Flexible and Efficient In-Vivo Enhancement for Grid Applications

Dong Kwan Kim, Yang Jiao, Eli Tilevich
Center for High-End Computing Systems (CHECS)
Dept. of Computer Science, Virginia Tech
Blacksburg, VA 24061
{ikek70, jiaoyang, tilevich}@cs.vt.edu

Abstract

In a grid application, some requirements may change while the execution is in progress. This paper presents in-vivo enhancement*–updating running grid applications to facilitate their perfective maintenance. Because applications in this domain are not only typically long-running, but also time-consuming to deploy, we propose a dynamic update technique that can change a running application flexibly and efficiently. Specifically, this paper presents a novel technique for dynamically updating grid applications deployed on the Java Virtual Machine (JVM). Our technique overcomes constraints of JVM HotSwap, a facility for replacing classes at runtime. While HotSwap precludes the programmer from adding new methods and fields, changing the signatures of existing methods, and has no support for transferring state between old and new objects, our approach effectively removes these constraints by rewriting program bytecode. Further, the rewritten programs incur only minimal performance overhead (less than 2% on average). We demonstrate the efficiency and extensibility of our approach through micro and macro benchmarks, as well as through a case study of dynamically updating a parallel bioinformatics application.*

1. Introduction

The behavior and performance of a distributed application can only be fully ascertained in a production environment. This, in turn, could lead to a change in program requirements. Grid applications are difficult to develop incrementally, as they are often time-consuming to deploy. Thus, the typical *try-change-try again* development cycle may 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 distributed 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. To address these inefficiencies, a running application could be updated dynamically, thus saving the programmer's time and computing resources. With dynamic updates, one could adapt a program for changed requirements, observing the results of the updates in almost real time.

The raison d'être of high end computing is to reduce *time-to-discovery*, the total time it takes from posing a problem to arriving to a solution. This metrics is the sum of the time it takes to run an application and the time it takes to develop and fine-tune it. To reduce time-to-discovery, this paper presents *in-vivo enhancement*–updating a distributed application dynamically, after it has been deployed in its intended execution environment. Since the ability to update distributed grid applications dynamically can shorten their development time, we argue that it should become an essential step in their development process.

This work is concerned with distributed computationally-intensive 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 [1]. The Java Virtual Machine (JVM) is one of the most advanced virtual execution environments ported to a multitude of different platforms.

The JVM features the HotSwap API [3], which replaces loaded classes in a running application. However, the signature of a replaced class must remain the same, and only method bodies can change. Thus, the programmer modifying the swapped classes is significantly constrained. To overcome these HotSwap constraints, allowing the programmer to update classes without restrictions, this paper presents a novel bytecode rewriting and code generation approach, using the standard HotSwap to replace the changed code in a running JVM. The flexible and efficient dynamic updates make it possible to enhance a running application at will. This, in turn, enables an incremental development model for grid applications, thereby reducing time-to-discovery.

The technical material presented in this paper makes the following novel contributions:

- A new methodology that uses dynamic updates to reduce the time it takes to enhance grid applications.
- A novel proxy indirection technique suitable for performance-sensitive applications due to its minimal performance overhead.
- A binary rewriting technique that leverages the proxy indirection to overcome limitations of JVM HotSwap, without changing its API or runtime libraries.

The rest of this paper is structured as follows. Section 2 motivates this work through an example from the bioinformatics domain. Section 3 details our approach to enabling in-vivo enhancement of distributed computationally-intensive applications. Section 4 evaluates the efficiency and expressiveness of our approach. Section 5 compares our approach with the existing state of the art. Section 6 discusses future work directions, and Section 7 presents concluding remarks.

2. Motivating Example

Our motivating example is concerned with pairwise sequence alignment, a well-known problem in bioinformatics. The use of computers has made it possible to answer a larger spectrum of questions in biology. Most of these problems are solved by representing a biological entity such as a gene computationally and manipulating the resulting representation using a variety of algorithms.

To motivate the need for flexible dynamic updates in in-vivo enhancement of distributed applications, we next describe how the well-known Smith-Waterman algorithm could be parallelized and developed incrementally to run in an ad-hoc grid environment. The sequential version of this algorithm [2] 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.

