DOI: 10.2298/CSIS130115070M

Extending Programming Language to Support Object Orientation in Legacy Systems

Hemang Mehta, S J Balaji, and Dharanipragada Janakiram

Department of Computer Science and Engineering,
Indian Institute of Technology Madras,
Chennai 600036, India
{hemang,sjbalaji,djram}@cse.iitm.ac.in

Abstract. The contemporary software systems written in C face maintainability issues because of tight coupling. Introducing object orientation can address these problems by raising the abstraction to objects, thereby providing better programmability and understandability. However, compiling a C software with a C++ compiler is difficult because of the incompatibilities between C and C++. Some of the incompatibilities such as designated initializers are nontrivial in nature and hence are very difficult to handle by automation such as scripting or by manual efforts. Moreover, runtime support for features such as global constructors, exception handling, runtime type inference, etc. is also required in the target system. Clearly, the traditional procedural language compiler cannot provide these features. In this paper, we propose extending programming language such as C++ to support object orientation in legacy systems instead of completely redesigning them. With a case study of Linux kernel, we report major issues in providing the compile and runtime support for C++ in legacy systems, and provide a solution to these issues. Our approach paves the way for converting a large C based software into C++. The experiments demonstrate that the proposed extension saves significant manual efforts with very little change in the g++ compiler. In addition, the performance study considers other legacy systems written in C and shows that the overhead resulting from the modifications in the compiler is negligible in comparison to the functionality achieved.

Keywords: g++, programming language, Linux kernel, legacy systems, object orientation.

1. Introduction

Many well-known large scale software systems such as Linux kernel[5], Apache webserver[7], PostgreSQL[6], etc. have been programmed in procedural language C. As these systems evolve with time, they become prone to issues related to cohesion and coupling. These issues make the systems difficult to maintain and reduce the understandability of the code. For example, Linux kernel has undergone studies[12,11] which reveal that it is a tightly coupled system and the instances of common coupling are increasing exponentially with new versions. Our previous work on object oriented(OO) wrappers[8] shows that the introduction of minimal OO features can help increase the maintainability. There are two main challenges when introducing OO concepts in a procedural language based system. The first is to compile all C files with a C++ compiler. The other challenge is to provide runtime support for features such as invoking of constructors for global and static objects.

Though C++ is perceived as a superset of C language, there are many incompatibilities between them which restrict compilation of the legacy system with a C++ compiler. The incompatibilities between C and C++ can be classified in trivial and nontrivial categories. The trivial issues can be addressed using scripting (e.g. renaming C++ keywords used as identifiers) or can be resolved manually if there are limited instances (e.g. pascal style function definitions). On the other hand, nontrivial issues can not be easily solved by the same means. An example of such incompatibility is support for nontrivial designated initializers.

Designated Initializers(DI) are used for initializing complex datatypes such as structures and arrays as shown in the following code snippet.

```
struct book { char name[]; int pages; }book_var = {
             ``Programming'', 200 };
       int arr[5] = { 10, 12, 21, 25, 32 };
```

Support for trivial DI was included in C89 standard[1] and C++ standard[3] also conforms to the same. On the other hand, nontrivial DI were introduced in C99[2]. The nontrivial DI provides the following features on top of trivial DI:
Labeled Initializing: Structure members can be selectively initialized out of the order in which they are defined.
Indexed Initializing: Assign values to specific array members using their indexes.
Ranged Initializing: specific range of an array can be initialized.
These features grant increased flexibility of initialization to the programmers. In addition, the absence of explicit constructors for structure variables leads to extensive and complex usage of DI (We shall use term 'DI' for nontrivial designated initializers henceforth.) in a C based software. Since C++ standard does not include DI, compiling a software written in C with g++ is not possible.

In this paper, we present an approach to extend g++ so that it recognizes the ranged, labeled, indexed initialization and nesting of them. With the help of a case study of Linux kernel, we show that it recognizes numerous instances of DI in the kernel and saves significant efforts. The main challenges we envisage are:
1. Different types of DI for structures and arrays,
2. Different combinations of types used in a single initialization and
3. Nested usage of DI in macro preprocessors.

