Annotation-driven code transformations for scientific models: Prototype report for COMPOSE-HPC project.

Geoffrey C. Hulette
Sandia National Laboratories

Matthew J. Sottile
Galois, Inc.

April 20, 2011

1 Overview

This document describes a demonstration intended to show an annotation guided transformation based on one that was identified as important in the COMPOSE proposal. Specifically, we will use annotations to allow a programmer to indicate data structures in a C program which should be subject to a struct transpose (i.e., array-of-structs to struct-of-arrays transform). The subsequent transformation will require the following changes to be automatically made to the program sources:

- Change the struct definition to move from singleton elements to pointers to 1D arrays of elements.
- Assuming dynamic heap memory allocation (aka, `malloc` or `calloc`) for the array of structs. Replace heap allocation of a single structure with stack allocation, and add multiple allocations for the internal fields.
- Identify all points in the code where the array is indexed and fields are accessed, such as `foo[i].x`. Replace with `foo->x[i]` or `foo.x[i]` as appropriate.
- Ensure that function signatures are also modified appropriately. In particular, pass the struct by value instead of by reference.
- Identify deallocation of the original array of structs, and replace with deallocation of the field arrays.

1.1 Intrastucture details

As part of this activity, we have also been testing our language processing infrastructure and revising our design choices based on our experience. This was expected at this early stage in which we are evaluating technologies against the needs of the research project in order to select appropriate tools and libraries for further work. Changes in this infrastructure later will be more disruptive than changes and adaptations at this earlier time.

- **PAUL**: C and C++ parsing is performed with Rose and an annotation parser based on the Lemon-parser generator[1]. Rose is more complex than other solutions we examined, but in addition to being very robust it provides built-in capability to associate comments with syntactic elements. We determined that the effort required to re-implement this feature in other parser frameworks would be prohibitive. The choice of Lemon versus other parser generators was based on a two significant factors. First, avoiding hand-written parsers allows easier growth and adaptation of the annotation grammar. Second, Lemon is a public domain tool that is in widespread use due to being a component of the SQLite project. This means that it can be included with PAUL without licensing burdens, and is

[1] `http://www.hwaci.com/sw/lemon/`

very robust due to its widespread usage in other projects. This reduces the prerequisite software requirements for PAUL.

- **ROTE**: At this stage COMPOSE is concerned with demonstrating how rewriting and transformation techniques can benefit HPC codes, and so we sought a rewriting system that was most flexible for a small set of common HPC languages. Our original recommendation was to look at ASF-SDF/Rascal or Stratego, but based on input from the San Diego Exascale meeting participants, we reprioritized Coccinelle[2]. Coccinelle proved robust and easy to use – we were able to design and implement the demonstration described in this document with relatively minimal effort.

We believe that the static analysis and transformation tools demonstrated here, along with the "version 0.1"-level PAUL prototype, are at a state of maturity that other groups in the COMPOSE team may start experimenting with them. We will continue to consider alternative rewriting systems while prototyping with Coccinelle. The primary discrepancy that is present with all systems is the lack of robust support for languages beyond C and basic subsets of C++. Coccinelle will support C-based work.

2 PAUL Annotations

The first goal of this prototype was to demonstrate a basic use of PAUL to drive subsequent transformation tools. In this example, we use the annotation to indicate the desired transformation and the data structure affected. This information is used to generate the Coccinelle transformation.

In the demo, we use the following PAUL annotation associated with a data structure definition:

```
/*% ABSORB_STRUCT_ARRAY outerAllocMethod=stack */
struct boid {
  Vec pos;
  Vec vel;
  Vec delta;
};
```

The annotation is embedded in the comment text, and is identified by the "%" prefix. Note that the prefix is configurable on a per-file basis, so as to avoid collision with other comment format schemes.

PAUL extracts the annotation from the comment, parses it, and associates it with the subsequent struct definition. This association informs ROTE that the target of the transformation is the struct named `boid`. The first element of the annotation is the *annotation identifier*; in this case, `ABSORB_STRUCT_ARRAY`. This identifier indicates which transformation is desired. The identifier is intended to be used by tools that process annotations to selectively extract annotations from a source file. If multiple types of annotations are present (e.g., transformation-oriented ones like these, along side performance-oriented annotations used by other tools), tools can distinguish one set of annotations from another.

Following the identifier are a sequence of *annotation options*, given as a sequence of key/value pairs. The syntax for options is quite simple: keys must be alphanumeric identifiers, and values may be identifiers, numbers, or strings (surrounded by quotes and admitting whitespace and the usual special character escape codes). Pairs are connected by `'='` and separated by whitespace.

PAUL will parse the options and pass them to the relevant transformation-generating routine. In the demo, we use only one option. The option key, `outerAllocMethod` is used to tell the transformation generator whether we want the outermost struct data to be allocated on the stack or on the heap. Here, we indicate that we would like it allocated on the stack, by passing the value `stack`. Note that the interpretation of this option is entirely up to the transformation generator – PAUL parses the options but does not attempt to interpret or validate them.

