Shortening Time-to-Discovery with Dynamic Software Updates for Parallel High Performance Applications

Dong Kwan Kim, Eli Tilevich, and Calvin J. Ribbens

Center for High-End Computing Systems (CHECS)
Dept. of Computer Science, Virginia Tech
Blacksburg, VA 24061, USA
{ikek70,tilevich,ribbens}@cs.vt.edu

Abstract. Despite using multiple concurrent processors, a typical high performance parallel application is long-running, taking hours, even days to arrive at a solution. To modify a running high performance parallel application, the programmer has to stop the computation, change the code, redeploy, and enqueue the updated version to be scheduled to run, thus wasting not only the programmer's time, but also expensive computing resources. To address these inefficiencies, this article describes how *dynamic software updates* can be used to modify a parallel application on the fly, thus saving the programmer's time and using expensive computing resources more productively. The net effect of updating parallel applications dynamically reduces their *time-to-discovery* metrics, the total time it takes from posing a problem to arriving at a solution. To explore the benefits of dynamic updates for high performance applications, this article takes a two-pronged approach. First, we describe our experience in building and evaluating a system for dynamically updating applications running on a parallel cluster. We then review a large body of literature describing the existing state of the art in dynamic software updates and point out how this research can be applied to high performance applications. Our experimental results indicate that dynamic software updates have the potential to become a powerful tool in reducing the time-to-discovery metrics for high performance parallel applications.

Key words: dynamic software updates; high performance applications; binary rewriting; HotSwap

1 INTRODUCTION

Among the most challenging computing application domains is parallel programming for distributed memory multiprocessors. Such systems range from compute clusters to ad-hoc grids, but fundamentally they coordinate a collection of distributed computing resources to solve a computationally-intensive problem in

parallel. Distributed memory multiprocessors help solve important computational problems in science and engineering, in domains including scientific simulation, image processing, and bioinformatics.

Writing software for distributed memory multiprocessors has been notoriously difficult due to a variety of factors. Parallel programming models over distributed memory abstractions are difficult to utilize effectively to achieve good performance. Distributed coordination is challenging and error-prone. The runtime behavior of a parallel program is difficult to predict from looking at its source code. Finally, applications in this domain are often written by domain experts—scientists and engineers—who are extremely knowledgeable in their respective domains but may lack a deep understanding of computing or experience with modern developments in software engineering.

The raison d'être of parallel high performance computing is to reduce *time-to-discovery*, the total time it takes from posing a problem to arriving at a solution. This metric is the sum of the time it takes to run an application and the time it takes to develop and fine-tune it. While high performance computing researchers have traditionally focused on reducing the time to run applications, another avenue for reducing time-to-discovery is to apply solid software engineering principles, novel techniques, and advanced tools [45, 8, 24]. That is, the software engineering approaches traditionally used to improve the construction and maintenance of traditional software can also benefit high performance software.

In particular, this article explores how *dynamic software updates (DSU)*—an advanced software engineering approach for updating software while it runs—can be applied to high performance applications, thus reducing their time-to-discovery. The idea behind dynamic software updates is simple. While a program is running, the programmer changes the program's source code, compiles the program into a binary representation, and then uses a dynamic update system to replace the running binary representation with the updated one. Implementing a safe and efficient dynamic update system is strewn with challenges, including the need to replace a binary representation of a program while preserving its runtime state as well as dealing with various features of the underlying programming language and the runtime environment.

Dynamically updating parallel high performance applications presents its own set of challenges. A typical parallel application includes multiple concurrent tasks executed on different processors, each running at its own pace and only periodically communicating with other tasks. These concurrent tasks must be updated consistently, which is non trivial. In particular, maintaining consistency during a dynamic update of multiple concurrent processes requires a distributed coordination protocol. This protocol must not only ensure some runtime invariant (e.g., no two divergent versions of a task are running simultaneously) but also do so without imposing an undue performance overhead on the parallel program.

Despite these challenges of applying dynamic software updates to a parallel high performance application, we foresee that this advanced software engineer-

ing approach is capable of drastically improving how we build and fine-tune applications in this important domain. High performance applications running on large numbers of processors are difficult to develop incrementally, as they are often time-consuming to deploy; the typical *try-change-try again* development cycle does not fit well for applications in this domain. In particular, any code change involves stopping the execution, changing the program, re-deploying the changed program, and re-starting the computation anew. In a traditional high-performance computing (HPC) environment, all these actions can be quite time-consuming and disruptive. Besides, this development model may not utilize expensive computing resources most effectively, wasting valuable processing time.

Furthermore, two important trends in HPC applications exacerbate the cost of starting and restarting a computation: *grid computing* and *real-time simulation*. In the case of grid computing, restarting a computation from scratch requires repeating several steps, all of which are expensive and some of which may be impossible (i.e., resource discovery and reservation, resource allocation, data staging, data streaming, and job launching). In the case of real-time simulation, "system-level" simulations, which use HPC resources, often have hard-to-anticipate requirements and real-time constraints, thus necessitating dynamic code updates to avoid stop/restart cycles. Examples of such computations include hurricane modeling, infrastructure and environment monitoring, epidemic modeling, and personalized medicine.

We believe dynamic software updating offers a pragmatic approach that can save human time and computing resources. With dynamic updates, one could change parameters, specify a different precision, or even switch to a different model or algorithm on the fly, observing the results of the updates in near real time. All of these can significantly reduce the time-to-discovery for high performance applications.

Having studied a large body of research on various aspects of DSU, we find that this advanced software engineering technique can provide the following benefits for HPC applications:

- **Satisfying new or changed user requirements on the fly**
  If the initial requirements change while a high performance application is executing, the computation must be interrupted, so that the code can be changed to reflect the new requirements. By contrast, dynamic software updates do not require interrupting the computation by enabling code changes on the fly.
- **Reducing software downtime**
  Off-line software updates result in software downtime—an application provides no services during updates. In HPC, such downtime is undesirable, as HPC resources are quite expensive to operate, with large electricity and cooling costs. Dynamic software updates can reduce software downtime, thereby avoiding the loss of intermediate data and using computing resources more productively.

– **Avoiding the re-deployment hassle**
Deploying HPC applications often requires interacting with a scheduler, a software module that allocates the required computational resources and schedules the execution. In heterogeneous grid environments, effective scheduling can become particularly onerous. In any case, re-deploying an HPC application imposes an additional burden on the user. Dynamic software updates can eliminate the need to re-deploy the application by applying the required changes on the fly.

This article leverages and expands on our recent research, which has put forward a novel approach to enabling flexible and efficient dynamic updates for high performance grid applications deployed on the Java Virtual Machine (JVM) [30, 29, 28]. Grounded in this work, this article further investigates the benefits of applying dynamic software updates to high performance parallel applications. Since a significant body of research explores various aspects of dynamic software updates, we present a thorough overview of this research and discuss the opportunities for applying this existing state of the art to HPC applications. In addition, we detail our experience in designing, implementing, and evaluating a dynamic update system for parallel, high performance applications, deployed on distributed memory multiprocessors.

