ABSTRACT
It is obvious that software after its initial deployment changes over time statically or dynamically. Some software systems such as servers and safety-critical systems cannot allow for static code changes—fixing and rebooting a running program. Dynamic software updates (DSU) make it possible to change executing program code without rebooting or stopping. DSU systems can be used to modify applications on the fly, thus saving the programmer’s time and using computing resources more productively.

This paper presents a lightweight dynamic updating system which replaces currently running software using dynamic linking. This update system implements software updates in the same process memory space. Thus, the new version of software is required to be loaded into the shard memory of the existing process’s virtual memory space. The state of the old version (i.e., local and global variables) is transferred to the new one. The experimental results indicate that the proposed updating system have the potential to become an effective tool for C applications.

KEY WORDS
Software maintenance, Dynamic software updating, Safe updating points, Code changes

1. Introduction
As the terminology "Software Evolution" implies, contemporary software experiences repetitive upgrades after the initial deployment due to a variety of reasons. Error correction such as bug fixes or security patches may cause software changes. In addition, the existing functionality of software is required to change over time so that it can adjust to rapidly changing computing environment and accommodate users’ new or changed requirements. Therefore, software maintenance is as important as software development. From this point of view, software maintenance including dynamic update is one of the major research topics in software engineering. Dynamic software updates (DSU) make it possible to change executing program code without rebooting or stopping [1, 2]. DSU systems can manipulate a running process itself or create a new process for dynamic updates. The former is called in-place updating and the latter state transfer-based updating. In-place updating systems, patch files are injected into memory area of a running process [3]. Therefore, in-place updating systems need a dedicated compiler to change code and a patch generator to produce patch files. State transfer-based updating systems [3] perform updates without applying patches into a running process. In state transfer-based updating, a new version of an updated program is executed separately while an old version is running. The state of the old process is transferred to the new process.

The rest of this paper is structured as follows. Section 2 discusses existing DSU systems and Section 3 proposes a novel dynamic update system for C programs. Section 4 presents concluding remarks and outlines future work directions.

2. Related Work
Typically, changes of a running process need system-level supports. Therefore, most DSU systems are closely dependent on virtual machines and operating systems as well as programming languages. Initial DSU systems can change FORTRAN programs and a variety of DSU systems support C, C++, or Java applications.

DSU systems for C programs: Ginseng [4, 5, 6] is one of the most famous DSU systems for C applications. It consists of a dedicated compiler, a patch generator, and a runtime system. The compiler makes programs updateable and then creates executable files. The patch generator creates a patch file including the difference between the two versions of the same program. The runtime system applies dynamic updates by linking dynamically the patch file to the running program. STUMP [6] is the enhanced version of the Ginseng system to support more effectively dynamic updates for multi-threaded programs. One of the major issues for dynamic updates for multi-threaded programs is to set safe update points. STUMP presents an approach to balance safety and timeliness. STUMP automatically induces safe update points from user-defined update points.

While Ginseng changes source code of the updated program using the dedicated compiler, POLUS [7] is to
change the binary code directly. POLUS supports binary compatibility. Moreover, it can change existing binary code and modify the binary code of the running software. Most of all, the POLUS system can fix already tainted state for currently running software before applying dynamic updates. In addition, the rollback capability is provided to nullify completed updates.

UpStare [8] uses call stack reconfiguration to support immediately dynamic updates for multi-threaded programs. The stack reconfiguration provides a way to handle local variables used inside the function. Upon receiving an update request, the UpStare system stores the stack contents of the currently running process and then loads a new version of the same program. The stack of the new version is reconfigured by the stored stack values. In addition, UpStare provides an effective method of updating the running main function and long-running loops.

**DSU systems for Java programs:** Some dynamic update systems for Java applications expand the standard APIs provided by the Java virtual machine [9, 10, 11, 12]. Others implement dynamic updates as an additional service of the Java virtual machine [13, 14]. The former approach is easy-to-implement, but not flexible. In contrast, the latter approach can support a wide range of updates including field changes, method changes, and class hierarchy changes. However, it is not easy to modify the Java virtual machine. Jvolve [14] enhances the existing Java virtual machines to support flexible and practical updates for multi-threaded Java applications. It also mitigates performance degradation which can occur during dynamic updates. For dynamic updates, Jvolve sees if the bytecode of the updated program supports safe type changes and determines safe update points where updates could be applied by examining the stack of the running thread.

**DSU systems for operating systems:** Dynamic software updates can be applied to change operating system kernels [15, 16] being used as well as high-level applications.

