In Proc. Workshop on Java for High Performance Network Computing at EuroPar'98, Southampton, September 2–3, 1998. Earlier version: Proc. 7th Int. Workshop on Compilers for Parallel Computers CPC'98, pages 34–41, Linköping, Sweden, June 29 – July 1, 1998.

Locality optimization in JavaParty by means of static type analysis

Michael Philippsen and Bernhard Haumacher
University of Karlsruhe, Germany
phlipp@ira.uka.de and hauma@ira.uka.de

Abstract

On clusters and DMPs, locality of objects and threads and hence avoidance of network communication, are crucial for performance. We show that an extension of known type inference mechanisms can be used to compute placement decisions that improve locality.

In addition to this general contribution, the paper specifically addresses the problems that are caused by the distributed Java environment. Since the JVM is assumed to be fixed, the optimization is done as source-to-source transformation.

1 Introduction

For programming languages that target distributed memory parallel computers (DMPs), locality is crucial for performance. Data structures that are used together should be stored on the same node, processes/threads should be executed where the data is located that they access. If either form of locality is not achieved, network access (high latency and low bandwidth) is likely to become a bottleneck.

For imperative programming languages, literature is full of techniques that enhance locality. Various parallel Fortran dialects have compiler pragmas to express distributed layout of array data [5, 12]. Some work has been done to determine distributed array layout through the analysis of index expressions [10, 18]. Loop restructuring techniques have been studied extensively in parallelizing compilers and for cache optimization purposes [2, 20]. The owner-computes-rule and the inspector-executor [11] are used to determine the scheduling of expression evaluation in distributed environments.

However, these techniques can rarely be applied to object-oriented code since, in general, it is not array-based. Moreover parallelism does not stem from forall-, doacross, or doall-loops but is instead expressed by means of thread objects. Little work has been done specifically for parallel object-oriented languages. From over a hundred existing imperative concurrent object-oriented languages (COOL) surveyed in [15] more than half do not consider the locality problem at all. The reasons are different: Some languages have only been implemented in a prototypical way on a single workstation, where network latencies do not occur; their developers have mainly been interested in the design of coordination mechanisms and a proof of concept. Other languages are restricted to shared memory multiprocessors, they rely on the cache systems provided by those machines. Most of the other languages are used to do research in concurrency coordination constructs. Threads and explicit synchronization as used in Java are not optimal for object-oriented languages because this approach suffers from various types of inheritance anomaly[13].[1]

There are at least three orthogonal approaches to deal with locality in parallel object-oriented languages. The basic approach is to let the programmer *specify placement and migration explicitly.* Since locality does not affect the semantics of a program, the programmer in general is required to express locality information by means of annotations. Several suggestions have been made, e.g. [19]. In contrast to explicit locality information in the code, both the runtime system and the compiler might help. *Dynamic object distribution* is based on a runtime system that keeps track of the call graph and the invocation frequencies. Clever graph distribution techniques are used to (re-) distribute objects and threads by migration. Dynamic object distribution has two disadvantages. First, since there is no knowledge about future call graphs and invocation frequencies, in general object placement decisions for newly created objects are far from optimal. Second, creation of objects that cannot migrate because they are only meaningful on a certain node (e.g.

[1] A very basic introduction to inheritance anomaly: The problem is that the lines of code that implement the synchronization requirements may be spread across all methods of a class. If a subclass has slightly different synchronization needs, inheritance anomaly is likely to occur: then instead of inheriting methods from the parent, nearly all methods must be re-coded in the subclass. However, in the re-implementations, the algorithms remain unchanged, just the synchronization code lines are modified. Code duplication results in higher maintenance efforts.

file handles, GUI-windows, etc.) often results in a broad re-distribution of other objects. *Static object distribution* is the compile time alternative. Whereas dynamic object distribution is predominantly aimed at the improvement of locality in running programs, static object distribution kicks in at the time of object creation. Based on a thorough program understanding developed during compile time, the compiler tries to predict the best node for a new object, i.e. the compiler predicts the node that will result in best locality during runtime.

