

Considerations for Level 3 Software Design

Bálint Joó, Jefferson Lab

January 10, 2007

1 Introduction

In this document I will examine SciDAC level 3 from the point of Software Engineering. I will look at key metrics of *cohesion* and *coupling* - to be defined later - as a guide to software re-usability. The discussion will focus on steps that can be taken which will improve the situation of software regarding these attributes.

The document is organized as follows. In section 2, I will discuss/define what I believe SciDAC Level 3 is. I will define what cohesion and coupling are in section 3. In sec. 4, I will then examine level 3 from from the point of view of cohesion and coupling and identify positive and negative features. I will also discuss ways in which the negative features can be softened or overcome. In section 5 I will discuss, as a case study, how the Chroma software system uses Level 3 components. Finally, I will summarize my key points in section 6

2 SciDAC Level 3

Perhaps the best way to introduce SciDAC Level 3 is to consider it together with the other SciDAC levels. These are:

Level 1: Nuts and Bolts – This level contains the QCD Message Passing (QMP) and QCD Linear Algebra (QLA) packages. QMP abstracts away communications hardware and is less complicated than MPI. By using QMP, QCD codes can share the communications infrastructure across all platforms. QLA provides a standardized naming convention and API for low level linear algebra routines.

Level 2: QCD Data Parallel (QDP) and QCD I/O (QIO) – This level provides data parallel primitives such as lattice wide operations and shifts. The data parallel interfaces are implemented in the QDP/C and QDP++ packages. This level also contains the QIO package for Binary I/O through LIME encoded files. This level essentially provides an abstract, data parallel, QCD specific computer abstraction onto which QCD codes can be layered.

Level 3: Optimization Layer – It was originally anticipated that most QCD development can occur with reasonable efficiency in Level 2. However, occasionally, very highly optimized components are needed to squeeze maximum performance out of a particular architecture on commonly executed tasks. The highly optimized, task and machine specific software to perform these tasks is classified as level 3. It does not need to rely on lower levels (although it may). This layer usually contains some form of optimized assembler code. Typical Level 3 components are Dirac Operators, Force Terms and Solvers.

As we can see, the key features of level 3 software are the following

- Architecture specific
- Task specific
- Performance oriented

Before we describe the effects of these attributes on re-usability let us consider reusable software from a software engineering perspective.

3 Cohesion and Coupling

In software engineering packages have attributes of cohesion and coupling. The term coupling is intuitive - it refers to the relation of one software module to other things. The term cohesion is a little more amorphous but essentially it refers to how easy it is for one module to be used by others.

3.1 Cohesion

A package is cohesive, if it can be used by other packages. The more packages can use a given package, the more cohesive that package is said to be. High Cohesion is a desirable feature, as it indicates that a package is highly reusable (by definition). Some factors which can make a package cohesive are:

- *Task Focus* - the package should do one thing and do it well.
- *High "market demand"* - the task which the package performs is in demand. Typically the package performs a task which is needed very often, but is not straightforward or undesirable to re-engineer.
- *Ease of Integration* - the package is straightforward to integrate with packages that use it (I will refer to these as *client* packages or client modules.) Ease of integration has several facets:
 - *In the code*: Clean API, closed namespace, straightforward resource (typically memory) and lifetime management
 - *In the build*: Uses standard language, and tools.
- *No Side Effects* - using the package does not have unexpected consequences elsewhere in the client code.

3.2 Coupling

The couplings of a package are all those components that a package brings with itself. Unlike cohesion, coupling is a negative attribute as it makes a package more difficult to maintain (maintenance may need to spread to the couplings). In some cases couplings can directly counteract the benefits of cohesion. Some typical couplings are:

- *Dependent Libraries* to which a module couples need to be ported whenever the package itself is ported. The dependent libraries may evolve and acquire bugs. Interface changes in dependent libraries need to be tracked.
- *Physical Couplings* are couplings that are induced by the use of tools - we will discuss some of these in the context of level 3 in section 4

Ideally one aims for highly cohesive packages with minimal couplings. This is not always possible. Layered software for example is necessarily coupled through the layers. Application frameworks can be highly coupled - often because their intention is in fact to assemble (and so couple) functionality from many sub modules. However, the reuse pattern of application frameworks is typically different from that of stand-alone level 3 packages. An analysis of cohesion and coupling in an application framework is a fitting subject for another discussion.

