program execution control among templates. These two operations

capture the interactions between object templates and the address table.

- The registration operation `register`. It registers the address of an object template in the address table. Registration operations are part of a local run-time specializer and are executed during the first phase to build the address table.
- The redirecting operation `redirect`. It directs the program execution control to the subsequent template at the end of execution of current template. The address of the subsequent template is recorded in the address table.

  Except for the templates derived from an action-annotated return statement, redirecting operations are inserted at the end of all templates, including totally dynamic templates and instantiated hybrid templates, so that program execution control will flow back to the address table after reaching the end of a template. Redirecting operations are executed in the second phase to execute the footprint properly.

The `register` and `redirect` operations are designed as follows: Suppose the name of the address table is `tab_add`, the name of the template counter which acts as a program counter is `tc`, and the symbolic label naming a template is *tem_lab*, then:

$$\begin{array}{lll} \texttt{register}\ \&\texttt{tem\_lab} & ::= & [\![\texttt{tab\_add}[\texttt{tc}] = \&\texttt{tem\_lab}]\!] \\ \texttt{redirect} & ::= & [\![\texttt{tc++; goto tab\_add}[\texttt{tc}]]\!] \end{array}$$

The template counter `tc` is initialized to zero at the beginning of the second phase at run-time.

The address of a footprint is the address of the first template to be executed during the second phase of run-time; this is be either a hybrid template residing in the dynamically allocated memory block or a totally dynamic template residing in the memory block allocated for the global template repository of a `GSC`.

In Figure 10 we present the code of a *local* run-time specializer derived from the action-annotated code $\texttt{power}^{\texttt{aa}}_{\texttt{pss2}}$ depicted in Figure 6. We also extend the templates listed in Figure 7 by inserting `redirect` operations. The design of `dump` and `inst` operations are the same as Tempo's, we omit the detail here.

```
/** power_pss2^rts **/

int power_rts (ss2, int e) {
 register (&p_t0);
 if (e == 0)
  register (&p_t4);
 else {
  inst(h0, e − 1) ;
  dump ;
  register (&p'_t1) ;
  power_rts(ss2, h0) ;
  register (&p_t2) ;
 }
}
```

```
/** Templates **/

p_t0 : {int z ; redirect ; }
p_t1 : {z = b * (tc++ , ((int(*)int))(&h0)(b));
       redirect ; }
p_t2 : {return z ; }
p_t4 : {return 1 ; }
```

Figure 10. The local run-time specializer and templates derived from $\texttt{power}^{\texttt{aa}}_{\texttt{pss2}}$

4. Our Algorithm

We have developed a modular static transformation algorithm (Figure 11, 12, 13) to create a `GSC` for a library `f`.

$$\begin{array}{l} \mathcal{GSC}_{\texttt{fs}} :: \mathcal{P}(\mathbf{Stat}^{aa}) \rightarrow (\mathbf{Dep_{temp}}\ ,\ \mathcal{P}(\mathbf{Stat})) \\ \\ \mathcal{GSC}_{\texttt{fs}}\ (\texttt{f}^{\texttt{aa}}_{\texttt{s}_1}, \ldots, \texttt{f}^{\texttt{aa}}_{\texttt{s}_n}) = \\ \quad let \quad \tau_0 = \emptyset \\ \qquad\quad \{[\![f_{s_i}]\!] = body\ (\texttt{f}^{\texttt{aa}}_{\texttt{s}_i}) \mid 1 \le i \le n\} \\ \qquad\quad \{(\tau_i\ ,\ \texttt{f}^{\texttt{rts}}_{\texttt{s}_i}) = \mathcal{GSC}_{\texttt{s}}\ [\![f_{s_i}]\!]\ \tau_{i-1} \mid 1 \le i \le n\} \\ \quad in \quad (\tau_n\ ,\ (\texttt{f}^{\texttt{rts}}_{\texttt{s}_1}, \ldots, \texttt{f}^{\texttt{rts}}_{\texttt{s}_n})) \end{array}$$