K42 [15] is an operating system developed by IBM and supports the dynamic update feature. In K42, system resources are considered as a set of object instances. They include a region of virtual memory, network connections, files, processes, etc. The object is encapsulated as an independent unit, separated by the interface and the implementation part. When applying dynamic updates, the implementation part can be changed easily. Since the execution time of the kernel thread is not long and the kernel thread runs in a non-blocking mode, the update of the kernel is not difficult in K42. According to update requests, an intermediate object between a thread and a callee object blocks the incoming calls from the new threads and waits until the execution of the running thread finishes. When the callee object reaches a quiescent state, the requested update takes place. After updating, incoming calls are passed to the new version.

Compared to other update systems, the proposed update system provides a relatively light-weight approach to update C applications. The update system relies on dynamic linking in Linux and minimizes the total elapsed time of dynamic updates. Therefore, the proposed update system may be applicable for embedded applications as well as server programs.

### 3. Proposed DSU System

In the following, the proposed DSU system is described in detail. The key design considerations of the update system are outlined and then implementation issues are described.

#### 3.1 Dynamic Linking for Updates

To implement software updates in the same process memory space, the new version of software is required to be loaded into the shared memory of the existing process's virtual memory space. The proposed update system makes it possible to change a running process by using dynamic linking functionality provided on Linux. For dynamic updates, the updated programs should be compiled into dynamic linking libraries (DLLs). Figure 1 illustrates the differences between static linking and dynamic linking. The major difference between them is the loading time when the library is bound to a program. While the static library provides some functionality that is bound statically at compile time, the dynamic library is loaded when an application is loaded and is bound at run time. Each running program has its own copy of the static library. Thus, the same static libraries can be loaded and executed. In contrast, the dynamic library is shared among multiple programs. The programmer can dynamically link a program with the shared library and load the library upon execution. The program can selectively call functions with the library in a process.

The running process can load the updated programs (i.e., DLL files) during program execution. The Linux operating system provides the GCC compiler to compile C programs into DLL files. The GCC compiler is also used to link these DLL files during program execution. The Linux operating system also allows the existing process to load and execute DLL files. With the help of the Linux operating system the running process can load DLL files into the shared memory and then figure out the address of a particular function included in the DLL file. The running process performs a function call to execute the found function. To support these tasks, the Linux operating system provides a series of system calls such as `dlopen`, `dlsym`, and `dlclose`.

769
3.2 System Structure

Figure 2 shows the basic steps to inject update management code to make target programs updateable. In most cases, an initial version of the target program is not ready to become updateable. It is common that software developers make more efforts to satisfy functional requirements of the program rather than non-functional ones during development. Similarly, dynamic updateability is one of the non-functional requirements and may not be of interest in software development. Thus, initially deployed software cannot be updated at runtime. To address such an issue, preprocessing shown in Figure 2 is required to allow software to be updated on the fly. The preprocessor automatically injects update management code such as safe update points, code for update log, system library calls, etc. For the fully automation, a parser is needed so that source code of the updated program can be analyzed. The preprocessed program becomes updateable. The compiler and linker produce shared libraries (.so files in Linux) from the updateable program. The shared libraries are deployed on the execution environment and can be updated during operation. Code changes by the preprocessor should not modify the original functionality of the updated program. In addition, the preprocessing step does not cause considerable performance degradation.

Figure 3 shows the system structure of the proposed dynamic update system. The update system consists of a preprocessor, a code transformer, an update runtime, and tools. The preprocessor is closely related to the code transformer—they read source code of a target C application and then insert update management code in order to make the target program updateable. The preprocessor marks update points where dynamic updates can take place. The current version of the proposed system does not support a static analysis to extract safe update points. The code transformer changes source code of the target application. The added code is used to seamlessly transfer the state of the target application. The values of local and global variables are transferred into a new process for safe dynamic updates. For such a state transfer, the original source code is required to be modified. However, the functionality of the original program remains the same. The update runtime applies requested updates to the running process at runtime. It loads a new version of the same program into the shared memory area. At this point, the update runtime relies on dynamic linking facility on Linux. Once the new shared library has been loaded, the stack of the new process is configured by restoring the values of the old process. The new process can access the global variables of the old process. The new process starts to execute at the point where the update request takes place. The tools of the proposed update system provide some useful functions for
users or developers during dynamic updates. The tools include update commands, logging, and libraries.

Upon receiving a user’s update request, the update system waits until the currently executing process arrives at a predefined update point. At the update point, the update system loads a new version, transfers states between old and new versions, and unloading the old version.

![Figure 3. System structure](image)

### 3.3 Updating Global and Local Variables

For dynamic updates, an updated program needs to be converted into an updateable version. The source code transformer injects update management code into the updated program. The update management code is used to set safe update points and to update local as well as global variables.

The proposed update system provides a function transformer and a global variable transformer. The function transformer modifies functions so that they can be ready for updates of their local variables. The function transformer reads a C program function and then makes it possible to be updateable by adding stack unroll and roll functionality. The input data of the function transformer includes function prototypes, local variables, and function calls.

