Adaptive Dispatch: A Pattern for Performance-Aware Software Self-Adaptation

Petr Kubát, Lubomír Bulej, Tomáš Bureš, Vojtěch Horký, Petr Tůma
Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 118 00, Czech Republic
first.last@d3s.mff.cuni.cz

ABSTRACT

Modern software systems often employ dynamic adaptation to runtime conditions in some parts of their functionality – well known examples range from autotuning of computing kernels through adaptive battery saving strategies of mobile applications to dynamic load balancing and failover functionality in computing clouds. Typically, the implementation of these features is problem-specific – a particular autotuner, a particular load balancer, and so on – and enjoys little support from the implementation environment beyond standard programming constructs.

In this work, we propose Adaptive Dispatch as a generic coding pattern for implementing dynamic adaptation. We believe that such pattern can make the implementation of dynamic adaptation features better in multiple aspects – an explicit adaptation construct makes the presence of adaptation easily visible to programmers, lends itself to manipulation with development tools, and facilitates coordination of adaptation behavior at runtime. We present an implementation of the Adaptive Dispatch pattern as an internal DSL in Scala.

ACM Reference Format:
Petr Kubát, Lubomír Bulej, Tomáš Bureš, Vojtěch Horký, Petr Tůma. 2018. Adaptive Dispatch: A Pattern for Performance-Aware Software Self-Adaptation. In *Proceedings of ACM/SPEC International Conference on Performance Engineering (ICPE'18 Companion).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3185768.3186406

1 INTRODUCTION

Modern software systems need to cope with increasingly complex and increasingly open environments. When engineering such systems, performance awareness – or, the ability to design for and manage software performance – is a concern on par with functionality. Assessing performance in such systems requires continuous evaluation in realistic environments with realistic workloads [8].

Compared to functionality, performance is much more of an emerging property, determined by a multitude of interactions inside the executing software system. This makes managing performance through static design difficult (but still possible for certain classes of systems, such as real-time software platforms). As an alternative or complementary solution, software systems can utilize dynamic adaptation [13, 19]. An adaptive system is able to observe (measure) itself or its environment and adjust towards optimal performance at runtime.

The said adaptation functionality can be implemented in specialized code, such as load balancer modules [2, 15] or autotuners [3], or with adaptation frameworks ranging from fairly specialized autotuners [17] to architecture adaptation [16]. Our goal is to extend this spectrum with adaptation support at programming language level.

Towards our goal, we propose Adaptive Dispatch as a generic coding pattern for implementing dynamic adaptation. The programmer can use Adaptive Dispatch to identify multiple implementation alternatives in application code but postpone the selection of a particular alternative to runtime, to be determined by observed runtime performance. In this work-in-progress paper, we present an implementation of the Adaptive Dispatch pattern as an internal DSL in Scala[1], summarize the potential benefits and prepare ground for eventual evaluation.

Section 2 introduces the basic design of our Adaptive Dispatch pattern. Section 3 continues with describing the runtime behavior. Section 4 provides some discussion and related work.

2 ADAPTIVE DISPATCH PATTERN

We illustrate the Adaptive Dispatch pattern on an example where the developer needs to encode and decode JSON content, perchance used to communicate with a remote web service via its REST API.[2] For that, the developer needs to choose one of the many available JSON libraries. The selection process is influenced by many factors, such as the API feature set, licensing, documentation, and – importantly – performance.

In our example, we assume the other factors help narrow the choice down to two or three candidate libraries. All other things being equal, the developer would now like to choose the fastest library. Unfortunately, the speed of each library typically depends on the input size and the structure of the input file, and performance

ICPE'18 Companion, April 9–13, 2018, Berlin, Germany

ACM ISBN ISBN 978-1-4503-5629-9/18/04…$15.00
https://doi.org/10.1145/3185768.3186406

[1] Our implementation of the Adaptive Dispatch pattern, as well as code fragments in this paper, use the Scala programming language. Scala executes on top of any standard Java Virtual Machine, but provides the flexibility required for seamlessly integrating the Adaptive Dispatch pattern. Scala code can be invoked from Java and vice versa, which makes our implementation widely applicable.