The rest of this article is structured as follows. Section 2 presents our synchronization algorithm and its reference implementation to update parallel high performance applications on the fly. Section 3 details our efficient and flexible approach to enabling in-vivo enhancement of parallel high performance applications. Section 4 describes the existing state of the art in dynamic update systems. Section 5 discusses future work directions, and Section 6 presents concluding remarks.

2 UPDATING HIGH PERFORMANCE CLUSTER APPLICATIONS DYNAMICALLY

Due to their cost efficiency, compute clusters are among the most widely-used high performance computing environments. A typical compute cluster features a large number of homogeneous processors connected to each other with a high performance interconnect. To submit a computational task to a cluster, users interface with a scheduler that queues up the submitted tasks for their turn to be run. To coordinate the execution between different processors, compute cluster applications use the Message Passing Interface (MPI) [5] middleware library, which has a standardized interface and is almost universally available. Next we describe our dynamic software update system that targets compute clusters and uses MPI for coordination. We start by outlining the main design goal of our system, then detail our implementation, and finally evaluate the performance of our system.

2.1 Design Considerations

Compared to applications run on a single machine, cluster applications coordinate the execution of multiple concurrent processes running on multiple compute nodes. Therefore, a dynamic update system targeting cluster applications must posses several properties.

One issue is delivering the updated binary representation of a program to each compute node. Since the nodes on a cluster typically run a shared file system, copying a new version of the binary will immediately make it available to all the nodes.

Another issue is updating multiple concurrent copies of a program consistently. In other words, some distributed runtime invariant has to be maintained. The nature of this invariant depends on the kind of updated application. For example, for embarrassingly parallel applications, in which the Master communicates exclusively with Workers and Workers do not communicate among each other, the required consistency guarantees are quite relaxed; each Worker code could be updated independently and the correctness of the computation will still be preserved.

Nevertheless, most high performance parallel applications have much more stringent update consistency requirements than a typical Master-Worker application. One invariant that may have to be maintained is that *no two divergent versions of a program are run simultaneously.* Since to ensure maximum parallelism, concurrent processors are synchronized sparingly, maintaining this invariant can be quite challenging.

Our distributed synchronization implementation has two parts. First, we have created a distributed synchronization algorithm that implements a consistency scheme using standard MPI calls. In addition, to apply this algorithm to an existing parallel application, we have to inject special code to each piece of software that will be run concurrently and will need to be updated dynamically. We describe each of these parts in turn next.

2.2 Synchronization Algorithm

Updating multiple concurrent tasks consistently entails receiving the update information from the user and applying the update consistently to all the running nodes. Each of these activities requires a distributed consistency algorithm, which we now describe.

Dynamic updates are initiated by the user who interacts with an update front end. The purpose of the front end is to accept from the user a new version of the running program. Then the front end may choose to calculate the delta between the running and new versions of the program to reduce the amount of code that would have to be replaced. Finally, the front end instructs the running nodes to update themselves to the new version.

The step of this procedure that requires distributed coordination is for the nodes to receive the update request from the front end. Figure 1 shows the algorithm, according to which the root node of the distributed computation receives

```
   OUTPUT: Update information which will be used for updating on all nodes.
 1 updateInfoQueue ⇐ φ
 2 create a communication channel ch_p with a port number p
 3
 4 REPEAT
 5   await the update data ud from the user
 6   IF ∃ ud THEN
 7     ch_p receives ud through the port number p
 8     add ud to the update queue updateInfoQueue
 9   ENDIF
10 UNTIL the application is running
```

Fig. 1. Waiting for update information from the user.

```
 1 updateInfoQueue ⇐ φ
 2 N ⇐ a set of the nodes involved in the computation
 3
 4 /* The root node (rank0) creates a thread which will wait for update information. */
 5 IF nodeRank ≡ 0 THEN
 6   create a update thread udThread
 7 ENDIF
 8 ...
 9
10 /* Broadcast the update information from the root node to all nodes. */
11 IF nodeRank ≡ 0 THEN
12   get updateInfoQueue from the update thread udThread
13   broadcast updateInfoQueue to all nodes N
14 ENDIF
15
16 IF updateInfoQueue ≠ φ THEN
17   /* Each worker node updates its application on the fly. */
18   update the application app_j on n_j based on updateInfoQueue
19   /* A worker node waits until the others finish their updates. */
20   WAIT UNTIL the updates on the remaining N − n_j have been completed
21 ENDIF
```

Fig. 2. Synchronizing concurrent dynamic updates on multiple nodes.

update requests. To that end, the root node spawns a new *update thread* that waits for an incoming socket connection request. An asynchronous IO facility can be used to avoid busy waiting. When a connection from the front end is received, the update thread creates a user communication channel (line 2), receives update data from the user (line 5), and enqueues it (line 8). The thread can repeat the above procedure continuously until the application is running, thus possibly enqueuing multiple update requests.

The code on each running node is patched with update management code, which constitutes a short sequence of method calls and conditional statements. The exact code location at which dynamic updates should take place depends on the semantics of the updated application. For some applications, update management can be seamlessly added to the beginning of the main compute loop; however, this particular location may turn to be unsuitable for other applications. Figure 2 displays the algorithm implemented by the update management code. To synchronize updates, the root node (i.e., the process with rank zero) broadcasts update information to all nodes. Upon receiving the update information, all the nodes, including the root node, update themselves dynamically using whatever local DSU infrastructure is in place, and synchronize on a barrier.

2.3 Example Implementation

To enable efficient and safe dynamic software updates for parallel high performance applications, our approach taps into their development cycle and also adds some runtime functionality. Figure 3 describes the process by which an unaware parallel high performance application is enhanced to be dynamically updateable. The enhancement process starts after the application has been implemented, thus following the principle of separating concerns. The programmer focuses on implementing the application logic, while a special post-compilation step adds the required functionality to enable dynamic updates. Furthermore, our approach does not assume that the application is built with dynamic updates in mind.

As our example implementation, we have targeted parallel applications that use the Java^{TM}technology to operate seamlessly in a heterogeneous environment. The Java technology has been successfully applied to the domain of distributed parallel computation: heterogeneous computational grids are commonly Java-based [2]. The JVM also is one of the most advanced virtual execution environments ported to a multitude of different platforms. However, our primary motivation for choosing JVM as our experimentation platform is convenience. The relatively high level of Java bytecode makes it easier to add functionality to classes at the binary level, thus simplifying our proof-of-concept implementation. We believe, however, that our example implementation can be ported to other HPC platforms with only minor adjustments in the programming tools.

One of the key issues of dynamic software updates is safety. That is, updating a running application must not put it in an unsafe state that may affect the application's correctness and stability. Thus, it becomes important to identify a point in the execution of a parallel task at which its software can be safely

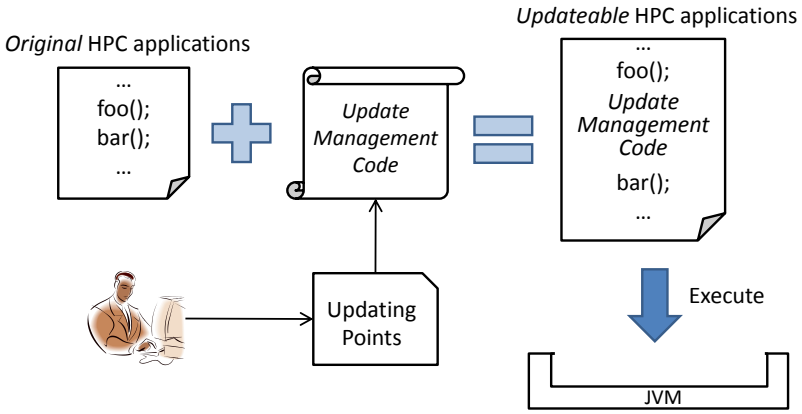


Fig. 3. Making HPC applications updateable.

updated dynamically. Upon reaching this point, the task should check whether the update is available, perform the requisite synchronization with the other tasks, and then update itself if so instructed.