Figure 11. Static transformation over action-annotated libraries

The main function $\mathcal{GSC}_{\texttt{fs}}$ is defined in Figure 11. The inputs to $\mathcal{GSC}_{\texttt{fs}}$ are various action-annotated libraries $\texttt{f}^{\texttt{aa}}_{\texttt{s}_1}, \ldots, \texttt{f}^{\texttt{aa}}_{\texttt{s}_n}$ that are the result of *profitability-oriented action analysis* [21] performed over a library `f` with respect to different specialization scenarios $\texttt{s}_1, \ldots, \texttt{s}_n$. $\mathcal{GSC}_{\texttt{fs}}$ returns a set of *local* run-time specializers $\texttt{f}^{\texttt{rts}}_{\texttt{s}_1}, \ldots, \texttt{f}^{\texttt{rts}}_{\texttt{s}_n}$ derived from those action-annotated codes respectively and a global template repository $\tau \in \mathbf{Dep_{temp}}$. τ is an action-annotated code indexed table of the form $[\mathbf{stat}^{\mathbf{aa}} \mapsto \mathbf{stat}]$. Template repository operators $\bowtie$ and $\uplus$ are defined as:

- $\bowtie$ extend a table τ with a new entry $\{s^{aa} \mapsto s\}$.

  $\tau \bowtie \{s^{aa} \mapsto s\}$ returns an appended table if s^{aa} is distinct from all existing indexes of τ. Otherwise, it returns the original τ.
- $\uplus$ build a new table by merging two tables τ_1 and τ_2 without duplication.

  $\tau_1 \uplus \tau_2 = \bigcup\{\tau_1 \bowtie ele_\tau \mid \forall\ ele_\tau \in \tau_2\}$

$$\begin{array}{l} \mathcal{GSC}_{\texttt{e}} :: \mathbf{Exp}^a \rightarrow (\mathbf{Exp}, \mathbf{Stat}) \\ \\ \mathcal{GSC}_{\texttt{e}}\ [\![e^{\texttt{ev}}]\!] = \\ \quad let \quad h \text{ be a fresh hole variable} \\ \quad in \quad ([\![h]\!]\ ,\ [\![\texttt{inst}(h, e)]\!]) \\ \\ \mathcal{GSC}_{\texttt{e}}\ [\![e^{\texttt{id}}]\!] = \\ \quad ([\![e]\!]\ ,\ [\![;]\!]) \\ \\ \mathcal{GSC}_{\texttt{e}}\ [\![(e^a_1\ \ b_{op}\ \ e^a_2)^{\texttt{rb}}]\!] = \\ \quad let \quad ([\![e'_1]\!], [\![rts_1]\!]) = \mathcal{GSC}_{\texttt{e}}[\![e^a_1]\!] \\ \qquad\quad ([\![e'_2]\!], [\![rts_2]\!]) = \mathcal{GSC}_{\texttt{e}}[\![e^a_2]\!] \\ \quad in \quad ([\![e'_1\ b_{op}\ e'_2]\!]\ ,\ [\![rts_1\ ;\ rts_2]\!]) \\ \\ \mathcal{GSC}_{\texttt{e}}\ [\![f\ (e^a_1, \ldots, e^a_n)]\!] = \\ \quad let \quad \{([\![e'_i]\!], [\![rts_i]\!]) = \mathcal{GSC}_{\texttt{e}}[\![e^{aa}_i]\!] \mid 1 \le i \le n\} \\ \qquad\quad [\![rts_e]\!] = [\![f^{gsc}(ss\ ;\ e'_1, \ldots, e'_n)]\!] \\ \qquad\quad [\![rts]\!] = [\![rts_1\ ;\ \ldots\ ;\ rts_n\ ;\ rts_e]\!] \\ \quad in \quad ([\![\texttt{tc++}\ ,\ fp)]\!]\ ,\ [\![rts]\!]) \end{array}$$

Figure 13. Static transformation over action-annotated expression

Function $\mathcal{GSC}_{\texttt{fs}}$ is defined in terms of two auxiliary functions $\mathcal{GSC}_{\texttt{s}}$ and $\mathcal{GSC}_{\texttt{e}}$. Function $\mathcal{GSC}_{\texttt{s}}$ (Figure 12) takes in an action-annotated statement and a global template repository, and returns a (possibly) updated template repository and code forming the local run-time specializer.

The Seq-Rule dealing with action-annotated sequential statements enables the transformation to descend recursively to the basic language constructs (i.e. assignment statement, return statement and conditional tests) to build templates.

