Advanced Compiler Design
Spring Semester 2013
Project
Due date: Apr 24, 2013, 23:59

1 Introduction

The final assignment for the course is to implement an advanced compiler optimization or feature. ¹

2 Tasks

Your tasks are:

1. To agree on a project that your team will complete by the end of this semester. See the next section for projects proposals.

2. To prepare a 3 (three) minute presentation on this project proposal and a description (one page is enough, at most three pages). Present the proposal April 24 in class. You can re-submit the description after the meeting (but no later than May 3) if you want to take into account any feedback given during the meeting.

   The proposal must include:
   
   • The names of all team members.
   • The objective of the project.
   • Why do you find this project interesting?

   Usually one single slide (foil) is enough, the proposal should be 1 page (using 10 pt font) but if you really want, you can use up to 3 pages.

3. After completing your project, you will be expected to give a presentation and submit a short report on your results. This report should include some details of your implementation and experimental evaluation.

¹Selection of a different framework than Javali needs to be discussed with and approved by the professor and the teaching assistant(s) by Apr 19, 2013
3 Projects

Missing features in Javali

Javali has been kept simple as it’s is targeted for teaching. The obvious consequence is that some features present in modern programming languages are missing. You are free to propose new features or pick some of the ideas listed:

- Class Loading, Open World Assumption (OWA)
- Generics

Exceptions

In this project you will implement exceptions.

SSA-PRE

In this project you will be asked to implement PRE based on SSA for Javali.

Register allocators

In this project you will implement a register allocator for the code generator. You can choose between different algorithms described by the classic literature, or research papers.

Escape analysis

Implement escape analysis algorithms to determine if an object escapes (or may escape) a method. If an object does not escape, the compiler can then apply optimizations. For example, you can decide to stack-allocate an object if the object doesn’t escape a method. Try to see what are other scenarios in which escape analysis can be useful.

Leak Detection

In this project you will implement a leak detector. The garbage collector (GC) usually takes care of the typical memory management problems. However the GC cannot resolve all the memory problems. In the Java terminology a leaked object is an object that is still alive (i.e., there is a reference pointing to it), but not reachable (e.g., there is no program path accessing that object). The garbage collector is then not able to free memory – thus we have a leak. There are a number of steps that can lead to such a situation. In addition to identifying leaks, there is the problem of deciding when to free a leaked object.
Loop optimizations

Javali don’t provide a “for loop” construct. A for loop can be easily added (for loops can be mapped into while loops), but the purpose of this project is to investigate loop optimizations; apply at least two out the following optimizations:

- Loop unrolling, loop peeling.
- Loop reversal.
- Loop splitting.
- Loop fusion/loop fission.
- Loop interchange.
- Loop invariant code motion.
- Loop unswitching.

You can apply the following restrictions to implement loop construct:

- Use Java-like syntax.
- Only one variable is allowed to determine the loop iterations.
- The variable needs to be declared at the beginning of the method.

Data-structure optimization heuristics

Library collections include a series of data-structures. For example, the Java libraries usually provide different kinds of implementations for lists (e.g., LinkedList, ArrayList). It is then the programmer’s decision which implementation use for an algorithm. As with any human activity, programmers sometimes make a decision that is sub-optimal. In this project you are required to detect data-structures access patterns (e.g., sequential vs casual). Subsequently, based on the discovered usage you can transparently exchange the implementation with one that better matches the access patterns.

Parallelism in Javali

The goal of this project is to introduce parallelism constructs in Javali. No specific directions needs to be followed.

You can choose to use an approach similar to Java where you provide classes. For example Runnable, Thread, and/or specific constructs like synchronized. Alternatively you can create new specific constructs to indicate parallel regions inside your code.
Object splitting

Object splitting is an optimization that tries to improve cache performance by splitting object fields into two categories: hot and cold fields. Field frequently accessed are marked as hot, and less frequently accessed fields are marked as cold. To determine if a field is hot or cold, you can profile the program or adopt static analysis techniques. An object instance is then split into two distinct parts, literally creating two new objects instead of one. The hot part contains a reference to the cold part. In this project you implement the infrastructure to support object splitting inside the framework. You will be required to build a profiler that will collect field accesses for each class. Given a profile, the compiler will then produce modified code for the classes that are marked for splitting. When the basic infrastructure is in place you use it to study the behavior of the optimized programs in different scenarios.