Identifying a specific program execution point at which it is safe to perform a dynamic update requires a thorough understanding of the semantics of the parallel application. While identifying such safe update points automatically can be an immensely complex task, the programmer can easily specify them by hand. To that end, our approach allows the programmer to specify update information through a simple configuration file that contains the Java class, method, and line number, before which the synchronization code should be added. The code is added directly at the bytecode level, a common software practice for Java applications with multiple bytecode engineering toolkits readily available [15, 1, 11].

Figure 4 shows that the synchronization code is quite straightforward. The process with rank zero creates a thread to receive update requests from the user. To ensure that the thread does not waste computing resources, all the communication takes place through Java asynchronous IO, which wakes up the thread only when a new update request is ready to be processed. The rank zero process then broadcasts the update information to the remaining processes. In the case when the user initiated a dynamic update, then each process updates itself and synchronizes on a barrier. If no update is needed, then the only additional cost is the broadcast (line 5).

Figure 5 illustrates our runtime support for dynamic updates. Our approach leverages the capacities of the standard JVM HotSwap facility. Upon receiving an update request, the running JVM sends itself a HotSwap request, relying on the HotSwap machinery to safely interrupt the execution, reload the updated version of the specified classes, and restart the computation. Upon restarting the computation, the first executed method is a barrier synchronization that ensures

```
1  if (MPI.COMM_WORLD.Rank() == 0) {
2    classNames = SelectSockets.INSTANCE.getClassNames();
3  }
4
5  MPI.COMM_WORLD.Bcast(classNames, 0, classNames.length, MPI.OBJECT, 0);
6
7  if (!classNames.isEmpty()){
8    ExecutionManager.update(classNames);
9
10   MPI.COMM_WORLD.Barrier();
11
12 }
```

Fig. 4. An example of the synchronization code.

that all the concurrently-running copies of the computation have been updated consistently.

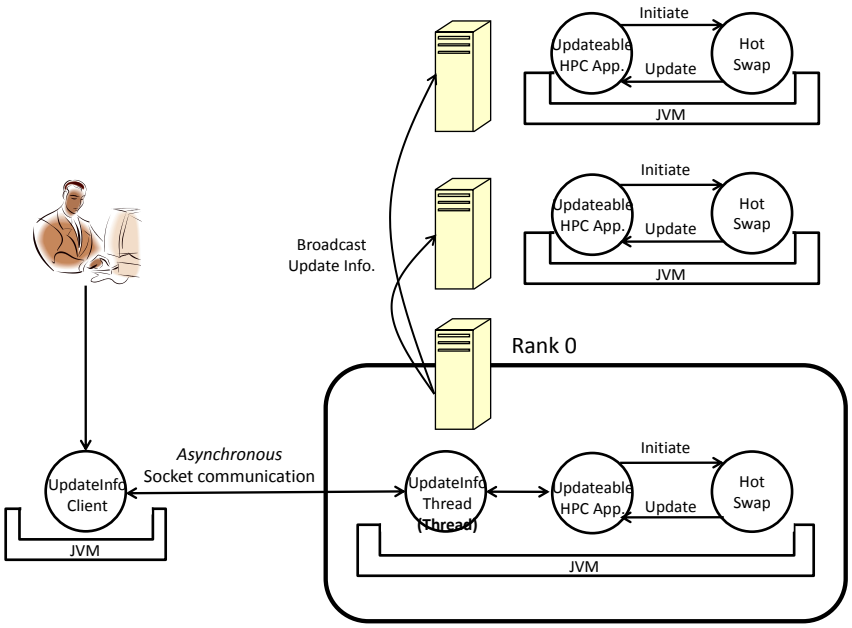


Fig. 5. Updating parallel HPC at runtime.

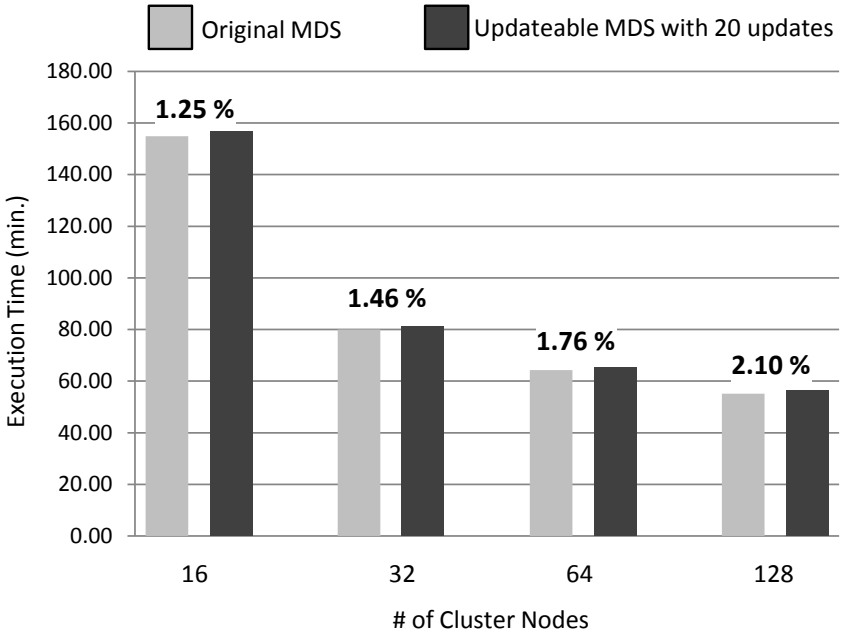


Fig. 6. Performance overhead on the update management code. `Bcast` and `Barrier` have been executed 200 and 20 times, respectively.

2.4 Evaluation

As our evaluation environment, we have used System G—a state-of-the-art research supercomputer recently constructed at Virginia Tech. System G features top-of-the-line components, typically found in a modern compute cluster: each compute node runs two Intel Xenon processors with 4 cores each (a total of 8 cores), 8GB RAM, and Fedora Core 10. The nodes are connected by InfiniBand (10Gbit+). For all the experiments, we used JDK 1.6.0_0.

The goal of our evaluation was to ensure that the injected update management code, discussed above, imposes a negligible performance overhead. If the goal of dynamic software updates is to reduce time-to-discovery, then high costs of executing the update management code would offset—if not eliminate altogether—the desired overall performance improvements. Since our update management code implements a distributed synchronization algorithm, its performance is thus dependent on the number of nodes. The higher the number of nodes that have to be updated consistently, the higher will be the cost of synchronizing them.