In the If-Rb-Rule dealing with if statement annotated by `rb`, we first derive templates for two branches, and then build a template for the action-annotated conditional test. The latter template will

$$
\begin{array}{lcll}
\mathcal{GSC}_{\mathtt{s}} & :: & \mathbf{Stat}^{aa} \rightarrow \mathbf{Dep_{temp}} \rightarrow (\mathbf{Dep_{temp}}, \mathbf{Stat}) & \\
\\
\mathcal{GSC}_{\mathtt{s}}\ [\![(s_1^{aa}; s_2^{aa})]\!]\ \tau & = & let & (\tau_1\ ,\ [\![rts_1]\!]) = \mathcal{GSC}_{\mathtt{s}}\ [\![s_1^{aa}]\!]\ \tau \qquad \text{(Seq-Rule)} \\
 & & & (\tau_2\ ,\ [\![rts_2]\!]) = \mathcal{GSC}_{\mathtt{s}}\ [\![s_2^{aa}]\!]\ \tau_1 \\
 & & in & (\tau_2\ ,\ [\![rts_1; rts_2]\!]) \\
\mathcal{GSC}_{\mathtt{s}}\ [\![(\mathtt{int}\ v)^{\mathtt{id}}]\!]\ \tau & = & let & [\![s^{aa}]\!] = [\![(\mathtt{int}\ v)^{\mathtt{id}}]\!] \qquad \text{(Decl-Rule)} \\
 & & & [\![temp]\!] = [\![\mathtt{int}\ v\ ;\ \mathtt{redirect}]\!] \\
 & & & [\![rts]\!] = [\![\mathtt{register};]\!] \\
 & & in & (\tau \bowtie \{[\![s^{aa}]\!] \mapsto [\![temp]\!]\}\ ,\ [\![rts]\!]) \\
\mathcal{GSC}_{\mathtt{s}}\ [\![(v = e^{\mathtt{id}})^{\mathtt{id}}]\!]\ \tau & = & let & [\![s^{a}]\!] = [\![(v = e^{\mathtt{id}})^{\mathtt{id}}]\!] \qquad \text{(Ass-Id-Rule)} \\
 & & & [\![temp]\!] = [\![v = e\ ;\ \mathtt{redirect}]\!] \\
 & & & [\![rts]\!] = [\![\mathtt{register};]\!] \\
 & & in & (\tau \bowtie \{[\![s^{a}]\!] \mapsto [\![temp]\!]\}\ ,\ [\![rts]\!]) \\
\mathcal{GSC}_{\mathtt{s}}\ [\![(\mathtt{return}\ e^{\mathtt{id}})^{\mathtt{id}}]\!]\ \tau & = & let & [\![s^{aa}]\!] = [\![(\mathtt{return}\ e^{\mathtt{id}})^{\mathtt{id}}]\!] \qquad \text{(Ret-Id-Rule)} \\
 & & & [\![temp]\!] = [\![\mathtt{return}\ e]\!] \\
 & & & [\![rts]\!] = [\![\mathtt{register};]\!] \\
 & & in & (\tau \bowtie \{[\![s^{aa}]\!] \mapsto [\![temp]\!]\}\ ,\ [\![rts]\!]) \\
\mathcal{GSC}_{\mathtt{s}}\ [\![(v = e^{\mathtt{ev}})^{\mathtt{ev}}]\!]\ \tau & = & & (\tau\ ,\ [\![v = e]\!]) \qquad \text{(Ass-Ev-Rule)} \\
\mathcal{GSC}_{\mathtt{s}}\ [\![(\mathtt{return}\ e^{\mathtt{ev}})^{\mathtt{ev}}]\!]\ \tau & = & & (\tau\ ,\ [\![\mathtt{return}\ e]\!]) \qquad \text{(Ret-Ev-Rule)} \\
\mathcal{GSC}_{\mathtt{s}}\ [\![(v = e^{aa})^{\mathtt{rb}}]\!]\ \tau & = & let & ([\![e']\!], [\![rts_e]\!]) = \mathcal{GSC}_{\mathtt{e}}\ [\![e^{aa}]\!] \qquad \text{(Ass-Rb-Rule)} \\
 & & & [\![s^{aa}]\!] = [\![(v = e^{aa})^{\mathtt{rb}}]\!] \\
 & & & [\![temp]\!] = [\![v = e'\ ;\ \mathtt{redirect}]\!] \\
 & & & [\![rts]\!] = [\![rts_e; \mathtt{dump}; \mathtt{register}]\!] \\
 & & & \tau' = \tau \bowtie \{[\![s^{aa}]\!] \mapsto [\![spe]\!]\} \\
 & & in & (\tau'\ ,\ [\![rts_e]\!]) \\
