Optimizing across interfaces

Tim Harris
Microsoft Research
tharris@microsoft.com

Eric Koskinen
Brown University
ejk@cs.brown.edu

Abstract

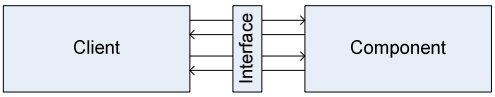
Programmers writing distributed systems are poorly served by current optimization technologies. Although there are well-known ways to improve performance – batching requests, caching results, speculation and so on – these must be applied manually, adding to the burden already faced when writing distributed systems. We argue that this kind of transformation should be done automatically by a compiler or runtime system. To do this we propose that components' interfaces provide high-level summaries of the semantics of their operations. These are used to identify when operations can be batched together, when the results of one operation can be cached on a client, when cached results must be invalidated, and when operations can be issued speculatively. We present a prototype which we use to improve the performance a multi-tier web application that accesses a remote database and business tool that queries an LDAP server.

1. Introduction

Programmers face a raft of challenges that have emerged over the last decade: (*i*) processor road-maps show increasing degrees of *parallelism* rather than increased clock rates: software must use this parallelism if its performance is to increase on newer hardware, (*ii*) *distribution* across the internet means that applications are often structured as interacting components raising problems of independent failures, high-latency wide-area communication and, in turn leading to (*iii*) *security* challenges when interacting with untrusted parties or across untrusted network links. We are investigating how languages and software stacks should be structured to tackle these challenges.

This paper focuses on one part of this problem: the challenge of optimizing RPC-like interactions between components in distributed systems, for instance optimizing a series of calls on a set of database stored procedures, or the sequence of messages exchanged between an application's front end on one machine and its back end on another, or a series of XML-RPC requests from a JavaScript client to a Web Service. Our goal, in each case, is to reduce the number of communication delays incurred.

For the majority of the paper we consider a very simple system model in which a single client interacts with a component by making invocations and receiving responses on a typed interface that the component implements:



In this model, when we talk about a 'component' we mean any part of a system whose internal state is encapsulated by an interface – this means that we can analyze and transform the stream of invocations and responses on the interface without the encapsulated state being modified 'behind our back'. As we illustrate with some of our examples, this can include suitably structured modules within a single process, as well as our motivating examples from distributed systems.

The role of the optimizer in this model is to reduce the number of round trips that must actually be made to the component. We assume that the remote component and its interface are written by expert programmers, but that the client code may be written by less experienced programmers – for example the client may be a quick web 'mashup' or prototype business tool. The optimizations we use are well known: client-side *caching* of the results of previous requests, *batching* of multiple requests into a single message, and *speculating* about future requests so that they can be performed before their results are needed.

The novelty is that we propose automating these techniques rather than requiring programmers to use them by hand. Of course, a suitably skilled programmer can outperform our system: the value of our approach is that it delivers benefits without requiring such expertise; if we can do this well enough then we can make it easier for more programmers to write distributed systems that perform well.

Our approach is for a component's interface to include what we call an *abstract implementation* (AI) containing a simple reference implementation of the interface's operations. These AIs are used to identify optimizations to make to client code: if the reference implementations commute then so do the underlying operations on the component, and if the reference implementation is side-effect free in a given context then the underlying operation can be ommitted or issued speculatively.

AIs are written in a simple imperative language that is designed to make it easy to answer this kind of question

at compile-time and at run-time, and also to make it clear to a component's author what kinds of transformation will be made. We give some examples of AIs in Section 3. Optimizations can be deployed incrementally: only a component's external interface needs an AI, and no transformations are made on un-annotated interfaces.

In Section 4 we show how to use AIs at compile time to derive data-flow equations to use in static optimizations like common sub-expression elimination and dead code elimination, and also how to use AIs dynamically to identify optimizations that cannot be made at compile-time. We conclude by demonstrating the benefit of these runtime transformations with two case studies (Section 5) where we have accelerated access to stored procedures by a web application, and to an LDAP directory by an business tool.

