Performance Engineering

Alexander Hulpke

April 1999

This talk is an attempt to give general strategies for a multi-faceted problem. Most real-world problems will need a closer, specific treatment. Even if the talk succeeds, the slides on their own might look extremely simple-minded.

My general assumption will be that you *have* code that runs too slow and that this is neither because of an inefficient algorithm design, nor because of the inherent difficulty of the problem.

Concentrating on GAP specifics I will talk about some traps and problems which I encountered again and again in GAP code and about remedies and ways to avoid these. These lists are certainly not comprehensive but may give some hints.

A la recherche du temps perdu

The first problem is to locate where unreasonably much time is spent. In simple cases interruption and backtrace or differences of `Runtime();` will suffice, in more complicated cases `Profile`ing might be necessary.

- The minimum clocktick difference is 10ms. Functions which take very little time are only profiled properly if they get called very often.
- Garbage Collections can influence single runs.
- Start GAP with command line option `-g -g` to get a display of all garbage collections. A call `GASMAN("collect");` will enforce a collection.
- The "culprit" function must take a substantial part of the runtime – otherwise it drowns in a sea of noise.
- Usually profiling gives only a "top-level" time distribution.
- Usually the "sum" of the profile information is relevant: The functions called as children are only called because of the function.
- Kernel methods are not profiled.

- Functions are only profiled if profiling is turned on for them deliberately.
- Do "elementary" operations show up high in the profile?
- Get an idea of "reasonable" runtimes for subtasks.
- Dissection of high-level functions might be necessary.
- Do the times of parts of the algorithm sum up?
- `TraceMethods` is ignored for profiled operations.
- If operation `A` is installed as a method for operation `B`, calls to `A` are *also* listed for `B`.

Example

Compute the conjugacy classes of a group of size 1152. (These are pruned outputs.)

The generic default method uses cyclic extension. There also is a (usually more efficient) method for solvable groups.

Define the group, turn on profiling for a few functions of the "solvable groups" algorithm.

```
gap> g:=TransitiveGroup(12,200);;
gap> ProfileOperationsAndMethods(true);
gap> ProfileFunctions(OCOneCocycles,
> SubgroupsSolvableGroup,
> InvariantSubgroupsElementaryAbelianGroup,
> ActionSubspacesElementaryAbelianGroup);
gap> l:=ConjugacyClassesSubgroups(g);;
gap> DisplayProfile();
```



```
  count  self/ms  chld/ms  function
 231659      770       30  ADD_LIST
    505       10     2730  StabChainOp
   1740       60     3370  Enumerator: for a collection with
  13541     2960      690  Meth(DepthOfPcElement)
      1        0     4270  Zuppos
    234      270     4480  Meth(InducedPcgsByPcSequenceAndGe
    234       10     4750  InducedPcgsByPcSequenceAndGenerat
    234       10     4760  InducedPcgsByGeneratorsNC
    234        0     4810  InducedPcgsByGenerators
    468    11150     9310  NormalizerOp: perm group
    234        0    20460  NormalizerOp
      2     1290    35270  LatticeSubgroups: cyclic extensio
      1        0    36570  LatticeSubgroups
      2        0    36580  ConjugacyClassesSubgroups
          36660            TOTAL
```

Now the same with the solvable groups method

```
gap> g:=TransitiveGroup(12,200);;IsSolvableGroup(g);;
```



```
  count  self/ms  chld/ms  function
      2        0      110  InvariantSubgroupsElementaryAbe
      6       30     2090  ActionSubspacesElementaryAbelia
    115      990     5320  OCOneCocycles
    572       10     6600  StabChainOp
   3576       70     6620  ClosureGroup
  72354      510     6400  DepthOfPcElement
    226      540     7290  OrbitStabilizerAlgorithm: for pc
     72       20     8190  ExternalOrbitsStabilizersAttr
   1580       40     8910  InducedPcgsByPcSequenceAndGenera
    272     6650     9790  NormalizerOp: perm group
    148       10    16950  NormalizerOp
    200       10    26980  CentralizerModulo
      2      230    78240  SubgroupsSolvableGroup
      1       20    78550  LatticeSubgroups: elementary abe
          78800            TOTAL
```


Analysis

Analyzing the code we find that `CentralizerModulo` is only used to compute normalizers of the subgroups. The interface to the algorithm permits to turn this off:

```
gap> SubgroupsSolvableGroup(g,rec(retnorm:=false));;
gap> DisplayProfile();
  count  self/ms  chld/ms  function
  58778      440     4270  DepthOfPcElement
     33        0     6880  SmallGeneratingSet
    115      870     6360  OCOneCocycles
     72       10     7890  ExternalOrbitsStabilizersAttr: m
      2      270    33630  SubgroupsSolvableGroup
          33900            TOTAL
```

Finally, just to show the difference here are times without profiling:

	cyclic extension	elementary abelian extension
Profile	36	78
No Profile	24	41

What can go wrong ...

