Common Coupling as a Measure of Reuse Effort in Kernel-Based Software with Case Studies on the Creation of MkLinux and Darwin

Liguo Yu

Department of Informatics
Computer and Information Sciences Department
Indiana University South Bend
South Bend, IN 46634, USA. Fax: 1-574-520-5589
E-mail: ligyu@iusb.edu

The initial version of this paper appers in the 19th International Conference on Software Engineering and Knowledge Engineering.

Abstract

An obstacle to software reuse is the large number of major modifications that frequently have to be made as a consequence of dependencies within the reused software components. In this paper, common coupling is categorized and used as a measure of the dependencies between software components. We compared common coupling in three operating systems, Linux, FreeBSD, and Mach, and related it to the reuse effort of these systems. The measure is evaluated by studying the creation of two operating systems, MkLinux which is based on the reuse of Linux and Mach, and Darwin which is based on the reuse of FreeBSD and Mach. We conclude that the way that common coupling is implemented in Linux kernel induces large dependencies between software components, which required more effort in order to be reused to produce MkLinux, while the common coupling implemented in the Mach and FreeBSD kernels induces few dependencies between software components, which required less effort in order to be reused to produce Darwin.

Keywords: Reuse, common coupling, kernel-based software, MkLinux, Darwin

1. INTRODUCTION

Software reuse has become a topic of interest within the software community because of its potential benefits. These include increased productivity and quality, and decreased cost and time-to-market. Obviously the biggest savings are to be found in large-scale reuse, that is, the reuse of a large portion of an existing software product. A considerable amount of research has been undertaken in this area [1, 2, 3].

One problem with software reuse is that large software components may be dependent on other components. On the one hand, suppose that the components of a software product are classes that communicate exclusively by message passing. The dependency between the components is low, and it should be possible to reuse one component in a new software product *with* little difficulty. But if a software product consists of components, all of which reference a large number of global variables, it may be impossible to reuse any one component in a new product without first totally redesigning and reimplementing that component, thereby all but defeating the purpose of reuse.

Many software products, including operating systems and database management systems, are *kernel-based*. That is, each implementation consists of required kernel components, together with specific optional architecture-specific or hardware-specific non-kernel components. The word kernel is overloaded. It can refer to a nucleus that can execute certain instructions [4, 5], or to a set of modules that are included in every installation. In this paper, we use “kernel” in the latter sense.

Coupling is a measure of the degree of interaction between two software components. It reflects the modifiability and the maintainability of a software product [6]. There are many different categorizations of coupling, all of which include common (global) coupling (two software components are *common coupled* if they reference the same global variable). Certain types of coupling, especially common coupling, are considered to present risks for software development and, in particular, for maintenance [7]. Common coupling can also be used to measure the dependencies between software components [8]. To reuse a kernel component in another software product, it is important that the kernel component should have minimal dependency on other components. Accordingly, it is important that the kernel components have as little common coupling as possible.

In a previous study, Yu et al. [8] defined a new categorization of common coupling within kernel-based software, and used it to measure the maintenance effort of kernel-based software. In this paper, we extend the categorization and use it to evaluate reuse effort in kernel-based software. Reuse and maintenance are different in many ways. For example, reuse is almost always optional, and often just one component or a few related components are reused. In contrast, maintenance is usually required, especially corrective maintenance, and maintenance has to be performed on a software product as a whole (hence the need for regression testing). On the other hand, reuse and maintenance have at least one feature in common: Both are adversely affected by common coupling.

MkLinux and Darwin are the outcomes of Apple Computer’s endeavor to create new operating systems based on the reuse of existing operating systems, in which MkLinux is produced through the reuse of Linux and Mach, and Darwin is produced through the reuse of FreeBSD and Mach. However, MkLinux and Darwin have different fates. MkLinux is active for only several years (1996 to 2001) and the development is dormant now. In contrast, Darwin is considered a successfully project and has continually being developed and used since 1998. In this paper, we use common coupling as a measure to study the reuse effort in the creation of the two operating systems.

The remainder of the paper is divided into six sections. Section 2 discusses component dependencies and reuse effort. In Section 3, we review the categorization of common coupling and discuss its relation to reuse effort. We introduce additional terminology in Section 4. Section 5 describes MkLinux, Darwin, and other open-source operating systems. Section 6 contains the results of our study of open-source operating systems. The conclusions are in Section 7.