Additionally, we explain how runtime support for C++ can be added in a legacy system with an example of Linux kernel.

The rest of the paper is organized as follows: Section 2 explains the motivation behind extending g++, as compared to other possible approaches. The design and implementation of the g++ extension are presented in Section 3. Section 4 describes the usefulness of the proposed approach using Linux kernel as a case study and also presents a performance study of the same. In Section 5, we explain how the runtime support for global constructors and volatile typecasting was included in the Linux kernel. The concluding remarks with future working directions are presented in Section 6.

2. Motivation

The first motivation of extending g++ was the absence of support for DI in g++ compiler. It is not included in the latest C++ standard [4] that was developed in 2011. To the best of our knowledge, there have been no attempts in the literature to explore this area. In case of C++, the complex datatype widely used is class and constructors are used to initialize objects. There was no need previously to statically initialize structure variables because of this, and hence the need for supporting DI was not felt in C++. However, with Linux kernel, we need some structure variables to be initialized at compile time since their initial state is required for system booting. The constructors are ill-suited in this case as they are called only after basic system initialization is complete. Secondly, many systems are implemented with both C and C++ like MySQL[10] and Windows kernel[9]. The primary reason behind this is that C++ provides higher abstractions in form of objects and many other useful features such as inheritance, polymorphism, templates, etc. If g++ could compile DI, the efforts in the development of these system can be significantly reduced. Thus, the primary objective is to make g++ recognize DI. We explain different possible approaches to tackle this issue. We motivate the need of extending g++ as a solution by comparing them with those solutions.

2.1. List Initialization

C++ standard supports list initialization of structure variables. The list method allows assigning values to *all* members of a structure in the *exact order* in which they are defined in the structure. Hence one approach is to replace labeled DI with list method using an automatic process such as scripting. However, converting labeled DI to list construct has two major issues. Firstly, the uninitialized members of the structure should be assigned their default values. Thus the script performing the replacement has to infer the datatypes of the member variables and their default values. Secondly, the values of members to be initialized have to be ordered. This becomes difficult especially when there is a nesting of DI (i.e. a member of a structure is also a structure and is initialized using the DI) and the nested element is initialized with a macro.

Apart from the labeled DI for structures, indexed and ranged initialzers are also required to be replaced by the list method. The list method is ill-suited if the array is very large and uses indexed or ranged methods.

2.2. Constructor for Structures

Another way to make g++ compatible with DI is to replace usages of DI with constructors for structures. However, this method adds another function call at runtime for initialization of structure members adding to overhead. If these variables are in global scope, then there are two issues. The first is that static (compile time) initialization is not possible, which is a requirement in case of Linux kernel. This is because when the system boots, some global system variables should be initialized before global and static constructors are called. Moreover, the order among global constructors can not be guaranteed. This means that if an uninitialized global structure variable is being used in another initialization, it may lead to system crash. Finally, this method is not suitable for array initialization and hence the issues of ranged and indexed initializations remain unresolved.

2.3. Extending g++

We examined two different approaches other than extending the C++ compiler. We found that both the approaches are difficult in implementation as well as in verification of their correctness. This is because large systems like Linux kernel have various ways in which DI are used and it is a cumbersome process to examine all of them. An example of one such usage of DI in Linux kernel is shown in Figure 1.

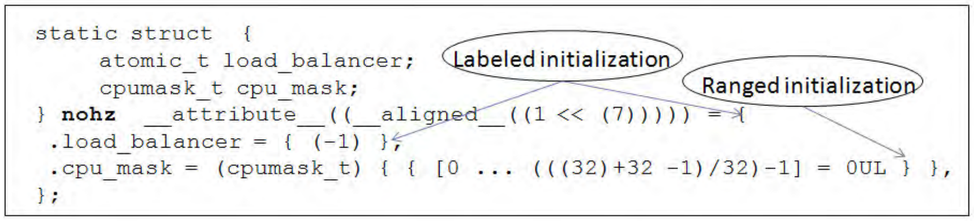
```
static struct  {
     atomic_t load_balancer;
     cpumask_t cpu_mask;
} nohz  __attribute__((__aligned__((1 << (7))))) = {
 .load_balancer = { (-1) },
 .cpu_mask = (cpumask_t) { { [0 ... (((32)+32 -1)/32)-1] = 0UL } },
};
```

Fig. 1. A precompiled code snippet from `kernel/sched.c`, the scheduler of Linux kernel. It shows nesting of labeled(`load_balancer`) and ranged initializer(`cpu_mask`) with labeled initializer(`nohz`).