\mathcal{GSC}_{\mathtt{s}}\ [\![(\mathtt{return}\ e^{aa})^{\mathtt{rb}}]\!]\ \tau & = & let & ([\![e']\!], [\![rts_e]\!]) = \mathcal{GSC}_{\mathtt{e}}\ [\![e^{aa}]\!] \qquad \text{(Ret-Rb-Rule)} \\
 & & & [\![temp]\!] = [\![\mathtt{return}\ e']\!] \\
 & & & [\![s^{aa}]\!] = [\![(\mathtt{return}\ e^{aa})^{\mathtt{rb}}]\!] \\
 & & & [\![rts]\!] = [\![rts_e; \mathtt{dump}; \mathtt{register}]\!] \\
 & & & \tau' = \tau \bowtie \{[\![s^{aa}]\!] \mapsto [\![temp]\!]\} \\
 & & in & (\tau'\ ,\ [\![rts_e]\!]) \\
\mathcal{GSC}_{\mathtt{s}}\ [\![(\mathtt{if}\ e^{\mathtt{ev}}\ s_1^a\ \mathtt{else}\ s_2^a)^{\mathtt{rd}}]\!]\ \tau & = & let & (\tau_1, [\![rts_{s_1}]\!]) = \mathcal{GSC}_{\mathtt{s}}\ [\![s_1^a]\!]\ \emptyset \qquad \text{(If-Rd-Rule)} \\
 & & & (\tau_2, [\![rts_{s_2}]\!]) = \mathcal{GSC}_{\mathtt{s}}\ [\![s_2^a]\!]\ \emptyset \\
 & & & [\![rts]\!] = [\![\ \mathtt{if}\ e\ rts_{s_1}\ ;\ \mathtt{else}\ rts_{s_2};\ ]\!] \\
 & & & \tau' = \tau_1 \uplus \tau_2 \uplus \tau \\
 & & in & (\tau'\ ,\ [\![rts]\!]) \\
\mathcal{GSC}_{\mathtt{s}}\ [\![(\mathtt{while}\ e^{\mathtt{ev}}\ s^a)^{\mathtt{rd}}]\!]\ \tau & = & let & (\tau_s, [\![rts_s]\!]) = \mathcal{GSC}_{\mathtt{s}}\ [\![s^a]\!]\ \emptyset \qquad \text{(While-Rd-Rule)} \\
 & & & [\![rts]\!] = [\![\ \mathtt{while}\ e\ rts_s;\ ]\!] \\
 & & & \tau' = \tau_s \uplus \tau \\
 & & in & (\tau'\ ,\ [\![rts]\!]) \\
\mathcal{GSC}_{\mathtt{s}}\ [\![(\mathtt{if}\ e^{\mathtt{rb}}\ s_1^a\ \mathtt{else}\ s_2^a)^{\mathtt{rb}}]\!]\ \tau & = & let & ([\![e']\!], [\![rts_e]\!]) = \mathcal{GSC}_{\mathtt{e}}[\![e^{\mathtt{rb}}]\!] \qquad \text{(If-Rb-Rule)} \\
 & & & (\tau_1, [\![rts_{s_1}]\!]) = \mathcal{GSC}_{\mathtt{s}}\ [\![s_1^a]\!]\ \emptyset \\
 & & & (\tau_2, [\![rts_{s_2}]\!]) = \mathcal{GSC}_{\mathtt{s}}\ [\![s_2^a]\!]\ \emptyset \\
 & & & [\![rts]\!] = [\![rts_e\ ;\ rts_{s_1}\ ;\ rts_{s_2}]\!] \\
 & & & \tau_{tmp} = \tau_1 \uplus \tau_2 \uplus \tau \\
 & & & [\![temp]\!] = \ \ [\![\mathtt{if}\ e'\ \mathtt{goto}\ \&(hd(\tau_1))\ ;\ \mathtt{else}\ \mathtt{goto}\ \&(hd(\tau_2));]\!] \\
 & & & \tau' = \tau_{tmp} \bowtie \{[\![e^{\mathtt{rb}}]\!] \mapsto [\![temp]\!]\} \\
 & & in & (\tau'\ ,\ [\![rts]\!]) \\
\mathcal{GSC}_{\mathtt{s}}\ [\![(\mathtt{while}\ e^{\mathtt{rb}}\ s^a)^{\mathtt{rb}}]\!]\ \tau & = & let & ([\![e']\!], [\![rts_e]\!]) = \mathcal{GSC}_{\mathtt{e}}[\![e^{\mathtt{rb}}]\!] \qquad \text{(While-Rb-Rule)} \\
 & & & (\tau_s, [\![rts_s]\!]) = \mathcal{GSC}_{\mathtt{s}}\ [\![s^a]\!]\ \emptyset \\
 & & & [\![rts]\!] = [\![rts_e\ ;\ rts_s]\!] \\
 & & & [\![temp]\!] = [\![\mathtt{while}\ e'\ \{\ \mathtt{goto}\ \&(hd(\tau_s))\}]\!] \\
 & & & \tau_{tmp} = \tau_s \uplus \tau \\
 & & & \tau' = \tau_{tmp} \bowtie \{[\![e^{\mathtt{rb}}]\!] \mapsto [\![temp]\!]\} \\
 & & in & (\tau'\ ,\ [\![rts]\!])
\end{array}
$$