2. Software Dependencies and Reuse Effort

Coupling is a measure of the degree of dependency between two software components (classes, modules, packages, or the like). A good software system should have high cohesion within each component and weak coupling between components. Coupling between components strengthens the dependency of one component on others and increases the probability that changes in one component may affect other components. There are several different coupling categorizations [6, 9], all of which include *common coupling*. Common coupling is considered to be a strong form of coupling, that is, it induces strong dependencies between software components, making software components difficult to understand, maintain, and reuse [7].

If a software component has strong dependencies (such as common coupling) on other components, it requires more effort to be adapted to a new environment. Common coupling makes a software component difficult to reuse for two reasons. First, suppose that we wish to reuse component C_0, and that C_0 is common coupled to n components C_1, C_2, … C_n. One alternative would be to incorporate and reuse not just C_0 but also components C_1 through C_n as well. However, this would result in unnecessary reuse and make the resulting product hard to comprehend. A second alternative is to reuse component C_0 on its own, that is, without components C_1 through C_n. In order to do this, we would need to make major modifications to C_0. In fact, these modifications might well be so drastic that it would be cheaper and quicker to design and implement a new version of C_0 from scratch, rather than reuse the existing version. Therefore, it requires more effort to reuse a component with strong coupling, like common coupling.

The kernel is the common part of a kernel-based software product. It is the most frequently reused component; reuse of kernel-based software usually consists of reusing all or most of the kernel, together with certain other non-kernel components. Therefore, the effort involved in reusing the kernel reflects the reuse effort of a kernel-based software product. Common coupling within a kernel-based product may increase the dependency of the kernel on non-kernel components and, therefore, it would require more effort to reuse the kernel or the software product as a whole.

3. Common Coupling and Software Reuse Effort

Common coupling induces dependencies between software components. As described in [8], these dependencies are induced by the definition-use mechanism; we say component C_1 is dependent on component C_2 via global variable gv if C_1 *uses* gv and C_2 *defines* gv (that is, if C_2 changes the value of gv and C_1 utilizes that value).

In previous study [8], global variables were categorized in terms of five categories, as summarized in Table 1.

Table 1: Categorization of global variables in kernel-based software [8]

Category number	Description
1	A global variable defined in one or morekernel components but not used in any kernel components.
2	A global variable defined in one kernel component and used in one or more kernel components.
3	A global variable defined in more than one kernel component, and used in one or more kernel components.
4	A global variable defined in one or more non-kernel components and used in one or more kernel components.
5	A global variable defined in one or more non-kernel components and defined and used in one or more kernel components.

This categorization of common coupling was introduced within the context of kernel maintenance [8] and discussed the impact of the existence of global variables in each category on kernel maintenance. In this paper, the categorization of common coupling is applied to software reuse effort. Maintenance effort is not directly related to reuse effort, but both are dependent on component dependencies. As described in Section 2, strong dependencies affect not only software maintenance but also software reuse. In the remainder of this section, we discuss how our categorization of common coupling can be used as a measure for reuse effort.

We consider two types of reuse. We refer to the reuse of one or more independent kernel components as *kernel-component reuse* and the reuse of all of the kernel as *entire-kernel reuse*. When we wish to refer to either kernel-component reuse or entire-kernel reuse, we use the umbrella term *kernel reuse*.

Dependencies induced by common coupling affect the reuse effort; in general, more effort is needed to reuse a component with a large number of global variables. However, reuse effort is also affected by the category into which each global variable falls.

A category-1 global variable is not used in a kernel component, so definitions of the global variable in other components (kernel or non-kernel) cannot affect kernel components. All kernel components are independent with respect to this global variable. Accordingly, the presence of a category-1 global variable will not cause difficulties for kernel reuse. Therefore, no kernel reuse effort is associated with a category-1 global variable.

A category-2 or category-3 global variable is defined in one or more kernel components but not in any non-kernel component. It is used in kernel components. A category-2 or category-3 global variable therefore induces dependencies between kernel components. A kernel component that defines a category-2 or category-3 global variable can affect the reuse effort of any kernel component that uses that global variable. Turning to the reuse effort of the entire kernel, this is not affected by the presence of a category-2 or category-3 global variable because there is no definition outside the kernel.

A kernel component that uses a category-4 or category-5 global variable is dependent upon non-kernel components that define that global variable. Thus, the presence of a category-4 or category-5 global variable in a kernel component negatively impacts both kernel-component reuse as well as entire-kernel reuse. Hence, more effort for kernel reuse is associated with category-4 and category-5 global variables than for categories 2 and 3.