The automation used for one software may not work for other softwares as the use cases may be different for them. In this way, changing the C++ compiler is a practical and easy solution to the problem. The proposed extension of g++ allows out-of-order and selective initialization of members. It also facilitates static and global initialization as it is done by gcc for any C based system. In addition, it is independent of the target software and does not involve any effort on the developer's part to apply any script or perform any kind of manual modification. Finally, as will be explained in Section 3, it is relatively simple and involves changing only 3 files of g++ source code.

3. Design and Implementation

This section explains the design and implementation of the extension to support designated initializers in C++ compiler. We have designed the extension with g++ version 4.4.5 as the base compiler. The proposed design primarily involves recognizing DI and performing corresponding semantic actions. Though this process spans across only 3 files, identifying the places to modify required careful analysis of the compiler code. The files to be modified for the implementation of the design include `parser.c`, `decl.c` and `typeck2.c`. They are part of the g++ branch of gcc compiler source tree(`gcc/cp/`). Each file represents a phase which the code being compiled passes through. In this section we explain the extended functionality of each phase to recognize all 3 DI types.

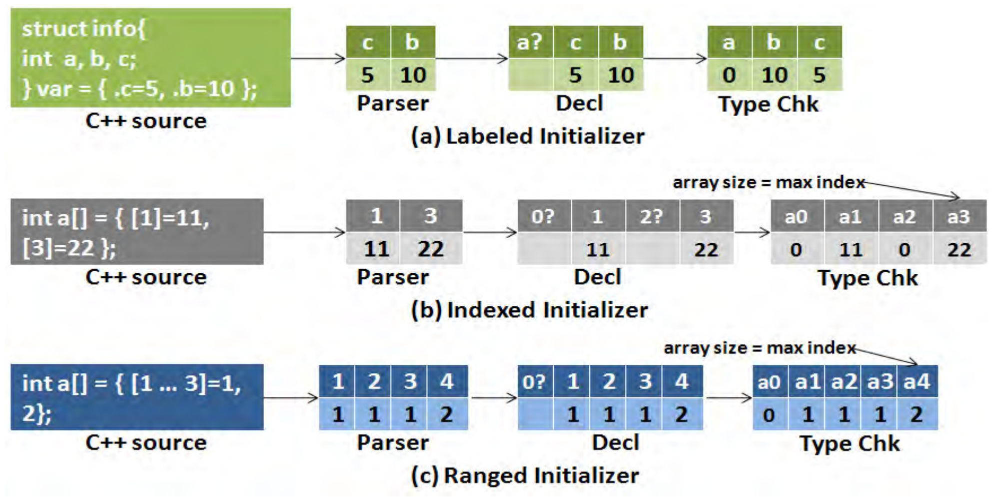

Fig. 2. The g++ extension design to recognize labeled, indexed and ranged Initializer involving parser, declarator and type-checker.

Parser: Since C++ standard does not include DI, the original parser throws syntax error when it encounters out of order labeled initializer. In order to support the same, we have added grammar rules in g++ parser. These rules recognize the signature patterns of DI and store initializer (values) and identifiers (member variables) in a list. This list is known as unprocessed vector, which is a fixed size array. The outcome of this process is shown in column 1 of Figure

2 (corresponding to the first phase). The parser passes the vector to declarator of g++.

Declarator: We have modified the declarator to process the unprocessed vector provided by the parser based on the type of initializer. The declarator validates the vector by checking if any identifiers are left uninitialized. It creates entries for uninitialized identifiers in the vector and marks them as 'erroneous' (See Figure 2, column 2). The partially processed vector is then passed on to the type checker.