As our benchmark application, we have used a parallel Molecular Dynamics Simulation (MDS) application, a benchmark that is distributed with MPJ Express [7], a middleware library that provides Java bindings to the majority of MPI calls. Thus, although each individual node runs Java, the nodes communicate

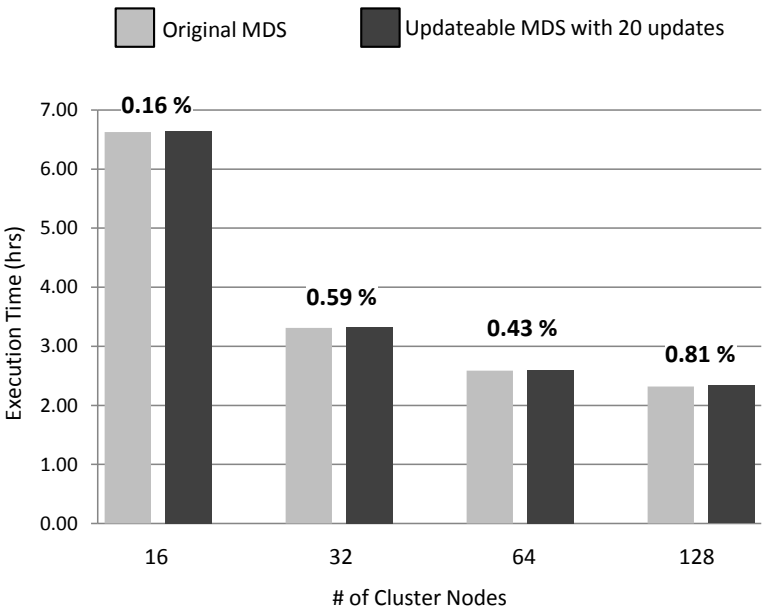


Fig. 7. Performance overhead on the update management code. `Bcast` and `Barrier` have been executed 500 and 20 times, respectively.

with each other through MPI. Therefore, our findings should be representative of a typical compute cluster application, and our setup does not unfairly benefit our approach.

MDS applications [44, 32] model the structure, motions, and interactions of molecular systems, including proteins, cell membranes, and DNA, at an atomic level of detail through a parallel computer simulation. The primary method of an MDS application includes the main execution loop, which simulates the movement of particles by running code on multiple nodes in parallel and synchronizing their execution through MPJ calls. Specifically, the main loop includes the routines that move particles, calculate the force on particles, and average the velocity of all particles. The size of the simulation is controlled by adjusting the number of times the particles are moved, `nstep`. We ran the simulation for `nstep` = 200 and 500. The update management code was inserted in the beginning of the main execution loop. Thus, the update management code was executed once per iteration step. We also randomly performed 20 dynamic updates. To capture the overhead of managing the updates, we have swapped the running class versions with the same classes. Thus, even though classes were swapped, the changes did not have any effect on the execution of the application.[1]

[1] See Section 3 for the description of realistic updates that can reduce the time-to-discovery in parallel high performance applications.

Figures 6 and 7 show the performance overhead imposed by our update management code for 200 and 500 simulation steps. We compared the total execution time of the original and the enhanced for dynamic updates versions of the MDS application on 16, 32, 64, and 128 nodes. The overhead is incurred mainly by the MPI `Bcast` and `Barrier` calls: while `Bcast` is executed on each iteration of the main execution loop, `Barrier` is called only if a dynamic update has taken place (i.e., `Barrier` is called 20 times). As expected, the performance overhead tends to increase with the number of nodes. Nevertheless, the total overhead never exceeds 2%. Furthermore, as the number of iterations increases, the total overhead of executing our update management code decreases.

Note that the total execution time of each simulation can be as high as 8 hours depending on the configuration. If the running code needs to be changed during these 8 hours, the ability to update the code dynamically, in our view, will be well worth the overhead of the update management code, which adds up to less than 10 minutes [2].

3 FLEXIBLE AND EFFICIENT DYNAMIC SOFTWARE UPDATES FOR JVM APPLICATIONS

The distributed consistency algorithm presented in the previous section is quite efficient, incurring a negligible performance overhead on parallel high performance applications running on the JVM. Furthermore, the JVM features a built-in facility—JVM HotSwap—which provides a standardized API for replacing classes in a running JVM. Unfortunately, as it turns out, HotSwap imposes serious constraints on what kinds of changes can be made to the swapped classes, thus significantly limiting the applicability of this facility for updating applications dynamically. To be able to dynamically update parallel high performance applications flexibly and efficiently, we have developed an approach, based on the binary rewriting of Java classes, that overcomes the limitations imposed by the design of JVM HotSwap. We present the main insights of our approach next.

3.1 HotSwap Constraints

The JVM HotSwap disallows any changes to the signature of a class: a swapped class has to contain the same set of methods and fields as the currently deployed version, and only method bodies can be changed. Whenever the programmer tries to perform any of the updates listed in the second column of Table 1 using the JVM HotSwap, the JVM throws an exception.

Because in Java, one cannot assume one-to-one correspondence between source files and their classes in bytecode, complying with HotSwap restrictions can be nontrivial. For example, a Java inner class is commonly translated by adding `synthetic` access methods to its enclosing classes, so that the inner class could

[2] 2% of 8 hours is $480\,mins * .02 \approx 10\,mins$.

Table 1. HotSwap Constraints (the addressed ones are shaded)

Targets	Changes
Method	Adding a new method
	Removing an existing method
	Adding formal arguments of a method
	Removing formal arguments of a method
	Changing the return type of a method
	Changing method modifiers
Field	Adding a new field
	Removing an existing field
	Changing the type of a field
	Changing field modifier

access their non-public members. This translation strategy is likely to leave the programmer unaware that a change to one class caused the compiler to add methods to other classes, thus violating the HotSwap constraint on adding new methods and rendering the enclosing classes unswappable.

3.2 Binary Refactoring for Proxy Indirection

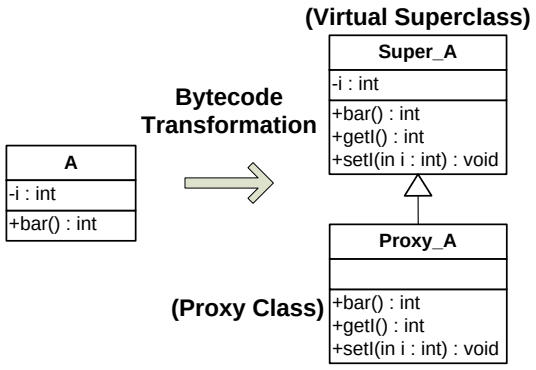


Fig. 8. Virtual superclass binary refactoring.

Binary refactoring applies structural semantics-preserving transformations to a program's binary representation, with the goal of enabling its functional enhancement. One of the most common binary refactorings in existence is changing direct references into proxy references. Our approach uses this refactoring

to address the limitations of HotSwap described in Section 3.1. A common implementation of indirect referencing is a binary refactoring that we call *Virtual Interface*[3]. Virtual Interface refactors the bytecode of a class into proxy, interface, and implementation classes. The bytecode rewriter makes the client part of the target version refer to the proxy class in the refactored version. This indirection style can incur between 8% and 44% performance overhead, which is prohibitively high for performance-sensitive applications.