Table 2 summarizes the impact of global variables in different categories on kernel reuse effort.

Table 2: The Impact of global variables on kernel reuse effort in kernel-based software

Category number	Kernel-component reuse effort	Entire-kernel reuse effort
1	No impact	No impact
2	Negative impact	No impact
3	Negative impact	No impact
4	Negative impact	Negative impact
5	Negative impact	Negative impact

4. New Terminology

As indicated in the previous section, reuse is hampered by definitions in non-kernel components that affect uses in kernel modules. In order to be able to quantify this phenomenon, we introduce additional terminology in this section.

Terminology 1: A definition of a global variable that induces a dependency of a kernel component on another component is called a *component-dependency-inducing* definition.

Terminology 2: A global variable is *kernel-on-non-kernel-dependency-inducing* if it induces a dependency of a kernel component on a non-kernel component.

Terminology 3: A kernel component is *use-*dependency-*induced* if it contains a use of a kernel-on-non-kernel-dependency-inducing variable.

Terminology 4: A non-kernel component is *definition-dependency-inducing* if it contains a definition of a kernel-on-non-kernel-dependency-inducing variable.

This terminology is discussed and utilized in Section 6.

5. MkLinux, Darwin, and Other Open-Source Operating Systems

MkLinux is short for *Microkernel Linux*, which is one of the outcomes of Apple Computer's endeavor to adapt a Unix-like kernel to create an operating system for Macintosh computers [10]. This project was started in February 1996 by integrating the Linux kernel with the Mach microkernel. In the summer of 1998, MkLinux Developers Association took over development of the system. However, the project appears to be abandoned now, having not had a release since 2002. Some updates are occasionally produced, but no further development [11].

Darwin is another outcome of Apple computer's endeavor to adapt a Unix-like operating system for Macintosh computers [12]. In contrast to MkLinux, Darwin was produced through the integration of FreeBSD and Mach. The first version of Darwin (version 0.1) was released on March, 1999. The latest version (version 8.9) was released on March, 2007. Darwin is considered a successful project. Until now it is still actively developing new versions.

In the creation of MkLinux and Darwin, the existing software components, Mach, Linux, and FreeBSD, designed and implemented separately, were customized and reused. However, none of Linux, FreeBSD, or Mach consists of ready-to-use building blocks. Modifications had to be made and effort had to be spent on each of those components in order to incorporate them into the new product [13].

In this study, we wished to understand the effort involved in modifying the different pieces that were reused to produce MkLinux and Darwin. Accordingly, we studied the three major pieces from which MkLinux and Darwin was built: version 3.0 of Mach, version 2.1.129 of Linux, and version 5.1 of Linux, from which Mach 3.0 and Linux 2.1.129 were used to create MkLinux 1.1 and Mach 3.0 and FreeBSD 5.1 were used to create Darwin 7.0 [14].

More precisely, we studied common coupling of the following open-source operating systems: Mach 3.0, FreeBSD 5.1, Linux 2.1.129 in order to understand the effort to reusing these systems. To examine the results of using common coupling to measure reuse effort, we compared the source code of Linux 2.1.129, Mach 3.0, and FreeBSD 5.1 with MkLinux 1.1 and Darwin 7.0.

All these operating system are written in C or C++. In this paper, a *component* is defined to be a source code file (".c" file, ".cpp", or ".h" file). The *size* of the product is measured in thousands of lines of code (KLOC). Data regarding the number of components and the number of lines of code of these systems are provided in Table 3. All these operating systems are kernel-based [15]. The column headed *Kernel components* in Table 3 shows the number of components in the kernel, and the number of lines of code in the kernel components is shown in the column headed *Kernel KLOC*. The column headed *Non-kernel components* shows the number of components in the non-kernel and the total number lines of code (both kernel and non-kernel) is shown in column headed *Total KLOC*. It should be noted that both MkLinux and Darwin are dual kernel systems, in which MkLinux contains Linux kernel and osfmk (Mach) kernel while Darwin contains BSD kernel and osfmk (Mach) kernel. This will be further illustrated in Section 6. It should be noted that in Table 3, the kernel components are determined according to their component names (kernel, for example) within the source code three.

