Refactoring for Performance with Semantic Patching: Case Study with Recipes

Michele Martone, Julia Lawall

Refactoring for Performance with Semantic Patching: Case Study with Recipes

Michele Martone[1] and Julia Lawall[2]

[1] Leibniz Supercomputing Centre
Michele.Martone@lrz.de
[2] Inria-Paris, Paris, France
Julia.Lawall@inria.fr

Abstract. Development of an HPC *simulation code* may take years of a domain scientists' work. Over that timespan, the computing landscape evolves, efficient programming best practices change, APIs of *performance libraries* change, etc. A moment then comes where the entire codebase requires a thorough performance lift. In the luckiest case, the required intervention is limited to a few *hot loops*. In practice, much more is needed. This paper describes an activity of *programmatic* refactoring of ≈200k lines of C code by means of source-to-source translation. The context is that of a so-called *high level support* provided to the domain scientists community by a HPC service center. The motivation of this **short paper** is the immediate reuse potential of these techniques.

1 Introduction

GADGET3 (from now on, GADGET) is a code simulating *large-scale structure (galaxies and clusters) formation*. Not counting forks, it has a codebase of around 200 kLOC (thousands of Lines of Code) in C. It is being developed by a geographically sparse community of domain scientists (astrophysicists). Code revisions progress using a revision control system.

GADGET revolves around the concept of *particle*. A particle and its associated quantities are modeled using a few C `struct`s, e.g.: `struct P {float x,y,z;} *p;`. Here three quantities are shown; GADGET has around a hundred of them, enabled/disabled via several `#ifdef`s. Memory use is dominated by a few dynamically allocated global arrays of similar `struct`s. The *number crunching* defining most of the code accesses these arrays using expressions similar to `p[e].x` (`e` being an indexing *expression*). This data layout goes by the name of *Arrays of Structures* (AoS). The GADGET community finds this layout very handy: the AoS definitions reside in one single header maintained by the project lead. Collaborators maintain separate source files with functionality (in jargon, *physics modules*) de-/activated by `#ifdef`s.

While appreciated by the community, AoS performs suboptimally on the frequently occurring *streaming access* loops. On current architectures, the *complementary* layout – *Structures of Arrays* (SoA) – performs better here, for

it favours *compiler autovectorization* [PHS15]. The SoA equivalent of the previously sketched AoS is: `struct P {float *x,*y,*z;} p;`, with dynamically allocated arrays `p.x`, `p.y`, and `p.z`. Besides redefinition of *particle* `struct`s and introducing a few new variables, porting AoS to SoA requires rewriting the bulk of the code. Namely, translating all the non-trivial expressions containing combinations of AoS accesses like `p[e].x` into using the `p.x[e]`-like syntax. Still, operations that move particles are more practical with AoS, than on structures scattered across a few hundred arrays. Thus, one may want to retain the AoS definitions, and use *scatter/gather* operations to occasionally convert *to/from* SoA, like in sorting or *load balancing* across distributed processes over MPI.

GADGET is a major astrophysical code and improving its performance is highly desirable. Baruffa et al. [BIHK16] estimated a performance improvement with SoA as exceeding 2×, but their study was limited to an excerpt of circa 1kLOC. The present article tackles the problem of *backporting SoA* to the entire ≈200 kLOC code, respecting its many build-time `#ifdef` variants.

The present work has been carried within a longer-term (so-called *high level support*) activity at LRZ, and has targeted both a legacy version (P-GADGET3) and a development one (codenamed OPENGADGET3, from non-public source repositories) forks, both from the Max-Planck Institute for Astrophysics in Garching, Germany. These ([RTB+15], [RDW+19]) are derivatives of GADGET2 [Spr05]

This article applies to both and addresses either by the namesake GADGET. Although we are aware of C++-based techniques to use a SoA semantics with AoS syntax, we had to rule those out: keeping the code in C has been a project requirement.

2 Prospective factorization steps and tool choice

Essential to any AoS to SoA translation are: 1. Identifying AoS members to be made into SoA arrays; 2. Declaring new SoA types and variables in a global scope; 3. For each new SoA array, adding de-/allocation and AoS ↔ SoA gather/scatter primitives; 4. Accessing each new SoA array by mutating corresponding AoS *expressions*. Such changes are constrained by interrelated factors:

- *timeliness*: How to quickly change so many lines of code?
- *correctness*: How to avoid introducing mistakes? Note the aggravation of having numerous build-time *code paths* implied by the `#ifdef`s.
- *flexibility*: Can we enact only a *partial* SoA translation, possibly on demand?
- *continuity*: Can we develop in AoS, transforming only before *build and run*?
- *acceptance*: How to have the community *accept* the proposed solution?