As an alternative, we have created a novel technique for introducing indirect referencing that we call *Virtual Superclass*, which incurs only minuscule performance overhead on the refactored programs. Our approach applies Virtual Superclass to all application classes loaded into the JVM; Figure 8 depicts the Virtual Superclass refactoring transformations. Every class `A` is changed to extend a *virtual* superclass `Super_A`, with the virtual superclass being inserted into the class's inheritance hierarchy.[4] Thus, the original class `A` becomes a proxy for the virtual superclass, which contains all the original method bodies and fields.

Another advantage of Virtual Superclass is its generality. The existing state of the art in enabling proxy indirection [16] creates subclass-based proxies, which have limitations for `final` classes and methods. By contrast, Virtual Superclass works for *any* class or method, as the `final` modifier does not constrain the creation of superclasses.

The performance efficiency of Virtual Superclass is explained by the sophisticated optimization capabilities of modern JVMs, which can inline delegating method calls, if the delegation does not involve dynamic dispatch. In Figure 8, the call to `super.bar` can be effectively inlined by modern JVMs, thus completely eliminating any indirection overhead in most cases. The call is translated into the `invokespecial` bytecode instruction, reserved for invoking constructors and methods in superclasses. The delegating call in Virtual Interface uses the `invokeinterface` instruction, which implements a form of dynamic method dispatch, and as such cannot be safely inlined, though its performance has been improved significantly in modern JVMs [6]. Thus, Virtual Superclass leverages the low-level differences of bytecode instructions to attain its performance advantages.

3.3 Flexible In-Vivo Enhancement with HotSwap

To be able to use the standard HotSwap to replace classes in a running JVM, our approach rewrites all the classes at the bytecode level before deployment. It is these rewrites that make it possible to change the signatures of replaced classes, without violating the HotSwap constraints.

The first phase, illustrated in Figure 9, refactors all the loaded classes at the bytecode level and generates their corresponding virtual superclasses. The

[3] The adjective *virtual* emphasizes the fact that the introduced interface is not seen by the client program and is only used as an implementation artifact. The client code never accesses the introduced "virtual" interface directly.

[4] The virtual superclass is inserted for each class in the inheritance hierarchy. Thus, `A` **ext** `B` ⇒ `A` **ext** `Super_A` **ext** `B` **ext** `Super_B`.

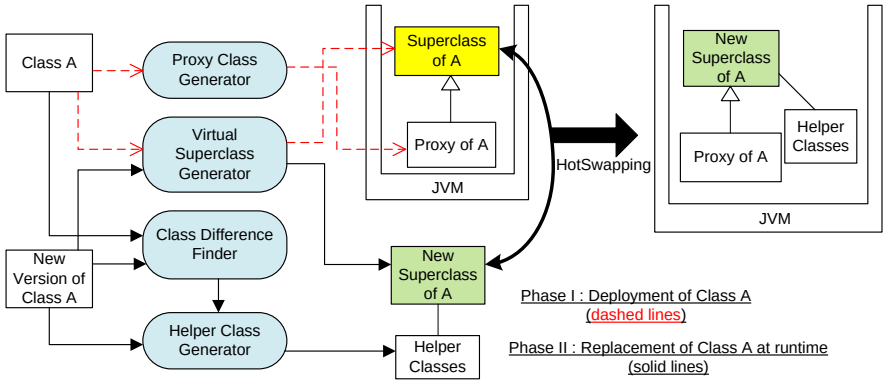


Fig. 9. Supporting a full range of dynamic updates using HotSwap.

virtual superclasses have actual methods implementing application-specific logic and are swapped by the updating system. Thus, the original code is rewritten into updateable software, structurally different from the original version, before being deployed on a virtual machine. When the initial program is changed, the programmer inputs the changed classes to the updating system, which refactors them into virtual superclasses and special helper classes. HotSwap can then replace older class versions of virtual superclasses with newer versions, as they have the same schema. Helper classes make the updates conform to the HotSwap API when new methods or fields are added. The new members are added to helper classes, so that the signatures of virtual superclasses remain the same.

In addition to transforming classes at load time with the Javassist library [15], our system includes a class differencing module and code generators for proxy, virtual superclass, and helper classes. The differencing algorithm operates at the bytecode level, and its output parameterizes the code generators and the bytecode rewriter. The rewriter translates newly-added methods, constructors, and fields to helper classes as follows.

New methods/constructors The rewriter adds a special `invoke` method to all the instrumented classes as a facility to invoke newly-added methods without changing the updated class's signature. Each new method is translated into a method in a helper class, whose invocation logic is added to the body of the `invoke` method. Each call site of a newly added method becomes a call to `invoke`, with the added method name as the first argument.

Figure 10 shows an example of adding a new method; the newer version of `A` has a new method `bar`. The first and second columns in Figure 10 illustrate class diagrams representing classes and their relationships at the source code and the corresponding bytecode, respectively. The special helper class `HelperClass` contains the new method `bar` and each proxy class contains the `invoke` method. Each invocation of `bar` is translated to invoke `invoke` instead.

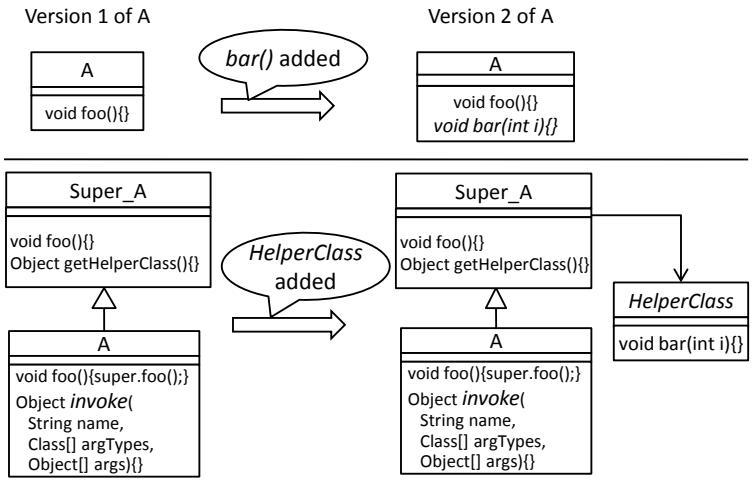


Fig. 10. Adding a new method using a helper class.

Each new constructor is translated into an invocation of a "do-nothing" constructor and a special initialization method that contains the added constructor's logic.

New fields New fields are translated according to two approaches, one optimized for performance, while the other for space. The first approach uses a separate helper class for the new fields whenever a class is replaced with a newer version. The second approach uses a single class that contains a mapping data structure that represents all the added fields for all classes. For performance-sensitive applications, we envision that the first approach will be preferable. Few HPC parallel applications run in memory-constrained environments that require the programmer to be space-aware. Modern JVMs can load new helper classes on demand quite efficiently.

