Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 1239–1255
The Journal of Systems and Software
www.elsevier.com/locate/jss

Fine-grain analysis of common coupling and its application to a Linux case study

Dror G. Feitelson [a,b,1], Tokunbo O.S. Adeshiyan [a], Daniel Balasubramanian [a], Yoav Etsion [b], Gabor Madl [a], Esteban P. Osses [a], Sameer Singh [a], Karlkim Suwanmongkol [a], Minhui Xie [a], Stephen R. Schach [a,*]

[a] Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37235, United States
[b] School of Computer Science and Engineering, The Hebrew University of Jerusalem, Israel

Received 18 October 2005; received in revised form 30 November 2006; accepted 19 December 2006
Available online 11 January 2007

Abstract

Common coupling (sharing global variables across modules) is widely accepted as a measure of software quality and maintainability; a low level of common coupling is necessary (but not sufficient) to ensure maintainability. But when the global variables in question are large multi-field data structures, one must decide whether to consider such data structures as single units, or examine each of their fields individually. We explore this issue by re-analyzing a case study based on the Linux operating system. We determine the common coupling at the level of granularity of the component fields of large, complex data structures, rather than at the level of the data structures themselves, as in previous work. We claim that this is the appropriate level of analysis based on how such data structures are used in practice, and also that such a study is required due to concern that coarse-grained analysis leads to false coupling. We find that, for this case study, the granularity does not have a decisive effect on the results. In particular, our results for coupling based on individual fields are similar in spirit to the results reported previously (by others) based on using complete data structures. In both cases, the coupling indicates that the system kernel is vulnerable to modifications in peripheral modules of the system.

Keywords: Common coupling; Data structures; Definition-use analysis; Fine-grain analysis; Kernel-based software; Linux case study

1. Introduction

The goal of software engineering is to produce high-quality maintainable software. But there is little agreement regarding how quality and maintainability should be measured, and whether they can be measured directly. Over the years, various indirect measures have therefore been proposed. The degree of coupling is one of them: Low levels of coupling are deemed necessary (but not sufficient) for high software quality and maintainability (Schach, 2007; Ince, 1988; Selby and Basili, 1991; Epping and Lott, 1994; Basili et al., 1996; Binkley and Schach, 1998; Briand et al., 1999; Ferneley, 2000; Rilling and Klemola, 2003).

"Common coupling" in particular refers to the use of global variables. It is generally agreed that common coupling induces an unacceptably high level of coupling. But what constitutes "a variable"? Programming languages allow the use of various constructs for organizing data: scalars, arrays, structures, and in some cases even more abstract types such as lists and hash tables. If a structure contains several scalars and two arrays, should the whole structure be considered as a single entity, or should its fields and subfields be considered independently?

The question of granularity is important because it affects the outcome of the evaluation of common coupling. Previous work using Linux as a case study has employed a coarse granularity, where large and complex structures are

* Corresponding author.
E-mail address: srs@vuse.vanderbilt.edu (S.R. Schach).
[1] Work done while on sabbatical leave at Vanderbilt University.

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.12.550

considered single entities in terms of coupling (Schach et al., 2002; Yu et al., 2004). We argue that a fine-grain approach may be more meaningful, especially if the fields are indeed functionally independent. In particular, it may be that considering a large structure as a single entity leads to "false common coupling," where some fields of the structure are used in one place and other fields in another place, but no fields are really shared among different modules.

To check whether this is indeed the case we have developed a procedure for analyzing fine-grain common coupling in a software product. To demonstrate how this procedure is applied in practice, we have re-evaluated the common coupling in the Linux system between kernel modules, and between kernel and non-kernel modules. In doing so, we determine whether the results of the study by Yu et al. (2004) may depend on the level of granularity adopted in that study when measuring common coupling in Linux.

Our work can be considered as an elaboration of the study by Yu et al., in considerably greater detail, and integrating operating system considerations along with software engineering issues. In particular, we claim that common coupling in Linux should indeed be evaluated at the level of the individual fields of large, complex data structures, rather than at the coarse-grain level of the compound data structures, as in Yu et al. (2004). The Linux case study thus also serves to evaluate the appropriate level of detail when determining common coupling in a large software product as a measure of software quality and maintainability.

The rest of this paper is structured in three main parts. Section 2 provides background on common coupling and reviews the categorization introduced by Yu et al. (2004). Section 3 explains the intricacies of analyzing common coupling when the global variables in question are complex data structures composed of many fields. Sections 4 and 5 then apply these concepts to the Linux kernel case study – the same case study as used by Yu et al. Section 6 presents our conclusions, both regarding the analysis of common coupling in general and regarding the specific case of the Linux kernel.

2. Common coupling

2.1. Metrics for open-source software

When measuring the quality of a software product, metrics based on the code itself have the advantage of being quantitative, objective, and amenable to mechanized evaluation. One such metric is the degree of coupling found in the code. Coupling between software modules measures the degree to which they are dependent on each other. One of the basic tenets of software engineering is that modules should have the lowest feasible level of coupling, because this enhances software quality and fosters maintainability (Schach, 2007; Offutt et al., 1993).

Coupling has been validated as a measure of development time and fault rate (Ferneley, 2000) and as a predictor of fault-proneness (Selby and Basili, 1991; Basili et al., 1996). Coupling has also been validated with respect to a number of aspects of maintenance, including comprehensibility (Rilling and Klemola, 2003), run-time failures and a variety of maintenance measures (Binkley and Schach, 1998), impact analysis (Briand et al., 1999), the cost of maintenance (Ince, 1988), and the cost of making changes (Epping and Lott, 1994).

Common coupling refers to the use of global (shared) variables, harking back to the COMMON keyword from FORTRAN. It is widely agreed that use of common coupling should be minimized, because of the high level of dependency induced between modules by this form of coupling (Schach, 2007; Yu et al., 2004).

2.2. Categorization of common coupling

Software is often built in layers. In many cases, there is a small, basic core of functionality, and on top of it a large loosely-knit set of tools. Examples are Emacs and Matlab: both have a stable, slowly evolving core, and many additional functions or packages, often created by users, that evolve quickly using the open-source paradigm (Raymond, 2000). In operating systems, the core is the *kernel*, and other modules include support for new functionality such as innovative file systems or new device drivers.[2] We term such software a *kernel-based system*.

The kernel is by definition the heart of the system. Everything depends on the kernel functioning properly. From a maintenance viewpoint, this means that it is highly desirable that the kernel be as independent as possible from other software modules. With this in mind, Yu et al. (2004) have defined five categories of common coupling, based on the roles that the global variables play.

Every occurrence of a variable in the code can be classified as either a definition or a use. A *definition* of a variable is the assignment of a new value to this variable. A *use* is the utilization of the current value of a variable. Yu et al. (2004) applied this classification to occurrences of global variables in the code, and then categorized the global variables as follows:

> *Category 1:* Global variables that are defined in kernel modules but not used in any kernel module. These can be interpreted as "kernel outputs"; in object-oriented terminology, they serve as "get" methods (accessors)

[2] We note that this terminology is not universal, and many would say that the file systems and drivers are in fact part of "the Linux kernel." Accordingly, we are actually referring to the core of the kernel, the part which is most important from a maintenance point of view because it is common to all builds and installations. Following Yu et al. (2004), we identify this as the code residing in the kernel subdirectory. Other possible definitions are considered in Section 5.3.

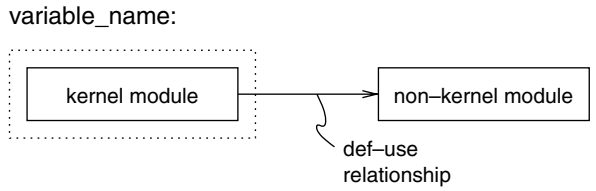

Fig. 1. Graphical notation to describe categories of common coupling.

for some internal kernel attribute. As such, their use is reasonable.

Category 2: Global variables that are defined in a single kernel module, and used in other kernel (and non-kernel) modules. Such a global variable can be interpreted as a "get" within the kernel in addition to being a "get" used by external modules. Again, this is reasonable.