2. Design space & related work

Optimization goal. Our transformations aim to make *fewer* calls on a component. In contrast, many domain-specific optimization techniques aim to make individual calls *faster*. These complementary approaches reflect the different bottlenecks systems may have: we want to reduce network round-trips caused by RPCs, while others want to reduce computation within each call. For example, Pu *et al.* use specialization and partial evaluation to accelerate system calls [7], and compilers like Broadway [5] and active library systems [9] select between different implementations of general operations – e.g. using the numerical examples that motivate Broadway's design, selecting based on whether or not a matrix passed to a method is in some known special form.

Generality. We aim to support a broad class of compile-time and run-time transformations, driven by the same semantic descriptions. Some distributed computing frameworks provide support for very specific optimizing transformations. The C-JDBC [2] database access middleware can cache SQL query results and invalidate the cache as updates are made. The .NET Framework `CacheDuration` property allows a result to be cached *on the server* for a fixed duration. This can avoid re-computing costly operations but does not reduce round-trips from the client. Extensible compilers, such as MAGIK [3], can support general purpose transformations. However, they do so by using additional compiler modules to define the new transformations. Such a framework could be used as a foundation on which to build our system, but using it directly requires programmers to be aware of the specific compiler's intermediate code format, working at a much lower level than our AIs.

Semantics preservation. A contentious view we take is that a component's AI *can* change the behavior of programs that use it – e.g. the abstract implementation may indicate that two operations commute when in fact they do not. If this happens then we argue that the component is incorrect: the AI *forms part of the interface specification* to the component albeit one that is not yet amenable to static checking. There is a complex trade-off here and our decision is experimental. Broadway also allows annotations to change the semantics of a program [5] and we agree with the argument that representing semantic information on an interface enables researchers to try to verify it, whereas just documenting it in comments encourages client code to use ad-hoc caching which can be a source of subtle errors. Other systems guarantee semantics preservation but the techniques do not seem to scale beyond small cases [8], or deal with non-memory side effects [4].

3. Abstract implementations

Consider the following code using a library-implemented extensible array:

```
for (int i = 0; i < v.Count(); i ++) {
  if (v.GetAt(i) == a) {
    ...
  } else if (v.GetAt(i) == b) {
    ...
} }
```

There are several opportunities for optimization: the first call to `GetAt` makes the second call redundant, and the call to `Count` would be better placed outside of the loop. Assuming for now that these are the only two operations on the array's interface, these transformations could be enabled by writing:

```
[FactType("ITEM(int) -> Object")]
[FactType("NUMITEMS() -> int")]
public interface ExtensibleArray {
  [Effect("NUMITEMS() = return")]
  public int Count();

  [Effect("ITEM(idx) = return")]
  public Object GetAt(int idx);

  ...
}
```

This example lets us introduce the basic concepts. `FactType` definitions declare the type of information that the abstract implementation will be working on. We call this information the *abstract state* and it comprises a set of *facts* that the AI reads and writes as operations are called on the component. As an example, Figure 1 shows the abstract state just before the second call to `GetAt(0)`: everything the facts indicate is true (10 is in the list, there are 3 items in total), but the facts omit some information (whether or not 20 is in the list), and abstract away details of the particular implementation (that the array is built out of chunks holding 2 elements).

`Effect` statements provide the AIs themselves, describing how operations change the abstract state: `GetAt`'s AI `ITEM(idx)=return` allows the second call to `GetAt(0)` to return the value `10` recorded in the abstract state by matching it with the fact `ITEM(0)=10` defined by the first call.

A more complex example comes from our earlier work on optimizing software transactional memory (STM) [6]. In outline, the STM library exposes operations `OpenForRead` and `OpenForUpdate` that are called before the first time an

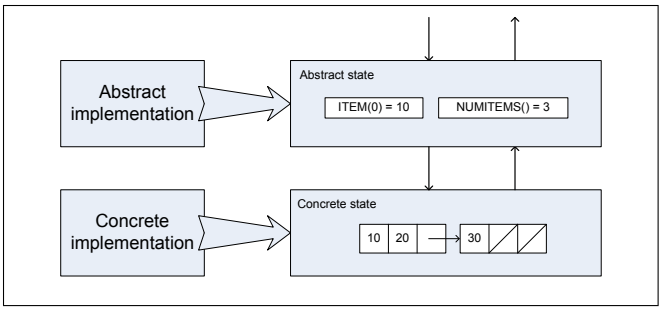


Figure 1: Abstract and concrete states for an extensible array interface and a list-based implementation of it.

object is read or updated within a transaction. The compiler we used inserts `Open*` calls before every data access and then removes those that it can prove are redundant (i.e. where the object is already open).