In the context of JavaParty – a transparent extension of Java for parallel programming of DMPs, [9, 16] – we follow all three basic approaches to improve locality; this paper focuses on the static object distribution through careful type analysis. Static type analysis of Java is more difficult than in imperative languages because of the dynamic dispatch. Object-oriented type analysis for improving locality is more difficult than type analysis for purposes of dispatch optimization [1, 3, 4, 14, 17], since concurrently executing threads must be taken into account.

In section 2 we present the details of JavaParty that are relevant for this paper. Section 3 discusses the basics of type inference, section 4 presents the extensions that are needed for threads and locality optimization. The transformations that result from the analysis are discussed in section 5. The paper is concluded after a brief discussion of the results in section 6.

2 Parallelism and Distribution in JavaParty

JavaParty is an extension of Java for programming DMPs; the details can be found in [9, 16, 8]. JavaParty provides the illusion of a shared address space, i.e., although objects can reside on different hosts, their methods can be invoked as in regular Java. JavaParty programs use thread objects as their sole means to introduce parallelism.

The JavaParty compiler and runtime system map this model to a system of Java virtual machines, one per node, where communication is handled by RMI. JavaParty threads are mapped to regular Java threads.

2.1 Activity-Centered Approach

It is necessary to understand JavaParty's execution model before reasoning about thread and object distribution.

Although a Java thread cannot migrate, the control flow – that will be called *activity* in the remainder of the text – can: whenever a method of a remote object is invoked, the activity conceptually leaves the JVM of the caller object and is continued at the callee object's JVM.

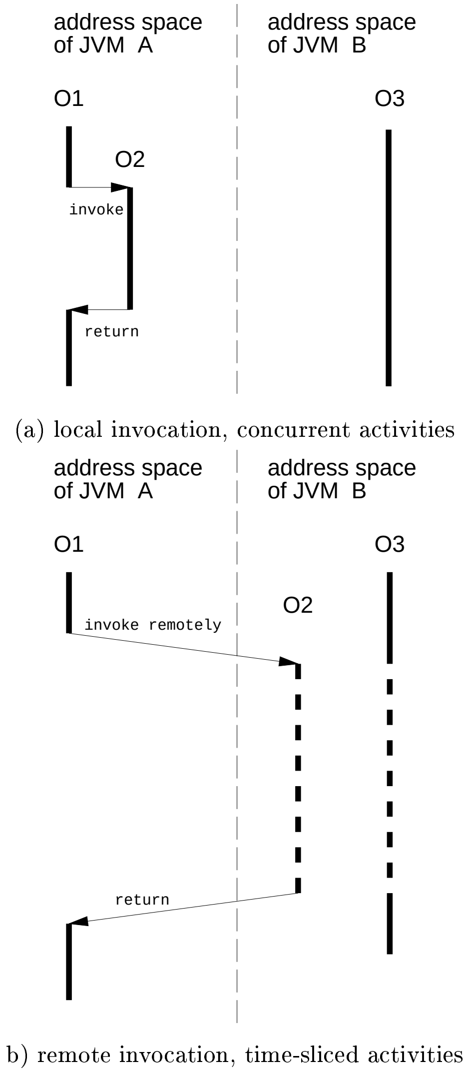

Figure 1: JavaParty's execution model

Figure 1(a) shows two activities, indicated by the fat lines, in two JVMs (A and B). Time advances from top to bottom of the graph. The activity in JVM A first executes a method of object O1, then calls one of O2's methods on the same JVM. The other activity works on object O3.

Figure 1(b) shows the situation, when object O2 is placed on JVM B. In contrast to a local invocation, the time needed for a remote method call is higher than for a local invocation. Moreover, during the execution of O2's method, the two activities face a time-sliced execution in JVM B while JVM A is idle. The total execution time increases.

In general, the following holds. Whenever an activity leaves its JVM, it competes with other activities that are concurrently executed on the target JVM. Since method invocation is synchronous in Java, the original JVM might run idle during the remote method invocation. Due to time-slicing and blocking, competing activities on one JVM decrease the total parallelism. Additional costs are introduced by the remote method invocation itself since communication latency and bandwidth must be taken into account.

The general distribution strategy therefore must be activity-centered: it is preferable to place different activities onto different JVMs. Objects are co-located to activities so that most often and as long as possible, method invocation is local. Local method invocation avoids costly network traffic and avoids competing activities.