Category 3: Global variables that are defined in several different kernel modules. This causes the different kernel modules to be dependent on each other, and is therefore an undesirable usage mode.

Category 4: Global variables that are defined in non-kernel modules and used in kernel modules. Although this creates a dependency of the kernel on non-kernel code, it may be necessary as an input mode; in other words, this is similar to a "set" method (mutator) of a kernel attribute. It therefore may be unavoidable.

Category 5: Global variables that are defined in both kernel and non-kernel modules, and used in kernel modules. This is an extreme form of coupling between kernel and non-kernel code, and is highly undesirable.

Yu et al. also introduced a graphical notation to describe the categories of common coupling. Fig. 1 shows a schematic example. The name of the global variable in question is noted at the top left. Modules are represented by rectangles. An arrow points from each module that contains a definition to each module that contains a use (regardless of whether these specific definitions can actually affect these specific uses). If two modules are connected by a two-headed arrow, then there are definitions and uses in both modules. A dashed or dotted line defines the kernel boundary: Modules that appear within it are kernel modules, and those that are on the outside are non-kernel modules.

3. Common coupling applied to structures

3.1. The two dimensions of global variables

The above discussion of common coupling implicitly assumes that global variables are independent monolithic entities. But in practice, computer programs are rife with complex data structures that include many different parts. In addition, they may have many distinct instances of each such data type. As a result, we find that global variables may be viewed in a two-dimensional space.

The first dimension involves the *components* of the data structure. When different modules access different components of the same data structure, this implies some sort of *syntactic* coupling between them. This is a distinct phenomenon from the common coupling discussed above, which occurs between modules that access the same global variable. Importantly, the two forms of coupling can interact, and this interaction can affect the results of the analysis.

Yu et al. (2004) considered each structure as a single, monolithic entity, thus allowing the syntactic coupling between fields to come into play. We claim that it may be more meaningful to decompose structures into their constituent fields, and treat each one independently. This reflects the fact that in many cases the fields are indeed independent, and different fields are used by different parts of the program. Treating the structure as a unit then leads to what we may call "false common coupling," as the different modules do not in fact access the same global variables – rather, they access *different* global variables that happen to be only syntactically related.

The second dimension involves *instances* of the data structure. A computer program typically creates many instances of each compound data type that it defines. We claim that it is proper to identify global variables that are distinct instances of the same structure (or rather, instances of the same fields within the same structure) with each other. The reason is that code is typically organized according to the data on which it operates. Accordingly, it is typical to find that accesses to a certain field are done from only a small number of functions. This is in contrast to the use of primitive data types – it would be ludicrous to suggest that all instances of, say, integer variables be identified with each other, as these variables are typically indeed independent and accessed by distinct code segments.

The important point is that the functions that handle a given complex data structure are called to handle *all* the different instances of the structure that may be instantiated at runtime. In particular, they may be called to handle the *same* instance. From a code maintenance point of view, this means that the functions may operate on the same data, and are therefore coupled to each other.

In short, we propose that instead of using the intuitive approach of regarding each instance of a structure as an independent global variable, one should decompose structures into their fields, and collapse the fields from different instances into a single entity (Fig. 2). The following two subsections elaborate on these concepts.

3.2. Decomposing structures into fields

Consider the following C declaration of a compound data structure:

```
struct struct_type_1 {
    int   f1;
    int   f2;
```

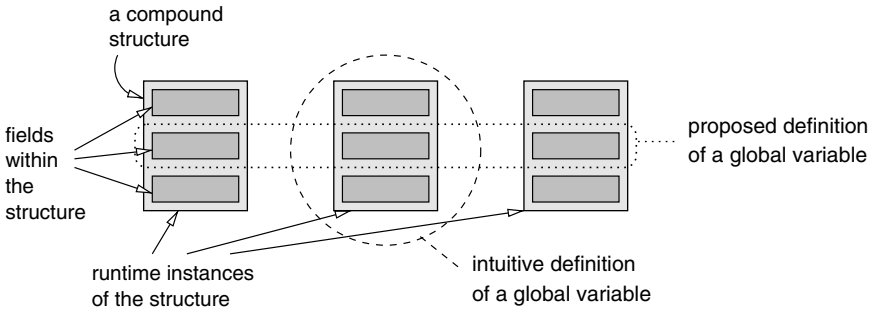

Fig. 2. The 2D space of compound global variables. Rather than defining a global variable as an instance of a structure, we decompose structures into their fields, and collapse fields across instances.

```
    int   f3;
}s;
```

That is, s is a variable of type struct_type_1, and has three integer fields.

Suppose now that fields s.f1 and s.f2 are functionally independent, that is, there is no relationship between the values of the two fields, and they are not used or defined in the same modules. Specifically, denote a module using s.f1 by m1, and a module using s.f2 by m2. Now consider the statement

```
s.f1 = 1;
```

in module m1, and the independent statement

```
s.f2 = 2;
```

in module m2. When performing coarse-grain definition-use analysis, these statements are both definitions of s, because variable s appears on the left-hand side of the assignment operator. As a result, we will find that modules m1 and m2 are coupled to each other. However, when performing fine-grain analysis, the first statement is a definition of only the field s.f1, and the second is a definition of only the field s.f2. As a result these statements do not testify to any coupling between module m1 and module m2, which remain decoupled when fine-grain analysis is performed.

The reason that this is important is that treating the whole structure as a single unit may create an impression of a high degree of coupling that is not really there. An example of how this may happen is given in Fig. 3. Assume that the fields of the structure declared above are accessed by kernel and non-kernel modules according to the pattern shown on the left of this figure: Field s.f1 is defined in kernel file 1 and used in non-kernel file 2; field s.f2 is defined in kernel file 3 and used in kernel file 1; and field s.f3 is defined in non-kernel file 4 and used in kernel file 1. Given such access patterns, each of the fields constitutes a well-behaved global variable: s.f1 belongs to category 1, s.f2 belongs to category 2, and s.f3 belongs to category 4. But if we look at the whole structure as a single entity (right of Fig. 3), we find that this pattern of accesses leads us to categorize s as a category-5 global variable.

The same considerations imply that structures should be decomposed even if they are nested at several levels. For example, consider a structure that includes another structure as one of its fields. We can then encounter statements such as

```
s.f1.sf2 = 3;
s.f1.sf4 = 5;
```

If subfields sf2 and sf4 are functionally independent in the structure s.f1, they should be treated as independent global variables. The same applies if a structure includes a pointer to another structure, and we find statements of the form

```
s.f2->sf6 = 7;
```

Here, subfield sf6 of field s.f2 (or rather, pointed to by field s.f2) should be treated as an independent global variable.

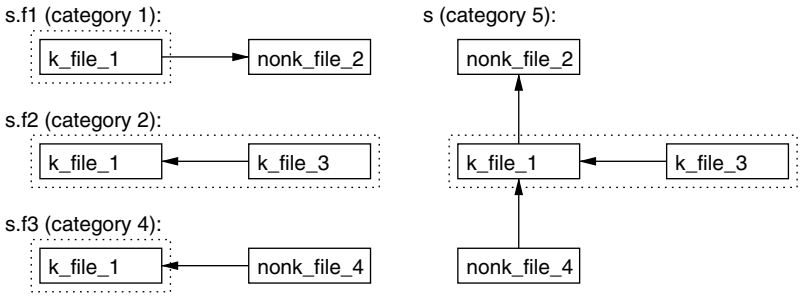

Fig. 3. Three fields of categories 1, 2, and 4, respectively (left) can make the whole structure look like a category-5 global variable (right). The dotted line denotes the kernel boundary.

In the sequel, when we talk of individual fields of a structure, we will typically mean all nested subfields as well.

3.3. Collapsing runtime instances of the same structure

Structures rarely appear only once in a program. It is much more common to have many instances of the same structure, organized as an array or linked to each other. This is analogous to the instantiation of multiple instances of an object in an object-oriented program.

