Static Latency Tracking with Placement Types

Pascal Weisenburger
Technische Universität Darmstadt
weisenburger@cs.tu-darmstadt.de

Tobias Reinhard
Technische Universität Darmstadt
tobi.reinhard@googlemail.com

Guido Salvaneschi
Technische Universität Darmstadt
salvaneschi@cs.tu-darmstadt.de

Abstract

Large-scale distributed applications, e.g., in geodistributed data centers, pose a performance challenge to developers which need to take high cross-data-center latency communication cost into account. We present a preliminary investigation of a type system that tracks latency and makes the cost of remote calls explicit, raising developers' awareness of communication overhead.

1 Introduction

To implement efficient and responsive geodistributed applications, it is imperative to avoid high-latency communication and take advantage of locality if possible [18]. In a distributed system, components trigger computations on other components, e.g., by sending events [10, 11, 17]. Such remote computations involve network communication, which entails significantly higher latency compared to local execution. Latencies between servers within a data center, for example, are often comparably low (under 2 ms), whereas latencies between geodistributed data centers, which may be located on different continents, can be orders of magnitude higher (over 100 ms) [1]. Ignoring this fundamental difference can lead to serious performance problems [7].

We investigate a design where the programming language makes latency explicit making the programmer aware of the latency attached to running computations. Our type system serves two purposes: (1) It infers an estimation for the upper bound of the latency attached to a computation. Developers can use this information to restructure programs that exhibit excessive latency. (2) Developers can explicitly ascribe types with a specific latency label, in which case, the type system guarantees that the estimated latency attached to a computation is not higher than specified upper bound. Otherwise, the type system statically rejects the program.

We built on our previous work on *placement types*, where the type system distinguishes between different components of the distributed system using placement specification for data and computations [16].

2 Latency Tracking

Placement Types in a Nutshell Placement types associate locations to computations. For example, the placement type `Int on A` denotes a computation *c* that produces an integer on a component `A`. We consider three data centers *A*, *B* and *C*. Using placement types, separate locations are represented by different types, modeled by distinct `Peer` subtypes in a Scala-like syntax:

```
trait A extends Peer
trait B extends Peer
trait C extends Peer
```

A computation `def c(n: Int): Int on A = 2 * n` with placement type `Int on A` executes on data center `A`:

```
def c(n: Int): Int on A = 2 * n
```

The execution of *c* can be triggered remotely (e.g., from data center `B`), in which case *c* computes an `Int` value on `A`, but the value will be available to `B` only after it is transmitted over network.

Latency Types The result type of a remote computation includes the latency *l* of the remote computation, e.g., the type of the remote value received on `B` is `Int withLatency 1`:

```
def d(): Int withLatency 1 on B = remote call c(5)
```

For simplicity, we simply count the number of remote calls in the type system to approximate the latency overhead for a distributed computation. In perspective, developers should be able to specify latency values for remote communication between components based on domain knowledge, e.g., from monitoring. Latency between two data centers is not entirely deterministic and depends on many factors. However, we assume that in the domain of geodistributed data centers, which are placed at fixed locations and typically exhibit communication latencies in the order of hundreds of milliseconds, it is feasible for developers to give adequate estimates.

Our approach goes beyond distinguishing between local and remote access. Our goal is to give developers a more faithful cost estimation of remote calls. Complex cases composing many different functions hinder keeping track of all (potentially nested) remote accesses for programmers. The type system makes the developers' assumptions on latency explicit and ensures that they hold. It tracks latency across function calls, which is especially important to guarantee that an application still complies to the annotated latencies after changing parts of the implementation, e.g., for maintenance.

Composition Composing values that may have a latency label, entails three cases:

(1) Nesting remote computations. When a computation (e.g., `d` in the example) with a latency label is called remotely, we increase the latency value, thus, tracking the accumulated total latency:

```
def e(): Int withLatency 2 on C = remote call d()
```

(2) Composing remote computations. Composing two computations (e.g., `c` and `d`) that both have a latency label requires adding latencies, which approximates the combination of two successive (synchronous) remote calls. In an asynchronous model, taking the maximum value of both latencies is more appropriate:

```
def f(): Int withLatency 3 on C = (remote call c(5)) * (remote call d())
```

(3) Interaction with unlabeled types. Traditional values of type `T` (e.g., `Int`) are assumed to have a latency of 0, i.e., `T` is treated as `T withLatency 0`. We allow unlabeled types for compatibility and to save developers from labeling every explicitly ascribed type:

```
def g(): Int withLatency 1 on C = (remote call c(5)) * 10
```