4 Cohesion and Coupling in Level 3 software

Let us now consider some general features of level 3 and see how they relate to couplings and cohesion.

4.1 Cohesion in Level 3

The first aspect, that the level 3 package should do one task well, is already embodied in the definition of level 3. Also, because of its task specificity, one would naively expect level 3 packages to be fairly self contained.

Let us consider the market demand. Level 3 was envisioned to supply the following components:

- Dslash operators
- Dirac operators
- Solvers for use in Force Terms, Energy calculations and Propagator Computations
- Force terms for Molecular Dynamics.

I separate Dslash and Dirac operators here, because the same Dslash operator can be used in the construction of several different Dirac operators.

We should notice, as we look upon this list from first item to last, that successive items may be coupled to preceding items, ie: Dirac operators use Dslash operators, solvers and force depend on the Dirac operators and force terms depend on solvers.

Another thing to notice is that as we move down the list of items from first to last, our space of choices and conventions constantly increases. In the case of Dslash operators perhaps the only thing to really worry about is checkerboarding conventions. In the case of Dirac operators we may also need to worry about parametrization, and normalization (eg: Should we use m or κ in Wilson Dirac operators?)

Preconditioning strategy affects the form of the preconditioned operators, which will have an impact on our solvers and force terms. In the case of even-odd preconditioned clover, for example, we can evaluate the $N_f \log \det M_{ee}$ part of the fermion determinant directly without pseudofermions, and this has a different force term than if we had used pseudofermions for both checkerboards. Different forms of preconditioning are also expected to have effects in the speed of convergence in solvers.

While in Level 2 it is easy to factor out the linear operator from the solver, in level 3 this kind of factoring may be hindered by the fact that the data layout of one level 3 linear operator may differ from the data layout of another, and because the solver may also need to know about the preconditioning strategy to prepare the sources and to reconstruct the solutions. Likewise, relating the solution of the system $M^\dagger M \phi = \chi$ and $M'^\dagger M' \phi = \chi$ where M and M' are different preconditioned operators, corresponding to the same unpreconditioned operator, is not necessarily straightforward to do efficiently. In order to factor off the Level 3 operator from the rest of the Level 3 solver, one needs to arrange that the fermion data layout in the solver be the same as in the operator. Choosing a single data layout for the solver and the factored operators may result in some operators not being as efficient as possible (if the chosen layout is not ideal for that operator)

The order of the components in the list above essentially is ordered in terms of the amount of “market demand” that they can supply. Dslash operators are most reusable (highest supply) and solvers and force terms are the most niche commodities - tied to particular preconditionings, data formats, conventions etc. They fulfill a demand that is potentially very small. On the other hand, once a set of production conventions, preconditionings etc is decided, the rest of the market effectively evaporates.

This poses a dilemma: In which level 3 components should a lot of effort be invested? The ones that can be most easily reused in many systems, or the least reusable niche pieces which will be used in a large production runs (eg: AsqTAD force term, and DWF inverter?)

Level 3 also has some factors that can reduce cohesion. The key ones among these are *task specificity* on which we have already touched above, and *machine specificity* which is a form of *physical coupling*.