We would be amiss if we did not mention up front that mature grid computing infrastructures often come with sophisticated simulators. These simulators make it possible for the programmer to test a grid application on a single machine and get a realistic picture how the application would work when deployed on the grid. Nevertheless, such simulators only come as a part of a mature grid infrastructure, and would not be available for light-weight environments such as ad-hoc grids using the JVM. Even if such a simulator were available, parameterizing it with the exact information about an ad-hoc grid would be a prohibitively-difficult undertaking, on par with deploying and running the application on the grid. Another capability of properly developed grid applications is checkpointing, which allows restarting an upgraded application without losing intermediate results. Again, many ad-hoc grid applications may not include any checkpointing functionality and thus could benefit from our approach.

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.

In-vivo enhancement makes it possible to fine-tune and troubleshoot a distributed application in its real deployment environment, while sidestepping the inefficiencies of a typical upgrade cycle through flexible dynamic updates.

Unfortunately, the dynamic updates required to address the change in requirements outlined above would be impossible with HotSwap. To address the insufficiencies

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

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

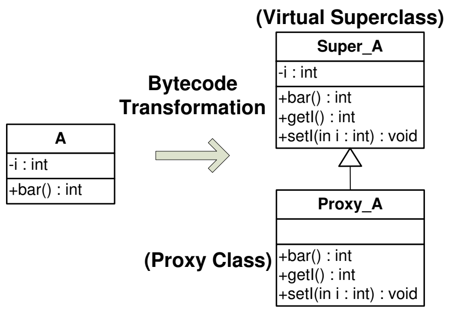


Figure 1: Virtual superclass binary refactoring.

of HotSwap to support the utility and efficiency of in-vivo enhancement, we present a novel binary rewriting approach that transforms the bytecode of a distributed computationally-intensive application. These transformations enhance the bytecode with the capabilities required to enable flexible dynamic updates with the standard HotSwap, but they do so while incurring only minuscule performance overhead on the rewritten programs, as detailed in Section 4. To validate the expressive power of our approach, we dynamically update the Smith-Waterman algorithm to address the deficiencies described above and thoroughly document our experiences.

3. Enabling In-Vivo Enhancement

Our approach to in-vivo enhancement of grid applications leverages the facilities offered by the standard JVM Hotswap API, which imposes serious constraints on what kinds of changes can be made to the swapped classes. In a previous publication [13], we have outlined our basic approach to overcoming these constraints. Next we summarize the main principles underlying our approach and describe how we have perfected it to support grid applications.

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 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

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 limitations of HotSwap described in Section 3.1. A common implementation of indirect referencing is a binary refactoring that we call *Virtual Interface*[1]. 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. As we describe in Section 4.1, 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 1 depicts the Virtual Superclass refactoring transformations. Every class A is changed to extend a *virtual* superclass `Super_A`, with the virtual superclass being inserted

[1]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.

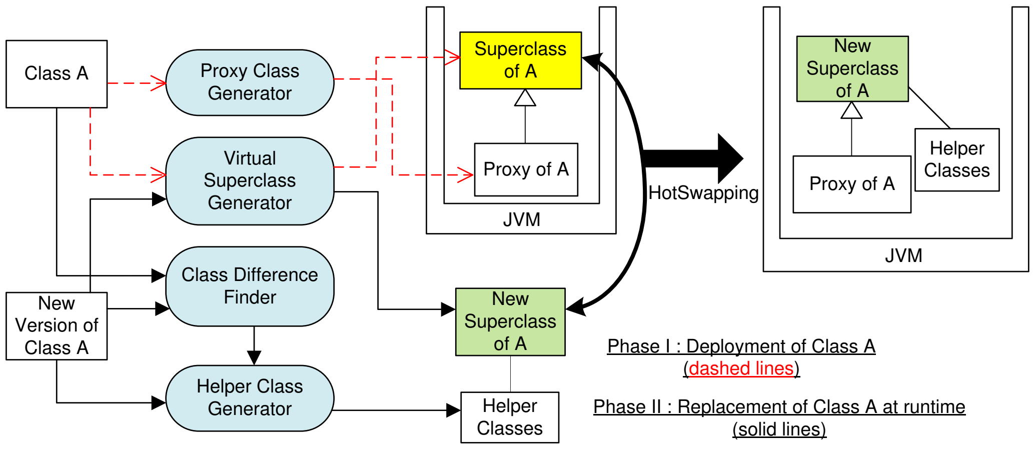


Figure 2: Supporting a full range of dynamic updates using HotSwap.

