Expressing Compiler Optimisations Using TRANS

by Robert Quill

An Example

How can the following loop be optimised?:

```
for(i = 0; i < 10; i++)
{
    a[i] = b[i] + n;
    if(k > 10)
        c[2m+n] = a[i] * b[i];
    else
        c[2m+n] = a[i] + b[i];
}
```

Possible Optimisations 1

The variable in the if statement does not change.

```
if(k > 10) {
    for(i = 0; i < 10; i++) {
        a[i] = b[i] + n;
        c[2m+n] = a[i] * b[i];
    }
}
else {
    for(i = 0; i < 10; i++) {
        a[i] = b[i] + n;
        c[2m+n] = a[i] + b[i];
    }
}
```

Possible Optimisations 2

2m+n is unchanged by the loop.

```
d = 2m + n;
for(i = 0; i < 10; i++)
{
    [i] = b[i] + n;
    if(k > 10)
        c[d] = a[i] * b[i];
    else
        c[d] = a[i] + b[i];
}
```

Caveats

- If statement may be a complex expression
  - Must be certain none of the variables are changed during the loop.
- Can only replace occurences where variable values have not changed.

What is TRANS?

- A language for writing compiler transformations.
- Transformations are applied to the CFG
- Specification has 2 parts:
  - Rewrite: modifies CFG
  - Side Condition: determines where to apply transformation.

Extensions

- Block Matching
  - More like representation used by a compiler
  - More efficient to implement
- Array Notation
- Loop Dependence Analysis

Loop Dependence Analysis

Which iterations of this loop can we exectue in parallel?

```
for(i = 0; i < n; i++)
{
   a[i+k] = b[i];
   b[i+k] = a[i] + c[i];
}
```

Answer: k iterations.

Loop Dependence Analysis

- If an array element is written in one iteration and read in another then there is a dependence between the iterations.
- Can only parallelize iterations where no dependencies exist.
- Solve i' = i + k
- Can parallelize at most k iterations at a time.
- Much more difficult with more variables, nested loops,etc
- Ananlysis for single variable subscript expressions in TRANS.

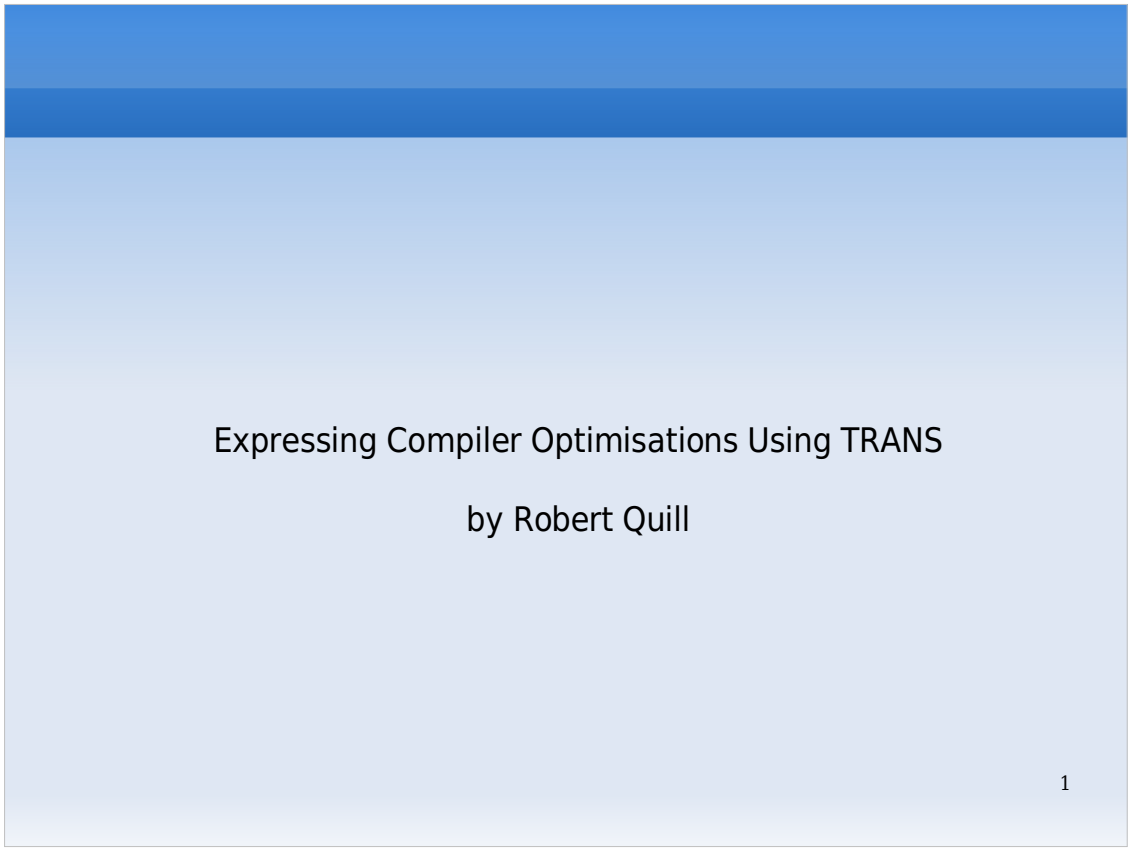
Transformation Catalogue

- Paper contains 51 transformations
- 23 have been implemented
- 10 more to implement
- Other will not be implemented
  - Too low level
  - Architecture specific
  - Parallelizing

Future Work

- Function calls
- Types
- Verfication

Any Questions?