Figure 12. Static transformation over action-annotated statement

be an if statement, each branch of which is a `goto` statement pointing to the address of the first template derived from each action-annotated branch respectively. This enables sharing of templates derived from identical action-annotated statements found in different action-annotated functions. Similar treatment is defined in While-Rb-Rule dealing with a while statement annotated by `rb`.

The functionality of our static transformation algorithm is similar to Tempo's algorithm of transforming action annotations [9] in that both aim to identify the code that should appear in the run-time specializer and the template file. The differences between these two algorithms are:

- Our static transformation derives a template at each basic language construct whereas the templates identified by Tempo may include the code derived for a sequence of statements, as shown in Figure 4.
- There are two more categories of operations used in our static transformation to manipulate templates, namely `register` and `redirect` operations, both for the purpose of directing program execution among templates during execution through the help of the address table.

- Dumping operations *dump* are only used in dispatching hybrid templates.

4.1 Building Local Run-time Specializers and Templates for Inter-related/Recursive Libraries

For inter-related/recursive functions, we only consider *unfolding* of recursive function calls so that function calls are substituted by their corresponding function bodies. We do not consider *specializing* recursive function calls in order to avoid the risk of infinite specialization. This strategy is similarly adopted by Tempo. Next, we explain in details the static transformation rules dealing with action-annotated return statement and function call.

The `redirect` operation is not inserted in the templates derived from action-annotated return statement, as illustrated by rules Ret-Id-Rule and Ret-Rb-Rule. This is because the program execution control flow for return statement will be directed by the underlying operating system conforming to the convention of handling function call/return.

The rule dealing with function call expressions is defined in Figure 13. Here, $[\![rts_e]\!]$ is the call (to be made during the first phase) to the corresponding `GSC` parameterized by ss. In the rule, ss is the specialization context which is available in the action-annotated function call. The resulting template contains a comma expression: Before the comma is an expression that increases the template variable `tc` by one; after the comma is a function pointer *fp* pointing to the address of the corresponding footprint constructed by executing $[\![rts_e]\!]$. When such template is executed, the function pointer will ensure the program execution control is transferred to the first template of the pointed footprint, following the convention of handling function call/return.

The rules dealing with action-annotated return statement and function call expressions ensures that the template counter `tc` obeys the following property throughout run-time execution:

> *Regardless of whichever template is executed,* `tc+1` *always points to the address of the next template to be executed.*

The above property holds regardless of how the next template will be reached. Indeed, a template can be reached by executing either the `redirect` operation at the end of the current template, or a function pointer which jumps to the first template of the function body, or a return statement which jumps to executing the code immediately following the function call.

In order to maintain this property, we insert an increment operator to increase the value of `tc` before a function pointer, and refrain from calling redirecting operations during the execution of the return statement.

4.2 Building the Address Table for Inter-related/Recursive Libraries

As demonstrated in function $\mathcal{GSC}_s$ and $\mathcal{GSC}_e$, the operations which are responsible for building the address table only exist in the code for the run-time specializers. The code in the templates of a `GSC`, which is constructed for either an intra-procedual library or an inter-procedual/recursive library, does not include any operations to build the address table.