into the class's inheritance hierarchy.[2] 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 [10] 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 1, 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 [5]. 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 2, refactors all the loaded classes at the bytecode level and generates their corresponding virtual superclasses. The 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 [9], 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

[2]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`.

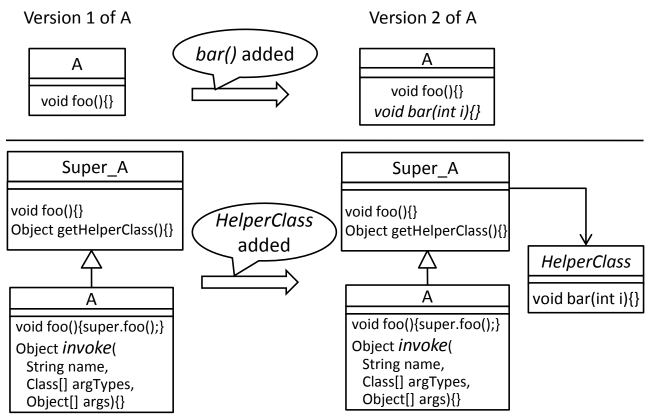


Figure 3: Adding a new method using a helper class.

added method name as the first argument.

Figure 3 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 3 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.

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 uses a separate helper class for the new fields whenever a class is replaced with a newer version. The second uses a single class that contains a mapping data structure that represents all the added fields for all classes.

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

3.4. Support of Language Features and Limitations of the Rewrite

Monitor Concurrency Control For `synchronized` original methods, we leave this attribute only for methods in subclasses only, but remove it from their proxies. Thus, if indirected methods call each other directly, they will lock on the same object, avoiding a potential deadlock.

Reflection The use of reflection to locate and invoke newly-added methods will render our approach invalid, as the new methods are not members of an updated class but of a helper class instead. Fortunately, the use of reflection is uncommon in high performance applications written in Java, and JVM-based languages such as X10 even disallow reflection completely to enable a wider range of optimizations.

Native Code Some of the functionality in JDK is provided as native code, executing as part of the JVM libraries. These native libraries are used in Java to achieve better performance and to obtain access to system resources. Since our approach changes bytecode, it would not be applicable for dynamic updates of native code. We do not foresee the need to update such native code dynamically in this application domain.

Removing methods and fields Our approach does not support the deletion of methods or fields, as the presence of unused methods and fields in a program does not affect its execution with one notable exception. If polymorphic dispatch is present and an overriding method in a subclass is deleted, the overridden method must be invoked in the new version. We ensure this behavior by changing the body of the deleted method in a subclass to delegate to the overridden method in a superclass.

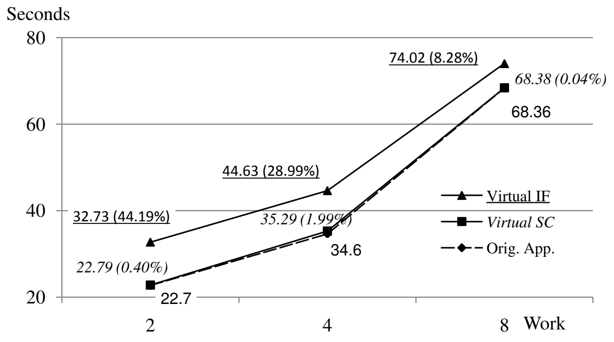


Figure 4: Overhead of binary refactoring microbenchmark.