Type-checker: The type checker has been refactored to consume the vector passed by the declarator and to perform final processing on the same. It infers the type of the identifiers of the erroneous entries for structures. It assigns default values to such identifiers with the assistance of back-end of the g++ compiler. The basic datatypes like numbers are assigned 0, pointers are assigned NULL, characters are assigned '\0' and boolean identifiers are assigned 0-bit. On the other hand, derived datatypes are broken into basic datatypes based on their members and the same procedure is followed.

For array (indexed and ranged) initializers, type-checker scans the initialization list till the end when explicit size is not provided. Then it allocates memory of the size according to the maximum index specified in the initialization. All erroneous entries are filled with 0. This accomplishes the processing of vector and the values are copied to the actual structure / array members as shown in the last phase of Figure 2.

4. Evaluation

The evaluation of the proposed g++ extension is divided into two parts. The first part explains how the extension reduces manual efforts by using Linux kernel as a case study. The second part presents the comparison of both original and extended compiler by measuring their performances.

4.1. Case Study: Linux Kernel

We explain the evaluation of the proposed extension to g++ in this section using Linux kernel as a case study. Linux kernel is a large software written in C, which makes extensive use of DI to initialize its global variables. We measure the increased productivity and ease of porting Linux kernel to C++ by counting the number of occurrences of DI in different subsystems.

We have extended gcc version 4.4.5 to count the number of DI instances everytime it encounters one. Table 1 shows the counter values calculated after the compilation of Linux kernel 2.6.23 with the modified compiler.

The experiment shows that more than 5600 variables are initialized using DI in core kernel and other subsystems of the Linux kernel. This result shows that making manual modifications is not a feasible solution. We have already seen how complex the usage of DI in the kernel can be, which renders scripting ineffective as an option. On the other hand, our extension only modifies 238

Table 1. Number of instances of designated initializers of different types in Linux kernel 2.6.23 as counted by the g++ extension (Core includes management of processes, timers, scheduling, etc.)

Kernel Subsystems	Labeled	Indexed	Ranged	Total
Core	750	30	109	889
Memory	165	1	2	168
Network	1415	0	225	1640
File System	509	0	41	550
Architecture	300	16	90	406
Device Drivers	1818	7	124	1949

lines (including addition, removal and modification) of the compiler source code. Thus, our exploration in modifying g++ is justified by the results.

4.2. Performance Study

Performance is one of the key concerns when a system such as compiler is refactored. In order to discover the overhead that results from the proposed extension of g++, we compared building times for 3 different systems; Linux kernel, Apache HTTPD (Web) server and PostgreSQL database. Our objective here was to ensure that performance is not sacrificed in order to gain more functionality. Additionally, this experiment verifies how the new compiler compares with the original one while compiling other systems than the Linux kernel.

The base system for the experiments consisted of Intel Core i7 quad core CPU at 2.4 GHz, 6GB RAM and Fedora 13 operating system. gcc compiler version 4.4.5 was used as original compiler and the same version was modified as explained earlier in the paper. The Linux kernel version was 2.6.23 while the versions of PostgreSQL and HTTPD were 9.2.4 and 2.4.6 respectively. The compilation times for building these systems were obtained using `time` utility of Linux to the precision of millisecond.

Table 2. Comparison of building (compilation) times (in seconds) for Linux kernel, Apache HTTPD server and PostgreSQL database

System Name	Extended Compiler	Orignial Compiler
Linux Kernel	774.060	774.039
Apache HTTPD	75.938	75.940
PostgreSQL	211.381	211.384

Table 2 summerizes the results of the experiment carried out for comparing the performance of both versions of compiler. It is evident that Linux kernel is the largest among all systems that were tested, as it takes longest to build. The extended compiler takes slightly longer time for Linux kernel than the original one since it has numerous instances of nontrivial designated initializers.

However, the overhead in this case is in order of milliseconds, which is a very small fraction of the total time taken for compiling the kernel and hence can be considered to be reasonable.