2.2 Co-location with Activities

Assume a hypothetical situation where the compiler has total knowledge about objects and activities that are created at runtime. Since for every single object its concrete type is known, no dynamic dispatch is needed, i.e., for each method invocation the code that must be executed can be selected statically. In addition, assume two heuristic approximations: an estimate of runtime cost for each method (including branches and loops) and an estimate of the probability that a specific method is invoked.

With all that information at hand, the compiler can build and trace the weighted call graph and can derive estimates for two values: $work(t, a)$ indicates the time an activity t would spend on methods of an object a if t and a are located on the same node. Note, that in the weighted call graph, for a specific thread object the execution of the `run()` method must be traced.

Similarly, we can determine $cost(t, a)$ for the communication time that would be necessary if t and a are *not* stored on the same node. The cost information takes into account that in the assumed hypothetical situation the concrete types of parameters and return values are known by the compiler. The compiler hence can estimate the communication costs caused by the sizes of the messages that must cross the network.

If $work(t, a) > \sum_{t_i \neq t} cost(t_i, a)$ it is wise to co-locate object a with activity t, since activity t spends more time working on a than all other activities t_i together need to call methods of a remotely. Hence, parallelism is increased.

For every single object a, the optimal activity $\tau = activity(a)$ is derived as the one that maximizes $work(\tau, a) - \sum_{t_i \neq \tau} cost(t_i, a)$.

2.3 Placement of Activities

In general, more activities are used than there are CPUs available. An activity increases parallel execution when working on a co-located object (first sum below); the overall parallel execution is decreased if the activity faces remote method invocations (second sum). Therefore, we define the parallelization win of t as $win(t) = \sum_{activity(a)=t} work(t, a) - \sum_{activity(b) \neq t} cost(t, b)$.

Activities are assigned in a round-robin fashion to the available JVMs in decreasing order of their parallelization wins until on every JVM a single activity has been scheduled. For each of the remaining activities, a new value of its parallelization win is computed. The new value takes into account the potential co-location with other activities and objects. With these new values derived, the next set of activities is scheduled to the JVMs. This is repeated until all activities are scheduled.

2.4 Reality Check

In sections 2.2 and 2.3 we motivated a perfect placement methodology for JavaParty objects and threads. Unfortunately, the presented techniques relied on severe assumptions.

First of all, the technique needed complete knowledge about concrete types of all objects that will be created at runtime. Although such precision cannot be available during compile time analysis, we show in sections 3 and 4 that less precise (but sufficient) information can be derived, categorizing objects and threads into equivalence classes with known concrete type information.

Second, heuristics have been used in optimizing compilers for the cost estimates and the invocation frequencies before, see for example [6]. We use similar heuristice in our prototype.

3 Type Inference

Consider a method invocation `a.foo(arg)` in Java. Due to Java's static type system, it can be determined at runtime which of several methods `foo(param)` is used if the concrete type of `a` is known. The types of `arg` and `param` are used at compile-time and are irrelevant at runtime. Hence, the goal of the type inference mechanism is to determine the concrete type of `a` since the computation of $work(t, a)$ needs to know exactly which version of `foo` is used.

Since there are different control flow paths that can lead to the invocation, in general `a` can hold one of a set of possible concrete types, called *imprecise type information*. Especially if `a` is declared to be of an `interface` type the set of possible concrete types can get quite diverse. The type inference algorithm uses constraint propagation and traverses the control flow and data flow graphs to determine such sets. Depending on the nature of `a`, there are known strategies to resolve the imprecise information: if `a` is the parameter of the surrounding procedure, conceptual method cloning can be used [1, 4, 3, 14]. If `a` is an instance variable, conceptual class cloning can improve precision [17]. We will now explain both techniques.

3.1 Method Cloning

If `a` is a method parameter, say of method `bar`, and if there is imprecise information about `a`'s concrete type, the imprecision is usually caused at the points where `bar` is called. Assume that there are two paths to two invocations of `bar`, as shown in Figure 2. Both invocations use a different concrete argument type (precise information, `C` or `D`). After the invocations, the concrete type of `a` can be either of these.

