Towards Highly Optimized Real-time Middleware for Software Product-line Architectures

Arvind S. Krishna[†], Aniruddha Gokhale[†] and Douglas C. Schmidt[†],
Venkatesh Prasad Ranganath[‡] and John Hatcliff[‡]
[†]Dept. of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN
[‡]Dept. of Computing and Information Sciences, Kansas State University, Manhattan, KS

Abstract

This paper provides the following contributions to the study of middleware optimization techniques for product line architectures in real-time systems. First, we identify different dimensions of generality in standards based middleware implementations. Second, we describe how specialization approaches used in other domains including OS, compiler and programming languages can be applied to address middleware generality challenges. Third, we present preliminary results from the application of our specialization techniques. Our results illustrate that specialization techniques represent a promising approach for minimizing time/space overheads in middleware.

1. INTRODUCTION

Emerging trends and challenges. *Product-line architectures* (PLAs) [1] are a promising technology for systematically addressing the challenges of large-scale software systems. In contrast to conventional software processes that produce separate point solutions, PLA-based processes create families of *product variants* [2] that share a common set of capabilities, patterns, and architectural styles. PLA based development processes are also desirable for Distributed Real-time and Embedded (DRE) systems [2, 3] that are characterized by their multiple, simultaneous constraints across different quality of service (QoS) dimensions (such as memory footprint, weight, and performance), which often makes them harder to develop, maintain, and evolve than mainstream desktop and enterprise software.

DRE systems QoS challenges have hitherto led developers to (re)invent custom applications that are tightly coupled to specific hardware/software platforms, which is tedious, error-prone, and costly to evolve over product lifecycles. During the past decade, therefore, a key technology for alleviating the tight coupling between applications and their underlying platforms has been *middleware*, which (1) functionally bridges the gap between applications and platforms, (2) controls many aspects of end-to-end QoS, and (3) simplifies the integration of components developed by multiple technology suppliers.

However, key challenges must be overcome before middleware can be applied broadly to support the QoS needs of *PLA-based* DRE systems. In particular, R&D is needed to help resolve the tension between (1) the *generality of standards-based middleware platforms*, which benefit from reusable architectures designed to satisfy a broad range of application requirements, and (2) *application-specific product variants*, which benefit from highly-optimized, custom middleware implementations. In resolving this tension, solutions should ideally retain the portability and interoperability afforded by standard middleware.

Specialization techniques for resolving middleware generality challenges. A promising solution approach to alleviate middleware generality for PLAs is the use of specialization techniques such as partial evaluation (PE). Jones et.al [4], define partial evaluation as a technique that creates a specialized version of a general program, which is more optimized for speed and size than the original program. Specialization techniques draw from and have characteristics of language mechanisms such as program optimization techniques [5], compilers [6] and program generation [7] and generative programming techniques [8].

2. OVERVIEW OF SPECIALIZATION TECHNIQUES

Specialization approaches tailor code based on ahead of time known invariant assumptions. Consider a given program p, and inputs $arg1$ and $arg2$ as shown below.

```
pow (n,m): /* Computes n power m */
  pre (n >= 0, m >= 0)
begin
 i := 1
 result = n
 while (i <=m)
 begin
  result := result * n
  i++
 end
 return n
end
```

Now given that $arg2$ is know a priori (the value of m is 2), *i.e.*, it is an invariant, specialization techniques can be used to produce a corresponding program p_{spl}. The code snippet below illustrates one such specialization of p that takes only input argument $arg1$.

```
spl_pow (n): /* Computes n * n */
  pre (n >= 0, m >= 0)
begin
  return n * n
end
```

The program spl_pow is called the specialization of program pow with respect to the invariance $m = 2$. For a given program p and its specialization p_{spl},

$$output(p) = output(p_{spl}) \quad (1)$$

$$speed(p) < speed(p_{spl}) \quad (2)$$

are necessary conditions and

$$size(p) > size(p_{spl}) \quad (3)$$

is a desirable condition. Specialization techniques simultaneously combine characteristics of: (1) **program optimizer**, by producing a specialized program, which has the same behavior as the original version, but takes lesser steps, (2) **compiler**, by using techniques like constant propagation (replacing $arg1$ with the constant value), and (3) **program generator**, by generating the optimized version of the program, either source or object code directly.