The global variable transformer takes a C program to be updated and then analyzes the global variables. The initial version of the global variables is collected and is stored into DATA and BSS memory areas.

The proposed update system assumes the old and new versions of the updated program are almost the same, but some of the functions and global variables can be different. The proposed update system stores the global variables of the old version into the DATA or BSS section of the virtual memory area. The update system allows the new version to access these global variables of the process’s virtual memory space of the DATA or BSS area. The functions of the new DLL are able to access to existing process’s data or global variables. When a program dynamically links DLLs, these DLLs do not have global variables, but could use the global variables in the BSS or DATA area as if they were declared in the DLLs. The new global variables of the new version cannot be added into the DATA or BBS area. They are stored on the heap memory area.

The proposed update system can unroll and roll the call stack of the C function so that it can deliver the local variables of the old version to the new one during dynamic updates. When the old version of the function performs an update point, the update system saves the contents of the call stack. And then the new version of the function is loaded and its call stack is constructed. The update system copies the stored contents of the old stack to the new stack. The new version continues to perform at the location where the old version of the function has been suspended for the dynamic update.
In this approach, the old and new versions of the updated program are modified at the source code level. When the old version reaches a predefined update point, the stack frame of the old version is unrolled and saved one by one. After the new version is loaded, its stack is configured by the stored stack frame. Typically, source code for stack unrolling and rolling is injected at the entry part of the function and right after of the function call.

Table 1 shows the code changes for updating global variables. The global variables in an old version (i.e., prog_v1.c) are declared as "extern" in a modified version. In addition, the global variables in a new version (i.e., prog_v2.c) are stored on the heap memory area. To do so, the code transformer makes the global variables pointer variables.

Table 1. Updating global variables

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Modified Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>// the original version of prog_v1.c</td>
<td>// the modified version of prog_v1.c</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>int x; //global variables</td>
<td>extern int x;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| // main.c | ...
| ... | int x; //global variables in prog_v1.c
| ... | ...
| int main(int argc, char * argv[]) | int main(int argc, char * argv[])
| { } | { } |
| // prog_v2.c | // prog_v2.c |
| ... | ... |
| extern int x; //global variables in prog_v1.c | |
| // double y; //original code | // double y; //original code |
| double *y_ptr; //modified code | double *y_ptr; //modified code |
| ... | ...
| void add_global_variables(void) | void add_global_variables(void)
| { | {
| y_ptr = malloc(sizeof(double)); | y_ptr = malloc(sizeof(double));
| ... | ... |

The code transformer of the update system inserts unrolling and rolling code for stack reconstruction shown in

Table 2. Updating local variables

```c
// prog_v1.c

foo() {
  if (check_stack_unroll() == 1) {
    return 0;
  } else if (check_stack_roll() == 1) {
    // roll the stack
    ...
    goto start_bar;
  }
  ...
  start_bar:
  bar();
  if (check_stack_unroll() == 1) {
    // unroll the stack
    ...
    return 0;
  }
}
```

To demonstrate the proposed updating system, it is applied to update a real-time video recording program at runtime. The video recording program shoots video and sends it to a remote client computer. For this case study, the two versions of the video recording program are used. They should be made in the format of a shared library. One version is to support a lower resolution and the other is a higher one. The video resolution determines the quality of the recorded video. The proposed updating system unloads a running video recording program and loads the other. After a successful update, one can find that the video output resolution has been changed. The experimental results indicate that the proposed updating system can be applied to update typical programs which are developed without considering dynamic updateability at development phase. The last row of Table 3 presents the exact statistics of the changes involved.
To update global variables in Table 3, the proposed update system uses a symbol table to store and restore them. For this case, two new functions are added and one function is changed. All added global variables are stored in heap memory area via malloc().

The second case presented in Table 3 is to update a file I/O program to demonstrate the state of the operating system is preserved after updating. The original version of the file I/O program starts to read a text file. According to an update request, the update system loads the new version of the same program into the same memory space where the original version has been loaded. Once the update has been completed successfully, the new version of the file I/O program will read the input text file in succession. For this case, two functions are changed and one new global variable is added.

To update the real-time video recording program, three global variables are changed. The update system changes a global variable which is used to store the resolution of the program. It also changes the size of the video frame at runtime.

4. Conclusion and Future Work

This paper has presented a lightweight dynamic updating system which replaces currently running software. For dynamic updates, the proposed system uses dynamic linking functionality provided by the Linux operating system. This update system implements software updates in the same process memory space. It also supports code change and state transfer between old and new versions. The initial results of the case study indicate that the proposed updating system can be applied to update C applications on the fly.

As future work, it would be interesting to apply the proposed update system to update real-world long-running applications. In addition, the performance of the update system could be improved by ruling out unnecessary or non-optimized code. Finally, it is worth comparing the proposed update system with existing DSU systems in terms of update performance and safety.

Acknowledgement

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012R1A1A1006022).

References