Timeliness requires an automated tool. *Correctness* calls for a tool having a model of a C program. A programmable, or at least parametric solution in choosing AoS quantities would be best: we do not know which *subset* of a complete SoA transition is most performant under which configuration (moreover, several code forks exist). *Continuity* by preserving the current AoS development culture would maximize *acceptance*.

Existing Integrated Development Environments (IDEs) are too primitive for these changes. *regular expressions* are tempting but brittle: C *expressions* may span multiple lines, and do not generally fit regular expressions. ROSE is a compiler-based infrastructure for developing program analyses and transformations. It requires working at the AST level and regenerates the affected code. Manipulating ASTs can be awkward for the developer, who is more familiar with the code base in its source code representation. The regenerated code does not follows the whitespace usage of the original code, which can lead to many superfluous diffs between the old and new versions. This is especially problematic in the case of a legacy code base, where users are mostly concerned with stability and any changes have to be checked carefully.

These factors led to consider the Coccinelle rule-based language for transforming C code [LM18]. Coccinelle is a *metalanguage* routinely used by the **Linux kernel** developer community for large-scale updating of API usages and data structure layouts. The use of Coccinelle on driver code results in mostly short `diffs`, which is not the case here. Its usage in HPC is in its inception.

3 Thinking out Coccinelle transformation rules

Coccinelle's rule syntax is a variant of the familiar `patch` syntax, thus also serving as documentation of the performed changes. A rule may *match* code at a function, definition, or declaration level, etc. Matching is independent of code layout, sensitive to control-flow, and takes into account type information. The modification specification is intermingled in the rule: that motivates terming this technology *semantic patching*. Rules can be chained; they involve notions of existential and universal quantifiers, as often found in *logic programming languages*.

Sec. 2 has enumerated well-defined factorization steps. Below, we describe their counterpart as real-life semantic patches.

3.1 Identify AoS variables for reuse in SoA

GADGET AoS variables involve variables of type `particle_data` and `sph_particle_data`, and their members. The variables are found by the Coccinelle rule `prtcl_str`, at right. Such a rule has two parts. Lines 1-7 describe a set of metavariables, which match any term of the indicated kind: `id` (line 2) matches any identifier, limited to a set of possible names (`particle_data` and `sph_particle_data`), while `I` matches any identifier without restriction. Lines 8-14 provide a *pattern* that matches a structure declaration. This pattern matches the declaration in three ways, connected as a *conjunction* by the enclosing `(` and `)` (lines 8 and 14), and by the occurrences of `&` on

```
1  @prtcl_str@
2  identifier id = {particle_data,
       sph_particle_data};
3  field list fs;
4  identifier I;
5  declaration d;
6  type ST;
7  @@
8  (
9   struct id { fs } *I;
10 &
11  ST        { fs } *I;
12 &
13  d
14 )
```

lines 10 and 12. The first pattern (line 9) exposes the details of the declaration, matching `id` to the name of the structure type, `fs` to the list of members (referred to by Coccinelle as a `field list`) and `I` to the name of the declared variable. The second pattern (line 11) matches `ST` to the entire type. The third pattern (line 13) matches `d` to the complete declaration. A match against a declaration in the C code only succeeds if all three patterns match the declaration.

Identifying members within the types of the variables matched by the previous rule is done by the following rule. This rule matches members of a previously selected structure (referenced by the redeclaration of `id` on line 2), such that the members are restricted to have one of a specified list of `type`s (line 5). Ellipses (`...`) (line 7) match the context around each matching member. This rule is applied once for each `struct` name for which a variable was found by the previous rule. There may furthermore be many matching members. This rule (and those dependent on it) is applied on each possible match.

```
1 @prtcl_str_mmbrs@
2 identifier prtcl_str.id;
3 identifier M, P;
4 typedef MyDouble, MyFloat, MyLongDouble, MyDoublePos, MyBigFloat;
5 type MT={double,float,MyDouble,MyFloat,MyLongDouble,MyDoublePos,MyBigFloat};
6 @@
7 struct id { ... MT M; ... } *P;
```