2.1 Specialization Example

In this section we show a concrete example of program specialization technique based on the C++ Standard Template Library (STL) that provides a set of containers (Abstract Data Types) and algorithms that can be used for PE. The code snippet below illustrates how template meta programming techniques can be used as a mechanism for partial evaluation.

```
template<int X>
struct fibo_num {
  static const num = fibo_num<X-1>::num +
    fibo_num<X-2>::num ;
}
template<>
struct fibo_num<0> {
  static const num = 1; /* 0th number */
}
template<>
struct fibo_num<1> {
  static const num = 1; /* 1st number */
}
template<>
struct fibo_num<2> {
 static const num = 1; /* 2nd number */
}
```

The code above computes the nth fibonacci number. However, this computation is done at compile time using PE as follows. Consider the following statement:

```
const int fibo_10 = fibo_num<10>::num;
```

To evaluate fibo_10, a C++ compiler recursively instantiates templates fibo_num<9...1> to compute the 10th number. Thus all occurrences of fibo_10 are substituted with the value directly thereby improving program space and speed.

2.2 Specialization Mechanisms Applied to Different Domains

Specialization mechanisms have been applied to different domains including scientific applications, functional programming, operating systems and database systems. In computer graphics for example, ray tracing algorithms compute information on how light rays traverse a scene based on different origination. Specialization of these algorithms [9] for a given scene have yielded better performance rather than general purpose approaches. Similarly in databases [10], general purpose queries have been transformed into specific programs optimized for a given input. Similarly, training neural networks [11] for a given scenario has improved its performance.

The earliest of the efforts in Synthesis Kernel [12] pioneered the idea of generating custom system calls for specific situations. The motivation was to collapse layers and to eliminate unnecessary procedure calls. Others have extended this approach to use incremental specialization techniques. For example in their work [13], Pu et al., have identified several invariants for a operating system `read` call for HP_UX platform. Based on these invariants, code is synthesized to adapt to different situations. Once the invariants fail, either re-plugging code is used to adapt to a different invariant or default unoptimized code is used.

Specialization techniques have also been applied to various generations of middleware. [14] describes the use of the Tempo C program partial evaluator tool to automatically optimize common software architecture structures with respect to fixed application contexts. For instance, the authors show how partial evaluation can be applied to fold together and optimize layers in early generations of middleware, *i.e.*, a remote procedure call (RPC) implementation, by specializing RPC invocations to the size and type of remote procedure parameters (yielding speed-ups of 1.7x and 3.5x).

3. SPECIALIZING MIDDLEWARE IMPLEMENTATIONS

Traditional specialization techniques have been used to optimize applications in function/logic programming. There does not exist any partial evaluation tool for object oriented programming languages such as C++ or Java. Other program specialization techniques are commonly used in optimizing compilers. Distributed Object Computing (DOC) middleware displays several characteristics amenable to specialization such as (1) ability to run on different platforms, (2) multitude of configuration options and (3) design for flexibility and generality. Using a similar approach as an optimizing compiler, specialization may be used to produce leaner and meaner middleware implementations more tailored to the operating context.

This section presents sources and methods of applying specialization techniques to middleware. The description for each of the specialization techniques are structured as follows: We first describe the motivation and opportunity for specialization, then at a high level illustrate how the specialization can be carried out. Finally, we show preliminary empirical results from our specialization application on the TAO [15] open-source C++ CORBA middleware that is widely used in production DRE systems (`www.dre.vanderbilt.edu/users.html`).

3.1 Opportunities for Middleware Specialization

To improve performance and footprint for different applications, middleware implementations incorporate several horizontal (general purpose) optimizations such as predictable and scalable (1) request demultiplexing techniques, that ensure $O(1)$ look up time [16] and collocation optimization, which bypasses the network when client and server reside in the same address space. However, these optimizations are still generic, for example redundant checks for remoting are performed to accommodate for generality, *i.e.*, capability to communicate over the wire as well. In the remainder of this section, we describe different dimensions of middleware specializations that we are working on to improve middleware QoS above and beyond existing general-purpose optimizations.

