Rapid Performance Prediction for Library Components

Sibylle Schupp, Marcin Zalewski, Kyle Ross
Dept. of Computing Science
Chalmers Technical University
{schupp,zalewski,kyle}@cs.chalmers.se

ABSTRACT
Component-based programming is a methodology for designing software systems as assemblages of components with a low degree of coherence and a high degree of orthogonality. Decoupling and orthogonality, however, require coupling and assembling on the side of the component’s client. This paper addresses performance problems that occur in the composition specifically of library components. We discuss the design and implementation of a composer, which assembles library components based on a classification of their declarative performance descriptions. Employing an off-the-shelf decision-tree procedure for selecting, and the C++ technique of traits for propagating the desired behavior throughout the whole library, our system allows for rapid performance predictions. It is applied to FFTL, an “STL-like” C++ library for the Fast Fourier Transform.

1. MOTIVATION
In this paper, we address performance-related component composition problems in the context of component libraries. A library adds the aspect of “completeness” to a general component-based system [8]. Aiming at capturing not only the functional properties of a computational domain or task but also their non-functional properties, a library always offers multiple variants of the same computational task—the more variants, the higher the reusability of a library. In practice, the number of variants a library offers is therefore often quite large. On the other hand, a library is usually delivered as-such, as a mere set of components, which leaves it to the users to compose them in some way; they must properly choose and assemble the components or else their program may not function properly and may suffer in terms of efficiency.

How overwhelming the number of choices can be, we realized ourselves during the development of a component-based library for the Fast Fourier Transform (FFTL) [10]; the initial stage of the library already offered 256 fully functionally equivalent combinations, and not even we, as the developers, knew which configurations were better than others, and under which circumstances. Fairly comprehensive tests, on the other hand, showed that wrongly chosen components result in performance losses of up to a factor of 11!

Yet, selecting the right components of a library is only half the battle. It is equally important to properly propagate any choice made. For example, choosing a certain memory component without linking container components to it, will not have the desired effect or may even result in a relative slow-down. As we experienced with our FFTL, however, not even an intimate knowledge of a library makes it easy to guarantee that all components fit together in a way so that the composition as a whole behaves (“runs”) as expected. For an arbitrary user, thus, it is absolutely necessary to have a tool at hand that automatically assembles the component composition with the best run-time performance: a component composer.

At first glance, such composer might require a lot of machinery. Since libraries often contain parameterized types and therefore often come partially as source code, the composer is concerned not only with low-level performance details, but also with high-level source code. For example our FFTL, like almost all libraries in C++, contains class templates and function templates, which a composer must properly instantiate with other library components. Composing library components might therefore involve a syntactic and semantic analysis of source code—and perhaps ultimately does. In this paper, however, we go a different direction and present a simple composition system for rapid performance predictions, which essentially plugs together two existing tools, a decision system and a compiler. Our composition system consists of three parts:

- declarative, XML-based descriptions of the performance factors of, e.g., the deployment or the compilation site;
- an off-the-shelf (external) decision tree system;
- traits classes, which (statically) interface a library and encapsulate its performance factors.

In short, the composition system performs two mappings: first it invokes a decision-tree procedure with the XML performance descriptions of the user’s actual environment along with all previous compositions that empirically proved “good.” Second, it maps the recommended configuration of the resulting decision tree to the static parameters of the traits interface classes. The traits technique [11] is a well-known idiom for meta-programming in C++, forcing an ordinary compiler to propagate certain information during its pass of template instantiation, as a substitution of parameter bindings. In our context, traits force the ordinary C++ compiler to propagate the recommended configuration of components automatically and without any further dependency analysis on part of the composer. All that is required from a library designer is the development of a few traits; and all that is required from a client, is...
an XML description of the actual computational environment. The latter ones, once created, are reusable, and we plan to provide them ourselves for the most common platforms and compilers.

The remainder of the paper is organized as follows: in the next section we further elaborate on component-based libraries, in particular in contrast to other kinds of component-based systems. We discuss the architecture of our composer in Section 3 and present its implementation, applied to our FFTL library, in Section 4. A brief discussion of related and future work concludes the paper.