Object state update One complication of using HotSwap for updating running applications is that it can only update classes—HotSwap has no facilities for upgrading objects created from an older version of a class to a newer version. In dynamic update systems, this operation is called *Object State Update*. Our approach also can efficiently transfer state between old and new objects, enabling instances of different versions of a class to coexist in the running application. Our system updates the state between old and new helper objects for new fields, based on their respective version numbers. In particular, the update system checks if the version of a helper object is older than the latest version. If so, a special helper object is instantiated for the newly added state (i.e., extra fields). The

values of the fields in the older helper object are then copied to the corresponding fields in the newer helper object.

3.4 Updating Smith-Waterman Parallelization Dynamically

To motivate the need for flexible dynamic software updates in in-vivo enhancement of parallel high performance applications, we describe how the well-known Smith-Waterman algorithm could be parallelized and developed incrementally to run in distributed multiprocessors. The sequential version of this algorithm [3] calculates a similarity score between two sequences. A parallelization of this algorithm will align an unknown sequence against an entire database of known sequences, with the database partitioned among different computational nodes. The resulting computation will follow a simple Master Worker model, with the Master node assigning tasks to the Worker nodes as well as collecting and filtering the results. Specifically, the Master accepts an unknown sequence as input and sends it to individual Worker nodes. Each worker node aligns the unknown sequence against its portion of the partitioned database. The sequences having the highest similarity scores (e.g., above a given threshold) are then sent back to the Master. The Master collects the results, sorts them, and reports the top-ranked results to the user.

Cases	Requirements	# of updates				
		Field	Method	Class		
				Method body	Sig. change	Replaced classes
Case1: Console ⇨ File	Saving alignment results as a file	1	1	1	1	2
Case2: float ⇨ double	Displaying alignment results in a double precision	5	11	6	4	9
Case3: SW ⇨ SWG	A need of more practical alignment algorithm	0	4	1	1	2

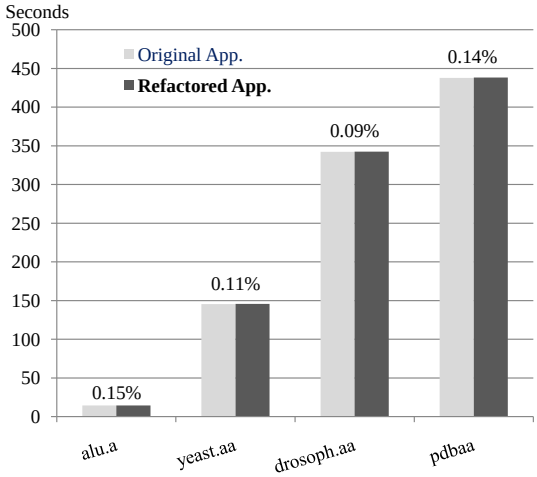
Fig. 11. Changes to Smith-Waterman program using extended HotSwap. SW:Smith-Waterman algorithm [46], SWG:Smith-Waterman-Gotoh algorithm [19].

Thus, after creating an initial parallelization of the Smith-Waterman algorithm described above, the programmer could deploy and test it in its intended deployment environment. One common difference between sequential applications and their parallelizations is that the parallel version produces much more output data. It is quite likely, for example, that while in the sequential version of Smith-Waterman algorithm, all the results could comfortably fit on the same output window, in the parallel version, the results would be more numerous. As a result, it is possible that the output data in the parallel version could only be properly examined, if they were saved to a disk file. Thus, the programmer may wish to change the piece of functionality that simply dumps the results to standard output to write them to a disk file instead.

It also may turn out that certain assumption made during the design phase would no longer hold true. For example, the programmer may have assumed that the `float` precision would be sufficient for representing similarity scores, while after seeing the initial results realize that the `double` precision is needed.

Finally, it may turn out that the implementation of the alignment algorithm does not satisfy the expected performance or accuracy requirements. A slight variation of the algorithm could satisfy these requirements to a greater extent.

As it turns out, all of these three updates involve structural changes to the bytecode, rendering the standard HotSwap facilities unsuitable for the task. Specifically, changing the display from the console to a disk file requires replacing classes `AlignCommentLine` and `FileOutput`, as well as adding a new method `writeToFile`, thereby changing the signature of class `FileOutput`. Such a seemingly trivial change as using `double` rather that `float` precision for the similarity scores requires modifications of 5 fields, 11 methods, and 9 classes! Because the similarly score is computed through the interaction of multiple methods in different classes, changing its type (i.e., from `float` to `double`) requires changing the signatures of all of the involved methods. Finally, modifying the alignment algorithm requires modifying the signatures of 4 methods in 2 different classes. Because the base algorithms use different parameter sets, the methods' signatures, invoked when the algorithm is executed, have to be changed accordingly. Figure 11 presents the exact statistics of the changes involved.



DB	Size (KB)	# of seq.
alu.a	221	1,962
yeast.aa	3,321	6,312
drosoph.aa	8,300	14,331
pdbaa	22,169	37,882

Fig. 12. Refactoring overhead on the worker portion of Smith-Waterman parallelization. x-axis: the databases names in FASTA format, y-axis: the total execution time.

For this case study, we used the Ibis [2] grid infrastructure, even though the grid nodes communicated with each other through Java sockets rather than through the MPJ middleware provided by Ibis. We have also included our binary rewriting infrastructure into the standard class loading process. All the dynamic updates are initiated from the Master node, which has remote debugging connections to each Worker node.[5] The programmer interacts with an upgrade script that takes the classes of a new program version, compares this version with the current version, computes the necessary updates, and applies them dynamically through the remote debugging connection to the remote nodes. Figure 12 shows the indirection overhead on the rewritten Worker code. Because the cost of indirection is incurred *only* when invoking methods, and the Worker process does most of the computation within a single method, the overall overhead is negligible. Thus, our novel binary rewriting approach made it possible to use the standard HotSwap to update a running distributed application, without either having to modify the JVM or having to degrade the performance. Furthermore, the updates were applied without having to stop the parallel execution and wasting valuable HPC resources. These results indicate that in-vivo enhancement can become a valuable tool for delivering parallel solutions under tight deadlines.

4 RELATED WORK

Research on dynamic software updates hails back to the early 1980's. Many researchers have focused on the challenge of changing a running application without stopping its execution. Each shift in the design of programming languages and models has brought about a new wave of research on dynamic software updates.

The following discussion, inspired by a similar overview included in Michael Hicks' doctoral dissertation [22], presents key properties of dynamic software update systems and demonstrates how representative systems support the presented properties.

4.1 Range of updates

The *range of updates* property refers to supporting arbitrary updates without significantly constraining which programming language features can be used in the updated applications. Ideally, DSU systems should have no limitations on the update unit such as classes, procedures, and processes.

DYMOS [33] is an integrated environment which consists of a command interpreter, source code manager, editor, compiler, and run-time support system. DYMOS permits changes to module definitions, data definitions, and infinite loop bodies. Ginseng [38] supports changes to function types and the type of global variables by tracking concrete uses of functions and global variables. Since

[5] Starting from JDK 1.4, remote debugging connections do not impose performance overhead, allowing programs to run at full speed.

the updating facility of PROSE [40, 39] is based on the HotSwap in the Sun JDK implementation, PROSE only supports updates to method bodies. Unlike PROSE, JVOLVE [47] can flexibly support schema changes such as additions and replacements of fields and methods because it uses a modified virtual machine. To support various kinds of updates, Bialek's approach [9] automatically partitions Java applications into a set of classes or Java packages.

