How to Have Your Cake and Eat It Too: Dynamic Software Updating with Just-in-Time Overhead

Rida A. Bazzi Bryan Topp
School of Comp. Inf. and Dec. Sys. Engineering
Arizona State University
Tempe, AZ 85287
Email:{bazzi, betopp}@asu.edu

Iulian Neamtiu
Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521
Email:neamtiu@cs.ucr.edu

***Abstract*—We consider the overhead incurred by programs that can be updated dynamically and argue that, in general, and regardless of the mechanism used, the program must incur an overhead during normal execution. We argue that the overhead during normal execution of the updateable program need not be as high as the overhead for the updated program. In light of the fundamental limitations and the differences in the overhead that must be incurred by the updateable and updated programs, we propose a new mechanism for dynamic software update based on a new shifting gears approach. The mechanism attempts to incur just the required overhead depending on the stage of update the application is in. Before an update the execution incurs low overhead and when an update occurs the execution incurs higher overhead which reverts to low overhead as the execution progresses. We evaluate the mechanism by modifying an application by hand. Preliminary performance numbers show that the mechanism performs better than existing mechanisms for dynamic software update.**

I. Introduction

Upgrading deployed software, whether for adding functionality or fixing bugs, is a significant part of the software lifecycle. Upgrading software typically results in substantial downtime needed to stop the old application and load and start the upgraded (new) application. For applications that cannot tolerate the interruption associated with traditional software upgrades, dynamic software update (DSU) offers the possibility of replacing the running application *in-memory* without the need for relinquishing system resources or terminating application processes and threads. Existing works on DSU concentrate on providing the system mechanisms to induce the update: suspending execution without stopping the application processes, copying the state from the old to the new version, and starting the new version [1], [2], [3].

Existing application-level mechanisms for dynamic sofware update use DSU compilers which instrument the application source code so that it is updateable at runtime. These mechanisms introduce substantial overhead due to indirection introduced by the mechanism, the loss of optimization opportunities by the compiler, as well as the changes in cache locality due to instrumentation by DSU compiler [4], [2]. While some existing low-level mechanisms can avoid that overhead, they are architecture dependent and harder to maintain, but more importantly they are not general mechanisms for dynamic software update. In this paper, we argue that any general mechanism for dynamic software update must suffer some unavoidable overhead unless knowledge and control of the compiler is assumed by the mechanism. This overhead must be introduced by dynamic software update mechanisms even for update systems such as Ekiden [5] which aim to reduce the overhead by restricting the generality of the mechanism and relying on the programmer to make up for the loss of generality.

By analyzing the overhead that must be introduced by current dynamic software update systems, we realized that we could develop general update mechanisms that incur high overhead only after an update has been initiated and have little to no overhead during normal execution (i.e., before the update is initiated and after the update has completed) The solution uses a novel shifting gears approach. The idea is to run in high gear (low overhead) during normal execution and only shift to low gear (high overhead) just before the update. Once the update is done, the execution shifts up to high gear as the execution progresses. To evaluate the overhead introduced by our approach, we performed a preliminary implementation of this scheme where we added the instrumentation manually, rather than automatically by a compiler, on an example program to show that the overhead can be greatly reduced during normal execution. We use as an example, the KissFFT program and show that the overhead due to instrumentation can be reduced to less than 10% during normal execution. This is significantly lower than the overhead introduced by the instrumentation of Upstare and Ginseng which can range from 40% to more than 100% depending on the platform and compiler used.

In summary, the paper has two main contributions:

1) An analysis of the fundamental limitations on the overhead introduced by general mechanisms dynamic software updates.