2. COMPONENT-BASED LIBRARIES

The particular problems that components impose on the performance prediction for applications on top of them, are well known, and there are many projects that address these problems. For example the COMPAS framework [9] can identify performance bottlenecks of enterprise applications and provide feedback for application developers at the design level of UML diagrams. Another example are the methodologies that the CAM group develops to determine, e.g., the optimal concurrency level of an Enterprise Java Beans (EJB) server [3]. In both, and many other, projects, “software components” are understood as the “binary units” of middle-ware systems such as EJB, CORBA, or .NET.

Our main intended application area is the family of so-called “STL-like libraries”, which includes the well-known Standard Template Library (STL) [1, 10, 12], with tens of thousands of users, as well as its numerous successors: libraries like MTL, BTL, VTL, GTL, and our own FFTL, which indicate already in their names an STL-like design for matrices, bioinformatics, container views, graphs, and FFTs, and in particular the Boost [5] initiative for library designers. In contrast to middle-ware components, these libraries center around fundamental data structures of computer science. Their performance costs, therefore, are less dominated by the costs of I/O or communication, as it is the case in enterprise applications, but depend much more directly on the choice of the right algorithm, iterator, or container type. At the same time, libraries are characterized by a high degree of parameterization and the explicit design goal to (ultimately) represent each performance-relevant factor as parameter. Although we have no space for further elaboration, we remark that an important “side effect” of the usage of traits is the systematic identification of the constituents that an interface of a performance-aware component needs to include. Finally, for notational clarification, we note that we use in the following the conceptual definition of components [4], in which a component is defined solely through its role in the construction of software:

A software component is any building block from which other software can be composed.

3. TRAITS-BASED ARCHITECTURE

Traits classes are widely used in modern C++, both in the libraries defined in the language standard and in newly developed libraries and applications, especially in the field of active libraries [14]. Stroustrup defines them as follows:

Think of a trait as a small object whose purpose is to
carry information used by another object or algorithm
to determine “policy” or “implementation details.” [7]

In our context the idea is to model the performance factors of a component-based system as static parameters and to represent them as policy templates via traits. Once the composer has determined which actual “policy” is appropriate, the choice of the best component is accomplished by simply rewriting, or specializing, the policy template that a trait carries. From there, as said before, the compiler picks up the policy and propagates it. In illustration, Figure 1 shows the role of traits in the interaction between the composer and FFTL. At design time, the “performance-aware” library exposes the (for the sake of the illustration: 3) static performance parameters, Decimation, Algorithm, Order, that define different performance behavior, and encapsulates them in a trait class. Internally it then refers to these parameters rather than to any particular specialization. The trait class itself forms the input of the composer, which, based on the recommendation of an external classifier, specializes each performance parameter to one particular component. For example, it specializes the Algorithm parameter to the particular algorithm component of FFTL the classifier has determined as the most appropriate.

Recommending component compositions according to their run time must, of course, be based on actual performance data. On the other hand, their selection is traits-based, thus requiring static parameters. The foremost task of the classifier therefore is to devise a mapping from the empirical data to their declarative representations in traits. This mapping has to take into account:

- Trade-off computations between different compositions
- Frequent updates of the software and hardware involved
- Uncertainty due to unknown performance parameters

Although, with the exception of MyPYTHIA [6], machine learning techniques have not yet been applied for software configuration, their choice seems quite natural. In the current implementation of the composer we use a decision-tree based approach but have designed the architecture to also allow for experiments with other strategies.
4. EXTENDED CASE STUDY: FFTL
At this point, our FFTL library contains 4 (radix-based) FFT algorithms, each of which, as always with FFTs, comes in two decimations (in time or frequency) and two input orders (bit-reversed or natural). We furthermore support two ways of computing twiddle numbers and two storage models for them. Since FFTL is designed in the style of STL, it works with potentially infinitely many (random access) containers, implementations of complex numbers, twiddle computations, etc. Even if restricted to the minimum number of components, though, there are still 256 legal compositions:

\[
4 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 256
\]

As mentioned before, all combinations are functionally equivalent but vary in their performance. Experiments have shown, however, that each of the (4 algorithms \(\times 2\) decimations \(\times 2\) orders) options is optimal in at least one combination of processor, compiler (version), input length, etc.