On the other hand, HTTPD and PostgreSQL are compiled in almost the same time by both the compiler versions. Again, the reason being the absence of nontrivial designated initializers. Thus, it can be observed that the modifications in the compiler do not have any adverse effect on the compilation of the software that does not utilize nontrivial designated initializers heavily.

5. Runtime Support for g++ in Linux Kernel

The Linux kernel needs built-in library support for basic operations since it is the only code in execution during the bootstrap of the system and it can not use any runtime linking for library functions. Hence certain C library functions and runtime have been included in Linux source tree at `lib/` directory. In order to include runtime support for g++, we added the necessary files from g++ source to this location. This section explains the issues that arose during this process and how they were addressed.

5.1. Volatile Typecasting of Complex Types for C++

Background: In some cases, certain compiler optimizations are a hindrance to the functional objective of the program. Usage of memory mapped I/O in Linux kernel is one such instance. In memory mapped I/O, an I/O device is mapped to a memory location. Accessing that location results in read/write operation on that device. However, compiler optimizes that location to be accessed from cache memory only and the operation does not happen on the device.

Problem: In order to stop compiler from optimizing operations on certain variables (memory locations), they are typecast as `volatile` in C. Linux kernel uses this mechanism very often. It uses `ACCESS_ONCE` macro to accomplish this task. Following is the definition of `ACCESS_ONCE` macro.

```
#define ACCESS\_ONCE(x) (*(volatile typeof(x)*)&(x))
```

This definition works well in C compiler for complex datatypes such as structures. However, this definition only works for basic datatypes in g++.

Solution: We have extended this definition to make it compatible with g++ using runtime type inference (RTTI) and reinterpret casting. Basically the central idea of the solution is as follows:
1. Infer the type of data at runtime using `typeid` construct of g++.
2. If the datatype is basic, the C style definition should be used.
3. In case the datatype is complex, reinterpret casting should be used.

The reinterpret cast, as defined by the C++ standard [4], allows casting of a pointer to any other (including unrelated) type. Additionally it ensures that if the pointer is cast back to the original type, its value is preserved. This is achieved by reinterpreting the bit pattern of the value to the target type. Thus, when used with `volatile`, reinterpret cast treats the variable as `volatile`, and directs the compiler not to apply any cache optimization on the variable.

The definition of `ACCESS_ONCE` macro, according to the proposed solution, is shown below:

```
#ifdef __cplusplus  /*g++ compiler*/
#include <iostream>
#include <typeinfo>
using namespace std;
#define ACCESS_ONCE(x)                              \
  (             typeid(x).name()[0] == 'P' ||    \
                typeid(x).name()[0] == '1' ||    \
                typeid(x).name()[0] == '2')      \
  ? reinterpret_cast<volatile typeof(x) &>(x)    \
  : (*(volatile typeof(x)*)&(x))
#else /*C compiler*/
#define ACCESS_ONCE(x) (*(volatile typeof(x)*)&(x))
#endif
```

In this definition, `typeid()` and `typeof()` are RTTI constructs, which explains the necessity of runtime support for C++ in Linux kernel. It returns ‘P’, ‘1’ or ‘2’ in case of pointers, structures and classes respectively, implying that the identifier is of complex type and reinterpret cast should be used to make it `volatile`.

5.2. Global Constructors

Background: The constructors for global and static objects are usually called by a special function named `__do_global_ctors_aux (void)`, which is inserted by g++ compiler in the linked object file. It is called before the `main()` function and thus before any possible usage of the global/static objects. Similarly, destructors are called after `main()` using a special function which is inserted by g++ compiler. In order to achieve this, a compiled object file is linked with `crtbegin.o` and `crtend.o` files. These pre-generated files are used by g++ to traverse through the given file to find global and static objects. For each such object its constructor and destructor are placed in the lists of global constructors and destructors respectively. The starting and ending of each list is denoted by g++ compiler variables `__CTOR_LIST` and `__CTOR_END` in case of constructors; and `__DTOR_LIST`, `__DTOR_END` in case of destructors. `__do_global_ctors_aux` function traverses the constructor list in downward fashion, i.e. from `__CTOR_END` to `__CTOR_LIST`.