6. The Effort to Reusing Linux, Mach, and FreeBSD

In the previous sections, we analyzed the relation between common coupling and software reuse effort, showing that common coupling can be used as a measure of software reuse effort. In the following subsections, we apply this measure to the three open-source operating systems and hence determine the reuse effort.

6.1. Common Coupling in General

We analyzed common coupling in the Mach, FreeBSD, and Linux operating systems. Global variables appearing in kernel components were identified by the Linux cross-referencing tool, lxr. Every instance of a global variable was determined to be either a definition or a use of that variable. An overview of our results is summarized in Table 4. As shown in the Table, there are 77 distinct global variables in the Mach kernel. Altogether, there are 332 instances of global variables in Mach kernel components. However, if multiple instances of a given global variable in a component are considered as one, there are 147 unique instances of a global variable in Mach kernel components. The other entries are similar. It worth noting that Linux has many more instances of global variables in kernel and non-kernel components than Mach and FreeBSD.

In general, global variables induce dependencies between software components and make the components difficult to reuse. However, as outlined in Section 3, the different categories of global variables have different effects on the reuse effort. To understand how global variables in the open-source operating systems affect the kernel reuse effort, each global variable was assigned to one of the five categories. Detailed results are shown in Tables 5 through 7 for Mach, Linux, and FreeBSD, respectively.

In the next two subsections, we discuss how dependencies within a kernel component and the kernel as a whole affect the reuse effort.

Table 3: The kernel and non-kernel structure of the five open-source operating systems

	Kernel components	Non-kernel components	Kernel KLOC	Total KLOC
Mach 3.0*	71	892	30.576	365.502
Linux 2.1.129*	20	3,597	9.829	1,512.314
FreeBSD 5.1*	131	3,157	108.475	1,821.619
MkLinux 1.1**	135	4,647	67.718	1,855.384
Darwin 7.0**	196	1,658	110.482	744.528
*single kernel system; **dual kernel system				

Table 4: Global variables in open-source operating systems

Operating system	Total number of global variables	Number of unique instances of a global variable in kernel components	Number of unique instances of a global variable in non-kernel components	Total number of instances of global variables in kernel components	Total number of instances of global variables in non-kernel components
Mach	77	147	99	332	228
Linux	76	137	2,027	759	6,964
FreeBSD	75	166	338	483	770

Table 5: Definitions and uses of global variables in Mach

Category number	Number of global variables	Kernel components			Non-kernel components		
		Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses	Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses
1	20	20	20	–	20	19	23
2	22	47	36	105	25	–	54
3	22	51	52	69	1	–	1
4	6	6	–	9	17	27	44
5	7	23	12	29	36	13	47
Overall	77	147	120	212	99	59	169

Table 6: Definitions and uses of global variables in Linux

Category number	Number of global variables	Kernel components			Non-kernel components		
		Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses	Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses
1	17	19	19	–	157	0	209
2	21	51	15	156	749	–	1,467
3	4	8	17	13	65	–	280
4	18	21	–	42	43	27	109
5	16	38	137	360	1,013	1,080	3,792
Overall	76	137	188	571	2,027	1,107	5,857

Table 7: Definitions and uses of global variables in FreeBSD

Category number	Number of global variables	Kernel components			Non-kernel components		
		Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses	Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses
1	22	23	53	–	73	0	87
2	35	104	74	251	172	–	504
3	8	18	23	28	36	–	70
4	4	8	–	25	24	22	21
5	6	13	6	23	33	24	42
Overall	75	166	156	327	338	46	724

6.2. Dependency of a Kernel Component

In order to analyze dependencies within the kernel and between kernel components and non-kernel components, we need to examine definitions and uses of global variables in more detail.

As stated in Section 4, a definition of a global variable that induces a dependency of a kernel component on another component is called a *component-dependency-inducing* definition. A reusable component should be dependent on as few other components as possible. A global variable in category 2, 3, 4, or 5 is used in a kernel component and defined in another component. Therefore, a definition of a category-2, -3, -4, or -5 global variable induces the dependency of a kernel component on another component, either in the kernel or the non-kernel. More specifically, a definition of a category-2, -3, or -5 global variable in a kernel component or a category-4 or -5 global variable in a non-kernel component is a component-dependency-inducing definition (see Section 4). Table 8 lists the number of component-dependency-inducing definitions in the operating systems we consider here. The entries in the columns headed "Number of component-dependency-inducing definitions per kernel component" and "Number of component-dependency-inducing definitions per kernel KLOC" were calculated by dividing the entries in column 4 by those in columns 2 and 3, respectively.