Specifically, during the process of executing specialized inter-related (or self-recursive) libraries, only one address table is built throughout the execution of all invoked run-time specializers. Because of the sequential nature of code execution, one template counter is adequate to be used to point to the address table and control the flow among templates. For example, the address table built after calling the run-time specializer $\texttt{power}^{rts}_{\texttt{pss2}}$ (presented in Figure 10) with input 2 during the first phrase at run-time, comprises the following sequence of template addresses:

$$[\&\mathtt{p}_{\mathtt{t0}}; \&\mathtt{p}^{\mathtt{1}}_{\mathtt{t1}}; \&\mathtt{p}_{\mathtt{t0}}; \&\mathtt{p}^{\mathtt{0}}_{\mathtt{t1}}; \&\mathtt{p}_{\mathtt{t0}}; \&\mathtt{p}_{\mathtt{t4}}; \&\mathtt{p}_{\mathtt{t2}}; \&\mathtt{p}_{\mathtt{t2}}]$$

Interested readers may wish to simulate the sequential execution of the templates listed in Figure 10 to verify that the templates are connected properly at run-time with the help of the dynamically-built address table.

4.3 Compiling the Template Repository

We organize templates in a way which is totally different than traditional run-time specialization systems do. The templates identified by those existing systems, such as Tempo, DyC and TickC [2, 16, 17], are syntactic subset of the original code. Thus those templates can be put together and compiled as a whole. This enables some global optimizations over those templates to be performed to produce an optimized object templates. In our approach, each individual template stored in the global template repository is derived from the action-annotated statement possessed by different action annotated libraries. The same statement with different action annotations lead to the generation of different templates. Thus, the template repository can not be compiled as a whole. Instead, we compile individual templates separately and then link all object templates to form a unique binary representing the binary of template repository.

This simple compilation scheme can be improved by grouping adjacent hybrid templates or adjacent totally dynamic templates into a cluster to allow global optimization. There will be only one registration operation and one redirecting operation associated with each cluster. The information conveyed by the web of action annotated codes (as introduced in Section 3.1) regarding the merging of sharable action-annotated code segments can help us to perform the clustering operation.

4.4 Wrapping `GSC`

In order to facilitate interaction between `GSC` and multiple applications, we wrap the `GSC` with other information, including multiple specialization scenarios and two APIs. These APIs act as interfaces to `GSC` clients, throughout the entire specialization process from compilation to execution, through reception of binding-time information (from clients) and return of proper run-time generating extension to be referred to by the specialized application.

Client access to the `GSC` is mainly guided by submitting a specialization scenario. As the `GSC` maintains only the run-time specializers associated with the *profitable* specialization scenarios and the *totally dynamic* specialization scenario, client access via scenarios not listed as "profitable" will have to be converted to profitable ones (or the totally dynamic one) by the wrapped `GSC`. Selection of the *most appropriate* profitable specialization scenario is guided by some specialization policy, and the selected scenario is called *minimal profitable context*. We refer interested readers to [21] for details.

The abstract data type of `wrapped GSC` for a function `f` is defined in Figure 14:

```
class f_wgsc{
private :
  static int        ss[ss_num];
  static (void*)    rts[ss_num];
public :
  int       gsc_bt(int ssv);
  (void*)   gsc_rtge(int ssv);
}
```

Figure 14. Abstract data type of `wrapped GSC`

In the figure,

- ssv refers to an encoding of *specialization scenario value* (SSV for short); ie., binding-time values `S` and `D` are encoded into 0 and 1 respectively, and a specialization scenario comprising a tuple of binding-time values is encoded into an integer representing a concatenated sequence of 0's and 1's. Thus, the SSV of the specialization scenarios **S1** and **S2** defined in Figure 3 are 00_2 and 10_2 respectively.
- ss_num is a constant representing the number of specialization scenarios supported by `GSC`. ss is an array of SSV's representing all acceptable (ie., profitable and totally dynamic) specialization scenarios.
- rts maintains an array of function pointers pointing to all *local* run-time specializers created for this `GSC`.