Class hierarchy changes In addition to ignoring the deletion of methods and fields, which does not affect the program execution, we also do not aim at supporting changes in class hierarchies. While our proxies maintain the original inheritance relationship, which allows subclass proxies to be used in place of a superclass, changing a class inheritance hierarchy is too destructive to the overall structure of a program to be of value for a dynamic update.

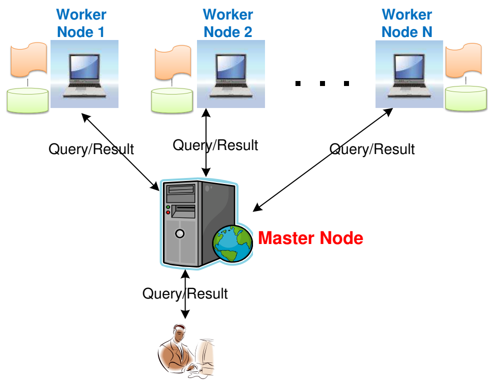


Figure 5: High-level view of parallelizing Smith-Waterman program

4. Evaluation

We have evaluated the performance and effectiveness of our approach. First, we have compared the respective performance of Virtual Superclass and Virtual Interface, and then we have dynamically updated a parallelization of Smith-Waterman algorithm.

4.1. Performance Evaluation

The following micro and macro benchmarks demonstrate the performance advantages of Virtual Superclass. The experimental environment consisted of a workstation with an Intel Pentium 4 (3.6GHz) processor, 1GB RAM, running Ubuntu Linux 7.10 (Gutsy Gibbon), JDK version 1.5.0_14.

The first micro benchmark assessed the overhead of indirecting a single method invocation. Figure 4 shows that the cost of indirection depends on the amount of computation performed by the indirected method. In this benchmark, the indirected method performed two, four, and eight multiplications, increments, and test operations. Each invocation is repeated $1 * 10^9$ times. The maximum overhead of less than 2% makes this refactoring suitable for introducing indirection to the majority of performance-sensitive applications.

To assess the performance of the Virtual Superclass indirection in more realistic programs, we used five different full program benchmarks from SpecJVM98 [4]. Figure 6 shows that similarly to the micro benchmark numbers above, the total overhead of Virtual Superclass never exceeds 2%, whereas for Virtual Interface it can go as high as 16%.

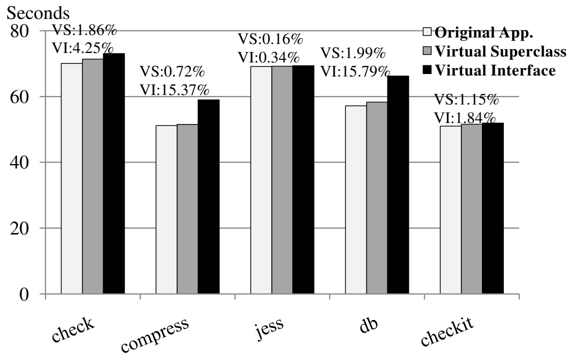


Figure 6: Overhead of binary refactorings on SpecJVM98.

4.2. The Smith-Waterman Parallelization Revisited

Next we demonstrate how the novel binary rewriting techniques presented above enable effective dynamic updates of the Smith-Waterman parallelization used to motivate this work, thereby making it possible to perfect and adapt this application on the fly. As our experimental environment, we have assembled a small grid of five nodes connected by a LAN. We used the Ibis [1] grid infrastructure, even though the grid nodes communicated with each other through Java sockets rather than through the MPJ middleware provided by Ibis. Figure 5 shows an example deployment of the parallelization.

Recall that the required dynamic updates included changing the display method, the precision of the results, and the alignment algorithm used. 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 required 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 the `double` rather that the `float` precision for the similarity scores required 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 algo-

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

Figure 7: Changes to Smith-Waterman program using extended HotSwap. SW:Smith-Waterman algorithm [18], SWG:Smith-Waterman-Gotoh algorithm [12].

rithm required 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, had to be changed accordingly. Figure 7 presents the exact statistics of the changes involved.

For this case study, we have 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.[3] 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 8 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.

5. Related Work

The existing state of the art in dynamic updates includes program transformation, custom virtual machines and runtime libraries, as well as special programming models.