Different calculations which yield equivalent results can take notably different time and GAP might do extra tests you do not really want to happen.

It is also possible that the operation chosen does more than what is needed and this "more" comes at a premium.

... and how to avoid it

To avoid such problems you should have an idea what you intend GAP to do and – if possible – also a rough idea how it should do it.

You also probably will want to ignore some of the (default) convenience mechanisms built in to save beginners from accidentally getting wrong results.

This also might mean that you have to write own code or modify existing library code.

Of course it is still possible that GAP library routines are underperforming. If you encounter a problem of this kind please let us know!

Asking the wrong question

GAP contains many operations and several of them might solve your problem. However the cost of the underlying algorithms can be quite different.

- In general, ask for the *weakest* result that will answer your question.
- The mathematical definition of an object might not be the optimal one for calculations. (Don't get Sylow subgroups by computing the Lattice.)
- Try to use the lowest (reasonable) level representation in inner loops (row vectors and matrices instead of generic vector spaces). If necessary rephrase the problem.
- Some operations in GAP are there to answer certain types of questions interactively with little parameter interface. (E.g. `Factorization`.) They are not suited to be called from within algorithms.
- When checking large sets of elements, can the question be phrased for elements up to conjugacy or automorphisms? (The function `MorClassLoop` provides a general interface to run through tuples of elements.)
- Do not compute all subgroups unless you really want (almost) all.
- It might be worth to change an algorithm to run it only partially (only on some classes,

modulo a normal subgroup, stop if one solution was found – don't wait for all). Some operations already provide an extended interface of this kind.

- The pigeonhole principle is your friend. (If there is only one candidate left and theory tells there must be a solution this is it.)
- Is the same question asked again and again? (Dynamical Programming.)
- Does the code create *equal but not identical* objects (e.g. `InducedPcgs`) anew?

A dual problem is not asking for enough, respectively using an operation interface that will not return everything that has been computed.

- Computing the orbit and the stabilizer separately.
- It is possible that the library interface of various operations does not yet permit to obtain all intermediate data. Complain.

Calling the wrong method

If different methods are applicable for the same object all will return a valid result. Thus you won't note immediately if the method called is not the (optimal) one you believed it to be.

- `ProfileOperationsAndMethods` lists the identification strings of the methods.
- `TraceMethods` displays the identification strings whenever a method gets called.
- `ApplicableMethod` can be used to check which method will be used. (**Note** however: Some functions still dispatch on operations and a few operations dispatch first in the kernel.)
- The good method might not be applicable because some (obvious) filters are not set. (`IsFinite` is a well-known culprit.)
- Set the filter once the object is created.
- Should the filter be implied? (`InstallTrueMethod` takes care.)
- The better method is actually ranked lower than a not-so-good default.
- Is the method ranked so low, because filters are missing (which are set for the default method)? Again `IsFinite` is a likely candidate.

- Explicitly rerank the method higher.
- If it happens for many methods it might be worth to rerank filters.
- The only implemented method may be too generic.
- Is it necessary to call the operation? (`NC` versions might avoid element tests.)
- Don't let GAP check properties you know already.
- When not debugging use `NC` versions of functions/operations.
- A function might (implicitly) ask for properties of an object you already know. Tell such properties to the object by setting the corresponding attributes.
- But don't spend all time just maintaining information.

Using the wrong representation

The choice of a suitable data structure can substantially influence the performance of an algorithm. This choice of a representation can happen unknowingly: While one might be interested only in an "ideal" object, GAP must to be given a concrete representation, the (right!) choice of this may be crucial for performance.

- It can be worth to change the representation in which the calculation takes place. (For example computing in an isomorphic PcGroup so that all pcgs calculations are done with respect the FamilyPcgs.)
- Matrix groups translate almost all calculations via a `NiceMonomorphism` into a permutation representation.
- Lists or matrices of finite fields that are concatenated are not automatically in the matrix representations. Matrix arithmetic then can be comparatively slow.
- There are compact kernel data structures for vectors and matrices over small finite fields. The kernel implements fast addition/multiplication for these. `ConvertToVectorRep` and `ConvertToMatrixRep` change objects accordingly. (It does not improve element access.)

- Operations for words usually convert it to an internal list format, access via Syllables is much quicker.
- Multiple operations on complicated objects are often done quicker in the permutation group image of the operation homomorphism.
- Use the implicit homomorphisms of the group operation functions ($\rightarrow$ talk on external sets)
- Try to build intermediate objects in a way that they can be used directly.
- Is the algebraic context reasonable? (Can one change from rationals to integers? From rational functions to Laurent polynomials?)
- Integers $> 2^{28}$, finite fields $> 2^{16}$ require a more complicated arithmetic.
- In algorithms for solvable groups try to use a Pcgs or ModuloPcgs instead of explicitly creating many factor groups (each with its own collector).
- For deeply nested lists the type determination can be expensive.
- Plain lists are typed in the kernel, a list may not (yet) know to be sorted, dense or homogeneous.