Problem: For normal application programs, this is handled by g++ and the linker automatically. However, for Linux kernel, we have to provide this runtime support. Hence we had to add that support in Linux kernel by adding

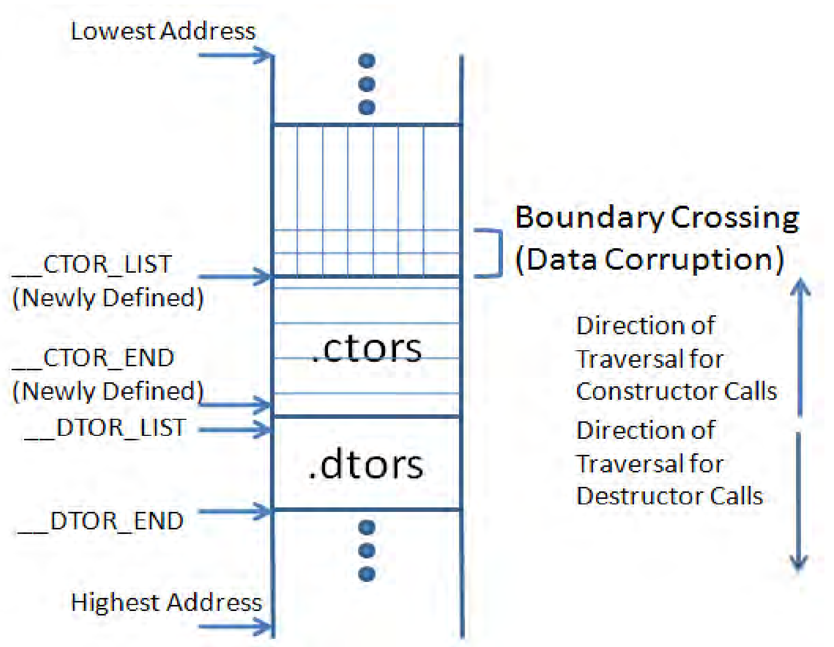
Lowest Address
Boundary Crossing (Data Corruption)
__CTOR_LIST (Newly Defined)
.ctors
__CTOR_END (Newly Defined)
Direction of Traversal for Constructor Calls
__DTOR_LIST
.dtors
Direction of Traversal for Destructor Calls
__DTOR_END
Highest Address

Fig. 3. The memory layout of vmlinux kernel image depicting boundary crossing of .ctors section and its solution

`crtstuff.c` file and `libstdc++` directory from source of g++. We also made suitable changes in the kernels makefiles at different levels, so that a C++ file can be compiled with g++ compiler. However, in the existing g++ files for that support, while traversing through constructor list, the initial boundary (`__CTOR_LIST`) was getting missed. This led to function `__do_global_ctors_aux` getting into the previous section of constructors. This resulted in execution of non-executable data and subsequently crashing of kernel.

Solution: Figure 3 shows the layout of `vmlinux`, the kernel image in the memory. It is an `elf` image that is made of different sections. We have added a new boundary for `__CTOR_LIST`, the beginning of the constructor section which `__do_global_ctors_aux` function checks when it iterates through the list. We have added this symbol as a kernel image (`vmlinux`) symbol, which g++ compiler can access in `__do_global_ctors_aux` function. The actual definition of the symbol is as follows:

```
#ifdef CONFIG_CONSTRUCTORS
#define KERNEL_CTORS() . = ALIGN(8);                          \
                       VMLINUX_SYMBOL(__ctors_start) = .; \
                       *(.ctors)                              \
                      VMLINUX_SYMBOL(__ctors_end) = .;
#else
#define KERNEL_CTORS() VMLINUX_SYMBOL(__ctors_start) = .; \
                      *(.ctors)                          \
                      VMLINUX_SYMBOL(__ctors_end) = .;
#endif
```

6. Conclusions and Future Work

This paper presented how object orientation can be supported in a large scale system such as Linux kernel by extending g++ compiler. As a part of compile-time support, a g++ extension for nontrivial designated initializers(DI) for structures and arrays was added. It handles usage of ranged, indexed, labeled and nesting of all types of DI in an application transparently. Furthermore, the paper showed how global constructors and volatile typecasting in C++ can be supported in Linux kernel. Finally the experiments proved that the proposed approach saves a lot of manual efforts with a very reasonable overhead.