[2] JSON is a data exchange format based on the notation for dictionary literals in JavaScript. The format is widely used in systems exporting an HTTP REST API, which thus need to encode and decode JSON content when sending a request or receiving a response.

```
import scalaadaptive.api.Implicits._
val parseJson = (
    parseFlexjson
    or { (s: String) => gson.fromJson(s) }
    or jacksonMapper.readValue
)

parseJson("""{"JSON":"some text", ...}""")
```

Listing 1: Use of Adaptive Pattern for selection of the fastest JSON library. Note that the framework allows us to freely combine functions, lambdas and methods.

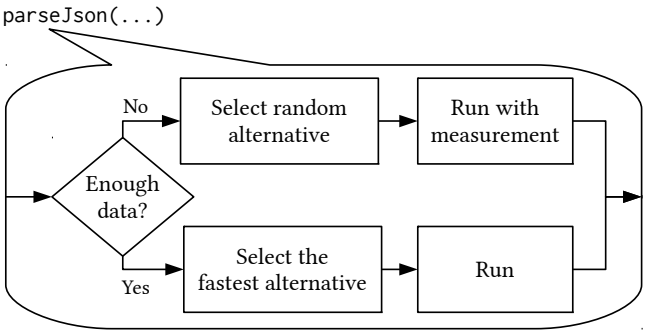

Figure 1: High-level overview of the Adaptive Dispatch pattern semantics. The call hides the actual selection of the fastest alternative and the initial measurements of all alternatives.

improvements appearing in new library versions may change the relative ranking of the libraries.

A standard approach in this situation would be to benchmark the candidate libraries and use the benchmark results to choose the library. This assumes that the performance of the library in the benchmark is representative of the performance under practical workload. When this assumption does not hold, the choice may end up being wrong. Instead, the developer can postpone the selection until runtime, employing the Adaptive Dispatch pattern.

The use of the Adaptive Dispatch Pattern in Scala is shown in Listing 1. In essence, the pattern provides the developer with a mechanism to aggregate the encoding or decoding functions of the candidate JSON libraries into a single function, which selects among the alternatives adaptively at runtime as illustrated in Figure 1.

The pattern is syntactically similar to function composition known from functional languages, where the `andThen` operator is used to chain function execution. Unlike function composition though, only one of the alternative functions is executed when the aggregate function is invoked. In this sense, the pattern is analogous to dynamic dispatch as one of the core concepts of object-oriented programming. Where dynamic dispatch chooses the method to invoke based on the target object type, the Adaptive Dispatch pattern chooses the function based on the observed performance history. As is the case with dynamic dispatch, the pattern assumes that the functions have compatible interfaces.

In more detail, the Adaptive Dispatch pattern shown in Listing 1 relies on user defined infix operators and implicit typecasts[3]. The `parseJson` function is of a custom type `AdaptiveFunction`[4], which conforms to the standard `Function` trait[5] and behaves like a normal function from the user perspective. The caller is not aware of the implementation variants – dispatched to different libraries – and works only with the aggregate interface.

Finally, the DSL provides a way to specify what factors the performance of the adaptive function depends on. In our example below we specify that the performance is likely to depend (in some way unknown to the developer) on the size of the input collection. Further, the DSL allows us to scope the measurement to distinguish that the same adaptive function is used in multiple distinct situations, which may need different decisions. Technically this determines which measurements are used to build the statistics to decide which alternative is the best given the particular input.

```
val parse = (
    parseFlexjson or parseJackson
    by (_.length)
    selectUsing Selection.InputBased
    storeUsing Storage.Global
)
```

The complete implementation consists of two core parts – the internal DSL that the developer uses to specify the interchangeable implementations of adaptive functions, and the runtime component that handles the collection of performance information and the alternative selection.

3 RUNTIME COMPONENT

At runtime, the implementation of the `AdaptiveFunction` type collects performance observations and uses them to select the function to call in each Adaptive Dispatch pattern invocation. The decision process is highly configurable to fit various application needs – each Adaptive Dispatch pattern configuration consists of a chain of Selection Policies, which determine how the data for the eventual decision is collected, and a Selection Strategy, which implements the selection once the data is available.

3.1 Selection Policies

The Selection Policies serve to regulate the overhead of selecting the function alternative on each invocation. The key idea here is that gathering performance observations and selecting the best alternative is typically computationally more demanding than just reusing the selection from the previous invocation. We permit the developer to freely configure the choice of policies in chains, such as:

- Select a new alternative every N-th call, use the last selected alternative otherwise.