Even in a non-object-oriented language like C, the code that handles such structures is typically generic. It can (and often does) handle any of the instances that are created at runtime. It is extremely uncommon to have distinct pieces of code handling distinct instances of the same data type. This motivates the notion that the different instances should be collapsed and treated as one for the purpose of analyzing common coupling.

For example, consider the following definition of a structure that can be used as an element of a linked list:

```
struct struct_type_2
{
   int    f1;
   int    f2;
   struct struct_type_2 *next;
}
```

A program might then include the following code segment, which traverses the list, using and modifying the data in it. This C notation assumes that head points to the head of the list, that the list is terminated by a pointer with value NULL, and that ptr is a pointer to type struct_type_2:

```
for(ptr = head; ptr != NULL; ptr = ptr->next){
   sum += ptr->f1;
   ptr->f1 /= sum;
}
```

The question is which global variables are being accessed. At runtime, many instances of type struct_type_2 may be created and linked to each other. But the code does not really discriminate among them; in effect, it treats the whole linked list as a single data structure, and plucks out a specific field from all the different instances. Another piece of code could do a similar computation using the field ptr->f2; this would be unrelated to the first computation, because it is using a separate field, even though it is traversing the same linked list.

Similar considerations apply to array global variables. The reason is that array cells are typically accessed in a dynamic manner, using other variables as an index. Accordingly, when performing fine-grain definition-use analysis, it would be wrong to treat array cells as independent. Instead, the cells should be collapsed and the whole array should be treated as a single global entity. This still holds even if each cell of the array is itself a structure.

Returning to the linked-list example, the above code segment actually refers to *two* fields of struct_type_2: the integer field f1 and the pointer field next. As a result, a similar code fragment using ptr->f2 instead of ptr->f1 would be coupled to this one by virtue of their shared use of ptr->next. However, this is a weak coupling because ptr->next is only used, and not defined. The code would have a stronger coupling with other code that actually defined ptr->next, which is appropriate, because such code really changes the structure of the linked list being traversed.

3.4. Handling pointers

When structures appear in arrays or linked lists, they are typically accessed via pointers (as shown above). Thus if pointer ptr points to an instance of struct_type_2, we might see a statement of the form

```
ptr->f1 = 1;
```

The question is precisely how to interpret this in terms of definitions and uses. The problem is that this simple statement involves no fewer than three variables: the pointer ptr, the structure s to which it points, and the field f1 within that structure.

When using coarse-grain analysis of complete structures, this assignment is an assignment to the structure s. However, if the pointer ptr is itself a global variable, it may be more convenient to use it as a representative of all the instances of struct_type_2 to which it might point. With this interpretation, we would say that the above statement is an assignment to ptr. In effect, this was the approach employed by Yu et al. in their analysis (Yu et al., 2004). Note, however, that the identification of s with ptr may be problematic, as the application may contain other means of accessing structures of type struct_type_2 apart from global pointer ptr. When basing the analysis on ptr, such additional references will be missed. This indeed happens in the Linux case study, as discussed in Section 4.4.

When using fine-grain analysis of fields, the picture is different. The above statement actually translates into two distinct accesses: First, there is a use of the (dereferenced) pointer ptr. Second, there is a definition of the integer field s.f1. In principle this makes it easier to analyze all uses and definitions of each field, independent of how they are referenced.

A somewhat subtle situation arises when a field is a pointer to the *same type* of data structure, as in the linked list example shown above. When such a field exists, we might find statements of the form

```
ptr->next->f1 = 2;
```

Based on our previous considerations of collapsing instances of accesses to the same field, this should also be interpreted as a definition of field s.f1, despite the extra

level of indirection. The indirection just adds a use of field s.next.

4. Common coupling and the Linux case study

4.1. Previous work

A case study in which common coupling has been investigated concerns the Linux kernel. Linux is a popular object of study due to its prominence and the availability of its source code for all versions since 1994 (Bowman et al., 1999; Tran et al., 2000; Schach et al., 2002; Albinet et al., 2004; Paulson et al., 2004; Yu et al., 2004). Regarding common coupling, it was shown that whereas the size of the Linux code base grows linearly with version number, the degree of common coupling grows exponentially (Schach et al., 2002). This agrees with an independent study of the architecture of Linux, which concludes that it has many more dependencies than it should (Bowman et al., 1999), and a study showing that open-source software is not necessarily more modular than closed source (Paulson et al., 2004).

A subsequent and more detailed study showed that not only is there significant common coupling, but that much of it is of the especially insidious category 5, causing vulnerability of the kernel to modifications in non-kernel modules (Yu et al., 2004). This is especially troubling in the context of an operating system, because it prevents the use of hardware support for protection. Practically all processors on the market have at least two protection levels: user and kernel. This allows the operating system to protect its data structures from being manipulated by user code. But many have more than two levels. For example, the popular Intel Pentium processors have four levels: a user mode, and three protected modes with increasing levels of protection (IA-32 Intel Architecture Software Developer's Manual, 2004). This is intended to be used to support layering within the operating system. For example, the kernel can use the most protected mode, and thus be shielded against faults that might be introduced by device drivers that run with a lower level of protection. But when global variables are used, all parts of the operating system must run at the same protection level, and the kernel data is left vulnerable. And indeed, Linux uses only a single protection mode for all operating system code.

An analysis of version 2.4.20 of the Linux operating system by Yu et al. (2004) has found 99 global variables, of which 4 are of the undesirable category 3, and no fewer than 20 are from the even worse category 5. In particular, the variable current stood out as especially problematic; it was defined and used by 12 kernel modules, used by an additional 6 kernel modules, and also defined and/or used by an astounding 1071 non-kernel modules.[3] These figures made a dominant contribution to the finding that 62–63% of all occurrences of global variables in Linux are category 5.

Fig. 4 depicts the definitions and uses of current according to Yu et al. (2004). module_name (n, m) denotes that the module in question contains n definitions and m uses of global variable current. The dashed lines separate the 12 kernel modules with both definitions and uses of current from the 6 kernel modules with just uses.

The methodology used to derive these results was as follows. First, all lines in the Linux source code in which current occurs were extracted. These were classified as definitions if current appeared to the left of an assignment operator, and uses otherwise, that is, coarse-grain definition-use analysis was performed. Then definitions and uses of current in all the modules were counted. The 26 files in the kernel subdirectory were identified as being the kernel modules, and all the rest as non-kernel (Linux 2.4.20 has about 11,000 files).

4.2. A closer look at current

In our case study we take a closer look at current. In particular, we incorporate operating systems knowledge into the analysis, and make the following observation: current is not a simple global variable. In fact, it has two independent roles. First, it serves to identify the currently running process. Second, it is a pointer to a structure containing many fields used to describe this process.

In Linux, as in other variants of Unix, data about each process are maintained in a *process descriptor*. In Linux, this is a structure called task_struct. In some versions of Unix, the kernel contains a hardcoded table of such structures. However, this limits the number of processes that can be created. In Linux, task structures are allocated dynamically together with the kernel stacks (Bovet and Cesati, 2001). Each process has a unique area in memory, 8 KB in size, that contains its task structure and its kernel stack. The address of this memory block is used by the kernel to identify this process, in place of the conventionally used process identifier, or pid (but a pid is still maintained for use in the programming API, e.g., the fork and signal system calls).

Because the kernel most often deals with the currently running process, the address of the memory block describing this process is made available using current. For efficiency reasons this is not a normal variable in memory, but rather a macro that returns the contents of a specific register (Bovet and Cesati, 2001). As a further optimization, Linux does not waste a general-purpose register for this; instead, it masks the low-order 13 bits of the stack pointer. This works because, when kernel code is running, the stack pointer points into the kernel stack of the currently running process, which resides in the same 8 KB memory block as the process descriptor. Thus current (the pointer) is actually never explicitly defined! Instead, it is implicitly defined when the stack pointer is defined

[3] In this study and in ours modules are taken as equivalent to C source files.