[2] `http://coccinelle.lip6.fr/`

In this example, the programmer has deployed the annotation to selectively transform a particular data structure (i.e., `struct boid`). We assume that the programmer knows (or can discover through experimentation) that this data structure will benefit from the transformation. We would eventually like to explore more advanced techniques that automatically suggest data structures to transform based on an estimated benefit analysis.

We show preliminary results of this transformation applied to a very simple flocking model (the classic "Boids" model), and demonstrate that by transforming the structure layout we can see significant changes in cache efficiency with a corresponding runtime reduction. This performance discussion appears later in Section 4.

3 Workflow

The basic workflow that this demo illustrates involves the following steps (also see Figure 1).

1. Code annotation
2. Annotation extraction and transformation template generation
3. Coccinelle transformation execution

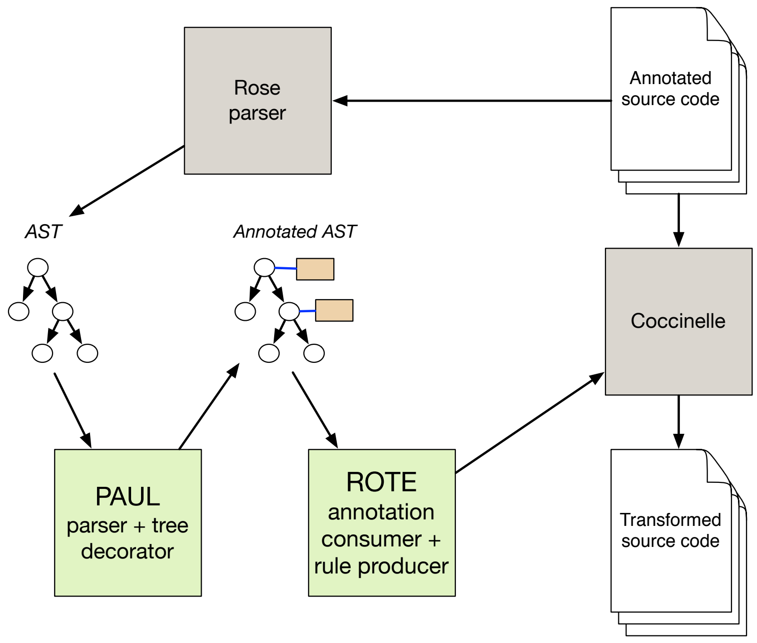

Figure 1: COMPOSE transformation workflow with PAUL and ROTE.

3.1 Code annotation

First, the programmer must annotate the source code. As shown in the example above, annotations are embedded in comments and placed in juxtaposition with the syntactic elements to which they are intended to apply. The annotation syntax is discussed in Section 2.

We rely on Rose's pre-existing association of comments with syntactic elements to determine where the annotation should apply. Rose seems to do a good job with this process, although some effort may be involved if we decided we needed different association behavior.

3.2 Annotation extraction and transformation template generation

Annotations are extracted by coupling the PAUL annotation parser to some language processing infrastructure. We started using a basic C parser, but rapidly found it lacking due to a number of factors that come into play with production codes:

- Lack of ability to deal with preprocessor directives. In particular, most parser frameworks invoke the preprocessor before trying to parse the code, resulting in macro and include expansion and thus altering our view of the code as it was written. Furthermore, frameworks lacking this capability limit our tools' ability to transform conditionally compiled sections of code (e.g., code within `#ifdef` blocks).
- Limited or non-existent support for parsing comments.
- Limited ability to decorate AST and other data structures with metadata derived from annotations.

Our conclusion was that we should move to a robust compiler infrastructure sooner than later, and Rose is the most capable candidate. Rose does have its downsides – in particular, it requires a convoluted and difficult build process. In a supplemental document we will provide guidance on the Rose build process, addressing common mistakes and subtle configuration flags for common platforms.

Annotation processing is straightforward. Rose parses the source code and generates an abstract representation, and associates comments and compiler pragmas with nearby syntactic elements. PAUL then iterates over all the comments, identifying those that represent annotations by examining the prefix. Comment text that is identified as an annotation is parsed in the identifier-plus-options format (detailed in Section 2), and the stored in an object instance. The object is then associated with the AST or other structural elements of the program, and is made available to subsequent traversals.

In the next phase, the AST is traversed again and each annotation is processed to emit the appropriate Coccinelle rewriting rules. Finally, these rules are invoked by ROTE on the original source files and the transformed source code is generated.

Using Coccinelle allows us to leverage the term rewriting approach instead of attempting to manually transform the code in Rose. The generated rules are are far simpler than the equivalent Rose API calls, and act as a bridge between PAUL and the ROTE layer.

