УДК 004.43 https://doi.org/10.15407/pp2026.02.077

D. Shcherbak, K. Zhereb

IMPLEMENTATION OF HOT RELOADING IN COMPILED PROGRAMMING LANGUAGES

Hot reloading is a powerful software developing tool which allows the programmer to make changes to the codebase, and see those changes be applied to the program while it is running. This feature is naturally easy to implement in interpreted programming languages, such as Python, JavaScript and Ruby. This feature is highly demanded in compiled languages as well, because it allows to write and debug programs much faster and simpler, without the need to recompile the whole project to test out new functionality. It is especially useful when working on user interface or investigating problems that require the program in a specific state, which is hard or time-consuming to reproduce. This paper explores existing approaches to solving this problem in compiled languages. We take a look at approaches used in languages with additional runtime, such as Java and C#, and explore languages which are compiled to native binaries, such as C++, Rust and Zig. We also discuss current challenges that arise with this technology, and what solutions are possible in new generations of programming languages.
Keywords: hot reloading, hot swapping, dynamic software updates, dynamic library reload, runtime code patching, programming languages

Д.В. Щербак, К.А. Жереб

РЕАЛІЗАЦІЯ ГАРЯЧОЇ ЗАМІНИ КОДУ В КОМПІЛЬОВАНИХ МОВАХ ПРОГРАМУВАННЯ

Гаряча заміна коду – це потужний інструмент для розробки програмного забезпечення, який дозволяє програмісту вносити зміни коду і бачити їх вплив на програму під час того, як вона виконується. Цю функцію легко реалізувати в інтерпретованих мовах програмування, таких як Python, JavaScript та Ruby. Ця функція також дуже затребувана в компільованих мовах, оскільки вона дозволяє писати та відлагоджувати програми набагато швидше та простіше, без необхідності перекомпілювати весь проєкт для тестування змін. Це корисно під час роботи над інтерфейсом користувача або у процесі дослідження проблем, які вимагають фіксацію програми у певному стані, особливо, якщо відтворення такого стану займає багато часу. У цій статті досліджуються існуючі підходи до вирішення цієї проблеми в компільованих мовах. Ми розглядаємо підходи, що використовуються в мовах з додатковим середовищем виконання, таких як Java та C#, та досліджуємо мови, які компілюються безпосередньо в бінарні файли, такі як C++, Rust та Zig. Ми також обговорюємо поточні проблеми, що виникають з цією технологією, та рішення, можливі в нових поколіннях мов програмування.
Ключові слова: гаряча заміна коду, динамічні оновлення програмного забезпечення, динамічне перезавантаження бібліотек, виправлення коду під час виконання, мови програмування

Introduction

Hot reloading (or hot swapping) refers to the ability to modify the source code for existing programs and see the changes appear in the program without having to restart it [1]. This feature is native to interpreted and dynamic languages since they often run straight from the source code. Hence, changes to the source code will be instantly reflected in the program behavior. By contrast, compiled languages, which produce executables tailored to the target system, face some difficulties implementing this feature. This paper researches the history of the hot reloading feature, from its first implementations in dynamic languages to adoptions to current popular compiled systems languages like C++, Java and C#. We also discuss limitations that might be imposed on this feature in compiled languages, for instance immutable function definitions and class definitions, and explore how those limitations can be overcome in the future.

© Д.В. Щербак, К.А. Жереб, 2026
 CC BY 4.0

Hot reload in dynamic languages

Hot reloading has its roots in the earliest dynamic languages and environments. For example, Smalltalk, which was introduced in the 1970s, presented programmers with an environment which allowed them to make changes to the classes during the debugging session [2]. The simple approach of Smalltalk, which relies on objects, their methods and messages being sent between those objects, provides a uniform interface for all language features, and allows making changes in runtime easily.

Common Lisp also allows for reassigning symbols at runtime [3]. This functionality can be implemented thanks to the interpreter that runs the program and can incorporate the changes at runtime, pausing the execution if needed and providing safety checks to ensure validity of program state. Since code isn't converted to binaries, the runtime environment can manage updated definitions of methods and functions, calling the new version after any changes were made. Following Lisp and Smalltalk, Python also incorporates a similar approach: as an interpreted language, it allows for changes in functions and class definitions in a live developing environment, and it can also load changed versions of modules on the fly using importlib.reload [4]. This flexibility usually comes at the cost of efficiency and runtime execution speed, which are typically lower in interpreted programming languages. This paper addresses the challenge of bringing hot reloading capabilities to compiled languages without the need to sacrifice the efficiency of compiler optimizations.

Hot reload in compiled languages

By contrast, compiled languages achieve significantly better performance by producing machine-readable artefacts, which are tailored to the target operating platform. This approach allows to perform great performance optimizations – for example, unused variables might not be calculated at all [5].