The entries in Table 8 may be interpreted as follows: Suppose we wish to reuse a kernel component K of Mach. On average, we will then have to modify 1.97 definitions of global variables in other components that induce dependencies in K and thereby affect its reuse. Similarly, if we wish to reuse 1,000 lines of Mach kernel code, on average we will need to need to modify 4.58 definitions of global variables in other components that induce dependencies and thereby affect the reuse of this code. In contrast, if we wish to reuse a Linux kernel component, on average we will need to modify 63.8 definitions of global variable in other components; to reuse 1,000 lines of Linux kernel code, on average we will need to modify 129.82 definitions of global variables in other components.

From Table 8, we can see that Mach and FreeBSD have a relatively small number of component-dependency-inducing definitions per kernel component and per kernel KLOC. This shows that, on average, a kernel component K in Mach and FreeBSD has few dependencies on other components, which makes K comparatively easy to reuse. The relatively independent property of a kernel component in Mach and FreeBSD means less effort is needed to perform kernel component reuse, while the relative dependent property of Linux means more effort is needed to perform kernel component reuse.

6.3. Dependencies of the Kernel as a Whole

In this paper, we are more concerned with entire-kernel reuse than kernel-component reuse. As we mentioned before, in most cases, successful reuse of kernel-based software depends on the reuse effort of the entire kernel. As stated in Section 4, a global variable is *kernel-on-non-kernel-dependency-inducing* if it induces a dependency of a kernel component on a non-kernel component.

As mentioned before, a category-4 or category-5 global variable is the most undesirable. According to terminology 2, such a global variable is kernel-on-non-kernel-dependency-inducing, because it has a definition in a non-kernel component and a use in a kernel component. That is, the definition of a category-4 or -5 global variable induces a dependency of a kernel component on a non-kernel component; these dependencies adversely affect the entire-kernel reuse effort. Table 9 enumerates the kernel-on-non-kernel-dependency-inducing global variables in the open-source operating systems we consider here.

Table 8: Dependencies of a kernel component

Operatin g system	Number of kernel components	Kernel size (KLOC)	Number of component-dependency-inducing definitions	Number of component-dependency-inducing definitions per kernel component	Number of component-dependency-inducing definitions per kernel KLOC
Mach	71	30.576	140	1.97	4.58
Linux	20	9.829	1,276	63.80	129.82
FreeBSD	131	108.475	149	1.14	1.37

Table 9: Kernel-on-non-kernel-dependency-inducing global variables

Operating system	Number of global variables	Kernel components		Non-kernel components	
		Number of unique instances of uses	Number of instances of uses	Number of unique instances of definitions	Number of instances of definitions
Mach	13	22	38	22	40
Linux	34	57	402	421	1,107
FreeBSD	10	18	48	42	46

Table 10: Dependencies of kernel components on non-kernel components induced by kernel-on-non-kernel-dependency-inducing variables

Operating System	Kernel			Non-kernel	
	Number of components	Number of use-dependency-induced kernel components	Number of instances of uses	Number of definition-dependency-inducing non-kernel components	Number of instances of definitions
Mach	71	12	38	19	40
Linux	20	16	402	395	1,107
FreeBSD	131	14	48	25	46

Considering entire-kernel reuse, there are 13 global variables that make Mach kernel components dependent on non-kernel components. These 13 global variables are used 38 times in kernel components. If multiple instances of uses of the same global variable in the same component are ignored, there are 22 unique instances of uses in kernel components. Also, these 13 global variables are defined 40 times in non-kernel components. If multiple instances of definitions of the same global variable in the same component are ignored, there are 22 unique instances of definitions in non-kernel components. The other entries in Table 9 are similar.

An implication of Table 9 is that, if we wish to reuse the entire Mach kernel, we either need to modify the 38 uses of kernel-on-non-kernel-dependency-inducing variables in kernel components to remove the dependencies, or we also need to incorporate 40 definitions in non-kernel components (or some combination of the two alternatives). Combining Table 9 with Table 4, we see that, although there are 332 instances of global variables in the Mach kernel, only 38 of them induce dependencies of a kernel component on a non-kernel component. Furthermore, of the 228 instances of global variables in non-kernel components, only 40 of them make it difficult to reuse the entire kernel. A similar result is found for FreeBSD, but not Linux, which contains more kernel uses and non-kernel definitions of kernel-on-non-kernel-dependency-inducing global variables.

