Efficient Specialization for Applications Using Shared Libraries

Ping Zhu and Siau-Cheng Khoo

Department of Computer Science, National University of Singapore

1 Introduction

The past few decades have witnessed the prevalence of using *shared libraries* to provide reusable and essential services in many systems or application domains. It has been widely acknowledged that building applications through reusing shared libraries achieves *memory sharing* in the sense that there is one single copy of memory allocated for a shared library and this memory is sharable across various applications which reuse it. *Memory sharing* contributes to significant runtime performance improvements.

Unfortunately existing program specialization frameworks were designed for specializing *static libraries*. They are inefficient in specializing applications which reuse shared libraries in the following two aspects.

Firstly, specializations for libraries were application-driven. Libraries were not specialized independently. We proposed independent library specialization to address this inefficiency [4]: We aimed for specializing a library without taking into consideration its use contexts. Independent library specialization produced, for a library:

- A set of specialization scenarios which are deemed profitable with respect to user-specified *profitability declarations*, and
- A *generic specialization library* which is a synthesis of (multiple) profitable specialization scenarios and the original library. It can be efficiently adapted to different specialization contexts.

Secondly, existing program specialization frameworks adopted *static linking* to link specialized libraries with the application to produce the final executable specialized application. Thereafter they do not possess the *memory sharing* merit. To address this inefficiency, we adopt *dynamic linking* to link *generic specialization libraries* with the application for specialization purpose. A novel approach to runtime specialization, which is the focus of this poster, is applied aiming for maximize the *memory sharing* effect.

2 A novel approach to runtime specialization

Conventional runtime specialization technique [1, 3, 2] created a *runtime generating extension* P_{rtge} for a program P with respect to ss a binding time division

of it parameters. P_{rtge} produced different runtime specialized codes P_{rtsc} with respect to different values of static parameters during runtime. P_{rtge} was commonly comprised of two parts:

- Several *object templates*, each of which is a binary code compiled from a (partially-)*dynamic* program fragment parameterized by *static* expressions.
- A *runtime specializer*, which 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 expressions in the template (which is termed as *template instantiation*), and *dumping instantiated templates*. When creating a P_{rtsc} from a P_{rtge}, a memory is dynamically allocated to store instantiated templates. The address of this memory is returned as a function pointer to P_{rtsc}.

Our novel approach to runtime specialization continues to value this two-part structure of runtime generating extension. We categorize object templates into two types: *Totally dynamic* templates without any embedding static expressions and *hybrid* templates parameterized by at least one *static* expressions. *Totally dynamic* templates are sharable among different runtime specialized codes. Thus we choose not to dump *totally dynamic* templates into dynamically allocated memories during runtime. For *hybrid* templates, they are instantiated and dumped into dynamically allocated memories.

Given that *totally dynamic* templates and *hybrid* templates are located in two logically separate memories now to form the runtime specialized code, these templates need to be explicitly connected together so that the execution of the runtime specialized code can proceed. We tackle this problem by

1. building a table which records the starting address of each object template to which control will be passed during the process of creating runtime specialized code and;
2. adding two more types of operations in the *runtime specializer* to manipulate object templates. These two extra operations are:
   - *Registration operation* which registers the address of the object template to the table;
   - *Redirecting operation* which directs the program execution control to the subsequent template at the end of execution of current template. The subsequent template is located by the address recorded in the table

In summary our novel approach minimizes the needs to dump *object templates* to dynamically allocated memories. This approach achieves *memory sharing* by sharing *totally dynamic templates* among different runtime specialized codes, at the expense of building an extra table at runtime. Together with our earlier solution to independent library specialization, this novel approach to runtime specialization act as two pillars of the framework of efficient specialization for applications using shared libraries.

References

1. 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, St. Petersburg Beach, Florida, United States, 1996. ACM Press.
2. 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.
3. 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, New York, NY, USA, 1999. ACM Press.
4. Ping Zhu and Siau-Cheng Khoo. Towards constructing reusable specialization components. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 154–164, New York, NY, USA, 2007. ACM Press.