The possibility of using hot reloading in compiled languages was intriguing. Java introduced HotSwap [6] in 2002, and C# added EditAndContinue [7] in 2004, introducing some limited hot swapping features, mainly in debugging mode, for the convenience of the developer. These languages run on virtual machines (JVM for Java) and managed runtimes (.NET CLR for C#), which allows for easier implementation of hot swapping, than in languages with fully native binaries. On the other hand, systems languages like C++ and Rust historically lacked built-in hot reloading, but various attempts of implementing it were made. In this section we discuss the most notable implementations of hot swapping in Java, C#, C/C++ and some other compiled languages, focusing on their capabilities and constraints.

Java and the JVM

Some hot reload functionality can be accessed in Java through the JVM HotSwap mechanism [6]. It was first introduced in Java 1.4, and gives the ability to change method bodies while in debug mode on a live program. This can be useful while fixing minor bugs, but it imposes major limitations: developers are not allowed to add or remove methods and class fields, change their type, name and access level, or mutate class hierarchy. The original design of HotSpotVM held an assumption of immutability of classes during runtime, hence all changes to class hierarchy still required full application restart. This limitation prompted further research, such as the Dynamic Code Evolution VM (DCEVM), which was a patch for HotSpotVM developed by Thomas Würthinger in the Institute for System Software in Linz [8]. It drastically increased the abilities of classic HotSwap, allowing for changing class fields, adding and removing methods, and changing class hierarchy. This is achieved using a special $transformer method, which is applied to every object on the heap of the JVM, which initializes new fields, and helps ensure the consistency of class invariants. This instrument is very powerful, allowing for many types of changes to the running program, with no ongoing runtime cost – benchmarks show [8] that the program

returns to original performance metrics after the changes are applied. The drawbacks of this approach might include the slightly longer reload times, additional complexity and the fact that this is JVM-specific: the same virtual machine that provides all the class information at runtime, which is not available in system languages like C++, introduces additional memory and performance overheads. However, with so many advantages for a considerably low price in terms of performance, the adoption of this instrument is low: the project repository wasn't updated for six years as of time of writing this paper. The fact that DCEVM is a separate tool could be a factor that limits the adoption. It needs a separate installation and users need to learn how to use it alongside usual Java tools. We believe that if such powerful hot reloading was a native language feature, with the main language features built around it, much more developers would be using it.

Other notable implementations include JDrums [9] and JVolve [10]. JDrums implements a dynamic JavaVM, which can allow for many runtime changes, but are limited in changes to class hierarchy. JVolve uses Jikes ResearchVM to implement dynamic code updates, but falls short of DCEVM as well, as it is not able to change running methods.

.NET and C#

In the ecosystem of .NET framework and the C# language, the ability to make changes while the program is running was represented since the early 2000s by the "Edit and Continue" feature [7]. This feature was available in the official .NET IDE, the Visual Studio, and allowed for minor changes to be done while the program was in Debug Mode. Those changes will then be applied to the program, without the need to restart the application. However, the number of changes that were available was quite limited, primarily changes inside method bodies were allowed, with no changes to class hierarchy or layout.

This feature was recently extended substantially by the new .Net Hot Reload feature, released first for .Net 6 in 2021 [11]. First major improvement was the ability to make changes without the need for explicit breakpoints, while the Edit-And-Continue feature required the program to be completely stopped before you could make any modifications to the running executable. The number of changes that are allowed was also significantly increased: the new .Net Hot Reload supports adding methods, fields, constructors, properties to classes, including async methods and changing method's return type [12]. However, some changes still remain unavailable, such as changes to the interfaces, adding a destructor to existing class, modifying a class parameter.

Akin to Java using its JVM to smooth out hot reload, .NET is using its Just-In-Time compiler to seamlessly introduce updated method bodies to the existing application. However, some systems are not suited to incorporate the additional overhead of Just-In-Time compilation and Garbage Collection, and instead rely on low-level languages, such as C and C++. In the next section we review the state of hot reloading in C++, as an example of a language that is compiled to a native executable, and how it handles hot reloading.

C and C++

Implementing hot reloading in languages that compile directly to machine code is a much more challenging task. It is further complicated by major compiler optimizations, which can drastically change the layout of instructions in the executable. There are two main approaches to this problem. The first approach is Dynamic Library Reload, described in [13]. This is achieved by splitting a program into a host subprogram, which starts the application and calls the main executing functions, and the rest of the logic, which is stored in Dynamic Link Libraries (DLL) or Shared Objects (SO). When the new version of a function is compiled, the host loads the new Shared Object instead of the old one and redirects function pointers, ensuring that all future calls will execute the new logic. The state of the app is stored by the host, so that changing the Shared Objects will not affect the application

data. This approach has some drawbacks: it requires ahead-of-time planning. Since changes cannot be applied in the host part of the application, signatures of the functions being called from Shared Objects cannot change, and need to be decided in advance. Changing object layouts in memory is also problematic with this approach, since state is most often managed by the immutable host, and if new SOs try to access objects with a different schema, they will encounter unexpected errors [14].