FTfJP'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5939-9/18/07…$15.00
https://doi.org/10.1145/3236454.3236486

Under- and Over-Approximation The types given in the examples are the ones computed by our type system. It is usually not a problem to over-approximate latency since this means that the developer considers a higher latency value when, in fact, the result will be available faster at run time. Hence, we allow explicit type ascriptions that over-approximate latency:

```
def h(): Int withLatency 3 on C = remote call d()
```

However, under-approximating latency can lead to performance problems at run time. Hence, we do not consider an under-approximation sound and the following code produces a type error at compile time:

```
def i(): Int withLatency 1 on C = remote call d()
```

The inferred or explicitly annotated latency value always estimates an upper bound for the latency attached to a computation.

2.1 Core Language

We sketch the type system for a core calculus. We annotate standard types T with a latency value l, assigning refined types (T, l) to terms. We distinguish between types T and placement types T on P. Terms t include standard terms and remote access through remote call. We model the definition of placed values as nested p-terms binding t-terms to names. Thus, p-terms express top-level placed bindings (expressing the program's top-level definitions, we do not assign types to p-terms) and t-terms are placed expressions, which evaluate to the value that is bound. Values are placed using the following let x with latency l on P construct:

$$\text{(T-Place)}\quad \frac{\Gamma; P \vdash t : (T, l_0) \text{ on } P \qquad \Gamma, x\colon (T, l_1) \text{ on } P \vdash p}{\Gamma \vdash \text{let } x \text{ with latency } l_1 \text{ on } P = t \text{ in } p} \quad l_0 \leq l_1$$

The type system guarantees that the computed latency label l_0 for the term t is not higher than the latency l_1 given explicitly by the developer. In case l_0 is higher, the program is rejected.

To count the number of remote calls to approximate the latency overhead, for every remote call to a computation t, we derive the refined type (T, l_0) for t. The term remote call t is given type (T, l_1), where T is the same but the latency annotation l_1 is increased:

$$\text{(T-Remote)}\quad \frac{\Gamma; P_0 \vdash t : (T, l_0) \text{ on } P_1 \qquad P_0 \neq P_1}{\Gamma; P_0 \vdash \text{remote call } t : (T, l_1) \text{ on } P_0} \quad l_1 = l_0 + 1$$

We model type-level latency tracking for operations performed on values and for control structures. For example, for an if expression, we evaluate the condition expression *and* we evaluate *either* the then branch *or* the else branch:

$$\text{(T-If)}\quad \frac{\begin{array}{c}\Gamma; P \vdash t_0 : (\text{Bool}, l_0) \text{ on } P \\ \Gamma; P \vdash t_1 : (T, l_1) \text{ on } P \\ \Gamma; P \vdash t_2 : (T, l_2) \text{ on } P\end{array}}{\Gamma; P \vdash \text{if } t_0 \text{ then } t_1 \text{ else } t_2 : (T, l) \text{ on } P} \quad l = l_0 + \max(l_1, l_2)$$

To (over-)approximate latency, the latency of the resulting type is the sum of (i) the condition and (ii) the maximum of the branches, i.e., we calculate the upper bound over both branches.

In our current model, we do not support latency computation over unbounded loops or for general recursion. Collection types with statically known sizes – e.g., as in Shapeless [13] – could be used to give an upper bound, e.g., on the number of iterations. Operating in the domain of a fixed set of geodistributed data centers also helps making the case of unbounded loops less common.

2.2 Outlook

We plan to conduct a case study consisting of multiple geodistributed data centers. Clients query the nearest data center to achieve decent user experience. We want to guarantee certain properties statically, e.g., that data paths dealing with local data do not involve remote communication or that certain operations do not exhibit unexpectedly high latency. Although we need to resort to a synthetic setting, we intend to base the locations of the data centers on real ones [5], specifying realistic latency values for remote communication between data centers.

We are currently working (i) on completing our formal model to prove our system sound and (ii) on a Scala embedding for latency tracking integrated into our ScalaLoci[1] language, which provides an implementation of placement types. Latency values in our Scala embedding can be represented using type-level Peano numbers. Since latency refinements are only used statically but not accessed dynamically, they can be phantom, imposing no run time overhead. Alternatively, literal singleton types (as implemented by the Type-level Scala compiler [15] and Dotty [4]) can be used for latency refinements, performing arithmetic operations using a macro.

We currently do not consider lost messages, but message loss and retransmission increases latency and should be considered as part of the latency estimation. A further refinement is to associate probability estimates to latency values.