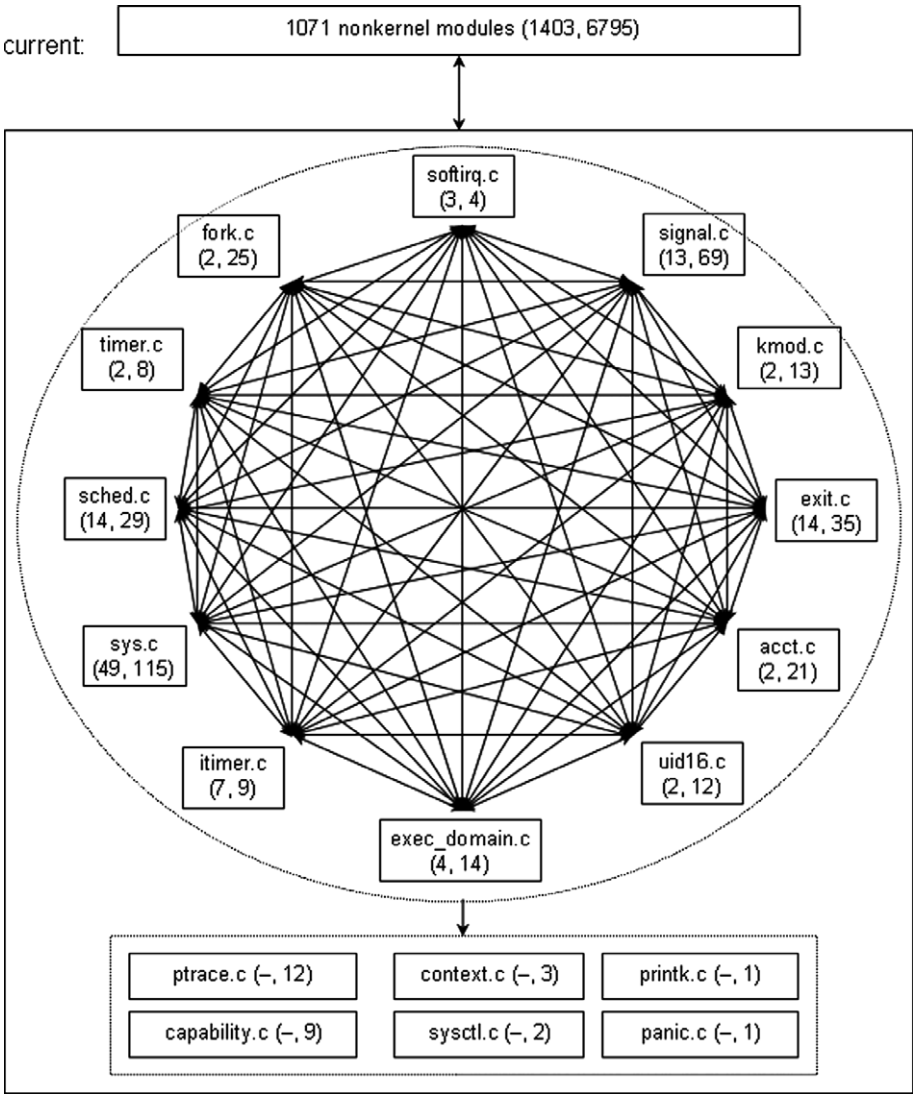

Fig. 4. Definitions and uses of current in Linux version 2.4.20, from Yu et al. (2004).

as part of a context switch (in the switch_to macro) (Bovet and Cesati, 2001). So, from the viewpoint of fine-grained definition–use analysis, current itself is not category 5, but rather category 2; it is defined in one place, and used extensively both in the kernel and in other modules.

Lines of source code in which current appears to the left of an assignment are not definitions of current. Rather, they are definitions of *fields of* the process descriptor to which current points. The original study by Yu et al. implicitly considered the whole process descriptor as a single entity, so a definition of any field was considered to be a definition of the process descriptor. But an alternative approach is to consider the fields individually, as we do in this paper. This is motivated by the fact that the process descriptor is actually a somewhat disorganized assembly of different pieces of data, used for different purposes. In principle, it may be that each of these fields is actually well behaved, and belongs to categories 1, 2, or 4 (as explained in Section 3.2). This would imply that common coupling in the Linux system is not as problematic as suggested by the results of Yu et al. (2004).

4.3. The fields of task_struct

The process descriptor structure in Linux is rather complex. Some of its fields are scalars. Others are structures, pointers to structures, or arrays. The question is if and when to fragment the structure into its constituent scalars. We decided to perform such fragmentation, following the approach outlined in Section 3.2.

Based on the arguments presented above, independent scalar fields should clearly be considered as distinct global variables. An example is current–>pid, the process identifier used in the programming API. However, there are cases where several such fields are actually related. For example, there are several different fields that express nuances of user identification for the purpose of granting permissions: current–>uid, current–>euid, current–>suid,

current–>fsuid, and similarly for groups. In principle these could have been grouped into a "uid" structure instead of cluttering task_struct, but they were not. The problem with grouping them when performing the definition-use analysis is that it requires an understanding of the semantics of each field, and the relationships between them; furthermore, it may be somewhat subjective. We therefore decided to take the more formal approach, and treat all scalar fields as distinct global variables, even where there seemed to be some relationships between them.

This then leads to the notion that fields that actually *are* structures should also be decomposed, and *their* fields should also be regarded as distinct global variables. Moreover, this should also apply to fields in structures that are pointed to by fields of current, rather than being part of the task_struct structure directly. This can go on for several levels.

The only case where pointers to structures were not followed and fragmented was when they point to other instances of task_struct (see Section 3.4). There are quite a few such pointers, used for two functions: keeping track of the family relations among processes (pointers to the parent, first child, and sibling processes), and maintaining lists of processes (such as the runqueue or processes waiting for an event). Subfields accessed via such pointers were identified with the fields of current itself.

Arrays, as distinct from structures, were not decomposed, but were treated as a single global entity. The reason is that array cells are typically accessed in a dynamic manner, using other variables as an index. In other words, array cells are more similar to instances of a data structure than to independent data structures. Accordingly, when performing fine-grain definition-use analysis, it would be wrong to treat array cells as independent. Instead, whenever a field is an array, the whole array should be collapsed and treated as a single global entity. This still holds even if each cell of the array is itself a structure – which is decomposed into its subfields, as described above.

4.4. Miscategorization caused by aliasing

Another issue that has surfaced is aliasing, which may be considered as a variant of clandestine common coupling (Schach et al., 2003). As before, let ptr be a pointer to a variable of type struct_type. Assume further that ptr itself is a global variable (like current).The statement

```
newptr = ptr;
```

creates an *alias* for ptr. Either one of them can now be used to access the fields of the structure to which ptr is pointing.

In particular, when the alias is used to define and use fields or subfields of variables of type struct_type, it becomes harder to detect instances of common coupling. These fields and subfields are global variables, but can now be accessed using different names!

In the original analysis of Linux by Yu et al., aliases of global variables were ignored. Consequently, none of the accesses made using aliases were considered, so there are potentially many more definitions and uses that were not identified (Orso et al., 2001). This is problematic because such missing information can lead to misclassification of global variables.

A specific example we found in Linux is current–>state, which we originally categorized as a category-1 field, because it is only defined but not used in the kernel (it is also defined in non-kernel code, but the categorization allows this as it focuses on the kernel dependencies). But in reality it *is* used in the kernel: the scheduler creates an alias of current called prev in anticipation of switching to a new process, and then uses the value of prev–>state in a switch statement. Therefore field state should actually be category 5, as it is indeed categorized after taking aliases into account.

Aliasing also partially accounts for the large number of fields that have only a single occurrence in the whole system, or seem to never be defined. They actually have more occurrences, but those are achieved using aliases rather than using current. For example, the field current–> did_exec appears only once in the whole system, where it is defined, but seems never to be used. But in fact it *is* used in the form p–>did_exec, after p is aliased to current.

Focusing on the use of current in Linux, we find that there is an additional special case related to aliasing. When a new process is created, its task_struct is initialized as part of the fork system call. At this time current is still pointing to the parent process. Thus, many fields of the new process seem never to be defined, because these definitions happen before current is made to point at this instance of a process descriptor. Likewise, some definitions and uses are performed by routines that loop over all processes, regardless of which process is the currently running one. We did not count such accesses, because our analysis was specifically based on those source-code statements that involve current itself. Therefore our results may be conservative, because additional definitions and uses may involve additional modules, and propel the affected fields to higher (and worse) coupling categories.