4.2 Robustness

The *robustness* property represents safety, well-timedness, and the ability to rollback DSU systems. Whenever a robust DSU system updates an application dynamically, the risk of the updated application crashing has to be minimized.

The DYMOS system mentioned above ensures resiliency to crash or system failure of the updated applications by enforcing type checking. A new version of a procedure is compiled within the environment used by the old version. To improve safety further, DYMOS uses update pre-conditions provided by the programmer and enforced by the system to ensure that the updated system does not become unstable as a result of an update. Argus [12] provides crash recovery facilities that interact with the Argus' process abstractions called *guardians*. A portion of a guardian's running state is stored in permanent storage. If the program crashes, a replacing guardian can be restarted using the persistent old state. Guarded Software Updating (GSU) [48] can update extremely long-running applications (e.g., satellite software running for a decade). GSU allows different versions of a program to be deployed simultaneously, so that the new version could be tested under the current runtime conditions. If the testing establishes confidence, then the system can be transitioned to the new version, using the old one as a backup accomplished through message logging, checkpointing, and rollback recovery.

JVOLVE verifies the updated bytecode for type safety and checks the running thread's activation stack to reach a safe update point, delaying the updates if necessary. POLUS [13] can rollback committed updates to the original versions of the code and data, which are stored in memory. While using extra memory, this approach provides fast and easy rollbacks.

4.3 Performance overhead

The *performance overhead* property refers to the total amount of extra time it takes to enable dynamic updates. To be applicable for high performance applications, DSU systems should impose a negligible performance overhead.

POLUS, a DSU system for C applications, uses binary rewriting to redirect function calls between versions. The introduced function redirection may impose some performance overhead. However, Chen *et al.* claim that such overhead is minimal and less than 1% for most applications [13]. PROSE updates Java applications dynamically with a negligible run-time performance overhead, thanks to the aggressive inlining of the existing method calls in the replaced method

bodies. JVOLVE uses a custom virtual machine, providing a highly efficient DSU service with virtually no performance penalty.

4.4 Ease of use

The *ease of use* property refers to the simplicity with which a DSU system can be used to apply dynamic updates. Ideally, the update processes should be transparent to the programmer, with most of the functionality completely automated. That is, programmers should not have to write any dynamic update code, and the development process should be separated from the dynamic update process. Segal and Frieder have observed that the degree to which a particular DSU system is used is directly proportional to its transparency [43].

To minimize user involvement, Bialek's system automatically adds dynamic update functionality to the applications. POLUS also needs only minimal update information from programmers and automatically generates update patches. Furthermore, to provide good usability, POLUS enables patch processes to be visible to programmers. Ginseng features a compiler and tool suite for constructing updateable C applications from programs written without dynamic updates in mind—it produces updateable programs dynamically and generates dynamic patches automatically.

4.5 Portability

The *portability* property refers to the ability to perform dynamic updates on multiple platforms. Some dynamic updating systems only work on specific operating and/or middleware systems.

Dynamic C++'s approach [23] requires no special preprocessor or compiler support. Its lightweight proxy classes can be compiled by any standard C++ compiler.

On the other hand, some DSU systems need comprehensive infrastructure support such as a custom operating or middleware system. Hauptmann *et al.*'s approach [21] leverages certain properties of the Chorus operating system, including dynamic process loading, port migration, and thread scheduling. The Eternal system [36] dynamically updates CORBA-based systems, thus being portable across CORBA-enabled platforms.

Kang *et al.* [27, 26] adapt HPC programs, in Fortran, C, and C++, for new requirements. Their approach leverages the function interception capabilities of a framework that operations at the assembly language level, thus ensuring programming language independence.

Some Java DSU systems, including JDRUMS [4], JVOLVE, and Dynamic Virtual Machine (DVM) [35], use custom JVMs. The portability of these approaches is thus constrained by the necessity to provide a customized JVM for every supported platform.

4.6 Multi-threading support

The *multi-threading support* property refers to the ability to safely update a program that has multiple concurrent execution threads.

DYMOS can update multi-threaded applications due to its design, which uses the multi-thread variant of the Modula language called StarMod; this particular dialect makes it easier to synchronize those DYMOS function calls that access global structures. The design of Argus naturally supports the replacement of *guardians*, which are groups of distributed, multi-threaded processes.

To update multi-threaded C programs efficiently, Ginseng improves on barrier-style synchronization. Its extension, STUMP [37], further facilitates safe and timely updates of multi-threaded programs by enabling the programming to easily reason about the safety of updates. To that end, STUMP takes programmer's input in the form of safe program update locations and calculates an extended set of such locations. It then relaxes the built-in properties of synchronization constructs to ensure that threads not be blocked at these update locations.

JVOLVE can safely update multi-threaded programs by ensuring that all the threads have reached a safe state, in which updated methods are no longer allocated on the runtime stack. If a safe state cannot be reached, the updates are postponed until a later time.

4.7 Distribution support

The *distributed support* property refers to the ability to update distributed applications, whose execution spans multiple address spaces, possibly separated by a network.

Some systems dynamically update distributed applications by using custom middleware. PolyLith [25] runs C programs on top of a special reconfiguration-enabled runtime system.

The Conic [31] distributed programming system coordinates multiple distributed processes. The provided entry points, called channels, are used for inter-process communication, and a configuration manager can redirect the channels on the fly.

JDRUMS provides a special communication layer that enables accessing a remote JVM through JINI using RPC. The JD reconfiguration tool coordinates distributed components using a JINI communication protocol.

To adapt HPC applications without degrading their performance, Kang *et al.*'s approach injects adaptive code at the existing global synchronization points (e.g., where MPI_Bcast and MPI_Barrier are invoked).

4.8 Language constructs support

The *language constructs support* property refers to accommodating various programming language constructs during dynamic updates. Of interest to this discussions are various language constructs found in C, C++, and Java.

PODUS, PolyLith, On-line Software Version Change (OSVC) [20], and Ginseng are specifically designed for dynamically updating programs written in C. Dynamic C++, Eternal, and Chorus aim at dynamic updates of C++ applications. Argus, Conic, Erlang [49], Dynamic ML [18], and DYMOS work with custom versions of various languages that facilitate dynamic updates.

The rapid, widespread adoption of Java technology has served as an impetus for creating many DSU systems for updating Java applications on the fly. Orso *et al.*'s technique [41] refactors bytecode to enable dynamic updates. Bialek *et al.*'s system also rewrites the updated software at the source or bytecode levels to enable its dynamic updates. Several approaches [42, 35, 17] have introduced custom virtual machines to support dynamic updates of Java applications. Some approaches [34, 14, 50, 10] introduce new languages features, middleware systems, or require that software developers abide by specific component models or programming rules. Bierman *et al.*'s UpgradeJ [10] is a Java-like language for type-safe upgrading classes dynamically.