3.2 Clone structures and make them SoA

Firstly, we want to derive SoA type identifiers `id1` from previously matched AoS identifiers `id`. For this, we exploit the Python scripting functionality. Likewise, we create SoA variable identifiers.

```
1 @script:python new_prtcl_str_id@
2 id << prtcl_str.id;
3 id1;
4 @@
5 coccinelle.id1="%s_soa_t"%(id)
```

```
1 @script:python new_prtcl_str_var_id@
2 I << prtcl_str.I;
3 J;// same as fresh identifier J=I ## "_soa";
4 @@
5 coccinelle.J="%s_soa"%(I)
```

Once the SoA identifiers are ready, we can add the new type definitions in the main header, `extern` variable declarations in most sources, and the variables themselves in one specific point. The choice of the *attach points* is crucial here.

```
1 @insert_new_prtcl_str depends on prtcl_str@
2 identifier new_prtcl_str_id.id1;
3 field list prtcl_str.fs;
4 type T;
5 @@
6 extern int maxThreads;
7 ++struct id1 { fs };
```

```
1 @insert_new_prtcl_str_var_extr@
2 identifier new_prtcl_str_id.id1;
3 identifier prtcl_str.I;
4 fresh identifier J = I ## "_soa";
5 @@
6 struct id1 { ... };
7 ++extern struct id1 J;
```

```
1 @insert_new_prtcl_str_var@
2 identifier new_prtcl_str_id.id1, prtcl_str.I, new_prtcl_str_var_id.J;
3 @@
4 struct global_data_all_processes All;
5 ++struct id1 J;
```

A few adjustments are still needed in SoA: we want to exclude `union`s. So we first match them, then erase them from the recently created definitions.

```
@match_anon_union@
identifier id;
identifier J;
field list[n] fs;
identifier new_prtcl_str_id.id1;
@@
struct id1 { fs
 union { ... } J;
  ... };
```

```
@rm_union_from_struct
 depends on match_anon_union@
field list[match_anon_union.n] fs;
identifier new_prtcl_str_id.id1;
field fld;
@@
struct id1 { fs
- fld
  ... };
```

One can *shortlist* member types to be made into allocatable C arrays. The remaining members, which are not transformed into pointers, can be deleted.

```
@make_ptr@
identifier new_prtcl_str_id.id1, M;
typedef MyDouble, MyFloat, MyLongDouble,
    MyDoublePos, MyBigFloat;
type MT={double,float,MyDouble,MyFloat,
    MyLongDouble,MyDoublePos,MyBigFloat};
@@
struct id1 { ...
- MT  M;
+ MT *M;
  ... };
```

```
@del_non_ptr@
identifier new_prtcl_str_id.id1, J;
type T;
type P != {T*};
@@
struct id1 { ...
- P J;
  ... };
```

Only pointer fields now remain – these will serve as the allocatable SoA arrays. Now a little trick is needed, to overcome Coccinelle's limited preprocessor support. We insert a special `__define` symbol just before each member (to be later replaced with `#define` via a script). At compile time that symbol will only be defined if the member had no deactivating `#ifdef`s context.

```
@define_per_field_syms@
identifier new_prtcl_str_id.id1, M; type MT;
typedef __define; fresh identifier si =  "HAVE__"##id1##"__"##M;
@@
struct id1 { ...
+ __define si;
  MT *M; ... };
```

3.3 Helper functions for SoA arrays memory management

Each of the new SoA `struct`s's arrays needs memory management statements. We cluster those in specific new functions.

```
@insert_per_type_soa_functions@
identifier new_prtcl_str_id.id1;
fresh identifier soa_init_fid="soa_init__"##id1;
fresh identifier soa_alloc_fid="soa_alloc__"##id1;
fresh identifier soa_free_fid="soa_free__"##id1;
@@
void allocate_memory(...) { ... }
++ void soa_init_fid(struct id1*P) { }
++ void soa_alloc_fid(struct id1*P, size_t N) { }
++ void soa_free_fid(struct id1*P) { }
```

Populating them with `malloc`/etc. statements proceeds by referring to members `P.M` of `id1`. In order to support multiple build-time configurations, each `P.M` allocation statement needs a surrouning `#ifdef`/`#endif` unique to that `id1`, `M` pair (recall `define_per_field_syms.si`). Given the overlap with

previous rules, we omit these rules. We proceed similarly for deallocation and *gather/scatter* to/from AoS (e.g. in I/O and network communication).