Another approach to hot reloading in C++ is Runtime Code Patching, described in [15]. This way, the changes are made directly to the executable code of the process in memory. This is a much harder task, since compilers are relying on the fact that the program code is static, and can use this information, as was mentioned, to perform optimizations. One of the tools that can be used to achieve this is Live++, a proprietary tool developed by Molecular Matters [16]. It requires the program to be compiled with special build flags that add a small patchable entry point at the beginning of each function. This way Live++ can easily insert a jump instruction which provides an indirection for the execution flow, allowing to delay execution before changes are made to the target function, and then return to normal execution, or if the size of the function is growing more than the allocated scope allows, the function instructions can be moved to a different slot in memory altogether. Another powerful feature is Hot Restart: if the changes made by the developer cannot be handled in runtime, they will be applied through the full program restart. However, since the app was already precompiled, the changes can be applied much quicker, skipping most of the warmup time. All in all, Live++ provides a great set of tools to perform hot reloading in C++, allowing for many types of changes during runtime with minimal overhead. A few of its drawbacks include some compile parameters limitations: the /FUNCTIONADMIN linking parameter is compulsory and some optimization flags, like Whole-Program Optimizations and Link-Time Optimizations are prohibited. It is prohibited to introduce new global or static variables into thread-local storages, and some behaviour changes might occur when debugging a patched program.

From the academic side, the most notable solutions for dynamic software updating in C and C++ included Ginseng, which implemented hot reloading for C in [17], and later improved it with an algorithm for a multi-threaded solution in [18]. However, its approach required automatically introducing additional variables to each struct and function to provide indirection, which could cause performance issues. A later project Kitsune [13] proved to outperform it in terms of runtime update speed, while maintaining the ability to make almost arbitrary changes to the program. This, however, was achieved by a requirement for the developer to explicitly mark code points, in which hot reload is possible. While such technique can work as a safety guarantee (or at least shift the blame for the inconsistencies on the developer, who didn't properly choose the update points), it makes the adoption process of Kitsune much harder for many projects, especially for those with an extensive code base.

Other notable implementations

Zig, as a low-level language that is compiled to a native executable, can easily implement the dynamic library reload, introduced in the C++ section of [19]. An implementation of runtime code patching was proposed in 2022 by one of the language maintainers [20]. The proposal outlined an incremental compilation mode, which left the compiler running as a background service. After receiving a command to update the live executable, the compiler determines which changes need to be made and updates the corresponding memory of the running process. This is only implemented as an experimental feature, and no handling of program state was described, but it shows potential for future implementations of hot reloading in Zig, leveraging its low level access to memory.

Erlang is a programming language built around the idea of scalable, fault-tolerant systems with long running time [21]. The main idea of Erlang is implementing multiple

concurrent processes, that don't share any memory or state, and communicate via asynchronous messages. Those processes are designed in a way that makes them easy to halt and start new instances, which provides a perfect environment for hot reloading. Erlang supports multiple versions of the same process running at the same time, with the main assumption that all old processes will eventually die out, and only the newest version processes are created as soon as it is compiled. Though Erlang is using a virtual machine, which restricts it from being used in performance-critical environments, its ability to change running systems, among other advantages, made it widely used in telecommunication systems, bankings and more.

Rust is another example of a low-level language, that is known for its long compile times, and, as such, would benefit greatly from hot-reloading capabilities. There are solutions for Dynamic Library Reload like dynamic_reload [22] or hot_lib_reloader [23], but no implementations of Runtime Code Patching were introduced. With **Go**, another popular compiled language, the most successful solutions like Air [24] provide the capability of Live Reloading, which essentially fully restarts the application in an automated way.

Mun [25] is a programming language in the early stages of development, which aspires to have a Rust-like syntax and feature set, combined with native capabilities of hot reloading. As of writing of this paper, it is too early to tell how it compares in terms of speed, and patches runtime overhead to other solutions in this paper. However, it shows promise, and has potential to become a great example of native hot-reload implementation in a compiled language.

Proposed Approaches for High-Performance Hot Reload

Since hot-reloading is a powerful and useful tool, we believe it is very likely to be a part of new languages that will be developed in the future. Yet, it is complex and hard to implement, especially post-factum in a language that is already in use, because current memory layouts in existing programs cannot be changed.