Now we determine how many kernel components have to be changed, or how many non-kernel components have to be reused if we want to reuse the entire kernel.

Dependencies between components caused by global variables are induced by the definition–use relationship. As stated in Section 4, a kernel component is *use-dependency-induced* if it contains a use of a kernel-on-non-kernel-dependency-inducing variable, and a non-kernel component is *definition-dependency-inducing* if it contains a definition of a kernel-on-non-kernel-dependency-inducing variable. Use-dependency-induced kernel components use the value of a kernel-on-non-kernel-dependency-inducing variable; definition-dependency-inducing non-kernel components define the value of a kernel-on-non-kernel-dependency-inducing variable, which means that a use-dependency-induced kernel component is dependent on at least one definition-dependency-inducing non-kernel component. A kernel is difficult to reuse if it has too many use-dependency-induced kernel components and if there are too many definition-dependency-inducing non-kernel components.

Table 10 shows the number of use-dependency-induced kernel components and definition-dependency-inducing non-kernel components in the operating systems we consider here. Multiple occurrences of the same component are counted as one. For example, if kernel component K contains multiple uses of kernel-on-non-kernel-dependency-inducing global variables gv1 and gv2, it is nevertheless counted as only one use-dependency-induced kernel component, because modifications will have to be made to kernel component K irrespective of the number of uses of kernel-on-non-kernel-dependency-inducing global variables. Similarly, if non-kernel component NK contains multiple definitions of kernel-on-non-kernel-dependency-inducing global variables gv3 and gv4, it is likewise counted as only one definition-dependency-inducing non-kernel component.

Using Mach as an example to explain the entries of Table 10, there are 71 kernel components in Mach, 12 of which are use-dependency-induced kernel components. There are 19 definition-dependency-inducing non-kernel components. This means that 12 kernel components have dependencies on 19 non-kernel components via common coupling. More precisely, 12 kernel components use at least one kernel-on-non-kernel-dependency-inducing variable in a total of 38 instances, which depend on 19 non-kernel components that define a kernel-on-non-kernel-dependency-inducing variable in a total of 40 instances.

Now, suppose that all 71 Mach kernel components are to be reused. Two extreme approaches could be taken. First, we could modify the 12 use-dependency-induced kernel components in 38 places to remove the dependencies of kernel components on non-kernel components. Second, we could reuse the 19 definition-dependency-inducing non-kernel components together with the kernel. Clearly, any combination of these two extreme approaches could also be adopted. Turning now to reusing the FreeBSD kernel, we could similarly modify the 14 use-dependency-induced kernel components in 48 places, reuse the 25 definition-dependency-inducing non-kernel components together with the kernel, or adopt some combination of the two extreme approaches.

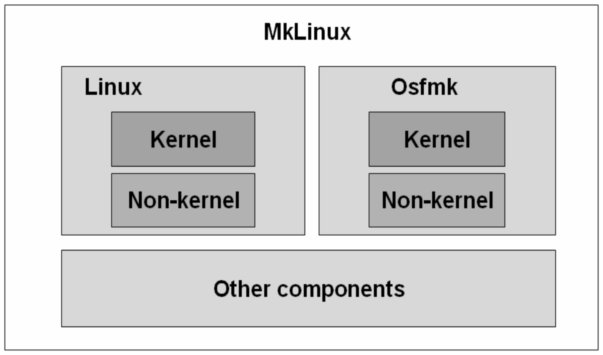
Recapitulating, suppose we wish to reuse the entire Mach kernel, from Table 4, it appears that we would have to modify 332 instances of global variables in kernel modules. By considering only those instances that induce dependencies of a kernel component on a non-kernel component, we see from Table 9 that only 38 of the 332 instances would have to be changed. Finally, by considering use-dependency-induced kernel components, we see from Table 10 that the number of kernel components that would have to be changed is 12. Alternatively, 19 definition-dependency-inducing non-kernel components would have to be reused together with the entire kernel. This shows that the Mach kernel and the FreeBSD kernel are relatively independent as a whole, which means less effort is needed to perform entire-kernel reuse.