4.2 Couplings in Level 3

One of the defining features of level 3, is task specificity, which suggests self containment. Typically the amount of dependent libraries are small (QMP, and potentially some custom memory management code). However, due to the

need for specialist optimization (eg: assembler), level 3 packages have a large potential for physical couplings. Some of these are defined below:

- use of compiler features such as `#pragma` statements, vector mathematics, compiler intrinsic functions or inline assembly is typically not portable between compilers and couples the software to the compiler.
- use of special purpose compilers (eg: custom code generating software or modified compiler) also couples the software to that compiler
- The use of assembler files couples the code to the particular assembler. Eg: the xlc family assembler and the GNU assembler are not compatible in terms of their register naming and their preambles to function definitions etc.
- Use of tools such as GNU Autotools can aid the manageability of dealing with different compilers in different modules. However, experience shows that the resulting software is sensitive to the Autotools version and builds can fail for example if `aclocal` is the wrong version. This is very easy to fix (run a new version of `aclocal`) but is an unnecessary waste of time.
- by definition level 3 software is coupled to the hardware architecture
- Level 3 software is occasionally extracted or exposed from a larger framework in which case it can couple the client software to that framework, bringing with it contextual baggage and pollution of the namespace of the level 3 package. This can complicate building and installation, as well as API level integration in the code.

Some of these couplings are not as severe as they could be in principle. For example recently there has been convergence between several key compilers on the handling of compiler intrinsics especially for the Intel X86 architectures and compatibles. Recent versions of the Intel, GNU and PathScale compilers all use the same set of compiler intrinsic functions and are moving/have moved towards compatibility in the area of inline assembler. Likewise there is convergence at least at the level of linkage between the xlc and gcc compilers on AIX and BlueGene.

4.3 Increasing Cohesion and Decreasing Coupling

Let us consider some strategies do increase cohesion and decrease coupling.

One key aspect that gets in the way of cohesion is lack of agreement on conventions (such as preconditioning). Currently, for example, there are three different kinds of even odd preconditioning for Domain Wall Fermion inversion in use in USQCD. The CPS system uses a 5 dimensional preconditioning, the Chroma code uses a straightforward Schur preconditioning where a linear operator is block decomposed as:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ CA^{-1} & 1 \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & D - CA^{-1}B \end{pmatrix} \begin{pmatrix} 1 & A^{-1}B \\ 0 & 1 \end{pmatrix} \quad (1)$$

and the MIT CG-DWF domain wall fermion solver uses a 4D preconditioning with a slightly different Schur Decomposition:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} D & 0 \\ CA^{-1}D & D \end{pmatrix} \begin{pmatrix} D^{-1}A & 0 \\ 0 & 1 - D^{-1}CA^{-1}B \end{pmatrix} \begin{pmatrix} 1 & A^{-1}B \\ 0 & 1 \end{pmatrix}. \quad (2)$$

In addition to this there are slightly different conventions for parameters and normalization in the three systems. As an aside, only the MIT CG-DWF solver is a true level 3 package from the point of view of self containment. Both Chroma and CPS have essentially level 2 implementations using level 3 and level 2 linear operators. However, this does not necessarily make these solvers less performant.

In this case essentially two options are open to us to increase cohesion:

- One way is to agree on conventions and some standard interface. This would increase cohesion in several ways:
 - It would boost the market share supplied by the resulting package.

- It would provide the same look and feel to all clients (ease of integration)

However, it has several disadvantages:

- It forces the same convention on everyone who uses the package and thus may stifle innovation especially if users are *required to use it* or are *strongly pressured* to use it.
- One standard interface may not be appropriate to all platforms. The needs of a package on one hardware may suggest one particular interface whereas to accomplish the *same task on a different platform* a different interface may be beneficial. This is particularly true for the specification of the data layout of external fields that the clients must provide.
- Most significantly, it may be extremely difficult to reach a consensus especially for pieces of software which are already in existence and in production. Once a piece of software is in production, changing it is dangerous. It may introduce or expose existing bugs.¹ Also for the parties that already have their own code in their framework there is little impetus to adopt a different convention from what they have, unless it will provide them with a truly significant performance increase. A change in preconditioning for example can be extremely pervasive (human cost) throughout the framework. Finally, in this situation moving towards a standard is actually a replication of effort. In this situation what tends to happen is that each party will instead push to have their version become the standard and a lot of effort is wasted in abstracting or extracting their code into an independent package, which then may not be used by others.
- The other avenue is to employ an adapter (follow the Adapter pattern). An adapter is a piece of software which takes a piece of software with one interface and presents to a client software a different interface. It is also known as a wrapper. They have the following advantage
 - Adapters increase cohesion by making a package usable to a client that otherwise could not use it.
 - The adapters can be used to unify the interfaces of different packages that may do the same job on different architectures. This allows client code to be flexible by softening its couplings.