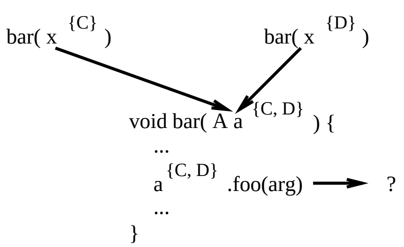

Figure 2: Reason for method cloning

Now, the type inference algorithm conceptually duplicates the called method. The invocation of `bar` that has been processed first remains unchanged. For processing the second invocation, `bar` is cloned and instead of `bar`, the copy `bar'` is called. Therefore, the analysis has precise concrete type information for the parameter `a` in `bar` and as well for the parameter `a` of `bar'`.[2] Thus, type inference can determine which version of `foo(param)` will be called in both `bar` and `bar'`.

3.2 Conceptual Class Cloning

If `a` is an instance variable, local method cloning is insufficient. Conceptually, whole classes must be cloned instead. That in turn causes the type inference mechanism to start over again – at least for a significant part of the code. The reason is that while for parameters there is only a single reaching definition (namely at the very beginning of the method) for instance variables there might be arbitrarily many reaching definitions all over the code.

Assume a situation (see Figure 3) where two assignments in the code assign objects of two different concrete types to `a`. Let `a` be an instance variable of class `A`. From these reaching definitions the type inference algorithms uses data flow information to track back to the points where a `new A(.)` statement is used to create the objects. (Since the paths to the reaching definitions might have common segments, the graph can have nodes with imprecise type information. Separation of common path segments may require cloning even during inverse tree traversal.)

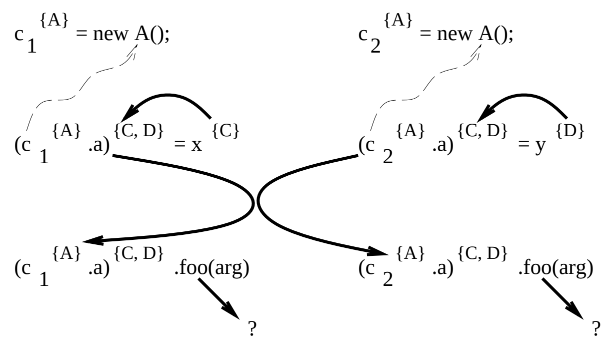

Figure 3: Reason for class cloning

At the `new A(.)` statements, class `A` is conceptually cloned, one of the `new` statements is modified to create an object of class `A'` instead.[3] Due to this cloning, there are suddenly two different instance variables: `a` of class `A` and `a` of class `A'`. Let us rename the latter to `a'` for clarity. Both instance variables keep their concrete types, they are not combined by the two assignment statements that now affect different instance variables. Thus, type inference can determine which version of `foo(param)` will be called at runtime in both `a.foo(arg)` and `a'.foo(arg)`.

3.3 Reality Check

Of course, type inference cannot determine the concrete classes for every method invocation. Especially,

[2]The presentation is simplified. Special care must be taken to avoid endless code duplications in recursive methods.

[3]Actual cloning would break the type system: Neither is there a well-defined relationship between `A` and `A'` in Java, nor can objects of type `A'` be used instead of `A`, and vice versa. Section 5 briefly explains how conceptually cloned classes are transformed back into regular Java.

method and class cloning cannot handle imprecise type information for local variables. Similarly, for arrays or polymorphically used recursive data structures, no precise type inference can be done.

4 Type Inference for Thread Locality

Although after type inference the target method of each invocation is known, still no placement decision can be derived, the analysis might not be able to tell thread objects apart. Consider the following example where, unfortunately, standard type inference will neither clone methods nor classes.

```
class MyThread extends Thread {
  public void run() {
    A x = new A();
    x.foo();
  }
}

class A {
  void foo() {
  }
}

- - - - - - - - - - - - - - - - - -

  new MyThreaad().start();
  new MyThreaad().start();
```

The analysis does not realize that each of the threads[4] uses a private object of type `A` since everything is monomorphic. We will extend the mechanism, so that both the classes `MyThread` and `A` will be cloned. Then `MyThread` uses `A` whereas `MyThread'` uses `A'`. This fact can be used to guide placement decisions so that the activity `MyThread` and the object of type `A` end up in the same JVM whereas the primed versions end up in another JVM.

4.1 Introduction of Helper Polymorphism

We now present how enough polymorphism can be introduced to force the necessary cloning. This is done in two steps: First, wherever a thread is created, the corresponding thread class is cloned. Objects that implement the `Runnable` interface are treated analogously. Second, the original source code is (conceptually) transformed so that it carries and keeps thread ids as additional parameters through all method invocations.