5 FUTURE WORK

As our experimental platform is the JVM, it would be interesting to explore how our approach works for new high-productivity languages that target the JVM. The new language features of these languages are likely to pose new challenges for dynamic updates. Furthermore, we would like to further improve the usability of our approach, making it accessible to non-expert programmers. It remains to be seen whether the complex functionality enabled by our infrastructure can be exposed through an intuitive GUI. The usability of our infrastructure can greatly affect its adoption rate. Finally, our approach to dynamic updates of parallel applications could benefit next-generation software systems including large scale grid applications, conscientious systems, and autonomic computing.

6 CONCLUSIONS

This article has considered the value of using dynamic software updates to reduce the time-to-discovery metrics of high performance parallel applications. We have presented an extensive overview of the state of the art in dynamic software updates and reported on our own work on dynamically updating high performance parallel applications. Our approach demonstrates the benefits of dynamic software updates for the domain of high performance parallel applications and contributes several innovative techniques. One technique employs binary rewriting to overcome constraints of the HotSwap API to support a wider range of dynamic changes. Another technique introduces an algorithm for ensuring consistency when dynamically updating multiple concurrent parallel tasks. To demonstrate the efficiency of the algorithm, we benchmarked a parallel molecular dynamics simulation, comparing the performance of the original version and the one enhanced with the implementation of our distributed synchronization algorithm. The results show that the overhead of our approach is negligible and

can be certainly justified by the added ability to update parallel execution on the fly.

High performance computing researchers and practitioners alike are starting to realize the potential of reducing the development and fine-tuning component of the time to discovery in parallel high performance applications. This research has demonstrated the benefits of applying dynamic software updates to parallel high performance applications, an approach that can improve many aspects of engineering software in this domain.

References

1. Byte Code Engineering Lab (BCEL), `http://jakarta.apache.org/bcel/`.
2. Ibis: Grids as Promised, `http://www.cs.vu.nl/ibis/`.
3. JAligner, `http://jaligner.sourceforge.net/`.
4. JDrums, `http://www.ida.liu.se/~jengu/jdrums/`.
5. Message Passing Interface, `http://www.mcs.anl.gov/mpi`.
6. B. Alpern, A. Cocchi, S. Fink, and D. Grove. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 108–124, 2001.
7. M. Baker, B. Carpenter, and A. Shaft. Mpj express: Towards thread safe java hpc. *Cluster Computing, IEEE International Conference on*, 0:1–10, 2006.
8. V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE Software*, 25(4):29–36, 2008.
9. R. P. Bialek. Dynamic updates of existing Java applications. *Ph.D. Thesis, the University of Copenhagen*, pages 1–216, June 2006.
10. G. Bierman, M. Parkinson, and J. Nob. UpgradeJ: Incremental typechecking for class upgrades. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2008.
11. W. Binder, J. Hulaas, and P. Moret. Advanced Java bytecode instrumentation. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM.
12. T. Bloom and M. Day. Reconfiguration and module replacement in argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, 1993.
13. H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. Polus: A powerful live updating system. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society.
14. X. Chen. Extending RMI to support dynamic reconfiguration of distributed systems. *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 401–408, 2002.
15. S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE'03)*, pages 364–376, 2003.
16. P. Eugster. Uniform proxies for Java. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 139–152, 2006.

17. B. Gharaibeh, D. Dig, T. N. Nguyen, and J. M. Chang. dReAM: Dynamic refactoring-aware automated migration of Java online applications. *Technical Report, Iowa State University*, August 2007.
18. S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without dynamic types. *Technical Report, The University of Edinburgh*, December 1997.
19. O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.
20. D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996.
21. S. Hauptmann and J. Wasel. On-line maintenance with on-the-fly software replacement. In *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, page 70, Washington, DC, USA, 1996. IEEE Computer Society.
22. M. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.
23. G. Hjálmtýsson and R. Gray. Dynamic C++ classes: a lightweight mechanism to update code in a running program. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 6–6, Berkeley, CA, USA, 1998. USENIX Association.
24. L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2005. IEEE Computer Society.
25. C. Hofmeister. Dynamic reconfiguration. *Ph.D. Thesis, University of Maryland*, 1993.
26. P. Kang, Y. Cao, N. Ramakrishnan, C. J. Ribbens, and S. Varadarajan. Modular implementation of adaptive decisions in stochastic simulations. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 995–1001, New York, NY, USA, 2009. ACM.
27. P. Kang, N. K. C. Selvarasu, N. Ramakrishnan, C. J. Ribbens, D. K. Tafti, and S. Varadarajan. Modular, fine-grained adaptation of parallel programs. In *ICCS*, pages 269–279, 2009.
28. D. K. Kim, Y. Jiao, and E. Tilevich. Flexible and efficient in-vivo enhancement for grid applications. In *9th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2009)*. IEEE, 2009.
29. D. K. Kim, M. Song, E. Tilevich, C. J. Ribbens, and S. A. Bohner. Dynamic software updates for accelerating scientific discovery. In *The International Conference on Computational Science 2009 (ICCS 2009)*, 2009.
30. D. K. Kim and E. Tilevich. Overcoming JVM HotSwap constraints via binary rewriting. In *First ACM Workshop on Hot Topics in Software Upgrades (HotSWUp 2008)*. ACM, 2008.
31. J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng.*, 11(4):424–436, 1985.
32. A. Kumar. Molecular Dynamics Simulations. `http://www.personal.psu.edu/auk183/MolDynamics/Molecular%20Dynamics%20Simulations.html`.
33. I. Lee. DYMOS: A Dynamic Modification System. *Ph.D. Thesis, University of Wisconsin, Madison*, April 1983.
34. Y.-F. Lee and R.-C. Chang. Java-based component framework for dynamic reconfiguration. *IEE Proceedings - Software*, 152(3):110–118, June 2005.

35. S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. *Proceedings of the 14th European Conference on Object-Oriented Programming*, 1850:337–361, June 2000.
36. P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the internet inter-ORB protocol interface to provide CORBA with fault tolerance. In *COOTS'97: Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 6–6, Berkeley, CA, USA, 1997. USENIX Association.
37. I. Neamtiu and M. Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2009.
38. I. G. Neamtiu. Practical dynamic software updating. *Ph.D. Thesis, University of Maryland*, pages 1–212, August 2008.
39. A. Nicoara and G. Alonso. Dynamic AOP with PROSE. In *Conference on Advanced Information Systems Engineering (CAiSE)*, pages 125–138, 2005.
40. A. Nicoara, G. Alonso, and T. Roscoe. Controlled, systematic, and efficient code replacement for running Java programs. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 233–246, New York, NY, USA, 2008. ACM.
41. A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, October 2002.
42. T. Ritzau and J. Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, London, May 2000.
43. M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, 1993.
44. D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang. Anton, a special-purpose machine for molecular dynamics simulation. *Commun. ACM*, 51(7):91–97, 2008.
45. F. Shull, J. Carver, L. Hochstein, and V. Basili. Empirical study design in the area of high-performance computing (hpc). *Empirical Software Engineering, International Symposium on*, 0:10 pp., 2005.
46. T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.
47. S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates for Java: A VM-centric approach. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2009.
48. A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, and W. H. Sanders. On low-cost error containment and recovery methods for guarded software upgrading. In *ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, page 548, Washington, DC, USA, 2000. IEEE Computer Society.
49. R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.
50. A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with Expanders. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 37–56, 2006.