In contrasting, it is hard to find a good strategy for reusing the 20 Linux kernel components. On one hand, if we modify the 16 use-dependency-induced kernel components in 402 places, we may completely change the functionality of the kernel. On the other hand, reusing the 395 definition-dependency-inducing non-kernel components together with the kernel would result in widespread unnecessary and redundant reuse. Furthermore, a kernel is generally difficult to reuse if it references a kernel-on-non-kernel-dependency-inducing variable gv and there are many definitions of gv in non-kernel components and many uses in kernel components. Linux has more instances of uses of kernel-on-non-kernel-dependency-inducing variables in kernel components and instances of definitions in non-kernel components than Mach or the three BSDs, which means that the Linux kernel is strongly dependent on non-kernel components.

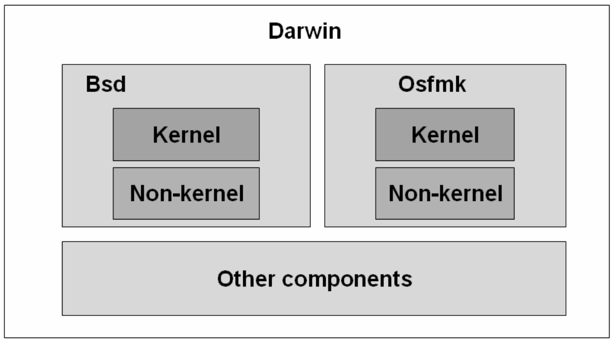
Software reuse depends on a large number of disparate factors [16]. One factor is the effort spent on customizing and reusing these components. The reuse effort of a kernel-based software product depends on the reuse effort of its kernel and this, in turn, depends on the definitions and uses of global variables within the kernel and non-kernel components. From the viewpoint of dependencies, reusing both the Mach and FreeBSD kernels is relatively effortless, irrespective of the precise reuse mechanism followed, while reusing Linux kernel will consume more effort.

6.4. THE CREATION OF MKLINUX AND DARWINOLE

As described in Section 5, MkLinux and Darwin were created by reusing Linux, FreeBSD, and Mach. Both MkLinux and Darwin are dual-kernel systems. Their structures are shown in Figure 1. In MkLinux, the Linux part is reused from Linux operating system, the Osfmk part is reused from Mach; in Darwin, the Bsd part is reused from FreeBSD, and the Osfmk part is reused from Mach.



(a)



(b)

Figure 1. The structure of (a) MkLinux; and (b) Darwin.

In order to evaluate our measures of using common coupling to represent reuse effort, we compared the source code of (1) MkLinux 1.1 with Linux 2.1.129 and Mach 3.0, from which MkLinux 1.1 is produced; (2) Darwin 7.0 with FreeBSD 5.1 and Mach 3.0, from which Darwin 7.0 is produced. The comparison is performed using a Perl program that integrates the source code *diff* function. Table 11 lists the number of components of Linux, Mach, and FreeBSD that were reused (might with modifications) in the creation of MkLinux and Darwin. Table 12 lists the number of new components added to MkLinux and Darwin. These new components could be added to kernel or non-kernel. The *other components* parts in Figure 1 are considered as non-kernel.

Table 11: The number of components reused from Linux, FreeBSD, and Mach

Target system	Reused from	Kernel		Non-kernel	
		Original	Reused	Original	Reused
MkLinux	Linux	20	17	3,597	1,582
	Mach	71	64	892	395
Darwin	FreeBSD	131	48	3,157	223
	Mach	71	59	892	189

Table 12: The number of components in MkLinux and Darwin

	Kernel		Non-kernel		Total
Target system	Reused	New	Reused	New	
MkLinux	81	54	1,977	2,670	4,782
Darwin	107	89	412	1,246	1,854

The effort to producing new systems by reusing existing components can be roughly divided into two parts, (a) the effort of reusing (including identifying and modifying) existing components, and (b) the effort of creating new components. Assume the effort to identifying components is minor comparing with the effort to modifying the component. We use the number of lines of code modified on original components to represent the effort of reusing existing components. Table 13 shows the effort of reusing existing components in the creation of MkLinux and Darwin (The size of the reused component refers to the size of the modified component, not the original component). The effort is represented with the total number of lines of code modified on (including added to, deleted from, and changed on) the original components. It shows that about 445.655k and 269.346k lines of code need to be modified on reusing the existing components in the creation of MkLinux and Darwin respectively.