[4]In Java a thread object has a `run()` method that is invoked by calling `start()`. `start()` returns immediately to the caller; `run()` is executed concurrently.

In the above example, we first use different classes `MyThread` and `MyThread'`.

```
  new MyThreaad().start();
  new MyThreaad()'.start();
```

Second, the source-to-source transformation carries and keeps thread ids. The idea is to pass the current thread as an additional parameter `$thread` to each method invocation. Each non-static method stores this thread parameter as a supplementary instance variable of its class (called `$thread`). The `run()` method is the only exception of this rule. This is the only method which is never called from within another method. The `run()` method is invoked in a new thread and as its first action stores the `this` variable in its `$thread` variable. For the example, the transformed code is shown below.

```
class MyThread extends Thread {
  Thread $thread;              //new instance var
  public void run() {
    Thread $thread = this;     //keep thread id
    this.$thread = $thread;    //keep thread id
    A x = new A();
    x.foo($thread);            //new argument
  }
}

class A {
  Thread $thread;              //new instance var
  void foo(Thread $thread) {   //new parameter
    this.$thread = $thread;    //keep thread id
  }
}
```

4.2 Helper Polymorphism is Sufficient

We now explain why the above two-step modification is sufficient to cause the desired cloning.

All the activities of a given program commence at the `run()` methods of the thread classes. As shown in Figure 4 there are two paths in the example program that reach `foo($thread)`. Along the two paths, thread objects of different conceptual types are used: One path carries a thread object of type `MyThread`, the other carries a thread object of type `MyThread'`. The type inference algorithm therefore faces an imprecision in the parameter of `foo`, since the set of possible concrete types has more than one element. This causes `foo` to be cloned so that `foo` is called with `MyThread` whereas `foo'` is called with `MyThread'`; see section 3.1.

Since both `foo` and `foo'` belong to the same class `A`, they refer to the same instance variable `this.$thread`. When `foo` and `foo'` assign the thread

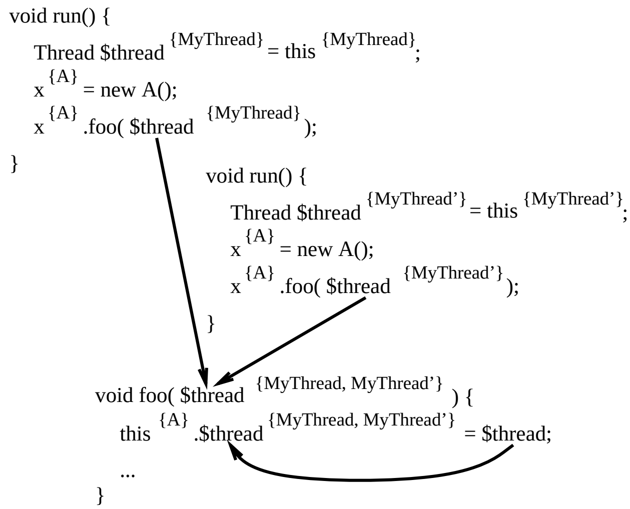

Figure 4: Type imprecision after introduction of helper polymorphism and cloning of thread creations. Two `run()` methods invoke the same method `foo(.)`, causing an imprecision in the `$thread` parameter.

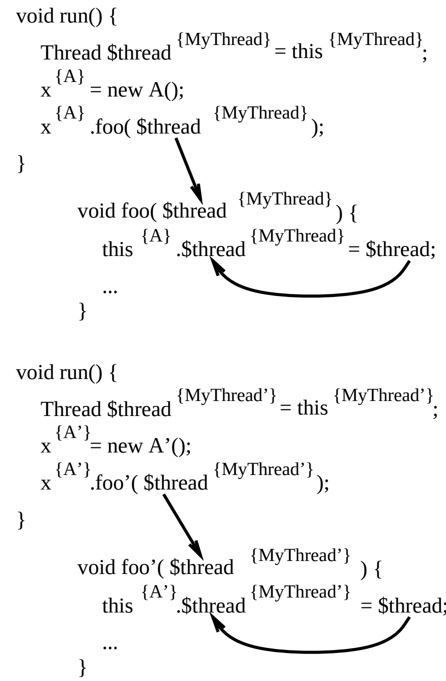

Figure 5: Resulting type information. Method `foo` is cloned. The imprecision in `this.$thread` has been resolved by cloning class `A`.