Object splitting benefits from temporal related object accesses. This can be achieved by implementing a garbage collector/allocator that moves temporal related objects together in the heap. Try to create programs in which this is done artificially. Show what are (if they are) benefits of placing temporally related objects together when object splitting is applied.

In your first implementation you will implement the redirection to the cold part as described above. You can try to implement different allocations and indirections strategies, and evaluate the impact:

- Allocate the cold part only when necessary. You can do this checking when there is the first access to a cold field. You can try statically analyze the code trying to determine the first access, or add a check before every cold field access, or you can profile this information.

- Allocate the cold part in a different memory area (or CPU). You can study the benefits/drawback in terms of performance (e.g. cache) in case of a multi-core architecture.

- Use fixed offsets instead of pointers. For example you can allocate hot instances in a designated part of the heap, and cold instances at a fixed offset in a different memory area. In this way you should be able to save the space for the extra indirection pointer and an extra memory indirection.

- You can also try to divide the heap in class areas, or try to interleave cold and hot instances in different manners.

Splitting data structures for NUMA systems

In multicore systems with a non-uniform memory architecture (NUMA) it is crucial that each thread of a multi-threaded program accesses data that are allocated on the processor where the thread is running.

The goal of this project is to extend the Javali language with support for simple parallel constructs (e.g., fork-join parallelism) and also constructs for localized data allocation (memory allocation routines that allocate memory regions at a specific processor of the system). You can evaluate your extension by implementing some simple algorithms (e.g., matrix-matrix multiplication, matrix-vector multiplication, or tree traversal) using the original and the NUMA-aware version of the Javali language, and comparing the performance of the two versions of the same algorithm at the end.
Type checking and method resolutions optimizations

The current type-checking mechanism is implemented by walking-up the inheritance tree encoded on the method tables. This a simple implementation and saves memory space compared to other approaches. Approaches like Cohen display or Schubert Numbering use a different strategy. A Cohen display assigns to each class an identifier that is checked at run-time. The identifier is stored in the method table at a specific offset for each class. This approach requires that the method-table of a class contains entries for all the super-classes ID at a fixed offset. Then it is possible to perform type-checking using a simple table look-up.

Interfaces

Javali doesn’t have interfaces. In this project you first implement interfaces inside the framework. This means introducing a new kind of declaration type. The compiler then must perform semantic checks to see if all the interface methods are implemented in the class or in one of its super-class(es).

Multiple inheritance

Multiple inheritance is a popular feature that introduces many interesting problems to solve. In this project you implement multiple inheritance in the framework. Different programming languages have implemented multiple inheritance in different ways, for example, see the approaches taken by python or C++. Try to get inspired by existing solutions, explore what are the implementation consequences from a compiler design point of view. Pick the most appropriate choice for the time restrictions imposed by the course’s time-schedule.

Implement a profiling infrastructure inside the compiler.

Many proposed projects require to profile programs. In this project you need to introduce a profiling infrastructure inside the framework. You will be asked to develop a domain specific language or annotations to describe which kind of information you want to retrieve, and the granularity of this information (e.g., program, method, basic block). We want you to think about different instrumentation/profiling techniques. For example, you can apply program analysis to instrument just certain parts of the code. Try to get inspired by existing profilers (e.g., BTrace/DTrace, gprof, hprof).

Method calls optimization

Method resolution optimizations in the context of object oriented systems have been of interest to various research groups. The main idea behind these optimizations is to avoid the indirection necessary for resolving the method to be invoked for an object. You implement at least one of these optimizations:

- Inline caching
- Method inlining
- Procedure cloning/specialization
Remember that you probably need to perform some analysis (e.g. profiling, class hierarchy analysis) to apply these techniques.