A detailed description of the effect of aliasing has been written in a separate paper (Schach et al., in press). The results reported here include all occurrences of current, including those using direct aliases. However, they do not include possible accesses to fields that were passed by reference to other functions, thereby effectively creating additional indirect aliases.

4.5. Classifying operations as definitions and uses

When performing fine-grain definition-use analysis, it is too simplistic to categorize occurrences of global variables as only simple definitions or simple uses. Real complex systems use various language constructs which imply additional categories. The following is a list of the C-based categories that we found in Linux (all the examples are actual code from the Linux 2.4.20 kernel).

Simple definition: This is a simple assignment, such as to field processor in

```
current->processor = 0;
```

Simple use: Similarly, this is a straightforward utilization of the current value of a variable. Two examples are the following statements using the field current->pid:

```
q->info.si_pid = current->pid;
if(current->pid != 1){...}
```

Combined definition and use: This is using the shorthand available in C, as in

```
current->link_count++;
current->flags |= PF_SIGNALED;
```

For the purpose of categorizing common coupling, such statements need to be counted twice, both as a definition and as a use. But when counting the number of occurrences of a global variable, they are counted only once.**Atomic operation:** Linux supports an atomic combination of definition and use, as in

```
atomic_inc(&current->files->count);
```

In addition, lock and unlock operations are actually atomic operations that both use and potentially modify their parameter, as in

```
read_lock(&current->fs->lock);
```

Such atomic operations were therefore also counted twice, as above.
Passing by value: This is equivalent to a use, because only the variable's value is passed to the function.
A special case occurs if the field in question is itself a pointer. In this case, passing the pointer by value is equivalent to passing the pointed-to structure by reference. From a maintenance viewpoint, the possibility of defining elements of the structure therefore dictates that this be classified as passing by reference and not passing by value.
Passing by reference: In this case, a variable may be modified by the called function, so this is potentially both a definition and a use.
A special case occurs when global variable current is passed as a parameter to a function. This seems strange, as current is global anyway. The answer is that current doubles as an identifier for the currently running process, and as such is sometimes passed to functions that accept a process descriptor as an argument, for example

```
send_sig(SIGKILL, current, 0);
```

Such cases are counted as a passing by value despite the fact that current is also a pointer to the whole task_struct structure.
Pointer dereference: Each time a pointer is dereferenced its value is actually used. Therefore statements such as

```
current->fs->altrootmnt = mnt;
```

actually represent two uses (of current itself and of the field fs) and a definition (of the subfield altrootmnt).
Execution: This is using the facility to define a variable that points to a function, and then calling that function. For example:

```
current->exec_domain->handler(segment, regp);
```

We interpret this as using the value of the variable.
Sizeof: A special case is sizeof, which looks like a function call but behaves more like an operator, for example:

```
charcorename[6 + sizeof(current->comm) + 10];
```

However, as this is resolved at compile time and uses only the type of the variable in question, we ignore such instances in our definition-use analysis.

5. Results of the Linux re-categorization case study

To assess the impact of the considerations described in the previous sections, we performed a complete re-categorization of common coupling as related to current in Linux.

5.1. Technicalities

The version used was Linux 2.4.20, as in the study by Yu et al. (2004), so that our fine-grain results could be compared to those of the earlier coarse-grained study.

The analysis started with all source code lines that include the identifier current. These include both .c and .h files. Note that .h files are considered as independent modules. Therefore, macros defined in .h files count as definitions and uses in that file, and not in the .c files that include the .h file.

The kernel modules were identified as the 26 .c files in the kernel directory, again as done in the study by Yu et al. Files in the arch/*/kernel directories were *not* considered to be part of the kernel. The reason for this decision is that, from a maintenance point of view, we define the kernel as the heart of the system that is crucial for any installation. Accordingly, architecture-specific aspects of the kernel are excluded. Alternative definitions of the kernel are discussed in Section 5.3.

Each occurrence of the fields of current was classified manually into one of the definition-use classes of Section 4.5. This was then checked by another person and any differences were reconciled. These classifications were then automatically analyzed by Perl scripts to categorize the fields into the five categories of Section 2.2.

We did not consider the occurrences of global variables in assembler code, but rather set them aside until we have done the necessary research into the nature of common coupling between a second-generation language (assembler) and a third-generation language (C). The only consequence of our not analyzing the assembler code is that we may have slightly undercounted the number of occurrences

of common coupling (current appears in assembly instructions only 31 times).

5.2. Results

In the remainder of this paper, the term "fields" also includes all subfields of current, as explained in Section 3.2. Categorizing the subfields of current according to the procedure outlined above leads to the results shown in Table 1. The first obvious result is that a refinement of the original categorization is needed. As indicated in the table, we have added another category, namely,

> *Category 0:* Global variables that are neither defined nor used in the kernel.

because this seems to be the case for many subfields of current. In addition to the 247 fields in Table 1, there were no fewer than 80 fields for which we identified uses but did not identify any definitions. Of these, 17 were used in the kernel, and the rest were not. Some of these fields received values during initialization in the fork routine; a prime example is current–>pid, which is used 22 times in the kernel and 765 times in non-kernel modules. In fact, the whole of task_struct is initialized in fork by simply copying the parent structure; in addition, many fields are initialized individually, including 15 of the 80 fields cited above. Other fields most probably received values from some other code that did not involve current directly or through an alias. This includes various actions that are typically unrelated to the currently running process, such as handling I/O operations (for file access or memory management), sending signals, and making scheduling decisions.

Using the extended categorization, we see that some 61% of the fields are in category 0, that is, not accessed by the kernel. This high percentage may mean that we are still missing additional definitions or uses that are not implemented using current or its aliases; indeed, a simple check based on analyzing the initialization in the fork system call reveals that including those definitions reduces the fraction of category 0 fields to 56%. It may also mean that our definition of "kernel" is lacking – an issue that we address again below in Section 5.3. On the other hand, the vast majority (109 of 152, or 72%) of these fields are actually subfields of the thread_struct structure that is embedded in task_struct. This structure is used to encapsulate architecture-specific state of the processor, and therefore is typically used by architecture-specific code, and not by kernel code. It is therefore reasonable to disregard these fields when discussing the coupling of the kernel to non-kernel modules.

Ignoring all the category 0 fields, we find that the majority of the other fields (53 of 95, or 56%) belong to the problematic category 5. This is much higher than the results obtained by Yu et al., who found that only 20% of the global variables were in category 5. Note, however, that these results are not directly comparable, because we are counting fields of current whereas Yu et al. were counting independent global variables (of which current was one).

In an effort to understand the significance of the above results, we note that some of the fields are locks or counters that are used atomically. These fields are explicitly designed to be accessed and modified by multiple modules, and their usage reflects this. Could it be that they are the source of the many category 5 fields? Upon inspection, it was found that only 15 fields are of this type, and only 8 of them were category 5, so the above results are not largely influenced by them.

Table 2 shows the breakdown of occurrences of fields of current of different categories in kernel and non-kernel modules. Here, occurrences that are both a definition and a use are counted only once. The results are that accesses to category-5 fields dominate the use of current's fields in both the kernel and non-kernel code. In the kernel, the second most common type of access is to a category-2 field. In non-kernel code, the second most common is category 0. Note that there are nearly 3 times as many different category-0 fields as category-5 fields, but together they occur just over a third as many times as category-5 fields.

5.3. What is the kernel?

The above results are based on the definition used by Yu et al. (2004), where the kernel is defined to be the kernel subdirectory of the Linux distribution. But code from other subdirectories is often also considered part of the Linux "core kernel." So we need a definition that specifically identifies the core kernel, that is, those parts of the kernel that are the most fundamental and used in all installations. We have come up with three possible alternatives for such a definition.