id to that instance variable, the same imprecision reappears. The instance variable `$thread` can either contain a thread object of type `MyThread` or of type `MyThread'`. At that point, the type inference algorithm uses conceptual class cloning to resolve that imprecision; see section 3.2. For that purpose, the type inference algorithm must find the point where the object is created to which the instance variable `$thread` belongs. Due to the cloning of `MyThread` there are two `run` methods, each of which calls `new A()`. Conceptual cloning of `A` solves the problem: The `run` method of `MyThread'` creates an object of type `A'` instead; there is no longer an imprecision at the assignment to `this.$thread`. The resulting types are illustrated by Figure 5.

Therefore, the added polymorphism is sufficient to clearly separate the activities and objects into two disjoint groups according to their locality requirements. More details can be found in [7].

5 Transformation

Instead of actual cloning of methods and classes, the JavaParty system modifies a "cloned" method to accept a new parameter. This parameter carries a version number into the method. Inside of the method, this version number is used to decide which versions of other methods are to be called. Similarly, "cloned" classes are mapped to classes that carry an additional version number.

The JavaParty system includes a ByteCode disassembler that is used to analyze existing Java programs that are not translated/optimized. To allow for seamless integration with existing code, in addition to the "cloned" method with its extended signature, there is a method with the original signature.

5.1 Transformation of cloned Methods

The method `bar(a)` of section 3.1 is roughly modified as follows:

```
void bar$clone(A a, int method$version)
   ...
   a.foo$clone(arg,
               bar$apply[method$version][42]);
   ...
}

void bar(A a) {
   bar(a, default$version);
}
```

There are two versions of `bar`; the second invokes a default version of the first. Since the two clones of `bar` introduced in section 3.1 invoke different versions of `foo`, a lookup table `bar$apply` is used. The lookup table is a static final array that is computed by the optimizer. There is one such lookup table per cloned method. It has entries for every single method invocation that appears textually within the method. In the above code, the invocation of `foo` is assumed to be the 42nd method that is called inside `bar`. The optimizer avoids the table lookup if a constant value will be returned for all versions.

In addition, type case operations on `a` might be necessary. In this case, the optimizer introduces another lookup table for switching between replicated invocations of `foo`, each of which is guarded with the correct type case.

5.2 Transformation of cloned Classes

Each "cloned" class is augmented with a new instance variable `class$version`. Two new parameters are added to constructor routines. In addition to the `method$version`, a `class$version` is passed into the constructor.

Let us study a `new A()` statement that appears inside of method `gee` and the (simplified) result of its transformation.

```
seq{
  RuntimeSystem.setTarget(
                gee$newAt[method$version][17]);
  return new A(gee$apply[method$version][29],
                gee$new[method$version][17]);
}
```

First of all, the runtime system is ordered to create the next object at a node whose number comes from the lookup table `gee$newAt`. This table has two dimensions: As before, the first dimension refers to the version of the method. The second dimension gives the numer of the `new` statement within `gee` (17 in the example). The constructor's first parameter refers to the version of `gee`, indexed by the number of method invocations inside `gee`, say 29. The second parameter selects the particular "clone" of `A` that is to be created. The `seq`-block is a shorthand notation that allows for a block of statements where Java expects an expression.

Again, the optimizer generates the necessary lookup tables. The details of the transformation can be found in [7].

6 Results and Future Work

As part of the JavaParty project, we have implented the type inference algorithm, a ByteCode disassembler that allows for the analysis of existing Java code, the transformation extending the type inference algorithm to handle threads and to deliver locality information. Finally, the "cloned" methods and "classes" are transformed back into regular Java that can compiled by `javac`.

The optimizer works fine for some JavaParty programs and performs badly for others. For a synthetic benchmark program with 7 classes and a total of 15 methods the type inference created 17 clones of classes and 41 clones of methods. On three JVMs the program performance was improved by more than a factor of 2.