Originally we did this by extending the compiler's intermediate code and analyses to handle these `Open*` operations. That was several weeks of effort for people already experienced in the compiler we were using. We can accomplish the same effect in 7 lines with an AI for the STM interface:

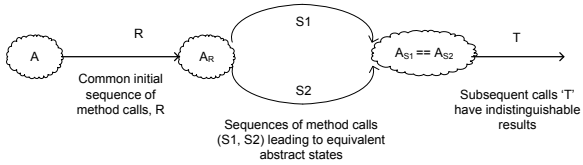
```
[FactType("OPEN_FOR_READ(Object) -> bool")]
[FactType("OPEN_FOR_UPDATE(Object) -> bool")]
[DefaultEffect("OPEN_FOR_READ(*) = ?;
                OPEN_FOR_UPDATE(*) = ?")]
public interface STM {
  [Effect("OPEN_FOR_READ(obj) = true")]
  public void OpenForRead(Object obj);

  [Effect("OPEN_FOR_READ(obj) = true;
           OPEN_FOR_UPDATE(obj) = true")]
  public void OpenForUpdate(Object obj);
  ...
}
```

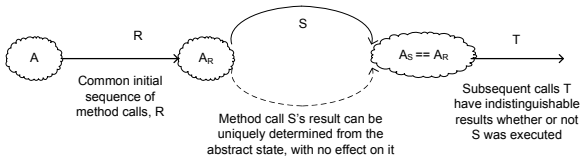
The `OPEN_FOR_*` facts show which objects are already open. These AIs allow the second and subsequent calls to open a given object to be removed: after one call, any subsequent ones have no side-effect on the abstract state and can therefore be removed (because the `OPEN_FOR_*` fact has already been defined by the first call). The `DefaultEffect` says that operations without their own `Effect` will leave objects in an unknown state – e.g. management operations to start/commit transactions.

4. Transformations

Informally, we treat the AI as a reference implementation: if transformations can be made on operations' abstract implementations then they can be made on the operations themselves. We can define this more formally in terms of traces of invocations and responses on the component. Figure 2(a) illustrates this: starting from abstract state A_R reached by a series of method calls R consider two further series of calls, $S1$ and $S2$, reaching abstract states A_{S1} and A_{S2} respectively, and then the subsequent execution of a final common series of calls T. For all series of calls R and T, we require



(a) If two series of calls have equivalent effects on the abstract state, then they must be indistinguishable on the concrete state.



(b) If an operation's result can be derived from the abstract state, with no side effect on it, then subsequent operations must be indistinguishable whether or not it is actually called.

Figure 2: Criteria for making transformations.

that if the two abstract states A_{S1} and A_{S2} are equal[1] then the calls in T must return the same results executed after $R;S1$ and after $R;S2$.

This definition lets us re-order operations (viewing the original program as $S1$ and the optimized program as $S2$) and remove an operation with a `void` return type where it does not change the abstract state. For methods with non-`void` return types, we use `Effect` attributes to derive results directly from the abstract state: if the set of assignments in an `Effect` attribute can be satisfied by binding a *unique* value to `return` then that value can be returned without invoking the underlying method. Figure 2(b) depicts this.

We call an AI with these two properties a *faithful abstraction* of the data type being manipulated through the interface: the abstract state must capture enough about the data type for transformations based on it to be those intended[2].

How can we check for conformance between a given component implementation and an AI that it claims to adhere to? Many of the transformations mentioned in Section 1 are evidently safe only given domain knowledge of interfaces in question – for example, the validity of eliding particular calls on the STM library is dependent on a complicated argument about the concurrent algorithms used by the STM. Other examples involve external devices or network services both of which remain research topics in terms of formal methods. If correctness relies on this kind of argument then what can we do to check a component against an AI?

Our current plan is to use dynamic testing. The component implementation under test will provide an *abstraction*

[1] We must be careful about what it means for two abstract states to be equal: each assignment of `?` creates a fact with a *distinct* unknown value which are not equal to one another.