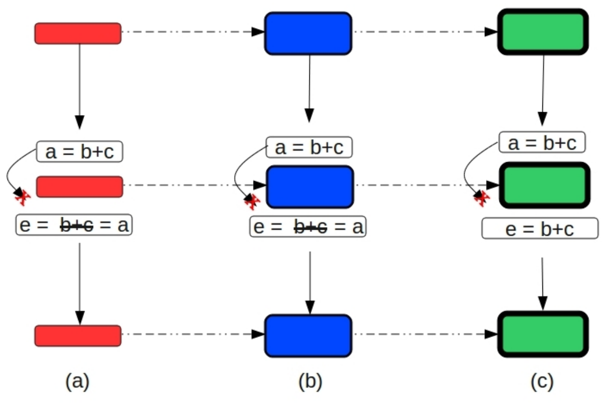

Figure 1. *Effects of DSU on optimization of code: (a) code motion not allowed across update point, but optimizations across update point are allowed; (b) same as (a) but instrumented code is larger which affects instruction cache locality; (c) optimizations not allowed across update point. This is especially problematic inside loops.*

2) A new dynamic software update approach that introduces less overhead than other existing approaches.

The rest of the paper is organized as follows. In Section II, we study the fundamental limitations on the performance of dynamically updateable programs. In Section III, we propose a new DSU mechanism that minimizes the overhead through a novel shifting gears approach. In Section IV we present preliminary performance results that show that our approach can reduce the overhead of dynamic software update. In Section V we compare our approach to related work.

II. Effects of DSU on performance

In this section, we argue that in general, providing dynamic software update must incur an overhead during the normal execution of the updateable program as well as the updated program. Interestingly, the overhead must exist independently of the type of mechanism used, so it applies to general systems such as Upstare and Ginseng [2], [4] as well as systems that explictly try to avoid the overhead such as Ekiden [5].

To simplify the discussion, we assume that the program consists of only one function, possibly with local variables. In application-level dynamic software update, a number of points in the old version are marked as potential update points—we call these points “potential” because, depending on which functions are on the stack at the time the point is reached, the update could be unsafe. Without loss of generality, we assume that the update is safe at the next update point.

When the update point is reached, dynamic software update requires that the execution state of the old version be mapped to a state of execution of the new version of the program (or to a hybrid version consistent with both the old and the new version). A state mapping has two components: saving the state of the old version and constructing a state of the new version. We discuss the effects of each on performance.

A. *Saving the state*

Mapping the state of the application requires that, at the time an update is effected, the application-level variables be accessible in order to be able to use their values to construct a state of the new version of the application. So, any update mechanism whether applied at the application level or at the executable level must have access to the application-level variables. This rules out the possibility for the compiler to optimize-out application level variables that might be needed to construct the state of the new version. Also, at the update point, the source level code that appears before the update point should be fully executed. Otherwise, we cannot have guaranteed semantics for the program state at the update point. This rules out any compiler optimizations that move code across the update point. Therefore, saving the state puts the following two restrictions on the compiler:

1) Application-level variables must be accessible and may not be optimized out at the update points
2) No code motion is allowed across the update point.

Otherwise, the compiler can optimize across an update point as show in Figure 1 (a). In the figure, the red colored rectangles represent update points. As shown in the figure, the compiler can optimize the expression $e = b+c$ to $e = a$ because that does not affect the saved values. The compiler cannot move $a = b + c$ to after the update point because the value of a captured at the update point should correctly reflect the semantics of the program.

The first execution, (a), shows only the effect of update point and assumes that these update points do not add to the size of the code. In the second execution, (b), the update points are shown to be larger because in general code needs to be added to the application in order to save the state. Adding that code can affect the cache locality of the application.

B. *Constructing a new state*

To construct the state, the programmer must be able to associate a point in the execution of the old version with a point in the execution of the new version. This requires that the point in the execution of the new version, which is specified statically, must be a barrier to any code motion so that update does not alter the application-level semantics. Also, since the state of the new version will be constructed at that point and code that reconstructs the state needs to be part of the new version, the compiler cannot tell whether the values before the barriers are the same as after the barriers. This would rule out any optimizations across the barrier. This situation is shown in Figure 1 (c). In the figure, the compiler cannot tell that $e = a$ after the barrier because the values of a, b, and c are read-in at the point of update.

```
a = b * c + d;      t = b * c;      t = b * c;
                    a = t + d;      a = t + d;

// update           // update       // update

e = b * c + f;      e = t + f;      t = b * c;
                                    e = t + f;

      (a)              (b)             (c)
```

Figure 2. Code example showing that the compiler cannot ignore the presence of code that reconstructs application-level state of the program: (a) source code; (b) incorrectly compiled code with optimization; and (c) compiled code with no optimization.