For some JavaParty programs, however, the approach did not improve the runtimes. We have identified two reasons for it. First, the analysis can handle separate creations of thread objects in the code. However, if threads are created in a loop (and stored in an array), there is no way to clone the corresponding thread classes and to introduce enough polymorphism so that the type inference can be used to derive useful placement decisions. On the contrary, without further processing, all these threads fall into the same activity class and are instantiated on exactly the same JVM. The second reason for weak performance showed up in other JavaParty programs where the programmer used the design pattern "workpile". Objects that describe/contain work to be done are stored into an internal data structure of the workpile that often is a list or an array. In either case, the type inference algorithm cannot handle the imprecision. Again, the problem is that all work objects are combined into one single equivalence class. Hence there is no way to distribute certain work objects to certain activities.

In the future, we will attack these two problems by using additional runtime information to further split equivalence classes determined by the type inference algorithm.

7 Conclusion

We have shown that standard type inference techiques for object-oriented languages can be applied to guide object and thread creation in a distibuted environment towards improved locality. The necessary extensions of the type inference mechanisms have been discussed in general.

The optimizer has been implemented as part of the JavaParty project. Depending on the nature of the optimized program, the optimizer can improve performance significantly. Cases where the optimizer fails have been identified and will be attacked in future.

Acknowledgements

We would like to thank the JavaParty group, especially Matthias Zenger, for their support of the JavaParty environment.

References

[1] Ole Agesen. The cartesian product algorithm, simple and precise type inference of parametric polymorphism. In *Proc. of. ECOOP'95, European Conf. on Object-Oriented Programming*, pages 2–26, Arhus, Danmark, 1995.

[2] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1988.

[3] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, Department of Computer Science, March 1992.

[4] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proc. of the SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pages 150–164, White Plains, NY, June 1990.

[5] Barbara Chapman, Piyush Mehrotra, and Hans Zima. User defined mappings in Vienna Fortran. In *Proc. of the 1993 Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, pages 72–75, Boulder, CO, September 30 – October 2, 1992, January 1993. ACM SIGPLAN Notices 28(1).

[6] T. Fahringer, R. Blasko, and H. Zima. Static performance prediction to support parallelization of Fortran programs for massively parallel systems. In *Int. Conf. on Supercomputing*, pages 347–356, Washington, D.C., July 1992.

[7] Bernhard Haumacher. Lokalitätsoptimierung durch statische Typanalyse in JavaParty. Master's thesis, University of Karlsruhe, Department of Informatics, January 1998.

[8] Matthias Jacob, Michael Philippsen, and Martin Karrenbach. Large-scale parallel geophysical algorithms in Java: A feasibility study. *Concurrency: Practice and Experience*, 10(11–13):1143–1154, September–November 1998.

[9] JavaParty. http://wwwipd.ira.uka.de/JavaParty/.

[10] Kathleen Knobe, Joan D. Lukas, and William J. Dally. Dynamic alignment on distributed memory systems. In *3rd Workshop on Compilers for Parallel Computers*, pages 394–404, Vienna, Austria, July 6–9, 1992.

[11] Charles Koelbel and Piyush Mehrotra. Supporting shared data structures on distributed memory architectures. In *Proc. of the 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPOPP*, pages 177–186, March 1990.

[12] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press Cambridge, Massachusetts, London, England, 1994.

[13] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press Cambridge, Massachusetts, London, England, 1993.

[14] Jens Palsberg and Michael I. Schwartzbach. Object oriented type inference. In *Proc. of OOPSLA'91, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, Phoenix, Arizona, November 1991.

[15] Michael Philippsen. Imperative concurrent object-oriented languages. Technical Report TR-95-050, International Computer Science Institute, Berkeley, August 1995.

[16] Michael Philippsen and Matthias Zenger. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[17] John Plevyak. *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1996.

[18] Thomas J. Sheffler, Robert Schreiber, John R. Gilbert, and Siddhartha Chatterjee. Aligning parallel arrays to reduce communication. In *Frontiers '95: The 5th Symp. on the Frontiers of Massively Parallel Computation*, pages 324–331, McLean, VA, February 6–9, 1995.

[19] David Stoutamire. *Portable, Modular Expression of Locality*. PhD thesis, University of California at Berkeley, Department of Computer Science, December 1997. Available as ICSI technical report 97-056.

[20] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Reading, Mass., 1996.