  This address of footprint is implemented as a function pointer pointing to the binary of the footprint, which is executed with respect to the values of dynamic variables in the second phase.
- gsc_bt takes in an SSV encoding of a specialization scenario, and returns an SSV encoding of a *minimal profitable context*, which will be used in the ensuing correspondence with this wrapped `GSC`, in replacement of the input scenario.
- gsc_rtge takes in an SSV encoding of a specialization scenario, which is returned by gsc_bt, and returns a function pointer pointing to the footprint instantiated from the field rts, indexed by the minimal profitable context in association with this specialization scenario.

5. A Novel Specialization Framework

Figure 15 depicts the complete picture of the novel framework for specialization of applications using shared libraries. There are three essential elements of this framework:

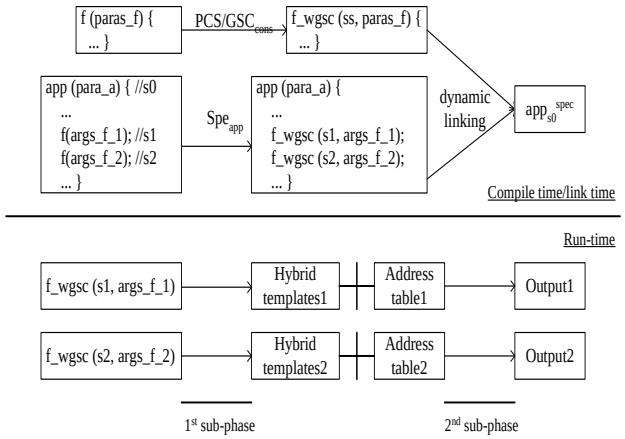

Figure 15. Complete picture of the novel framework for specialization for applications using shared libraries

- *Shared library specialization*: This is a process that constructs a `wrapped GSC` by performing profitability-oriented component specialization [21] and `GSC` construction `GSC`$_{\texttt{cons}}$ over a shared library at compile-time. It is an application-independent process.
- *Application specialization*: This is a process that specializes an application at compile-time with respect to its relevant binding-time information specified by application programmers. A typical example of the relevant specialization information is the specialization scenario for the application's parameters. Application programmers may also specify binding-time conditions for the called libraries at call sites inside the application as we proposed in [20].

  An application specializer `Spec`$_{\texttt{app}}$ is constructed to perform *profitability-oriented* binding-time analysis, action analysis and specialization over the application through interaction with `wrapped GSC` of called functions. The main specialization job undertaken by `Spec`$_{\texttt{app}}$ is to substitute each action-annotated function call with the call to corresponding `wrapped GSC` parameterized by the minimal profitable specialization context of that function call. In this way, we build a correct reference to corresponding `wrapped GSC` in the specialized application such that the subsequent dynamic linking can proceed properly and $f\_wgsc.gsc\_rtge(ss)$ returns the desired generating extension (i.e. the selected *local run-time specializer* the execution of which will manipulate the templates stored in the *global template repository*) at run-time. This specialization step produces a generating extension for the application, similar to what Tempo generates in run-time specialization.
- *Specialized application execution*: This is a process that runs the specialized application with respect to the concrete values for the whole input. Here, the approaches described earlier are employed to ensure minimal footprints of the specialized libraries throughout the execution.

6. Related Work

The idea of specializing libraries independent of their use contexts was first proposed by Schultz in his position paper [18], in which he called it "black-box program specialization". Relevant work also includes [3, 4, 10, 19]. However, none of these works address the sharing issue pertaining to dynamically-linked shared libraries. In fact, to the best of our knowledge, the issue has not been addressed by the partial-evaluation community.

Tempo [8] is the first run-time specializer that employs partial evaluation at the level of object code, and yet remains high-level in its specification. While it does not handle shared libraries, its implementation details have inspired many ways in our current implementation. Specifically, the division of generating extension into a run-time specializer and a set of template, and the generation of template file through execution of run-time specializer, have been the core technique of our implementation. We have also found similar implementations in Dyc [11, 12]. None of these works, however, consider the possibility of maintaining separate memory regions for different kinds of templates. Consequently, they fail to minimize library footprints in application execution.

There is much room for improvement in our implementation. One such improvement is to perform data specialization, as proposed by Lawall [13] to optimize the run-time specialization in Tempo system. More specifically, data specialization can be used to pre-calculate the precise size of the buffer needed to be allocated at run-time, rather than allocating buffer of default maximal size.