Adapters are typically client code specific (for example they convert from the data format of the client into the data format of the Level 3 package) and also tend to be fairly thin pieces of software that is not an onerous burden for the client code to provide.

Adapters can also have disadvantages

- Adaptation (of data format for example) can have a cost which may be high. Potentially Dirac Operators require adaptation of the gauge field on construction and the fermion fields on application. However, adaptation of the fermion fields on every application may be too costly. Typically Level 2 packages using Level 3 Dirac operators adopt the format of the fermion fields that the Level 3 Dirac operators expect so that fermion fields do not need to be adapted on application. The cost of adapting the gauge fields is generally amortized over a large number of applications. On the other hand, solvers may perform enough operations to amortize the cost of the adaptation of fermion fields.
- If one has to go through multiple adapters the previous disadvantage is proportionally magnified.

There are some things that adapters are not able to solve efficiently through no fault of their own – for example converting between preconditioning strategies.

Finally it should be noted that while adapters are thin pieces of code, it does not mean that they are trivial to write, since typically they involve data conversion, resource and lifetime management (all of which can lead to egregious bugs such as memory leaks).

5 Case Study: Using Level 3 routines in Chroma

We now try to illustrate several of the issues we discussed by describing how we use level 3 software in Chroma.

¹Is ignorance bliss?

5.1 A brief introduction to Chroma

Chroma is a lattice QCD framework built on top of the QDP++ software from SciDAC level 2. It is written in C++ mostly. Most of the Chroma library is written in terms of level 2 framework.

In addition to this the Chroma C++ object structure provides top level abstract classes for linear operators and their associated force terms, inverters for various systems (propagator or $M^\dagger M \phi = \chi$), monomials (2 flavour or one flavour rational), Hamiltonians, molecular dynamics, Hybrid Monte Carlo² and measurements.

Concrete, derived classes implement the functionality of the abstract base classes. Class instantiation is typically done through a factory mechanism which allows the selection and instantiation of concrete derived classes at run-time.

Level 3 packages are typically interfaced to in these derived classes. The derived classes then act as adapters, adapting the interface of the level 3 package to the interface of the abstract base class. The decision to use a version of a derived class written in level 2 or a version interfacing to level 3 is made using type aliasing (typedef statements) and is controlled through the build system (autoconf `--enable` and `--with` switches). Chroma manages lifetimes of objects using *reference counting handles*.

5.2 Chroma and Level 3 Packages

Chroma uses the following Level 3 packages:

- The *SSE Wilson Dslash* package is used to provide an optimized Dslash operator for SSE and SSE2 compatible architectures. This is used in the otherwise level 2 Wilson Dirac operator, and in an operator which applies the Dslash operator to a vector of vectors. This vector Dslash is used in otherwise level 2 Domain Wall like operators. It should be noted that a lot of the linear algebra here is already optimized in level 2, so that overall most Wilson like operators end up being almost completely in assembler. *Despite these accelerations the Chroma Linear Operators are essentially Level 2 objects*. These linear operators are used in solvers that also use assembler in their linear algebra, but are otherwise also level 2 objects.

The Wilson Dslash operator package is relatively self contained. Parallel versions of it require QMP for communications. Chroma and QDP++ have the same spin basis as the SSE Wilson Dslash package and the layouts of the fermion fields agree between the two as long as QDP++ is built with the two colour checkerboarded layout. Hence fermion fields do not need to be adapted for use of this Dslash by Chroma. The gauge field layout of the SSE Dslash operator is different from Chroma, and so Chroma as a client code must rearrange its gauge fields to use level 3 package. The SSE Dslash software contains functions to do this which Chroma can call. The gauge field rearrangement is done in the constructor of the client class (and need not be explicitly called by a user) and correspondingly, cleanup takes place in its destructor. Chroma places a reference count on the usage of this operator so that communications buffers are only allocated once no matter how many instances of the operator are in scope.