Table 1
Results of categorizing fields of current

Category 0	152	(61.5%)
Category 1	5	(2.0%)
Category 2	27	(10.9%)
Category 3	3	(1.2%)
Category 4	7	(2.8%)
Category 5	53	(21.5%)
Total	247	(100%)

Table 2
Results of analyzing individual occurrences of fields of current

	Kernel	Non-kernel
Category 0	0 (0.0%)	1863 (25.0%)
Category 1	6 (1.0%)	18 (0.2%)
Category 2	96 (15.2%)	197 (2.6%)
Category 3	13 (2.1%)	10 (0.1%)
Category 4	16 (2.5%)	166 (2.2%)
Category 5	499 (79.2%)	5208 (69.8%)
Total	630 (100%)	7462 (100%)

The first alternative is to use the distribution makefiles to find those modules that are always compiled, in all kernel configurations. This led to a set of 52 files (in addition to the 26 in kernel), which are listed in the Appendix. Some judgment has been applied in setting up this list, for example, modules related to networking were excluded as a system could in principle be stand-alone. On the other hand many file system modules have been included, because the file system serves as the main abstraction for naming and access to all hardware devices, and not only as the implementation of the file abstraction.

Another alternative is architecture-based, and includes all the files compiled for the simplest possible Intel-based i386 platform. To find this list of files, we simply created such a kernel build, and extracted the list of files that were used. The selected configuration was typical of a modern desktop, including Ethernet and USB. The list ended up containing 342 source files, and an additional 494 header files. The source files are also listed in the Appendix.

The third alternative is much simpler, and is based on exclusion rather than inclusion. It considers all the code to be the kernel except for two obvious subdirectories: arch, which contains architecture-specific code, and drivers, which contains device drivers.

Note that except in the i386 version .h files are not included in the kernel. Firstly, these files reside in the include directory, not in the kernel directory. Moreover, the same .h file may be included by multiple files, some of which are in the kernel while others are not. Therefore there is no straightforward way to associate .h files with the kernel.

The results of classifying the fields of current using these three alternatives are shown in Table 3, and compared with the original definition used by Yu et al. (2004). One obvious result is that as we include more files in our definition of the kernel, fewer fields are classified as belonging to category 0. These fields migrate to the other categories, mainly to categories 1, 2, and 3. However, the overall picture does not change very much, and the largest non-category-0 category by far is always category 5. The fraction of non-category-0 fields that are classified in the "bad" categories of 3 and 5 is also relatively stable, and stays in the range of 52–59%.

When looking at the fraction of occurrences of each category (Table 4), we again see a similar picture for all four definitions of the kernel. Between 63 and 79% of the occurrences are of category-5 fields. When using alternatives that define a larger kernel, the main growth occurs in occurrences of category-3 fields. In the extreme case, namely, defining the kernel as all subdirectories except the arch and drivers subdirectories, the fraction of occurrences of category-3 fields is an order of magnitude larger than for Yu et al.'s original definition of the kernel, and twice as large as the fraction of occurrences of category-2 fields. This indicates that a large part of the problem is indeed coupling between modules in these two subdirectories and the other subdirectories.

Table 3
Categorization of fields of current using different definitions of what constitutes a kernel

	Yu	Makefile	i386	Exclude
Category 0	152 (61.5%)	134 (54.3%)	98 (39.7%)	93 (37.7%)
Category 1	5 (2.0%)	7 (2.8%)	16 (6.5%)	17 (6.9%)
Category 2	27 (10.9%)	37 (15.0%)	52 (21.1%)	44 (17.8%)
Category 3	3 (1.2%)	17 (6.9%)	20 (8.1%)	26 (10.5%)
Category 4	7 (2.8%)	6 (2.4%)	3 (1.2%)	6 (2.4%)
Category 5	53 (21.5%)	46 (18.6%)	58 (23.5%)	61 (24.7%)

5.4. Threats to the validity of the Linux case study

A major threat to the validity of the results of the Linux case study reported above is that they are based on a lexical analysis of the code, rooted at uses of current. This does not allow for a full and precise identification of all data flows from one module to another. In particular, we do not follow the passing of current and its fields by value, or accesses using pre-processor macros. Moreover, we completely miss those fields and subfields of task_struct that are simply not accessed via current at all, and ignore those that are used but not defined. A semantic analysis using a compiler front-end is needed to correct these omissions. We are considering performing such an analysis, and comparing the results, to assess the severity of this methodological issue. Such an analysis could identify new fields in all the different categories, leading to changes in the observed distribution of fields in the different categories. However, regarding the fields that we have already analyzed, such future analysis can only *increase* the coupling between modules. Therefore, our results may be

Table 4
Results of analyzing individual occurrences of fields of current using different definitions of what constitutes a kernel

	Yu		Makefile		i386		Exclude	
	Kernel (%)	Non-k (%)	Kernel (%)	Non-k (%)	Kernel (%)	Non-k (%)	Kernel (%)	Non-k (%)
Category 0	0.0	25.0	0.0	21.8	0.0	17.4	0.0	17.6
Category 1	1.0	0.2	0.7	0.3	1.2	0.9	1.3	0.6
Category 2	15.2	2.6	14.8	3.2	15.1	3.0	11.8	2.8
Category 3	2.1	0.1	14.8	1.4	15.5	1.9	23.1	0.6
Category 4	2.5	2.2	0.7	3.9	0.5	4.3	0.9	1.9
Category 5	79.2	69.8	69.0	69.5	67.7	72.5	62.9	76.5
Total	630	7462	1206	6886	1821	6271	2420	5672

somewhat conservative, and the actual degree of coupling may be even higher, possibly even leading to a re-categorization of certain fields into worse categories.

Another threat to the validity of these results is that they obviously depend on the definition of what constitutes the Linux kernel. We compared four definitions, two based on the subdirectory structure and two based on the kernel makefiles. All four led to qualitatively similar results. But it might be that some refactoring can significantly reduce the coupling among any of these definitions of the kernel and other modules (Tran et al., 2000).

A third potential threat to the validity of the results of the Linux case study is that they pertain to only the Linux system. It may be that such a complex system, written in a non-object-oriented language, simply requires such patterns of global variables to be used. It may be that the resulting code is hard to maintain, but that by itself does not necessarily imply a low quality.

To achieve an absolute scale, comparison with other similar software systems is required. Yu et al. have in fact conducted such a study, comparing Linux to several versions of the BSD Unix operating system. This seems an appropriate comparison because the basic functionality of Linux and BSD are similar, but the BSD line has had a much more disciplined development history. The results of the study show that BSD has an order of magnitude less common coupling than Linux: 900–1600 occurrences of global variables vs. about 15,000 (Yu et al., 2006). Although striking, it should be remembered that the functionality of the different systems is not identical: Linux contains support for many more platforms and hardware devices, and therefore has many more device drivers, which may inflate the coupling numbers.

5.5. Discussion of the case study

The result of our re-categorization of global variables with regard to current in Linux is to uphold the concern raised by Yu et al. regarding the large number of category-5 global variables in Linux. We have shown that, even when global data structures are reduced to their constituent fields, there are many individual fields that are category 5, and, moreover, they receive a disproportionally large fraction of the accesses. As such, they lead to vulnerabilities of the kernel and dependencies on non-kernel code. In particular, we found a strong coupling with the arch and drivers subdirectories, indicating that

1. The kernel is exposed to manipulation by peripheral code such as device drivers and
2. There is a lack of a well-defined interface between the generic part of the system and the architecture-specific parts.

Examples that this coupling is a real problem include the following. The field current–>counter is the main mechanism by which the scheduler assigns priority and keeps track of CPU usage by a process. It should therefore be defined by only the scheduler and the timer. In actuality, it is also defined by two other kernel modules, by a host of architecture-specific kernel extensions (to reduce priority), and by several device drivers, presumably also to reduce priority. In fact, the scheduler and timer access current–>counter only via an alias.

The field current–>session is used by the system to keep track of session information and process relationships, so it should also be defined by only the kernel. However, we found that it is also directly set once in file system code, and once in a device driver (and indirectly, via an alias, another few times). In fact, there were 19 fields that are defined only once or twice in non-kernel modules, while being used up to 64 times. Eliminating these definitions would change their classification from category 5 to category 2.