This overhead is worst case overhead, but it must be incurred at update points. Here, it might be tempting to think that it would be enough to "instruct" the compiler to ignore the code that constructs the new state when optimizing in order to reduce the overhead. Unfortunately, that can lead to incorrect code as we show below. So, the loss of optimization is a fundamental cost that cannot be avoided.

To understand why instructing the compiler to ignore the code that reconstructs the state is not a viable solution, consider the code snippet shown in Figure 2. The source code is shown in (a). In the compiled code (b) the value of e is calculated using a temporary variable t and program variable f. The cause of the difficulty is the fact that t is not visible at the source level. If the state is reconstructed at the update point, the values of a, b, c, d, and f before the update point would be correct but e would not be correctly calculated because t is not part of the program-visible state. The compiler would have to compile the code so that t is recalculated as shown in (c). So, constructing the state puts the following three restrictions on the compiler:

1) Application-level variables must be accessible and may not be optimized out at the update points
2) No code motion is allowed across the update point.
3) No optimization can be made across update points.

In summary, dynamic updates at the application level would require a loss of optimization opportunities regardless of the mechanism being used. Our study of the fundamental constraints that dynamic software update introduces led us to a new approach for dynamic software update that incurs just the right amount of overhead. This is discussed next.

III. Shifting Gears: A New Approach for Dynamic Software Update

In this section, we present our approach to reducing the overhead of dynamic software update. The approach relies on the observation that we can run the system in high gear in the common case (pre-update and post-update) and only switch to low gear for a brief period around the update.

Essentially, the system operates in high gear, with the only overhead being mainly due to instrumentation used to save the execution state when an update is required. When an update must be performed, the state of the old version must be saved and the state of the new version must be constructed. The new version will be instrumented so that it can construct its state from the saved state of the old version. Finally, since the new version itself should be able to save its state when an update is required, the new version will first be loaded with ability to both save and restore its state. Note that a fully instrumented new version will incur high overhead in the worst case due to the presence of instrumentation to restore the state. So, there is a need for a way to eliminate the instrumentation that restores the state and replace the fully instrumented new version with a version that only contains instruentation to save the execution state. We will argue below that, in general, it is not possible to remove full instrumentation while a function is active. Instead we propose that as functions exit, they are replaced with versions that are instrumented only to save the state. As a consequence, all but the longest running functions (e.g., `main()`) can be replaced not long after an update is completed. To summarize, the stages for an update are:

1) Initially, code is instrumented to save the execution state.
2) When an update occurs, the execution state is saved.
3) A new version with the ability to both save and restore the execution state is loaded.
4) As functions exit, they are replaced with versions that are less heavily instrumented, that can only save the execution state.
5) When a subsequent update is required, the state of execution is saved, and go to stage 3.

A. Compilation and Gear Shifting

Normal operation (pre-update, high gear): During normal operation, the instrumented code must ensure that all application-level variables are available at an update point and that the update point be a barrier against code motion. Also, the instrumentation should increase the code size as little as possible.

To achieve these goals, we use the instrumentations illustrated in Figure 3 (which is an example of the instrumentation that we hand coded). All local variables of the function are encapsulated inside a structure and all variable accesses are transformed to accesses of the corresponding fields in the structure. For each function there is a corresponding `struct` type for the structure that holds the local variables of that function.