4.1 The Composition System

<table>
<thead>
<tr>
<th>value</th>
<th>Processor</th>
<th>attr</th>
</tr>
</thead>
<tbody>
<tr>
<td>architecture ∈ {i386, Sparc, SGI}</td>
<td>clock speed, cache, bus speed, multiplicity</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>type ∈ {RDRAM, DRAM, DDR-DRAM}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>name ∈ {GCC, VC++, Intel, MipsPRO}</td>
<td></td>
</tr>
<tr>
<td>version</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>name ∈ {Win32, Linux, Sparc, IRIX, FreeBSD}</td>
<td></td>
</tr>
<tr>
<td>version</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Configuration Attributes

In addition to traits and the mappings from the recommended composition to the specialized traits, the composition system for the FFTL consists of:

- XML descriptions for each of the tested configurations and one user-supplied description of the current configuration; Figure 3 lists all software and hardware factors we currently describe. Obviously, the configuration data has to be collected just once, and can then be reused by composers for other libraries as well as by any other tool. Currently, our system contains 3 files with nearly 1700 lines of configuration descriptions.

- An interface to a learning system, which predicts the best composition based on the user input and the existing knowledge; we have specialized this for integration of the Waikato Environment for Knowledge Analysis (WEKA) library [13]. We currently use its decision tree procedure J48, because a decision-tree-based composer helps reducing the complexity of the parameter space, which is important especially for a first prototype. At the moment, the decision procedure is restricted to the prediction of the best FFT algorithm, i.e., one of the following options: 4 algorithms \(\times 2\) decimations \(\times 2\) orders.

It is worthwhile to emphasize that users see neither the underlying classification nor the component transformation itself. All they need to do is to add a preprocessor directive to their code that includes a certain header file and then simply invoke `run_fft1` (parameterized by the length of the FFT to be run) with any iterator to a random-access container; the proper FFT implementation is then called automatically.

Once the classification system has determined the algorithm it predicts to perform best, the composer maps this recommendation to the C++ client of FFTL. More specifically, the composer creates traits defining the container used to hold the twiddle factors, the algorithm used to generate the twiddles, the permutation (e.g., bit-reversal for radix-2 FFTs), and, of course, the FFT algorithm selected by the composer. Figure 3 shows a (simplified) sample trait, used to permute the input to an FFT when necessary.

![Figure 3: Example Trait](image)

4.2 Experimental Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(M_0)</td>
<td>4 × Ultra Sparc II</td>
<td>4Gb</td>
<td>SunOS 5.8</td>
<td>GCC 3.2</td>
</tr>
<tr>
<td>(M_1)</td>
<td>8 × 195mhz Mips r10000, 4Mb cache</td>
<td>1.5Gb</td>
<td>Irix</td>
<td>MipsPro</td>
</tr>
<tr>
<td>(M_2)</td>
<td>1 × i386</td>
<td>FreeBSD 4.5</td>
<td>GCC 3.0.4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(M_3)</td>
<td>1 × Ultra Sparc II</td>
<td>SunOS 5.8</td>
<td>GCC 3.2</td>
<td></td>
</tr>
<tr>
<td>(M_4)</td>
<td>FreeBSD 4.7</td>
<td>GCC 3.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(M_5)</td>
<td>4 × Ultra Sparc II</td>
<td>4Gb</td>
<td>SunOS 5.8</td>
<td>GCC 3.0.4</td>
</tr>
</tbody>
</table>

Figure 4: Test Configurations
To test how well the composer is actually able to select an FFT algorithm we collected data on the 6 configurations illustrated in Figure 4. (empty entries simulate partial knowledge). The data from $M_0$, $M_1$, and $M_2$ were encoded in the training XML file and used to create a decision-tree; the configuration data for the remaining 3 machines were encoded as user configuration files. Each test machine is similar to a training machine configuration except for a change to a relevant element of the configuration: $M_3$ is identical to $M_0$ except for a change in the GCC version (viz., from 3.2 to 3.0.4). The machine $M_1$ is not similar to any of the test machines and was used to see if the composer would be distracted by “irrelevant” training data. The composer was run on each test machine for input sizes $2^3, 2^4, \ldots, 2^{26}$. Finally, the algorithm selected by the composer for each input size was compared against the empirical data for the particular machine to see how well the composer predicted the optimal (i.e., fastest) algorithm.