The SSE Dslash code is written using GNU inline assembly and relies heavily on preprocessor macros. It has been built with the gcc-3.x and PathScale 5.9 series of compilers. My most recent attempts to compile with the Intel 9.1 compiler failed on the inline assembler with an internal compiler error.

- The *BAGEL Dslash operator* is not a SciDAC package, but it is written in the Level 3 style and is used in that way by Chroma. It provides Dslash, vector Dslash and Wilson Dirac operators for RISC based platforms, most notably the QCDOC and the BlueGene/L. The Wilson Dirac operator provided uses the κ normalization which is different from Chroma convention and thus only the Dslash and Vector Dslash functionality is used in Chroma client linear operators.

The Dslash code is packaged in a similar way to the *SSE Dslash operator* with GNU autotools. The fermion field format in the QDP++ 2 checkerboard layout can be directly consumed by the BAGEL Dslash, but the gauge field must be transformed. In a similar way to the SSE case, this is done in the constructor and the destructor of

²Hybrid Monte Carlo + Rational Monomials = RHMC

the client class, and like the SSE case, a reference count is placed on the instances to make sure that buffers and infrastructure are only allocated and freed once.

The *BAGEL Dslash* package brings with it an unpleasant coupling of the BAGEL assembler generator framework which must be installed for the BAGEL Dslash operator to build. This coupling could be circumvented in principle but licensing restrictions do not allow this. Installation of the BAGEL generator is straightforward.

The BAGEL generator in its current form produces assembly that works with the GNU assembler. The assembly produced cannot currently be consumed by the native assembler in the x86 family of tools due to syntactic differences between that assemblers and the GNU assembler.

The BAGEL Dslash operator fulfills an important need for a very efficient Dslash operator on QCDOC and BlueGene however it has a few disadvantages:

- Although freely downloadable it is technically proprietary
- It is not straightforward to maintain the BAGEL code and so we rely on the author of the code for maintenance and support. This is a vulnerable position.
- For the BlueGene target, I have not yet found a single precision target. The so called sloppy precision (single precision internal half fermions) has not so far passed the chroma unit test.

Building the BAGEL Wilson Dslash package is straightforward.

- The *CG-DWF Inverter* package from MIT is used to perform Domain Wall propagator inversions. This package provides 5 interfaces
 - Single precision SSE, Double precision SSE
 - Single precision BG/L, Double precision BG/L
 - AltiVec (which is solely single precision) .

All of the interfaces have the same sets of arguments and only the function names change to indicate the architecture. Chroma provides 5 wrappers, of which at most two are built at any one time (to wrap single and potentially double precision interfaces for a given architecture) These wrappers have a low overhead as they essentially only transform function and type names. These wrappers are then used by two classes (single and double precision - except for AltiVec where there is only a single precision) which adapt the interfaces to Chroma and provide the indexing functions that the level 3 routine needs.

Conversions to internal data formats are handled by indexing functions, which the client code must supply. This way the internal layouts are completely hidden from the client. However, a down side is that the indexing functions are called for every single number in the external fields. It is believed that the largest proportion of time in the indexing functions from function call overhead.

This level 3 package currently relies on the built in vector data-types offered by the GCC compiler and the fact that the GCC compiler allows arithmetic to be performed between these types. Intriguingly the Intel compiler offers the same vector types, but does not provide for arithmetic between them.³

On the BG/L platform a special version of the GCC compiler is needed which can take the code with the arithmetic on vector types, and produce instructions for the BG/L Double Hummer. This is a potentially annoying physical coupling, but fortunately, sources for the modified compiler are available online and if for any reason the compiler cannot be installed, the code can be still built using the level 2 inverter and a level 2 DWF operator (with BAGEL vector Dslash acceleration). Benchmarks indicate that this latter approach is not necessarily slower (and is in some cases faster) than the MIT DWF inverter on BG/L systems.