We envisage that an automated tool for converting legacy systems written in C into C++ can be well appreciated by the software engineering community. At present, the proposed approach has limited features and there are still many incompatibilities between C and C++ that require addressing. In future, this tool can be made more enhanced and sophisticated by integrating the modified compiler with scripting support to tackle these issues. This tool can be used to cater to specific issues of other legacy systems, as opposed to just Linux kernel. At a later stage, this work can be extended to raise abstractions from objects to services in legacy systems. The services are more abstract than procedures or objects and hence are independent of the language they are implemented in, which can make maintenance of the legacy systems easier.

Acknowledgments. The authors acknowledge Prateek Dhawaalia, IIT Madras for his inputs in implementation of the g++ extension. The authors also thank DIT, Government of India for financially supporting this work.

References

1. ISO/IEC 9899:1990 - Programming languages - C (1989), http://www.iso.org/iso/catalogue_detail.htm?csnumber=17782
2. ISO/IEC 9899 - Programming languages - C (1999), http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899
3. ISO/IEC 14882:1998 - Programming languages – C++ (2005), http://www.iso.org/iso/catalogue_detail.htm?csnumber=25845
4. ISO/IEC 14882:1998 - Programming languages – C++ (2011), http://www.iso.org/iso/catalogue_detail.htm?csnumber=50372
5. Beck, M., Bohme, H., Kunitz, U., Magnus, R., Dziadzka, M., Verworner, D.: Linux kernel internals. Addison-Wesley Longman Publishing Co., Inc. (1996)

6. Douglas, K.: PostgreSQL. Sams (2005)
7. Fielding, R., Kaiser, G.: The apache http server project. Internet Computing, IEEE 1(4), 88–90 (1997)
8. Janakiram, D., Gunnam, A., Suneetha, N., Rajani, V., Reddy, K.V.K.: Object-oriented wrappers for the Linux kernel. Software Practice & Experience 38(13), 1411–1427 (2008)
9. Solomon, D.A., Custer, H.: Inside Windows NT. Microsoft Press, Redmond, WA, USA, 2nd edn. (1998)
10. Widenius, M., Axmark, D.: MySQL reference manual: documentation from the source. O'Reilly Media, Inc. (2002)
11. Yu, L., Schach, S.R., Chen, K., Heller, G.Z., Offutt, J.: Maintainability of the kernels of open-source operating systems: A comparison of Linux with FreeBSD, NetBSD, and OpenBSD. Journal of Systems and Software 79(6), 807–815 (2006)
12. Yu, L., Schach, S.R., Chen, K., Offutt, J.: Categorization of common coupling and its application to the maintainability of the Linux kernel. IEEE Transactions Software Engineering 30, 694–706 (October 2004)

Hemang Mehta is an MS research scholar at Department of Computer Science and Engineering, Indian Institute of Technology Madras, India. His research interests include design of operating systems, distributed systems and compilers. Specifically, his work focuses on applying principles of service oriented computing to improve the design of operating systems.

S J Balaji is an MS student of Computer Science and Engineering at Indian Institute of Technology Madras. He received a BE degree in Electronics and Telecommunication Engineering from Mumbai University in 2010. His research interests are operating systems design, distributed systems, cloud-based systems, energy aware system designs and manycore operating systems.

Dharanipragada Janakiram is currently a professor in the Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Madras, India, where he heads and coordinates the research activities of the Distributed and Object Systems Lab. He obtained his Ph.D from IIT, Delhi. His current research involves building large scale distributed systems focusing on design pattern based techniques, measurements, peer-peer middleware based grid systems, cloud bursting, etc. He is currently an associate editor of IEEE Transactions on Cloud Computing, the SIG Chair of Distributed Computing of Computer Society of India, Chair of ACM Chennai Chapter and is also the founder of the Forum for Promotion of Object Technology in India.

Received: January 15, 2013; Accepted: August 20, 2013.