Graphs of these comparisons can be found in Appendix A. As expected, the optimal algorithm choices for $M_3$ and $M_5$ were almost perfectly predicted (these two machines are similar to $M_0$, one of the training machines). With $M_4$, however, only the operating system and the compiler are defined; given that most of the configuration attributes are left undefined, the composer is able to perform fair predictions for some input sizes, but for others, (e.g., $2^{21}$), the algorithm selected is over 5 times slower than the optimal algorithm.

Overall, the composer provided good, though not always the best solutions. One explanation for this is that the decision tree does not take the interaction between the performance parameters into account; other machine learning techniques (e.g., neural networks) might give more accurate predictions.

5. RELATED AND FUTURE WORK

As we argued earlier, “software performance engineering” for component-based programming at middle-ware level and at library level share the fundamental problems, but are at the same time very different in the details. Since library components already parameterize and encapsulate performance factors, our composition problems are about the proper and consistent selection of, ultimately, algorithms and data structures; concurrency and I/O, for example, currently play no role. Another difference, especially with respect to feedback for the component designer, is that we can operate mostly at source code level.

Our work is very close to design maintenance systems [3], where the goal is to integrate non-functional properties of a software system. More generally speaking, our work falls in the category of software configuration management but is much more heavyweight than the usual (script-based) configuration tools. While traditional configurators select among existing programs or files, we take a partially generative approach by appropriately specializing the components of the final configuration. Because of the integration of machine learning techniques we are able to cope with partial knowledge, which usual configuration tools cannot.

There are a number of tasks that need to be tackled next. A natural extension of the current prototype is to support more sophisticated user demands. While currently users can only request the fastest configuration, it perfectly makes sense to “optimize” a configuration for other kinds of resource usage, e.g., memory usage, power consumption, or network bandwidth. Most likely, additional traits are necessary to allow for the new kinds of transformations. Another set of experiments is driven by the need to evaluate the precision of our approach: we want to experiment with different learning algorithms, to find out whether any of them can increase the precision of the classification. A new dimension of this project includes the integration of run-time parameter bindings. Not so much for the FFT, but for other applications it could be important to include specializations at run time.

6. REFERENCES

APPENDIX
A. EMPIRICAL RESULTS

$M_3$. A plot of the “efficiency” factor for $M_3$ shows that, with the exception of one input point, the composer was able to select an optimal algorithm; for the case where it made a sub-optimal choice, this algorithm was less than 16% less efficient than optimal.\(^1\)

$M_3$. A plot of the number of possible choices better than the composer’s; in the case of the last data point—the only case in which the composer selected a sub-optimal algorithm—there was only one algorithm better than the one predicted.

$M_4$. A plot of the efficiency factor for $M_4$ illustrates the imperfect nature of the composer’s prediction: since $M_4$ does not bear a large similarity to any of the training machines, suboptimal selections are to be expected.

$M_4$. A plot of the number of algorithms better than the one selected by the composer shows that in all but one case, the composer was able to select an algorithm that was better than half of the others.

$M_5$. A plot of the efficiency factor for $M_5$ shows that the composer was able to select the optimal configuration in all cases; this is expected because of the large degree of similarity between $M_5$ (test machine) and $M_0$ (training machine).

$M_5$. Likewise, a plot of the number of choices better than the algorithm selected by the composer confirms the optimality of the predictions.

\(^1\)Efficiency plots show the speedup of the optimal algorithm over the composer-selected algorithm; as such, 1 is the target value.