Hi. My name is Rob Quill and I'm going to talk to you about my research into formalising compiler transformations using a language called TRANS.

An Example

How can the following loop be optimised?:

```
for(i = 0; i < 10; i++)
{
    a[i] = b[i] + n;
    if(k > 10)
        c[2m+n] = a[i] * b[i];
    else
        c[2m+n] = a[i] + b[i];
}
```


I'd like to start with an example. How can the following loop be optimized? This loop may be run hundreds of thousands of times in a scientific application, so the faster we can make it the better. I'll give you a bit of time to look at this now.

Possible Optimisations 1

The variable in the if statement does not change.

```
if(k > 10) {
    for(i = 0; i < 10; i++) {
        a[i] = b[i] + n;
        c[2m+n] = a[i] * b[i];
    }
}
else {
    for(i = 0; i < 10; i++) {
        a[i] = b[i] + n;
        c[2m+n] = a[i] + b[i];
    }
}
```


Here are a couple of possible optimisations.

Firstly, the expression which is evaluated for the if statement within the loop does not change with each iteration of the loop, so we can move the if statement outside the loop and recreate the loop in each branch of the if statemnt.

This improves performance as we only need to branch once then execute the loop instead of branching every time inside the loop,

Possible Optimisations 2

2m+n is unchanged by the loop.

```
d = 2m + n;
for(i = 0; i < 10; i++)
{
   [i] = b[i] + n;
    if(k > 10)
        c[d] = a[i] * b[i];
    else
        c[d] = a[i] + b[i];
}
```


Secondly, each iteration we write to the 2m+n-th element of c, and as this is a constant we could assign the value of 2m+n to some variable d, and access the d-th element of c instead, saving us having to evaluate 2m+n every iteration.

Although I haven't shown it here, there is no reason that both these optimisations could not be applied together.

Caveats

- If statement may be a complex expression
  - Must be certain none of the variables are changed during the loop.
- Can only replace occurences where variable values have not changed.

5

However, there are some potential problems.

If the exoression being evaluated for the if statement is a complex expression we must be certain that the entire expression is not changed with each itertion of the loop. To do this we need to make sure that none of the variables used in the expression are changed by the loop.

Also, we can replace occurences of 2m+n with d if and only if the values of m and n have not be changed between the definition and use of d.

For these reasons it is important for us to be able to formalise these definitions so that a transformation is only applied at a place where it is valid. This is the purpose of TRANS.

What is TRANS?

- A language for writing compiler transformations.
- Transformations are applied to the CFG
- Specification has 2 parts:
  - Rewrite: modifies CFG
  - Side Condition: determines where to apply transformation.

So, what is TRANS? TRANS is a language for writing compiler transformations. Transformations in TRANS are applied to the control flow graph of a program. A transformation is TRANS is made up of two parts: a rewrites and a side condition.

The rewrite describes how the transformation should modify the control flow graph of the program. A rewrite may modify or replace the instrucion at a node, add or replace edges in the graph or combinations of these.

The side condition is a CTL formula which describes where in the graph a transformation can be applied by pattern matching instructions at nodes, or matching properties of the control flow graph.

Extensions

- Block Matching
  - More like representation used by a compiler
  - More efficient to implement
- Array Notation
- Loop Dependence Analysis

The original version of TRANS, which was presented by David Lacey in his PhD thesis matched programs written in L0, a simple imperative language which contains only assignments, if statements, goto statements and return statements. Nodes of the flow graph also only contained a single instruction.

I've extended TRANS to represent each node in the CFG as a basic block, i.e. a sequence of instructions where has one entry point, one exit point and no branch instructions inbetween.

This format more closely resembles how the control flow graph is represented within a compiler. It also makes the implementation more efficient as we can store all the variables used and defined at a node in a list instead of having to check every time we look at a node.

I've also extended L0 and TRANS to support arrays as scientific applications make heavy use loops and arrays.

I've also added features in TRANS to perform loop dependence analysis.

Loop Dependence Analysis

Which iterations of this loop can we exectue in parallel?

```
for(i = 0; i < n; i++)
{
   a[i+k] = b[i];
   b[i+k] = a[i] + c[i];
}
```

Answer: k iterations.

Which iterations of these loops can we execute in parallel?

The answer is k iterations, here's why...

Loop Dependence Analysis

- If an array element is written in one iteration and read in another then there is a dependence between the iterations.
- Can only parallelize iterations where no dependencies exist.
- Solve i' = i + k
- Can parallelize at most k iterations at a time.
- Much more difficult with more variables, nested loops,etc
- Ananlysis for single variable subscript expressions in TRANS.