Specialization for target location. The collocation optimization in middleware bypass the network completely when both the client and server objects are collocated. However, in this situation, middleware is also general purpose, *i.e.*, it still can send and request remote CORBA requests. Similarly, an object may be a "sink", *i.e.*, only receive events and updates from other sources but never send out any events itself. General purpose middleware works in both cases, however, considerable footprint and performance improvements can be obtained by eliminating unnecessary checks and code within the middleware.

For example, in the special collocation case where there is no remoting, *i.e.*, there is no need to make remote calls and the code required to make remote connections (connection handling code) can be eliminated; same case for sink components. Further, in the collocated case, as all calls are known a priori to be on the same node, checks to see if a call is remote or local can also be eliminated. These checks span multiple layers within the middleware, including message invocation classes in the I/O layer and object proxies in the ORB core layers.

Extrapolate rather than send. HTTP caching works by storing web pages in a local machine and servicing requests to the remote page from the local cache. After a given time, the web page expires and a remote request is sent. A CORBA client/server performance can be improved via caching. For example, before sending a request to the server, a client can check to see if it has a previous response which is still valid. This eliminates a roundtrip overhead. Some middleware implementations, including ACE+TAO support a mechanism called Smart Proxies [17] which enable extrapolation rather than sending a request to the server.

Specialize middleware framework implementations. Middleware is often developed as a set of frameworks that can be extended and configured with alternative implementations of key components, such as different types of transport protocols (*e.g.*, TCP/IP, VME, or shared memory), event demultiplexing mechanisms (*e.g.*, reactive-, proactive-, or thread-based), request demultiplexing strategies (*e.g.*, dynamic hashing, perfect hashing, or active demuxing), and concurrency models (*e.g.*, thread-per-connection, thread pool, or thread-pre-request). However, most applications only use a subset of the different features provided by the middleware framework. For example, certain applications use only the TCP/IP protocol for communication or use the thread-per-connection concurrency strategy. In this situation, the frameworks can be specialized to eliminate dynamic dispatching overheads based on the type of concrete component used by the application that is know a priori.

Specialize deployment platform characteristics. Another key dimension of generality stems from the deployment platforms on which middleware and PLA applications are hosted. Examples of this deployment platform generality include different OS-specific system calls, compiler flags and optimizations, and hardware instruction sets. Every OS, compiler, and hardware platform provide different configuration settings that perform differently and can be tuned to minimize the time/space overhead of middleware and applications. For example, sendfile() optimizations available on certain platforms, such as Linux can be used to avoid data copies between middleware and kernel buffers thereby minimizing end-to-end latencies for application using the middleware.

3.2 Summary of Results

In this section we present preliminary empirical results for target object location specialization discussed in Section 3.1. This specialization targets collocated components that do not require remoting capabilities. In this case, remoting checks along the request/response processing path within the middleware can be eliminated. This specialization is applied above and beyond the standard collocation optimization supported in a Real-time CORBA implementation and completely eliminates remoting tests (the generality) in the collocation optimization. The specialization is compatible with the CORBA specification since no changes are made to the CORBA interfaces.

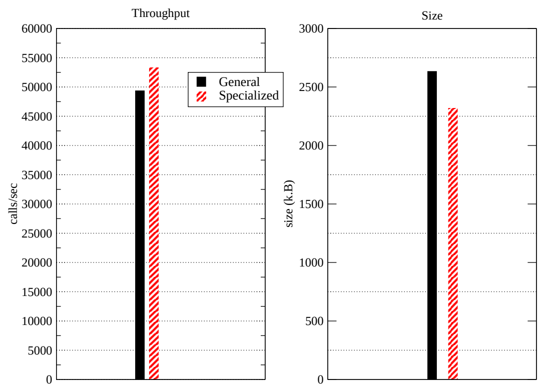

Figure 1: Results for Target Object Location Specialization

All experiments were performed on an Intel Pentium III 851 Mhz processor with 512 MB of main memory running on Linux 2.4.7-timesys-3.1.214 kernel, which contains a very predictable real-time kernel module. The TAO middleware used for the experiments was version 1.4.7, which was compiled with gcc version 3.2.2.