Lists, Collections and their elements

A favourite place to waste time is to find elements in a list or collection.

- Element tests become much quicker in a sorted list.
- A list that contains immutable objects may **not** store that it is sorted.
- A few "library lists" (elements of GF$(p)^n$, right transversals) have an intelligent `Position` function that is faster than an ordinary search. It will not inherit to sublists.
- For "property stabilizers" (GL, SO, wreath products) much simpler element tests are possible.
- If element tests turn out to be the time consuming part investigate other data structures. Frequent in tests in small objects that are handled by an algorithm might be better via sorted element lists.

Lists are also used to represent row vectors and careless arithmetic can create lots of garbage. There are in-place operations that avoid this.

For example instead of `sum:=sum+c*vec` use `AddCoeffs(sum,vec,c)`.

The devil in the detail

In many cases speed tuning at the cost of legibility, generality or maintainability may not be worth the price, but inner loops may warrant such measures bordering on the desperate.

- Do not create garbage.
- Try to use in-place operations whenever possible.
- Instead of `l=[]` compare `Length(l)=0`.
- Do not build intermediate objects which are not used further on (for example do a polynomial multiplication with the coefficient lists yourself and do not create rational functions.)
- Check the methods for operations whether they still dispatch. Call the innermost operation.
- Try to precompute subresults.
- If possible use kernel operations (e.g. sublist, scalar product) – even if they do a bit more than library code.
- There is about 10-15% benefit from "unraveling" a simple recursion by a stack.

Kernel vs. Library

In average C code is a factor 4-5 faster than the same code in the library.

However usually most time is spent in a small part of the code. (e.g. permutation multiplication or matrix arithmetic)

The philosophy of the GAP design is to put these routines in the kernel while preserving the rapid prototyping convenience of an interpreted language.

Compiling the whole library gives a disappointing increase in performance (10 %).

The improvement will be better if most time is spent in "elementary" calculations as Integer Arithmetic, List Handling or Combinatorics.

Compiled code still has "untyped" variables. If it is known that certain variables will be small integers hand-specialization can be worthwhile.

The compiler cannot use special C features (packed representation, bit manipulation) for GAP lists unless this is provided for by the GAP language.

Compilation is a last resort if the algorithms cannot be improved any more.

Creating many objects

GAP ensures that two equal types are identical. This makes creating or changing types expensive, if it happens often:

```
for i in params do
  o:=Objectify(NewType(family,filter),rec(param:=i));
  SetMyAttribute(o,value);
  SetMyProperty(o,true);
  Add(l,o);
od;
```

In every iteration of the loop a new type is created, GAP checks whether this type already exists (finds out it does) and returns the old type. An obvious improvement is:

```
typ:=NewType(family,filter);
for i in params do
  o:=Objectify(typ,rec(param:=i));
```

or even better to store a few default types in *family*.

However assigning attributes (or properties) also implies a type change. The quickest way to create the objects thus is:

```
typ:=NewType(family,filter and HasMyProperty and MyProperty);
for i in params do
  o:=ObjectifyWithAttributes(typ,rec(param:=i),
     MyAttribute,value);
```

Caveat: If any of the attributes has a separate setter method installed, `ObjectifyWithAttributes` cannot use the quick way but has to call the setter. This will be *slower* than the standard method.

Keeping many objects

Once many objects have been created it is possible they will remain in existence for some time and use up the memory. The numbers given are only the pure data content, there is some constant offset for the bag of each non-immediate object.

Memory requirements:

Type	Bytes (32 bit system)
small integer	4
integer $> 2^{28}$	$\lfloor \frac{(\log_2(n)/8)}{8} + 1 \rfloor \cdot 8$
FF Element	4
list	$4 \cdot l$ for pointers
GF(2) vector, blist	bit list
GF(q) vector	compressed in bytes
permutation	$2 \cdot deg$ (but $4 \cdot deg$ if $deg > 2^{16}$)
word	8, 16, 32 bit syllables, integers list

The best way is not to run into this problem, but here are a few desperate measures if you really **must** keep that many objects.

- Attributes which have been computed once remain stored.
- `NamesOfComponents` lists the components which are known for an object. (Currently) every attribute is stored that way.
- The values of "large" attributes which are likely not to be used later can be computed via `AttributeValueNotSet`.
- this does not help, if the method involves other attributes. (Say `AsList` calls `AsSSortedList`.)
- If this is suspected it might be worth to create **another** object with the same generators, compute its attribute value and delete this new object afterwards.
- You might want to assign a few attribute values for the new object, e.g. `Size`, to avoid expensive recalculations.
- Objects which are otherwise dead might be kept alive via pointers to families or homomorphisms.
- Inessential information of this kind can be kept via *weak pointers*. (This are pointers that do not enforce objects to stay alive over a garbage collection.)