3.3 Coccinelle transformation

Transformations in Coccinelle are based on term rewriting. At a high level, a transformation specifies a code pattern to be matched and then a replacement pattern (which may involve elements of the matched term) to be substituted. For example, consider this Coccinelle snippet to transform a data structure in C.

```
@def@
identifier s,x,y,z;
type T1,T2,T3;
@@
struct s {
- T1 x;
- T2 y;
- T3 z;
+ T1 *x;
+ T2 *y;
+ T3 *z;
};
```

The first part of the Coccinelle code states that we have a rule named "def" that is defined in terms of four distinct identifiers and three distinct C types. The rule is defined in terms of abstract types and identifiers. For example, if the first field in the struct is a double precision field named `foo`, then `T1` would be associated with `double`, and `x` would be associated with the symbol `foo`.

The remaining portion of the rule (below the "") defines the pattern to match and the changes to make after matching. The syntax is intentionally similar to the well-known "patch" file format. Lines beginning with "-" will be removed, lines beginning with "+" will be inserted, and lines beginning with neither will simple be matched in context. This example will match any named structure with three distinct non-pointer fields. The three fields will be removed and replaced with a set of fields, preserving the names but adding an additional pointer attribute. Thus, the struct of three values will be transformed into a struct of three pointers to the values.

Note that current versions of our tools require the transformation code for Coccinelle to be generated for specific contexts. For example, the above example transformation would look slightly different if the targeted struct had four fields instead of three. It is possible that Coccinelle supports or could be made to support general transformations, but we have found that generating Coccinelle code for specific contexts gives us a high degree of flexibility and control that might not be possible with Coccinelle alone.

In the demonstration example, we aim to replace an array of struct values with a single struct value that contains arrays for each field. As such, we need to not only transform the data structure definition, but the allocation expression and places within the code where the structure is used. For the sake of brevity, we present only the case where memory for the struct is dynamically allocated using `malloc`. We can define the following rule to appropriately transform memory allocation and deallocation.

```
@@
identifier decl.k,def.s,def.x,def.y,def.z;
expression E;
type def.T1,def.T2,def.T3;
@@
- k = malloc(E * sizeof(struct s));
+ k.x = malloc(E * sizeof(T1));
+ k.y = malloc(E * sizeof(T2));
+ k.z = malloc(E * sizeof(T3));
...
- free(k);
+ free(k.x);
+ free(k.y);
+ free(k.z);
```

This rule is a bit more complex than the rule used for the struct definition transformation. First, note that the set of identifiers and types reference names from other rules. For example, `def.T1` refers to the type `T1` named in the rule `def` that was presented earlier. This syntax allows rules to interact. In this case the rule for manipulating the calls to `malloc` and `free` is evaluated dependently based on the application of the rule for the struct definition, and it shares the matched symbols from that rule. In fact, the rule above depends on one other rule as well, named `decl`, and which is as follows.

```
@decl@
identifier def.s,k;
@@
- struct s *k;
+ struct s k;
```

This rule is used to transforms variable declarations of the relevant struct type. Taken together, the rules for changing `malloc` are subject to preconditions that A) a structure with three fields has been identified, and B) a variable has been identified with that structure as its type. Given these preconditions, and thus the prior application of the relevant rules, we transform the memory allocation. When a single `malloc` is detected for the array of structures, we replace it with a sequence of mallocs for the internal array fields.

4 Performance impact of transformation

The purpose of this transformation is performance. When using a data structure made up of numerous scalar fields, it is not uncommon to loop over arrays of the structure to examine a single variable at a time. For example:

```
for (int i = 0; i < n; i++) {
  arr[i].x = arr[i].x + 1;
}
```

Each iteration of the array will stride memory by the size of the overall structure. In systems with caches that retrieve lines that are contiguous in memory, we will observe a suboptimal cache usage due to the fields that are skipped over on each iteration. If we transpose the array of structs to instead make an array for each field that is contiguous in memory, we will observe an increased cache efficiency. Similarly, a transformation in the opposite direction would be appropriate for improving the memory behavior of code in which all fields of a structure are accessed at once.

The example code that was used for this example was written originally to use arrays of structures containing scalar fields. We transformed the code to use the structure-of-arrays design, and measured the performance of both the pre-transformation and post-transformation code to see if a difference could be detected.

We performed a basic experiment in which we fixed the number of entities in a simple flocking simulation, and measured the difference in execution time, L1, and L2 cache miss statistics. The results are shown below:

Version	L1 DCM	L2 DCM	Wall clock
Original	11263721904	685	1m20.624s
Transformed	3769398085	461	1m13.213

As we can see, the transformed code performed better than the original due to the striding behavior of the code. Each entity has a velocity, position, and delta vector associated with it. The main loop of the time stepping algorithm manipulates each of these in the outermost loop (in the calls to cohesion, separation, and alignment in the `step` function), but internally these may do a traversal over the set of structs looking at just one field (such as separation, which traverses the array and only examines the position field). Reordering the structure allows this traversal of the whole array looking at just one field to be more cache efficient by increasing locality of access.

5 Demo availability

All code described in this document can be found at the COMPOSE-HPC SourceForge.net repository:

`http://sourceforge.net/projects/compose-hpc/`

The sub-directory `paul/demo` contains the demo code, along with a README file with basic instructions for getting the demo working on your own machine. We have done basic testing on Linux (recent Ubuntu and Debian releases), along with OSX 10.6.