3.4 Transform Expressions from AoS to SoA, globally

Rules in this section can be independently applied to the bulk of source files. As in Sec. 3.1, matching begins on the original AoS particle structures.

```
@ostr@
identifier id = {particle_data,sph_particle_data};
identifier P;
type ST;
@@
(
 struct id { ... } *P;
&
 ST { ... } *P;
)
```

Create an SoA `struct` type `id1` based on AoS `id`, and match it in `nt`.

```
@script:python pps@
id << ostr.id;
id1;
@@
coccinelle.id1="%s_soa_t"%(id)
```

```
@nt@
identifier pps.id1, I;
type T;
@@
struct id1 { ...  T I; ... };
```

Create SoA identifiers `S` based on AoS `P` and substitute in all expressions.

```
@script:python pid@
id1 << pps.id1;
P << ostr.P;
S;
@@
coccinelle.S="%s_soa"%(P)
```

```
@soa_access@
identifier ostr.P, pid.S, nt.I;
expression E;
@@
- P[E].I
+ S.I[E]
```

Here we exploit that the variables referred to by *metavariables* `P` and `S` are declared globally. Their members `I` instead are bound to the given `struct`. `E` matches arbitrarily complicated index expressions.

At the cost of flexibility, one may have as well written SoA `struct`s by hand and applied only this section's rules: they account for the near totality of the code changed.

4 Current Status and Conclusions

This paper describes a technique to obtain an SoA port of GADGET. Preprocessor-related complications required a few by-hand small code changes. Besides that, as long as the matching points in the code remain stable, these *semantic patches* will stay valid and applicable. This preserves the original AoS-based development model; the *semantic patches* plus helper scripts are ready for use in the GADGET repository. Worthwhile to note, the output of `diff -r` with a SoA converted version amounts to $18K$ lines. A few last steps are still pending before an SoA-GADGET can be fully usable, and the expected $> 2\times$ speedup verified: I/O and MPI related functions need more intervention, postponed to a future collaboration with the GADGET lead developers. The quality of the changes has been ensured by using the internal test suite and developing a few

custom check scripts. The semantic patches have been developed on samples of the code, which allowed quick and terse visual inspection of the changes. Coccinelle preserves existing coding style (e.g. spacings, indentation) very well, which leads to a minimal `diff`.

Generally, HPC is witnessing a divergence in APIs and programming models. This may be a problem for small, geographically sparse domain scientist teams working on large codebases, risking 1) premature code obsolescence, 2) wasted, repeated effort, and 3) operation of hardware at below-optimal performance.

This paper illustrated novel use of a source-to-source translator in such a context. Adapting these recipes to introduce SoA in another codebase may be straightforward. However, the cleaner a codebase is, the easier it is to develop *semantic patches*. Adopting *coding guidelines* and following emerging *research software engineering* practices [ABD+20] will help, no matter how small the team.

References

ABD+20. Hartwig Anzt, Felix Bach, Stephan Druskat, Frank Löffler, Axel Loewe, Bernhard Y. Renard, Gunnar Seemann, Alexander Struck, Elke Achhammer, Piush Aggarwal, and et al. An environment for sustainable research software in Germany and beyond: current state, open challenges, and call for action. *F1000Research*, 9:295, Apr 2020.

BIHK16. Fabio Baruffa, Luigi Iapichino, Nicolay J. Hammer, and Vasileios Karakasis. Performance optimisation of smoothed particle hydrodynamics algorithms for multi/many-core architectures. *CoRR*, abs/1612.06090, 2016.

LM18. Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the Linux kernel. In *2018 USENIX Annual Technical Conference, USENIX ATC*, pages 601–614, 2018.

PHS15. Simon J. Pennycook, Christopher J. Hughes, and Mikhail Smelyanskiy. Chapter 8 - optimizing gather/scatter patterns. In James Reinders and Jim Jeffers, editors, *High Performance Parallelism Pearls*, pages 143 – 157. Morgan Kaufmann, Boston, 2015.

RDW+19. Antonio Ragagnin, Klaus Dolag, Mathias Wagner, Claudio Gheller, Conradin Roffler, David Goz, David Hubber, and Alexander Arth. Gadget3 on GPUs with OpenACC. In *Parallel Computing: Technology Trends, Proceedings of the International Conference on Parallel Computing, PARCO 2019, Prague, Czech Republic, September 10-13, 2019*, volume 36 of *Advances in Parallel Computing*, pages 209–218. IOS Press, 2019.

RTB+15. Antonio Ragagnin, Nikola Tchipev, Michael Bader, Klaus Dolag, and Nicolay Hammer. Exploiting the space filling curve ordering of particles in the neighbour search of Gadget3. In *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing (ParCo)*, volume 27 of *Advances in Parallel Computing*, pages 411–420. IOS Press, 2015.

Spr05. V. Springel. The cosmological simulation code GADGET-2. *MNRAS*, 364:1105–1134, 2005.