[2] For readers familiar with *abstract interpretation*, this property distinguishes the information tracked by an AI from that tracked by other abstract interpretations of the same operations.

function that maps its internal state to a set of facts in the abstract state. For instance, in the list-based set example of Figure 1, the abstraction function would create facts that represent the set's cardinality and each of the items present.

The test harness would invoke operations and compare the facts generated by the AI (F_A) against the facts derived from the concrete implementation (F_C). An error is reported if F_A contains a fact not in F_C: this would indicate that the abstract state includes knowledge that cannot be derived from the concrete state. An error is also reported if two runs leading to equal abstract states F_{A1} and F_{A2} are followed by different results to a subsequent operation, showing that a transformation that could be made according to the criteria in Figure 2 could change a program's behavior.

4.1 Static transformations

AIs can extend the reach of static analyses, allowing common subexpression elimination (CSE) [1] and dead code elimination (DCE) [1] to remove redundant calls.

CSE avoids re-computation of the same expression. It is based on *available expression analysis* which is a forwards data flow analysis that produces a set, $\mathrm{avail}(n)$ which, for an instruction n, includes expressions guaranteed to be evaluated on all paths that lead to n without any of the variables they depend on having been subsequently changed.

We extend these sets to include facts that are guaranteed to be in a component's abstract state. Facts are added to the sets by executing operations' AIs. Facts are removed from the sets whenever (*i*) explicitly storing `?` into an object's abstract state, (*ii*) replacing an existing fact with a new one with the same left hand side, (*iii*) updating a value on which an existing fact depends, or (*iv*) invoking a method on an object reference that may be aliased with the one that a fact relates to. For instance, a fact `l1:CONTAINS(r1)=True` indicating that list `l1` contains element `r1` would be removed by an update to the reference `r1` (because the new object `r1` refers to may not be in the list), or by an invocation of a method on an object that may be aliased with `l1`.

Given the result of this analysis, we use the rules from Figure 2 to identify calls that can be removed: if an operation's AI has no effect on the facts at a given point, then its effect on the concrete state must be indistinguishable.

DCE removes computations that have no effect on the program's subsequent behavior – e.g. those that compute values that are never used. We can use AIs as the basis of a backwards dataflow analysis to identify opportunities for this kind of transformation: we remove an operation if its effect on the abstract state will be overwritten by another operation without being read in the mean time.

4.2 Dynamic transformations

Working at run time it's often possible to identify transformations that are impossible to make statically – e.g. ones based on the particular path taken through a program, or based on knowledge of exactly which object a given reference refers to. Conversely, identifying transformations at run time introduces costs. This makes dynamic optimizations best suited to components with significant redundancy in their usage and which take a sufficient time to execute that the cost of identifying redundancy is worthwhile. The systems used in our case studies provide two such examples.

Static CSE is straightforward to transfer to a dynamic setting. The general idea is to track facts that must be present in a component's abstract state, and to use the interaction between this state and operations' AIs to identify and avoid redundant invocations. For each component we start with an empty abstract state, and iteratively apply the effects of each operation invoked. We eliminate calls whenever an operation with a `void` return type has equivalent abstract states before and after invocation or, for non-`void` methods, if the abstract state provides a unique binding for `return`.

Static DCE is a *backwards* data flow analysis and so is harder to apply during forwards execution at run-time. Our solution is to buffer operations in a *deferred queue*, in the hope that a later operation will allow a deferred operation to be removed without being sent to the component. Furthermore, if operations accumulate in the queue they can be issued in a batch rather than sequentially.

In our prototype, only certain methods can be deferred: they must have `void` return types, and the parameters of their AIs must be marked as cloneable with `*` in the `FactType` definitions so their parameters can be cloned when buffering them. A deferred operation op_i must be performed if a later operation op_j is performed and the two operations do not commute in their effect on the component's abstract state.

5. Case studies

We have built a prototype of our system which implements the dynamic optimization ideas from Section 4.2. Our prototype runs on Version 2 of the .NET Framework. The user's bytecode is rewritten so that classes that implement interfaces with abstract implementations wrap each of the operations with a call to a new library that is responsible for processing the abstract implementations and deciding whether or not to call the underlying method. In a full implementation we would use dynamic bytecode generation to compile the AIs to bytecode which could then be compiled to native code by the .NET Framework.