At the update point, the local variables are saved in a global `stack` variable. Each entry in `stack` (`stacknode`) will contain the local variables of a function that is active on the execution stack. A `stacknode` is large enough to hold any set of local variables. To save space, it is declared as a union of all possible local variables struct types. The copying of the local variables is achieved with a simple statement which assigns the local structure to the

```
typedef struct {            typedef struct {
    int _t;                       int t;
    int _r;                       union {
    int p;                           loc1t  loc1;
    int floor_sqrt;                  loc2t  loc2;
} loc9t;                             ...
                                     loc9t  loc9;};
                            } stacknode;

void kf_factor(int n,int * facbuf)
 {
   loc9t loc;
   loc._t = 9;  // tag

  loc.p=4;
  loc.floor_sqrt = (int)floor (sqrt (n));

  /*factor out ...  */
  do {
              if(__update_req) {
                  loc._r = 91;
                  stack[sp++].loc9 = loc;
                  return;
               }
    ...

  } while (n > 1);
```

Figure 3. Instrumentation to save state.

```
 void kf_factor(int n,int * facbuf)
 {
   loc9t loc;
   loc._t = 9;
   if (__recover) {
     switch (stack[sp].loc6._r)
     {  case 91: goto recovery91;
        case 92: goto recovery92;}
   }

   loc.p=4;
   loc.floor_sqrt = (int)floor (sqrt (n));

  /* factor out ... */
  do {       if(__update_req) {
                  loc._r = 91;
                  stack[sp++].loc9 = loc;
                  return;
               }
recovery91:   if(__recover) {
                 loc = stack[sp--].loc9;
              }
    ...

  } while (n > 1);
```

Figure 4. Instrumentation to save and restore state.

corresponding field in the union. In addition, the update point is stored in that structure as an integer identifier. It can be seen that the amount of instumented code is minimal: (1) store update point; (2) copy local variables to stack; and, (3) return to caller.

When an update is invoked, one by one, the functions will store the local variables in `stack` and when the `main()` function is reached, `stack` contains a copy of the execution stack (minus function parameters) and at that point a state mapping can be done but that does not interfere with the normal execution of the program.

During update operation (low gear): When an update occurs, the saved state needs to be mapped to a state of the new version. This requires a new version that can restore and save the state be loaded, and the state mapping be applied. This can be done in any way required (similar to the way it is done in UpStare for example) as long as it is done outside the main execution path. Naturally, code introduced by the instrumentation (in the main execution path) should be kept at a minimum. The instrumentation is illustrated in the hand-coded example of Figure 4.

Normal operation (post-update, high gear): After the state of the new version is retored (or constructed), we would like to get rid of the instrumentation that restores the state because that instrumentation introduces a lot of overhead. We propose that such an elimination can be achieved as functions exit. When a function is called again, a non-fully instrumented version is used.

B. Reducing the Overhead Further

One might hope to further reduce the overhead by eliminating the code that restores the state while a function is active. We argue that in general such an elimination is not possible. Essentially, we would like to be able to map the state of execution of a function with full instrumentation to the state of execution of a function with less instrumentation (and hence more optimization).

The difficulty of mapping the state of a fully instrumented to that of a less instrumented function (that only saves state) has to do with the differences in optimizations in general. Revisting the example in Figure 2, we see that in general replacing an active fully instrumnented function with a less instrumented function can result in the loss of temporary variables that are not visible at the application-level. Since it is not possible to assign values to these temporaries at the application level, we cannot expect an application-level mechanism to be able to achieve the replacement.

IV. Experiments and Conclusion

To test our approach, we coded the instrumentation by hand. We tested the instrumentation on Kiss FFT [6], the bête noire of dynamic software update systems due to its dependence on compiler optimizations for performance. We had two versions of the instrumentation; one version could save the state of execution and one version could both save and restore the state.

We did the testing on two machines: (1) Intel DUO 2Gb, 1.6 Ghz machine and (2) Xenon 4Gb, 2.8 Ghz. The results (additional overhead compared to the uninstrumented code) are shown in Table I. The results show a wide gap between the ovehead of instrumentation that saves the state and instrumentation that saves and restore the state. At around 10% overhead for this CPU-bound application, one can expect most applications to have considerably less overhead

	Instrumented to Save	Instumented to Save/Restore
Pentium Duo	12.68%	40%
Xeon	8.65%	39.42%

Table I
OVERHEAD FOR KISS FFT.

when executing in high gear. The result for the save-and-restore instrumentation are also competitive with those of Ginseng and Upstare.