[3] For object methods, this relies on the *eta-expansion* mechanism of Scala.
[4] Both `AdaptiveFunction` and `Function` types have multiple variants depending on the arity of the function, e.g. `AdaptiveFunction1`, `Function2`, etc. For simplicity, the arity is omitted from the type names in the text.
[5] Equivalent to interface in other object oriented languages.

– First gather data from M invocations, then select a new alternative on each of N invocations, and finally keep using the most often selected alternative but make a new selection every O-th call.
– For every M seconds spent on executing the adaptive function, use at most N seconds for gathering data, then at most O seconds on selecting new alternatives, and use the last selected alternative afterwards.

To illustrate the DSL syntax, here is an example of a policy chain that will gather data from 50 invocations, then select new alternative each one of 50 additional invocations, and then repeat the whole process unless the same alternative was selected the last 20 times:

```
val customPolicy: Policy = (
    gatherData
      until (totalRunCount growsBy 50)
    andThen selectNew
      until (totalRunCount growsBy 50)
    andThenIf ((stats: StatisticDataProvider) =>
        stats.getStreakLength >= 20)
      goTo (useLast forever)
    andThenRepeat
)
```

3.2 Selection Strategies

Given the collected performance observations for individual function alternatives, and possibly for individual input features, a Selection Strategy selects an alternative. We distinguish two basic types of selection strategies, mean-based and input-based. A mean-based strategy assumes the execution time of an alternative does not depend on the input, and uses standard statistical tests to select the alternative with the shortest average execution time.

The input-based strategies are intended for selecting alternatives whose execution time depends on the function input. We require that each relevant input can be reduced to an integer feature (for example a size for an input list that the function traverses), and use the performance observations associated with corresponding input features to construct a regression model that approximates the function performance. The model is then used to determine which alternative is likely to provide the best performance for a particular input. The framework currently implements three regression models which enable different trade-offs between complexity (and the implied computation overhead) and accuracy – the fastest and least accurate is a linear regression model based on least squares, followed by a window-bound linear regression, and finally a local regression (LOESS) [7].

4 DISCUSSION AND RELATED WORK

Our design goals for the Adaptive Dispatch pattern focus on ease of use from the developer perspective and general flexibility where the runtime component is concerned. We have briefly tested the design in basic use cases such as the library selection discussed in the introduction, or classical load balancing, and we envision more potential uses. On the downside, the Adaptive Dispatch pattern does not (yet) permit a simple expression of some more complex adaptation mechanisms such as the optimum search strategies used in autotuning. We have also examined the overhead associated with adaptation – the overhead obviously depends on the Selection Strategy, but for the use cases above reasonable combinations were easily available.

As a concept, the Adaptive Dispatch pattern is not limited to adaptation based on performance. With access to relevant data, the adaptation can be performed based on memory usage or energy consumption, which can be especially relevant for mobile and IoT devices. The condition itself can also change dynamically – for example, mobile applications can adapt for energy consumption while (low) on battery power but switch towards performance when connected to a charger. These changes are permitted by decoupling the dynamic invocation from the Selection Policies and the Selection Strategy – eventually, we envision the possibility of coordinating adaptation across the entire application in the style of a MAPE control loop.

If adopted, the Adaptive Dispatch pattern will make it easier to implement adaptation mechanisms, which should ultimately lead to applications with more adaptation points, i.e. more locations where a dynamic adaptation is performed. As a broader question, we should care whether such applications are still manageable, or whether the interaction of potentially many adaptation points will lead to unpredictable performance. We should, however, realize that between our highly adaptive processors, operating systems, virtual machines and middleware frameworks, this situation may already exist to a large degree. Perhaps we should pragmatically resign ourselves to the fact that our systems are and will remain highly adaptive – and in this context, introducing the Adaptive Dispatch pattern contributes to making the adaptation more explicit and more manageable.

On the related work side, the development of adaptive systems is a widely studied topic. In many cases, systems that are capable of some form of autotuning are among the fastest in their class – for example the FFTW library [10], which is considered to be the fastest non-commercial FFT implementation in the world, relies on adaptation performed statically by recompiling the application code [9]. Among well known autotuning frameworks, OpenTuner [1] can use similar process to find optimum application configuration.

Our earlier work on an adaptation framework with feedback from runtime measurements is presented in [4]. The framework focuses on architectural adaptation in component-based applications, with use cases including verification of component performance contracts or DevOps-style performance feedback integrated into performance documentation. A special language called SPL [5] is used to express the performance requirements and the adaptation conditions.