Our approach uses refactoring transformations– changing the structure of a program without affecting its functionality. Orso *et al.*'s technique [16] similarly refactors bytecode to enable its dynamic updates. Our approach differs by using HotSwap, allowing changes to the signatures of swapped classes, and introducing a more efficient implementation of the Proxy pattern. Bialek *et al.*'s system [6] also rewrites the updated software at the source or bytecode levels to enable its dynamic updates; however, instead of replacing classes using HotSwap, which is a standard JVM facility, their approach globally renames classes. Several approaches [17, 15, 11] have introduced custom virtual machines to support dynamic updates of Java applications. These approaches, however, require installing a custom JVM, which may have limited functionality and interoperability. Some approaches [14, 8, 19, 7] introduce new languages features, middleware systems, or require that software developers abide by specific component models or programming rules. Warth *et al.* presents Expanders [19], a new programming language construct that specifies new methods, fields, and interfaces. Expanders enable the programmer to express new methods and fields to be added to an existing program statically. Bierman *et al.*'s UpgradeJ [7] is a Java-like language for upgrading classes dynamically. UpgradeJ is concerned with type-safety for dynamic updates rather than with a particular implementation technique.

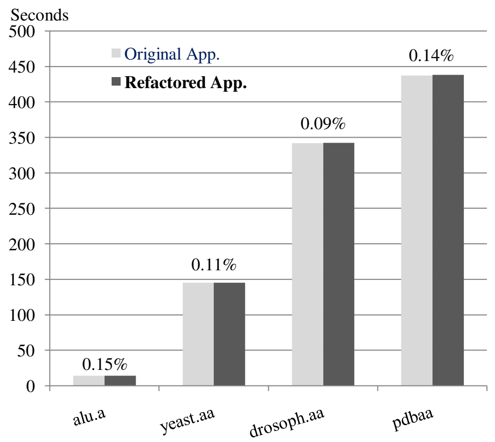
6. Future Work

Possible future work will include applying our approach to larger-scale applications. Meeting the goal of scalability is likely to uncover new research challenges. In addition, improving the usability of our approach further can make 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 can benefit new JVM-based high-productivity languages. Their new language features are likely to pose new challenges for dynamic updates.

7. Conclusions

This paper has presented flexible and efficient dynamic updates for JVM-based, distributed, computationally-

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



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

Figure 8: Refactoring overhead on the worker portion of Smith-Waterman parallelization.
x-axis: the databases names in FASTA format
y-axis: the total execution time.

intensive applications. The presented approach leverages a new binary refactoring that rewrites program binaries, thereby overcoming constraints of the HotSwap API. The resulting flexibility has been demonstrated via a case study of perfecting and adapting a parallelization of a popular bioinformatics algorithm on the fly. The minimal overhead of our approach and the flexibility it affords make it a powerful tool for shortening the time-to-discovery in distributed computationally-intensive applications. Further, as dynamic reconfiguration and maintenance are becoming an indispensable part of evolving modern software systems, our approach can benefit the broader software development community.

Acknowledgments The authors would like to thank Taweesup Apiwattanapong, Dennis Kafura, and the anonymous reviewers, whose comments helped improve the paper.

References

[1] Ibis: Grids as Promised, `http://www.cs.vu.nl/ibis/`.

[2] JAligner, `http://jaligner.sourceforge.net/`.

[3] Java HotSwap, `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/enhancements.html`.

[4] Specjvm98 benchmarks, `http://www.spec.org/jvm98/`.

[5] 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.

[6] R. P. Bialek. Dynamic updates of existing Java applications. *Ph.D. Thesis, the University of Copenhagen*, pages 1–216, June 2006.

[7] G. Bierman, M. Parkinson, and J. Nob. UpgradeJ: Incremental typechecking for class upgrades. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2008.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

[13] 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.

[14] Y.-F. Lee and R.-C. Chang. Java-based component framework for dynamic reconfiguration. *IEE Proceedings - Software*, 152(3):110–118, June 2005.

[15] 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.

[16] 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.

[17] T. Ritzau and J. Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, London, May 2000.

[18] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[19] 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.