A worrying example is that several uid-related fields are also category 5 (the uid is the user identifier, and is used to control access to private information). They are naturally set in the kernel, but are manipulated also by file-system code. For example, current–>fsuid is temporarily set to be 0 (the root user ID) in one place so as to obtain a privileged port. In principle, any new module may set these variables and introduce serious security risks.

These examples show that common coupling is more than a software engineering issue related to maintainability. As stated earlier in this section, common coupling also reflects a real vulnerability of the kernel, the most crucial code at the heart of the system, to manipulation by peripheral code-like device drivers, which are known to be fault-prone and are relatively unregulated (Chou et al., 2001). This demonstrates the problems that stem from a monolithic design based on direct access to global variables. A much safer design would be to use layers based on hardware protection, where peripheral code can call only functions exported by more protected code. For example, drivers should not manipulate scheduler data – they should call a function to request a reduction in priority. And there should simply be no interface that allows peripheral code to modify the uid. Such design ideas are not at all innovative, and have been known since the very first multi-tasking operating systems projects (Dijkstra, 1968; Schroeder et al., 1977).

6. Summary, conclusions, and future work

The degree of common coupling found among software modules may depend on the granularity of the analysis – whether we are considering the sharing of complete data structures or the sharing of their constituent fields. In particular, coarse-grained analysis may cause false coupling, where the syntactic coupling of fields in the same structure leads to apparent coupling between modules that in actuality use only distinct fields.

In fine-grain analysis we focus on the fields. We claim that this can potentially lead to a more accurate characterization of sharing patterns than using complete data structures. We further claim that the way to perform the fine-grained anal-

ysis is to collapse runtime instances of global data structures that have the same type. Our main contribution is the development of a technique for fine-grain analysis of common coupling in kernel-based software. We have shown by means of a case study that the technique can be used in practice.

Previous work on the common coupling in the Linux system used a coarse granularity, and found significant coupling between the kernel and more peripheral software modules. We have repeated this work using our fine-grain technique at the level of individual fields. The main result found is that significant coupling exists at this level as well, thereby alleviating the concern that those previous results were based on false coupling. At the same time, our results augment the concern regarding the long-term maintainability of Linux.

In future work we plan to extend our study in three directions. First, there is much more to learn about coupling in general. The most prominent example is to follow the effect of passing global variables by reference from one module to another. This creates a form of alias that we did not consider in the present study, and might increase the coupling significantly. Likewise, the use of macros to access global variables needs to be taken into account.

Second, we wish to further improve our understanding of coupling in the specific case of the Linux system. One aspect of this is to actually tabulate coupling that arises from passing by reference, as suggested above. Another is to extend the analysis to all references to task_struct (not just those made using current), and furthermore, to all other shared data structures. A third is to make more detailed comparisons with other systems, such as BSD Unix. This will answer the question of whether our results for fine-grain coupling are general or unique to the Linux system.

Third, a different line of research is to investigate alternatives to the massive use of global variables, in the interest of making operating systems more robust, and reducing the potential vulnerabilities to peripheral code. This harks back to studies of kernel structure in the Multics system (Schroeder et al., 1977), and can also exploit studies of the architecture and inter-dependencies of Linux (Bowman et al., 1999; Tran et al., 2000). It is obviously related to the software engineering study of partitioning (and re-partitioning) systems into modules (Parnas, 1972; Schwanke, 1991).

Appendix. Selection of kernel modules

In addition to defining the kernel according to the subdirectory structure of the code, we considered two other ways to identify the core modules based on the kernel's makefiles.

One alternative definition of the "kernel" is based on files that are compiled in all kernel configurations. Using this consideration, we suggest that the kernel comprises the following files (this is called the "makefile" version):

directory	files			
init/	do_mounts.c	main.c	version.c	
fs/	open.c	read_write.c	stat.c	fcntl.c
	ioctl.c	readdir.c	exec.c	devices.c
	block_dev.c	char_dev.c	namespace.c	namei.c
	dcache.c	inode.c	bad_inode.c	super.c
	attr.c	file.c	file_table.c	buffer.c
	iobuf.c			
ipc/	util.c			
kernel/	acct.c	capability.c	context.c	dma.c
	exec_domain.c	exit.c	fork.c	info.c
	itimer.c	kmod.c	ksyms.c	module.c
	panic.c	pm.c	printk.c	ptrace.c
	resource.c	sched.c	signal.c	softirq.c
	sys.c	sysctl.c	time.c	timer.c
	uid16.c	user.c		
lib/	errno.c	ctype.c	string.c	vsprintf.c
	brlock.c	cmdline.c	bust_spinlocks.c	rbtree.c
	dump_stack.c			
mm/	bootmem.c	filemap.c	memory.c	mlock.c
	mmap.c	mprotect.c	mremap.c	numa.c
	oom_kill.c	page_alloc.c	page_io.c	shmem.c
	slab.c	swap.c	swap_state.c	swapfile.c
	vmalloc.c	vmscan.c		

Another alternative is based on an actual compilation of a basic configuration suitable for an Intel-based desktop. Doing this led to the following list of files (called the "i386" version):

directory	files			
init/	do_mounts.c	main.c	version.c	
drivers/block/	ll_rw_blk.c floppy.c	blkpg.c	genhd.c	elevator.c
drivers/cdrom/	cdrom.c			
drivers/char/	mem.c raw.c vt.c console.c defkeymap.c	tty_io.c pty.c vc_screen.c selection.c pc_keyb.c	n_tty.c misc.c consolemap.c serial.c sysrq.c	tty_ioctl.c random.c consolemap_deftbl.c keyboard.c
drivers/ide/	ide.c ide-adma.c rz1000.c ide-disk.c	ide-features.c ide-dma.c ide-proc.c ide-cd.c	ide-taskfile.c ide-pci.c ide-probe.c	cmd640.c piix.c ide-geometry.c
drivers/net/e1000/	e1000_main.c e1000_proc.c	e1000_hw.c	e1000_ethtool.c	e1000_param.c
drivers/net/	eepro100.c net_init.c	mii.c loopback.c	Space.c auto_irq.c	setup.c
drivers/pci/	pci.c proc.c	quirks.c setup-res.c	compat.c	names.c
drivers/scsi/	scsi.c scsicam.c scsi_queue.c scsi_scan.c	hosts.c scsi_proc.c scsi_lib.c scsi_syms.c	scsi_ioctl.c scsi_error.c scsi_merge.c	constants.c scsi_obsolete.c scsi_dma.c
drivers/sound/	sound_core.c es1371.c	sound_firmware.c	i810_audio.c	ac97_codec.c
drivers/usb/storage/	scsiglue.c initializers.c	protocol.c	transport.c	usb.c
drivers/usb/	usb.c uhci.c	usb-debug.c	hub.c	hcd.c
drivers/video/	vgacon.c			
fs/	open.c ioctl.c select.c fifo.c dcache.c attr.c iobuf.c noquot.c	read_write.c readdir.c devices.c pipe.c inode.c file.c dnotify.c binfmt_aout.c	stat.c exec.c block_dev.c namespace.c bad_inode.c file_table.c filesystems.c binfmt_script.c	fcntl.c locks.c char_dev.c namei.c super.c buffer.c seq_file.c
fs/devpts/	root.c	inode.c		
fs/ext2/	balloc.c fsync.c namei.c	bitmap.c ialloc.c super.c	dir.c inode.c symlink.c	file.c ioctl.c