Figure 1 shows the footprint and throughput improvements accrued by the target object location specialization. As shown in the figure, footprint for a collocated application improves by ∼40 kiloBytes, which is a 12% improvement over the general-purpose TAO implementation. This specialization also removes redundant tests along the critical path, which improves end-to-end throughput by ∼7% over the general-purpose collocation optimization implemented by TAO. These results show how eliminating redundant remoting functionality can improve size and performance of general-purpose middleware.

Figure 2 illustrates the QoS improvements accrued by applying all of the middleware specializations discussed in Section 3.1 to a remote CORBA operation. The average end-to-end latency for the specialized TAO dropped by ∼43%, while the dispersion measure was twice as good as general-purpose optimized TAO implementation, indicating considerable improvement in predictability, which is essential for DRE systems.

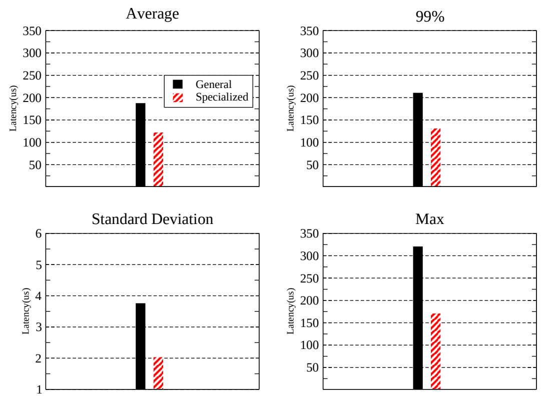

Figure 2: Results for Cumulative Specialization Application

Similarly, the 99% bound values for the specialized TAO improved by ∼40% while worst-case measures improved by ∼150μsecs, which is a 45% improvement over the general-purpose TAO implementation. End-to-end throughput measures improved by n average of ∼65%. To measure performance speed up for a complicated data structure, we ran the experiment using the complex data structure from our demarshaling experiments. For a sequence length of 64 average latency improved by ∼26%, while for a length of 4,096 latency improved by ∼51%.

4. CONCLUDING REMARKS AND FUTURE WORK

Traditional program specialization techniques such as partial evaluation have been used to specialize a given program based on ahead of time known invariant properties. This paper described how such specialization techniques are also applicable to standards-based middleware implementations for PLAs. Our preliminary results show that application of specialization techniques can minimize the time-/space overheads of applications using standards based middleware without (1) changes to the application code and (2) compromising compliance to the CORBA specification. Our ultimate goal is to enable other middleware developers to analyze and implement the specializations. We are working on developing a comprehensive CORBA *specialization model* based on [18] that – independent of a particular CORBA implementation – identifies (1) points in an ORB architecture where specialization is beneficial and (2) API extensions to the architecture that provide *hooks* for achieving effective specialization.

Our preliminary implementation of the specialization techniques illustrated that manually applying these specializations to thousands of lines of C++ middleware code would be infeasible. Our future work therefore focuses on developing languages and tools to automate static and dynamic analysis to identify opportunities for specialization and to collect information that can drive the specialization process. We are also developing transformation engines that automatically perform the refactoring, partial evaluation, and code weaving necessary to achieve specialized middleware implementations.

Figure 3 illustrates a futuristic view of a specialization process. In this approach, middleware models and PLA invariance specification is fed to a middleware specializer that generates a middleware configuration suitable for the PLA. From this base configuration model, the variability specifications, *e.g.*, the assembly and deployment aspects are woven by a variability weaver generating an optimized middleware implementation.