Inherent to the dynamic adaptation process is the need to predict future application performance (which the adaptation should reflect), in particular performance under certain (expected) workload. The work of Goldsmith [11] takes a runtime approach, where the basic blocks of a program are identified and their performance is measured for inputs described by selected features. Regression is used to find an approximate relationship between the feature values and the basic block execution frequencies, and the resulting model can be used to formulate predictions.

A relatively accurate prediction of execution time is provided by the Mantis framework [6], which is based on instrumenting the code and automatically identifying its essential features (loops, branches, variable values and so on). These features are characterized (branch counts, loop counts and so on) at runtime and machine learning is used to select those important for the overall performance. Finally, a predictive model can be constructed from the data [12].

Among earlier works, the authors of [18] obtain predictions in two steps. First, a greedy or genetic algorithm finds similar inputs in historical measurements, then simple mean or linear regression is used to generate prediction. Some similarities can be found also in the techniques used to construct black box performance models [14]. Our particular use calls for models that facilitate prediction in the course of function invocation, overhead is therefore of utmost importance.

5 CONCLUSION

In this work-in-progress paper, we have introduced our Scala based implementation of the Adaptive Dispatch pattern, a software construct aimed at simplifying the implementation of dynamic adaptation mechanisms. Our design aims to make the adaptation elements in software more explicit – this should not only improve readability, but also open the way for development tool support (such as tying the adaptation statistics from live systems into the information provided by the development environment) or runtime adaptation coordination (such as directing multiple adaptation points towards shared goals or preventing oscillations through interactions between multiple adaptation mechanisms). The implementation is already available at http://d3s.mff.cuni.cz/software/adp, we are currently working on use cases to assess the (currently still hypothetical) benefits.

ACKNOWLEDGMENTS

This work was partially supported by Charles University Institutional Funding (SVV) and by the Research Group of the Standard Performance Evaluation Corporation (SPEC).

REFERENCES

[1] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *PACT*. ACM, 2014.
[2] Apache Software Foundation. Apache Camel. https://camel.apache.org/, 2018.
[3] ATLAS. Automatically Tuned Linear Algebra Software. http://math-atlas.sourceforge.net/, 2016.
[4] L. Bulej, T. Bures, V. Horky, J. Keznikl, and P. Tuma. Performance Awareness in Component Systems: Vision Paper. In *Proc. 36th IEEE Computer Software and Applications Conference Workshops*, pages 514–519, July 2012.
[5] L. Bulej, T. Bureš, V. Horký, J. Kotrč, L. Marek, T. Trojánek, and P. Tůma. Unit testing performance with Stochastic Performance Logic. *Automated Software Engineering*, pages 1–49, 2016.
[6] B.-G. Chun, L. Huang, S. Lee, P. Maniatis, and M. Naik. Mantis: Predicting System Performance through Program Analysis and Modeling. *arXiv:1010.0019 [cs]*, Sept. 2010.
[7] W. S. Cleveland, S. J. Devlin, and E. Grosse. Regression by local fitting. *Journal of Econometrics*, 37(1):87–114, Jan. 1988.
[8] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and Deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.
[9] M. Frigo. A Fast Fourier Transform Compiler. In *PLDI*, pages 169–180. ACM, 1999.
[10] M. Frigo and S. G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proc. Intl. Conf. on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384 vol.3, May 1998.
[11] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring Empirical Computational Complexity. In *ESEC-FSE*, pages 395–404. ACM, 2007.
[12] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression. In *NIPS*, pages 883–891, USA, 2010. Curran Associates Inc.
[13] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.
[14] M. Kuperberg, K. Krogmann, and R. Reussner. Performance prediction for black-box components using reengineered parametric behaviour models. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08, pages 48–63, Berlin, Heidelberg, 2008. Springer-Verlag.
[15] Netflix. Ribbon. https://github.com/Netflix/ribbon, 2018.
[16] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime Software Adaptation: Framework, Approaches, and Styles. In *Companion of the 30th International Conference on Software Engineering*, pages 899–910. ACM, 2008.
[17] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, R. W. Johnson, M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. High Perform. Comput. Appl.*, 18:21–45, 2004.
[18] W. Smith, I. T. Foster, and V. E. Taylor. Predicting Application Run Times Using Historical Information. In *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–142. Springer, 1998.
[19] M. Wirsing, M. Holz, N. Koch, and P. Mayer. *Software Engineering for Collective Autonomic Systems - The ASCENS Approach.* 2015.