Finally, it should be noted that that as described in 4.3 this package uses an internal preconditioning that is different from the convention Chroma uses for 5D Dynamical Fermions. Hence Chroma cannot use this inverter in its force calculations and its usefulness is limited to propagator calculations at this time.

³Had this package been written with intrinsic functions instead of arithmetic operators it would port better between Intel and GCC compilers, but would become less portable between SSE, BG/L and AltiVec architectures - since the arithmetic instructions are common between these, whereas the intrinsic function names are not

6 Summary and Conclusions

We have examined level 3 software from the point of view of cohesion and coupling and illustrated how level 3 software is used by the chroma software system. Our chief points are the following:

- The higher the level of level 3 code we are attempting to write the more specialized it will be, and the smaller its market share (reusability potential) will be.
- This reusability potential can be increased by community agreement on conventions, and preconditionings.
- It is not clear, that it is in the interest of any one big code bases to move from a working production code to a newly standardized interface, especially if the change can have a large impact on its structure. The dangers of convergence include a potential for new bugs. At the same time there is little return on the investment of effort besides appearing to 'play nice' politically.
- As a question of principle, should SciDAC focus on writing many specialised pieces of code to accomodate different conventions and ensure that the 3 big codes run fast at the forthcoming leadership facilities, or should SciDAC urge a standard under the aegis of meeting a user demand, that none of the three big codes may end up using?
- Extraction of level 3 functionality from large frameworks is error prone and can bring with it excess baggage from the frameworks.
- Level 3 packages should have minimal couplings to the extent that this is possible.
- Cohesion can be increased by writing adapters (wrappers), however adapters alone cannot solve issues like different preconditionings.
- Adapters are typically specific to the client code and should therefore be the responsibility of the client to write.
- A combination of a framework written mostly in level 2, with judicious use of level 3 components may be equally effective as a monolithic level 3 package and may be more flexible.

Being pragmatic (or pessimistic) it is very difficult to see the entirety of USQCD reaching a concensus on conventions within the timeline of this project (after all they did not succeed in it during the last project.) This to my mind leaves the approach of overcoming integration and interface issues by writing adapters/wrappers. Since every package using the software may have a different internal set of interfaces each client package will probably need to write its own adapter (these in turn can be contributed to the package and distributed with it growing organically over time). What is most important is that the level 3 packages should be relatively uncoupled, have clean (preferably documented) interfaces for resource and lifetime management and for them to be fast.

7 Appendix: Personal Desiderata

To finalize and perhaps personalize this discussion, here are my personal desiderata for level 3 packages. I emphasize that these are rather personal tastes and should be taken separately from the preceding disucions.

- The packages should build using a (de facto) standard: `configure; make ; make install` build invocation. Customising compilers, linkers and flags should be straightforward in the style of GNU Autoconf. At the least the package should be provided in source form, so that such a system can be put on top of it.
- Compiling with and linking against the installed package should be straightforward, and may be time consuming.
- To the maximum extent possible, the packages should contain some kind of functioning testing framework.
- The packages should not pollute the namespace unnecessarily.

- Building the Level 3 packages should require the minimum of tools, and, as much as possible should work with versions of tools in common current use. (I should not need version 9.99.999 of arcane package *foo*)
- I should have the freedom, to extract, use, transform and redistribute with my own code parts of other SciDAC packages. I should not be bound by a restrictive use pattern license. I should be able to check in transformed code into a CVS repository of my own choosing and distribute it as part of my own framework as long as I make appropriate acknowledgements to the original authors, cite their publications, promulgate their copyright notices, or otherwise give due credit (ie really open source).
- The packages should be side effect free
- I am largely language agnostic, but prefer C++ first and then C, and finally assembler.
- It is OK for a given package to depend on other packages, as long as they are relatively few in number and satisfy the above desiderata for easy integration, redistribution etc.
- It should be straightforward for me to discover the conventions used and the data layouts the package requires, eg through documentation, comments or functioning tests. I can then write appropriate adapters.
- Just because a package exists I should not be forced to use it. Not using a given SciDAC level 3 should not count against me in any discussion of allocation. I extend the same courtesy to everyone else.