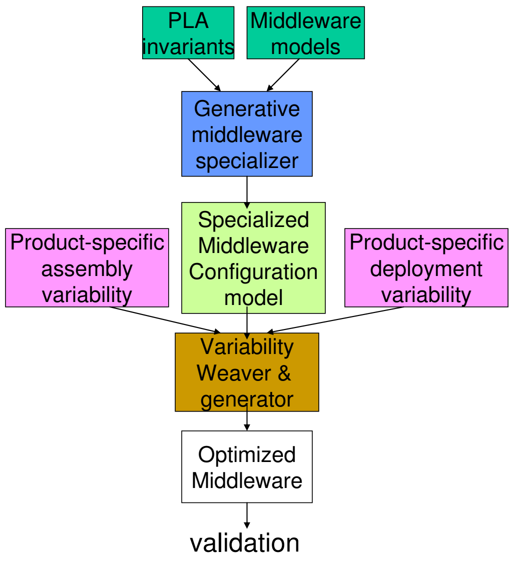
PLA invariants
Middleware models
Generative middleware specializer
Product-specific assembly variability
Specialized Middleware Configuration model
Product-specific deployment variability
Variability Weaver & generator
Optimized Middleware
validation

Figure 3: Model-driven Middleware Specialization Approach

5. REFERENCES

[1] D. L. Parnas, "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, pp. 1–9, 1976.

[2] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.

[3] B. S. Doerr and D. C. Sharp, "Freeing Product Line Architectures from Execution Dependencies," in *Proceedings of the 11th Annual Software Technology Conference*, Apr. 1999.

[4] N. Jones, C. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[5] V. Itkin, "On Partial and Mixed Program Execution," in *Program Optimization and Transformation*, pp. 17–30, CCN, 1983. (In Russian).

[6] S. Abramov and N. Kondratjev, "A Compiler Based on Partial Evaluation," in *Problems of Applied Mathematics and Software Systems*, pp. 66–69, Moscow, USSR: Moscow State University, 1982. (In Russian).

[7] P. Thiemann and M. Sperber, "Program Generation With Class," in *Informatik'97, Aachen, Germany, September 1997* (M. Jarke, K. Pasedach, and K. Pohl, eds.), Berlin: Springer-Verlag, 1997.

[8] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Boston: Addison-Wesley, 2000.

[9] P. Andersen, "Partial Evaluation Applied to Ray Tracing," DIKU Research Report 95/2, DIKU, 1995.

[10] C. Sakama and H. Itoh, "Partial Evaluation of Queries in Deductive Databases," *New Generation Computing*, vol. 6, no. 2,3, pp. 249–258, 1988.

[11] L. Lei, G.-H. Moll, and J. Kouloumdjian, "A Deductive Database Architecture Based on Partial Evaluation," *SIGMOD Record*, vol. 19, pp. 24–29, September 1990.

[12] C. Pu, H. Massalin, and J. Ioannidis, "The Synthesis Kernel," *Computing Systems*, vol. 1, pp. 11–32, Winter 1988.

[13] C. Pu, T. Autery, A. Black, C. Consel, C. Cowan, J. W. Jon Inouye, Lakshmi Kethana, and K. Zhang, "Optimistic Incremental Specialization: Streamlining a Commercial Operating System," in *Symposium of Operating System Principles*, (Copper Mountain Resort, Colorado), Dec. 1995.

[14] R. Marlet, S. Thibault, and C. Consel, "Efficient Implementations of Software Architectures via Partial Evaluation," *Automated Software Engineering: An International Journal*, vol. 6, pp. 411–440, October 1999.

[15] Institute for Software Integrated Systems, "The ACE ORB (TAO)." www.dre.vanderbilt.edu/TAO/, Vanderbilt University.

[16] A. S. Krishna, D. C. Schmidt, R. Klefstad, and A. Corsaro, "Towards Predictable Real-time Java Object Request Brokers," in *Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS)*, (Washington, DC), IEEE, May 2003.

[17] N. Wang, K. Parameswaran, and D. C. Schmidt, "The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware," in *Proceedings of the 6th Conference on Object-Oriented Technologies and Systems*, (San Antonio, TX), pp. 103–118, USENIX, Jan/Feb 2000.

[18] G. Daugherty, "A Proposal for the Specialization of HA/DRE Systems," in *Proceedings of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM 04)*, (Verona, Italy), ACM, Aug. 2004.