7. Conclusion

In this paper we investigate specialization for applications using shared libraries, aiming for preserving and maximizing the chances of sharing during the whole specialization process, which includes specializations of both libraries and applications. We propose a static transformation to construct a `GSC` for a shared library, aiming at eliminating code duplication occurring at compile-time. Instead of creating separate generating extensions with respect to different specialization scenarios as traditional specialization framework does, our `GSC` is composed of a set of *local* run-time specializers, each of which pertains to a specialization of the library with respect to a specific specialization scenario; and a *global* template reposi-

tory which is shared by those *local* run-time specializers. We also propose a novel run-time specialization approach to minimizes the need to dump *object templates* at run-time and maximize *sharing* by sharing the *totally dynamic templates* of a `GSC` among different footprints, at the expense of building an extra address table at run-time. Together with our earlier solution to *profitability-oriented component specialization*, the approaches proposed in this paper build a framework of specialization for applications using shared libraries.

Acknowledgements

We would like to thank Julia Lawall, Charles Consel, Brian Grant and Craig Chambers for their help in understanding the run-time specialization details of Tempo and DyC systems through personal communications with the first author. We are also grateful to Hugh Anderson for his comments regarding using shared libraries under Linux. This research is partially supported by the university research grant R-252-000-250-112.

References

[1] Tempo Specializer - A Partial Evaluator for C. Phoenix group. URL: http://phoenix.labri.fr/software/tempo/.

[2] 'C and tcc. PDOS group. MIT Computer Science and Artificial Intelligence Laboratory. URL: http://pdos.csail.mit.edu/tickc/.

[3] Gustavo Bobeff and Jacques Noyé. Molding components using program specialization techniques. In *Workshop on Component-Oriented Programming*, July 2003.

[4] Gustavo Bobeff and Jacques Noyé. Component specialization. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 39–50, 2004.

[5] Charles Consel. A tour of schism: a partial evaluation system for higher-order applicative languages. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, 1993.

[6] Charles Consel and Olivier Danvy. From interpreting to compiling binding times. In *European Symposium on Programming*, pages 88–105, 1990.

[7] Charles Consel, Luke Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, Julia Lawall, and J. Noyé. Tempo: specializing systems applications and beyond. *ACM Computing Survey*, 30(3es):19–24, 1998.

[8] Charles Consel, Julia Lawall, and Anne-Françoise Le Meur. A tour of Tempo: A program specializer for the C language. *Science of Computer Programming*, 2004.

[9] Charles Consel and François Noël. A general approach for run-time specialization and its application to c. In *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, 1996.

[10] Crispin Cowan, Andrew Black, Charles Krasic, Calton Pu, Jonathan Walpole, Charles Consel, and Eugen-Nicolae Volanschi. Specialization classes: an object framework for specialization. In *IWOOOS '96: Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, page 72, 1996.

[11] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Dyc: an expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science*, 248(1-2):147–199, 2000.

[12] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in dyc. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–304, 1999.

[13] Julia L. Lawall and Gilles Muller. Faster run-time specialized code using data specialization. Technical Report Research Report RR-3833, INRIA, December 1999.

[14] Anne-Françoise Le Meur, Julia Lawall, and Charles Consel. Specialization scenarios: A pragmatic approach to declaring program specialization. *Journal of Higher-Order and Symbolic Computation*, 17(1):47–92, 2004.

[15] François Noël. *Spécialisation dynamique de code par évaluation partielle*. PhD thesis, Université de Rennes 1, France, October 1996. In French.

[16] Massimiliano Poletto. *Language and compiler support for dynamic code generation*. PhD thesis, MIT, June, 1999.

[17] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transaction on Programming Languages and Systems*, 21(2):324–369, March 1999.

[18] Ulrik Pagh Schultz. Black-box program specialization. In *Workshop on Component-Oriented Programming*, 1999.

[19] Eugen-Nicolae Volanschi, Charles Consel, Gilles Muller, and Crispin Cowan. Declarative specialization of object-oriented programs. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 286–300, 1997.

[20] Ping Zhu and Siau Cheng Khoo. Request and assert: A pragmatic approach to generating specialization scenarios. Technical report, School of Computing, National University of Singapore, 2006.

[21] Ping Zhu and Siau-Cheng Khoo. Towards constructing reusable specialization components. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 154–164, 2007.