3 Related Work

To the best of our knowledge, no previous work explores type level latency tracking to promote low latency computations. Jost et al. [6] augment a type system with amortization (a complexity analysis that averages over time [14]) to statically analyze worst-case execution time and heap space bounds depending on the input of the program by annotating types with cost values. Their approach, however, targets embedded systems where both time and space bounds are important. Similar to their approach, we plan to annotate place types with latency values between components to compute the overall worst-case latency. End-to-end latency for data flows in cyber-physical systems has been analyzed based on architecture models, incrementally refining the analysis while refining the model [3]. Energy Types [2] raise the developer's awareness of potentially inefficient code by encoding *phases* (with distinct patterns of energy consumption) and *modes* (with an associated energy requirement, e.g., *high* or *low*) in the type system.

Information flow type systems, which have been used to enforce security policies [9], are related to our approach in the sense that they track additional information associated to data and computations at the type level. Opposed to security labels, we annotate computations with latency values. Explicit placement in types has been investigated in ML5 [12]: Possible *worlds*, as known from modal logic and Kripke structures, define placement of computations on different components. Encodings of types parameterized with numbers are discussed, e.g., by Kiselyov [8] in Haskell. A similar approach is applicable to Scala.

Acknowledgments

This work has been co-funded by the German Research Foundation (DFG) as part of project C2 within the Collaborative Research Center (CRC) 1053 – MAKI, and by the DFG project SA 2918/2-1.

[1] scala-loci.github.io

References

[1] Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. 2017. Geo-distribution of Actor-based Services. In *Proceedings of PACMPL (OOPSLA '17)*. ACM, New York, NY, USA.

[2] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. 2012. Energy Types. In *Proceedings of OOPSLA '12*. ACM, New York, NY, USA.

[3] Julien Delange and Peter H. Feiler. 2014. Incremental latency analysis of heterogeneous cyber-physical systems. In *Proceedings of REACTION '14*.

[4] Dotty. 2013. http://dotty.epfl.ch/. (2013). Accessed 2018-04-16.

[5] Google Data Center FAQ. 2012. http://www.datacenterknowledge.com/archives/2012/05/15/google-data-center-faq/. (2012). Accessed 2018-04-16.

[6] Steffen Jost, Hans-Wolfgang Loidl, Norman Scaife, Kevin Hammond, Greg Michaelson, and Martin Hofmann. 2009. Worst-case execution time analysis through types. In *Proceedings of ECRTS '09*.

[7] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. 1994. *A Note on Distributed Computing*. Technical Report.

[8] Oleg Kiselyov. 2005. Number-parameterized types. *The Monad.Reader* 5 (2005).

[9] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. 2009. Fabric: A Platform for Secure Distributed Computation and Storage. In *Proceedings of SIGOPS (SOSP '09)*. ACM, New York, NY, USA.

[10] Manisha Luthra, Boris Koldehofe, Pascal Weisenburger, Guido Salvaneschi, and Raheel Arif. 2018. TCEP: Adapting to Dynamic User Environments by Enabling Transitions Between Operator Placement Mechanisms. In *Proceedings of DEBS '18*. ACM, New York, NY, USA.

[11] A. Margara and G. Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: the Cost of Consistency. *IEEE Transactions on Software Engineering* (2018).

[12] Tom Murphy, VII., Karl Crary, and Robert Harper. 2008. Type-safe Distributed Programming with ML5. In *Proceedings of TGC '07*. Springer-Verlag, Berlin, Heidelberg.

[13] Miles Sabin. 2011. Shapeless. http://github.com/milessabin/shapeless. (2011). Accessed 2018-07-01.

[14] Robert Endre Tarjan. 1985. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (1985).

[15] Typelevel Scala. 2014. http://typelevel.org/scala/. (2014). Accessed 2018-04-16.

[16] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. In *Proceedings of PACMPL (OOPSLA '18)*. ACM, New York, NY, USA.

[17] Pascal Weisenburger, Manisha Luthra, Boris Koldehofe, and Guido Salvaneschi. 2017. Quality-aware Runtime Adaptation in Complex Event Processing. In *Proceedings of SEAMS '17*. IEEE Press, Piscataway, NJ, USA.

[18] Mike P. Wittie, Veljko Pejovic, Lara Deek, Kevin C. Almeroth, and Ben Y. Zhao. 2010. Exploiting Locality of Interest in Online Social Networks. In *Proceedings of CoNEXT '10*. ACM, New York, NY, USA.