Appendix (*continued*)

directory	files			
fs/isofs/	namei.c	inode.c	dir.c	util.c
	rock.c			
fs/lockd/	clntlock.c	clntproc.c	host.c	svc.c
	svclock.c	svcshare.c	svcproc.c	svcsubs.c
	mon.c	xdr.c	lockd_syms.c	
fs/nfs/	dir.c	file.c	flushd.c	inode.c
	nfs2xdr.c	pagelist.c	proc.c	read.c
	symlink.c	unlink.c	write.c	nfsroot.c
	mount_clnt.c			
fs/partitions/	check.c	msdos.c		
fs/proc/	inode.c	root.c	base.c	generic.c
	array.c	kmsg.c	proc_tty.c	proc_misc.c
	kcore.c			
fs/ramfs/	inode.c			
ipc/	util.c	msg.c	sem.c	shm.c
kernel/	acct.c	capability.c	context.c	dma.c
	exec_domain.c	exit.c	fork.c	info.c
	itimer.c	kmod.c	module.c	panic.c
	printk.c	ptrace.c	resource.c	sched.c
	signal.c	softirq.c	sys.c	sysctl.c
	time.c	timer.c	uid16.c	user.c
lib/	errno.c	ctype.c	string.c	vsprintf.c
	brlock.c	cmdline.c	bust_spinlocks.c	rbtree.c
	dump_stack.c		rwsem-spinlock.c	dec_and_lock.c
mm/	memory.c	mmap.c	filemap.c	mprotect.c
	mlock.c	mremap.c	vmalloc.c	slab.c
	bootmem.c	swap.c	vmscan.c	page_io.c
	page_alloc.c	swap_state.c	swapfile.c	numa.c
	oom_kill.c	shmem.c		
net/	socket.c	sysctl_net.c		
net/802/	p8023.c	sysctl_net_802.c		
net/core/	sock.c	skbuff.c	iovec.c	datagram.c
	scm.c	sysctl_net_core.c	dev.c	dev_mcast.c
	dst.c	neighbour.c	rtnetlink.c	utils.c
net/ethernet/	eth.c	sysctl_net_ether.c		
net/ipv4/	utils.c	route.c	inetpeer.c	proc.c
	protocol.c	ip_input.c	ip_fragment.c	ip_forward.c
	ip_options.c	ip_output.c	ip_sockglue.c	tcp.c
	tcp_input.c	tcp_output.c	tcp_timer.c	tcp_ipv4.c
	tcp_minisocks.c	tcp_diag.c	raw.c	udp.c
	arp.c	icmp.c	devinet.c	af_inet.c
	igmp.c	sysctl_net_ipv4.c	fib_frontend.c	fib_semantics.c
	fib_hash.c	ipconfig.c		

(*contiuned on next page*)

Appendix (*continued*)

directory	files			
net/netlink/	af_netlink.c			
net/packet/	af_packet.c			
net/sched/	sch_generic.c			
net/sunrpc/	clnt.c	xprt.c	sched.c	auth.c
	auth_null.c	auth_unix.c	svc.c	svcsock.c
	svcauth.c	pmap_clnt.c	timer.c	xdr.c
	sunrpc_syms.c	stats.c	sysctl.c	
net/unix/	af_unix.c	garbage.c	sysctl_net_unix.c	
arch/i386/kernel/	process.c	semaphore.c	signal.c	entry.S
	traps.c	irq.c	vm86.c	ptrace.c
	i8259.c	ioport.c	ldt.c	setup.c
	time.c	sys_i386.c	pci-dma.c	i386_ksyms.c
	i387.c	bluesmoke.c	dmi_scan.c	pci-i386.c
	pci-pc.c	pci-irq.c	head.S	init_task.c
arch/i386/mm/	init.c	fault.c	ioremap.c	extable.c
	pageattr.c			
arch/i386/lib/	checksum.S	old-checksum.c	delay.c	usercopy.c
	getuser.S	memcpy.c	strstr.c	
arch/i386/boot/	bootsect.S	setup.S		
arch/i386/boot/compressed/		head.S	misc.c	

References

Albinet, A., Arlat, J., Fabre, J.-C., 2004. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In: Int. Conf. Dependable Syst. and Networks, June, pp. 867–876.

Basili, V.R., Briand, L.C., Melo, W.L., 1996. A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. 22 (10), 751–761, October.

Binkley, A.B., Schach, S.R., 1998. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In: 20th Intl. Conf. Softw. Eng., April, pp. 452–455.

Bovet, D.P., Cesati, M., 2001. Understanding the Linux Kernel. O'Reilly.

Bowman, I.T., Holt, R.C., Brewster, N.V., 1999. Linux as a case study: its extracted software architecture. In: 21st Int. Conf. Softw. Eng., May, pp. 555–563.

Briand, L.C., Wust, J., Lounis, H., 1999. Using coupling measurement for impact analysis in object-oriented systems. In: Int. Conf. Softw. Maintenance, August, pp. 475–482.

Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D., 2001. An empirical study of operating system errors. In: 18th Symp. Operating Systems Principles, October, pp. 73–88.

Dijkstra, E.W., 1968. The structure of the THE-multiprogramming system. Comm. ACM 11 (5), 341–346, May.

Epping, A., Lott, C.M., 1994. Does software design complexity affect maintenance effort?. In: 19th NASA/GSFC Software Engineering Workshop, November, pp. 297–313.

Ferneley, E., 2000. Coupling and control flow measures in practice. J. Syst. Softw. 51 (2), 99–109. April.

IA-32 Intel Architecture Software Developer's Manual. Vol.3: System Programming Guide. 2004. Order number 253668. URL <http://www.intel.com/design/pentium4/manuals/25366814.pdf>.

Ince, D., 1988. Software Development: Fashioning the Baroque. Oxford University Press.

Offutt, A.J., Harrold, M.J., Kolte, P., 1993. A software metric system for module coupling. J. Syst. Softw. 20 (3), 295–308, March.

Orso, A., Sinha, S., Harrold, M.J., 2001. Effects of pointers on data dependences. In: 9th IEEE Intl. Workshop Program Comprehension, May, pp. 39–49.

Parnas, D.L., 1972. On the criteria to be used in decomposing systems into modules. Comm. ACM 15 (12), 1053–1058, December.

Paulson, J.W., Succi, G., Eberlein, A., 2004. An empirical study of open-source and closed-source software products. IEEE Trans. Softw. Eng. 30 (4), 246–256, April.

Raymond, E.S., 2000. The cathedral and the bazaar. URL <http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar>.

Rilling, J., Klemola, T., 2003. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In: 11th IEEE Int. Workshop Program Comprehension, May, pp. 115–124.

Schach, S.R., 2007. Object-Oriented and Classical Software Engineering, seventh ed. McGraw-Hill.

Schach, S.R., Jin, B., Wright, D.R., Heller, G.Z., Offutt, A.J., 2002. Maintainability of the Linux kernel. IEE Proc.-Softw. 149 (2), 18–23.

Schach, S.R., Jin, B., Wright, D.R., Heller, G.Z., Offutt, J., 2003. Quality impacts of clandestine common coupling. Softw. Quality J. 11, 211–218.

Schach, S.R., Adeshiyan, T.O.S., Balasubramanian, D., Madl, G., Osses, E.P., Singh, S., Suwanmongkol, K., Xie, M., Feitelson, D.G., in press. Common coupling and pointer variables, with application to a Linux case study. Software Quality J.

Schroeder, M.D., Clark, D.D., Saltzer, J.H., 1977. The Multics kernel design project. In: 6th Symp. Operating Systems Principles, November, pp. 43–56.

Schwanke, R.W., 1991. An intelligent tool for re-engineering software modularity. In: 13th Int. Conf. Softw. Eng., May, pp. 83–92.

Selby, R.W., Basili, V.R., 1991. Analyzing error-prone system structure. IEEE Trans. Softw. Eng. 17 (2), 141–152, February.

Tran, J.B., Godfrey, M.W., Lee, E.H.S., Holt, R.C., 2000. Architectural repair of open source software. In: 8th IEEE Intl. Workshop Program Comprehension, June, pp. 48–59.

Yu, L., Schach, S.R., Chen, K., Offutt, J., 2004. Categorization of common coupling and its application to the maintainability of the Linux kernel. IEEE Trans. Softw. Eng. 30 (10), 694–706, October.

Yu, L., Schach, S.R., Chen, K., Heller, G.Z., Offutt, J., 2006. Maintainability of the kernels of open-source operating systems: a comparison of Linux with FreeBSD, NetBSD, and OpenBSD. J. Syst. Softw. 79 (6), 807–815, June.