There is a dependence between two itertions of a loop if an element of an array is read in one of the iterations and written in another.

We can only parallelize iterations of loops where it is impossible for there to be a dependence, due to the non-deterministic order of execution.

In the previous example, if we formulate and solve the linear equation i' = i + k, then we see that there are dependencies between iterations (1,k), (2,k+1) etc, so we can do at most k iterations in parallel.

It becomes much more difficult to forumalte and solve these equations quicky when given nested loops and more complex epressions.

I've added support to TRANS for determining dependencies where the expression involves at most one loop indexing variable.

Transformation Catalogue

- Paper contains 51 transformations
- 23 have been implemented
- 10 more to implement
- Other will not be implemented
  - Too low level
  - Architecture specific
  - Parallelizing

10

The main aim of my thesis has been to implement an extensive catalogue of compiler transformation in TRANS. So for I have implemented 23 of the 51 transformations in the source paper. I have a further 10 yet to implement. The remaining transformations will not be implemented, for various reasons:

Some of the transformations are too low level. TRANS optimisations take place at the intermediate representation level, so transformations to save registers or organise memory are irrelevant.

Some transformations are architecture specific, for example transformations which use vector instructions, and the interediate representation is not meant to be architecture specific,

Some transformations involve paralizing loops but there is no language contruct in TRANS to say that a loop may be executed in parallel. We consider this to be carried out in a different optimisation phase.

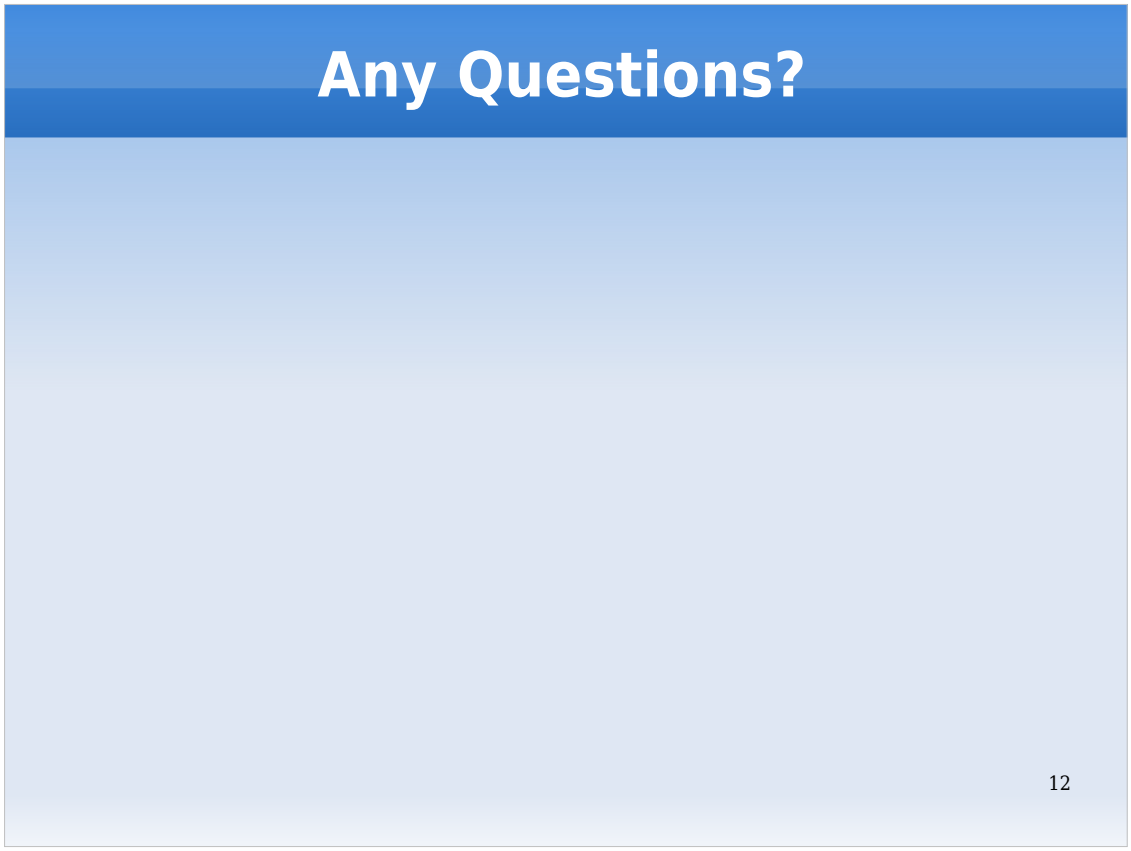
Future Work

- Function calls
- Types
- Verfication

11

Before I submit my thesis I am hoping to add support to TRANS for function calls, and implement optimisations relating to functions, such as function inlining.

Some future work which is beyond the scope of my Masters is adding support for types to TRANS as currently all the variables are integers and also formally verifying all the transformations in the catalogue, which I believe Richard is going to talk about later.



That it. Any questions?