The experiments do not measure the overhead for an application after some, but not all, fully instrumented functions are replaced with non-fully instrumented versions. We argue that such functions will not be a source of high overhead for well-written applications, since they will normally be dispatch functions that do not have much computation. Also, our discussion in the previous section shows that it is not possible to eliminate such full instrumentation with an application-level update mechanism.

In summary, we have studied fundamental limitations on the overhead introduced by dynamic software update and proposed a new scheme that preliminary experiments show to be a viable approach for low overhead dynamic update.

V. RELATED WORK

We discuss the most relevant related works. Ginseng [4] transforms off-the-shelf C programs into C programs that can be updated on-the-fly via two main techniques: type wrapping and function indirection. Ginseng allows a wide range of updates to C programs, however, the indirection and type wrapping impose a permanent performance overhead on programs compiled with Ginseng. This overhead ranges from 32% for I/O bound programs to 129% for the KissFFT CPU-bound program.

Kitsune [7] is a whole program-replacement DSU systems for C, i.e., it updates a program by starting the new version from scratch and transferring the state from the running version. In Kitsune state migration is automatic by default for global variables, but other variables must be marked manually for migration. Control migration means indicating update points, just like in our approach. Some form of stack reconstruction is manually achieved by the programmer who "must write code to direct execution back to the equivalent spot in the new program". The low overhead achieved by Kitsune (around 2% for the applications considered) does not contradict the results of this paper. We argued that for variables that need to be saved or restored, compiler optimizations *will be inhibited*. Variables that do not need to be saved or restored can still be optimized out. General update mechanisms like Ginseng and Upstare are overly conservative and instrument the code under the assumption that all variables need to be saved and restored. The numbers we provided for the KissFFT application can be considered as extreme because the application is highly sensistive to compiler optimizations and we instrumented it so that the whole state can saved and restored in any loop in the program. In contrast, for the updates applied using Kitsune, not all variables are saved and restored and stack reconstruction is done selectively which reduces the overhead. Also, Kitsune manages to reduce the size of instrumented code which reduces the effects of loss of cache locality.

Lucos [8] is an approach for applying dynamic updates to the Linux kernel using Xen virtualization. Lucos employs binary rewriting and paging to update functions and types. The performance overhead is less than 1% when dynamic updates consist of applying small patches to the Linux kernel. Polus [1] is a user-space follow-up to Lucos: calls to updated functions go to the latest version, but active functions continue to execute at the old version. Old data and new data may coexist, and maintaining coherence between them is relegated to the programmer. The performance overhead ranges from 5% (VsFTPd) to 30% (Apache).

UpStare [2] uses *stack reconstruction* to allow an actively running function to transition to a corresponding point in the new version of the same function when an update is applied. This technique has the same effect as Ginseng's code extraction, but is more flexible, as transition points can be specified at patch time, not deployment time.

Acknowledgments. This research was supported in part by NSF grants CSR-0849980 and CCF-0963996. We thank the anonymous referees for their helpful comments on this paper.

REFERENCES

[1] H. Chen, J. Yu, R. Chen, B. Zang, and P. Yew, "POLUS: A POwerful Live Updating System," in *ICSE'07*, pp. 271–281.

[2] K. Makris and R. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *Proceedings of USENIX Annual Technical Conference*, 2009.

[3] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis mutandis: Safe and predictable dynamic software updating," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 4, p. 22, 2007.

[4] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," in *PLDI'06*, pp. 72–83.

[5] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster, "State transfer for clear and efficient runtime upgrades," in *Proceedings of the Workshop on Hot Topics in Software Upgrades (HotSWUp)*, Apr. 2011.

[6] M. Borgerding, "Kiss FFT," http://sourceforge.net/projects/kissfft/.

[7] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for C," Jan. 2012, http://www.cs.umd.edu/~mwh/papers/hayden11kitsune.html.

[8] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew, "Live updating operating systems using virtualization," in *VEE '06*, 2006, pp. 35–44.