Table 13: The effort of reusing existing components in the construction of MkLinux and Darwin

Target system	Reused from	Number of reused components	Size of reused Components (KLOC)	Total Lines modified (KLOC)
MkLinux	Linux	1,599	741.959	332.668
	Mach	459	249.268	112.987
Darwin	FreeBSD	271	179.948	178.093
	Mach	248	123.442	91.253

The effort of creating new components can be represented by the number of new components created and the size of the new components. Table 14 lists the new components created in the construction of MkLinux and FreeBSD. It can be seen that more new components and more new lines of code are created in the construction of MkLinux than in the construction of Darwin.

Table 14: The effort of creating new components in the construction of MkLinux and Darwin

Target system	Number of new components	Size of new components (KLOC)
MkLinux	2,724	938.132
Darwin	1,335	441.703

From Table 13 and Table 14, we can see that more efforts are spent in modifying reused components and in creating new components in the construction of MkLinux than in the construction of Darwin. In particular, we found that in the construction of MkLinux, about 446k lines of code needs to be modified to reuse Linux and Mach components, and about 938k new lines of code needs to be created to add new components; in contrast, in the construction of Darwin, about 269k lines of code need to be modified to reuse FreeBSD and Mach components and about 442k lines of code need to be created to add new components. These observations indirectly support our argument of using common coupling as the measure of reuse effort and the subsequent conclusions—more effort is needed to reuse Linux kernel than FreeBSD and Mach Kernel due to kernel dependency induced by common coupling.

7. Conclusions

In this paper, we have utilized our categorization of common coupling based on definitions and uses of global variables to analyze the reuse effort for a software component. Common coupling in different categories has different effects on the reuse effort in kernel-based software. Our results show that common coupling within the Mach kernel and the FreeBSD kernel is well designed, inducing only a few dependencies of kernel components on non-kernel components. As a result, relatively less effort is required for entire-kernel reuse of these two operating systems. While common coupling within Linux kernel induces large amount of dependencies, which makes it difficult. The discussions were evaluated by analyzing the effort in the creation of MkLinux and Darwin.

Reference

[1] D. E. Perry, H. P. Siy, H. P. and L. G. Votta. Parallel changes in large-scale software development: an observational case study. ACM Transactions on Software Engineering and Methodology, 10(3): 308–337, 2001.

[2] K. J. Sullivan and J. C. Knight. Experience assessing an architectural approach to large-scale systematic reuse. In Proceedings of the Eighteenth International Conference on Software Engineering (ICSE-18), Berlin, pp. 220–229, 1996.

[3] W. B. Frakes and S. Isoda. Success factors of systematic reuse. IEEE Software, 11(5): 14–19, 1994.

[4] P. B. Hansen. The nucleus of a multiprogramming system. Communications of the ACM, 4(4): 238–241, 1970.

[5] T. Härden. New approaches to object processing in engineering databases. In Proceedings of International Workshop on Object-Oriented Database Systems, pp. 217, 1986.

[6] W. P. Stevens, G. J. Myers and L. L. Constantine. Structured design. IBM Systems Journals, 13(2): 115–139, 1974.

[7] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller and J. Offutt. Quality impacts of clandestine common coupling. Software Quality Journal, 11(3): 211–218, 2003.

[8] L. Yu, S. R. Schach, K. Chen and J. Offutt. Categorization of common coupling and its application to the maintainability of the Linux kernel. IEEE Transactions on Software Engineering, 30(10): 694–706, 2004

[9] J. Offutt, M. J. Harrold and P. Kolte. A software metric system for module coupling. Journal of System and Software, 20(3): 295–308, 1993.

[10] MkLinux.org. http://www.mklinux.org/, 2007

[11] MkLinux News. http://www.mklinux.org/info/index.html, 2007

[12] Apple Computer. Mac OS X hits stores this weekend. http://www.apple.com/pr/library/2001/mar/21osxstore.html, 2001

[13] J. West. How open is open enough? Modeling proprietary and open source platform strategies. Research Policy, 32(7): 1259–1285, 2003.

[14] Kernelthread. What is Mac OS X. http://www.kernelthread.com/mac/osx/arch_xnu.html, 2005.

[15] L. Yu, S. R. Schach, K. Chen, G. Z. Heller and J. Offutt. Maintainability of the kernels of open-source operating systems: A comparison of Linux to FreeBSD, NetBSD, and OpenBSD. Journal of Systems and Software, 79(6): 807–815, 2006

[16] G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. IEEE Computer, 24(2): 61–70, 1991.