5.1 Case study 1: stored procedure calls

Our first case study uses DotNetNuke (`http://www.dotnetnuke.com/`), a web forum application which uses stored procedure calls to query and update a database; generating a single web page can involve dozens of stored procedure calls. Many calls are for the same stored procedures, and often with the same arguments. For example, loading the "site settings" administration page generates 35 calls, of which 10 are made redundant by earlier calls in the same page – e.g. the `GetSkin` stored procedure takes three arguments to

looks up the appropriate skin (i.e. web site theme) from the database, and each page calls `GetSkin` several times.

We added AIs to the `SqlDataProvider` component in DotNotNuke which defines methods corresponding to the stored procedures. In this case methods such as `GetSkin()` return `IDataReader` objects which act as iterators over the rows that the stored procedure selects. We therefore provide custom clone operations to keep a buffer to remember values read from the database and recall them when an `IDataReader` is cloned and reused by a subsequent call. In a full implementation we would modify the .NET Framework rather than the application, and would attach the abstract implementations directly to the signatures of the stored procedures rather than needing to find an internal interface like `SqlDataProvider` within the application.

Quantitative results are clearly highly dependent on the test involved, but the AIs we used allowed all 10 redundant calls to be removed, providing some confidence in the expressiveness of our approach.

5.2 Case study 2: LDAP queries

Our second case study is `OrgNavigator`, an internal tool to view Microsoft's org-chart based on information from LDAP queries. It behaves responsively when used in Redmond (where the server it queries is located), but slowly when used from Europe because it makes synchronous trans-Atlantic queries. Queries are repeated from scratch each time the user navigates to another employee – even though the typical forms of navigation are to an employee's immediate manager, or one of the people that reports to them.

The tool is built around a class for querying management chains. Within this is the method:

```
public Employee FindEmployee(string name,
                             string[] extra)
```

The second argument is a list of strings to select additional information to collect from the LDAP server. For these management chain queries, the second argument is a singleton list containing the string `manager`. The simplest form of AI is to indicate that an `Employee` object can be re-used when it is the result of a query with the same `name` and `extra` information, using `Equals`/`Clone` methods on the mutable `extra` array. The FactType and annotation are therefore:

```
[FactType("EMPLOYEE(String, *String[])
                              -> Employee")]
...
[Effect("EMPLOYEE(name, extra) = return")]
```

An alternative form of annotation is possible at a lower level in the system – placing it in the implementation of the `DirectorySearcher` class that is used to query the LDAP server, without needing to change the application at all.

As expected, with our optimizations turned on, the tool can navigate between employees without incurring network traffic for those that have already been seen.

6. Future work and conclusions

In this paper we have introduced the idea of extending the interfaces between parts of a system with *abstract implementations* from which the compiler or runtime system can identify opportunities for optimizing series of calls across the interface. The general approach has been to use a simple imperative language for writing the AIs, in the expectation that this will provide a setting in which component authors will intuitively understand the transformations that the system will and will not perform.

There are several topics we do not have space to talk about in detail in this paper. In particular, we have focused on transformations made on a single client interacting with a component, rather than the case of multiple clients interacting concurrently with the same component.

This later case is more complicated and leads to interesting analogies with memory consistency models (when can reads and writes appear to be re-ordered by the compiler or memory subsystem?). One approach we are exploring is to follow that taken in memory subsystems by using AIs to build invalidation protocols: when a component performs an operation for one client then the component will send invalidation messages to other clients that may have cached state that the update conflicts with. An alternative approach, with synergy with our work on transactions, is to only perform transformations *within* a transaction since those operations will, in any case, run in isolation from other clients.

References

[1] A. W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, 1997.

[2] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *USENIX '04 Annual Technical Conference*, June 2004.

[3] D. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *Software Engineering*, 25(3):387–400, 1999.

[4] A. Greenhouse and J. Boyland. An object-oriented effects system. In *ECOOP 1999, LNCS 1628*, pages 205–229.

[5] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. *SIGPLAN Not.*, 35(1):39–52, 2000.

[6] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06*, pages 14–25.

[7] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: streamlining a commercial operating system. In *SOSP '95*, pages 314–321.

[8] M. Vandevoorde. Exploiting specifications to improve program performance. Technical Report MIT/LCS/TR-598, 1994.

[9] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM, 1998.