One way to overcome this is to start building the compiler around the idea of hot-reloading, extensibility and improvability. Most popular compiled languages don't support hot-reloading out of the box. We believe that possible growth of hot reload implementations and usage is limited, partly, by the general approach, which is common for most compiled languages – treating the executable as an immutable artefact.

Of course, such a way of thinking has its benefits. It provides for encapsulation, modularity and more strict separation of responsibilities. And in the early days of computing it was enough: simple programs that automated mathematical tasks, such as simple physics simulations, followed the same path. The program started at a clean slate, allocated and initialized some resources, executed some computations, freed the resources and died. Today, many programmers work on long-running servers that might have uptime of days, weeks and months. Yet, for compiled systems languages, these servers still need to be put to a complete stop for every change that needs to be introduced. Entire systems, like Kubernetes, were implemented for ensuring the uptime of big systems, and one of the features they provide is an ability to update a running service to a newer version without disrupting the users' workflow. This is achieved by starting multiple instances of the same service, and gradually switching some of them to a newer version, while redirecting users to them. While this solution is elegant, it does introduce a certain level of complexity, which is only required because the executables that are running the servers are immutable. Perhaps, it would be beneficial to write programs in an environment that allows them to change while running, providing evolution of systems, not revolutionary (breaking) changes.

Performance-critical systems, which cannot allow for large overhead of an orchestrator, such as Kubernetes, would still benefit from an ability to upgrade its software. In 1988, part of

a space probe software, which was written in Lisp, malfunctioned, and the mission was saved, because Lisp allowed to debug and change its code while live and in the process of a spaceflight [26]. While this shows how important it is to be able to debug and improve the program in runtime, a strict type system and compile-time checks can prevent many cases of undesired behavior. Combining both the safety of a compiler with the flexibility of hot reloading is a way to achieve the most stable and adaptable systems.

Compiler built around Hot Reloading feature

As was mentioned earlier, implementing the Hot Reload feature in an existing language is quite difficult, and it almost always results in some overhead. For example, in compiled OOP-languages, like Java and C++, one of the issues that could arise is related to the object's layout in memory. If a new field is added to an object via Hot Reloading, how to add this field to existing objects?

Such a problem would be solved, if the compiler is designed with the idea of hot reloading in mind. Then all the tooling, memory layouts and runtime can be implemented in such a way that allows to change them easily, even while the program is running. This also allows the implementation of different approaches to hot-reloading, exploring different trade-offs.

One approach is to do memory mapping in a way that makes the addresses of functions in memory predictable and easy to calculate. This allows to make the hot swapping process as fast and painless as possible, but might result in the final executable being larger, which sometimes could be an issue, if the target machine doesn't have a lot of spare memory. Alternatively, it is possible to provide the most efficient memory layout for the executable, and calculate which parts need to be changed at each hot swap event, factoring in additional factors like compiler optimizations. This will result in smaller executable sizes and, in some cases, faster code, for the cost of longer hot-reload times. Building the compiler with the concept of hot reload in mind allows to implement both approaches much easier. And of course, such a compiler can still be used in an old-fashioned way: just a simple executable, no hot-reloading capabilities, for the sake of security, speed and reliability.

This way of thinking, in terms of evolving, runtime-developed systems, allows for potential new approaches to programming in general. As an example, it might be possible to write some interface with default implementations, and combine it with logic that will invoke new implementations, when they are implemented, and until then will fall back to the default implementation. In other words, it becomes possible to not only reason about what the code *is*, but also about what it *will be*.

Proposed compiler architecture

In this section we present a concept of a future approach to compiler architecture, for a statically compiled language to achieve maximum native hot reloading capabilities. We propose to combine the Dynamic Library Reload (DLR) approach with Runtime Code Patching (RCP). In principle, DLR allows for the easiest approach, and the one providing the least overhead. However, it has its limitations in terms of program state and function signatures, which were already discussed earlier. RCP, in turn, allows for more flexibility, but can cause more additional overhead. What is meant here by combination of those approaches, is that the compiler separates the code, designating some of it to dynamic libraries for ease of hot swapping in runtime. At the same time, the controlling unit, which in regular DLR approach is left immutable, is compiled in a way that allows the use of the RCP approach, allowing for changes here as well, with the possibility of state transformations. This way it is possible to allow for maximum flexibility, while keeping memory overhead to a minimum. What is also important, compiler optimizations also take this layout into account, so the developer won't be forced to choose between powerful optimizations and hot reloading. The proposed design is presented at Figure 1. This approach

doesn't require a separate process to handle hot reloading, as it is done by the compiler itself, which can be executed as a background process. When the hot reload is triggered, the compiler checks for changes since the last compilation, compiles them, without recompiling the whole project, but using previous compilation info to provide type checks and other validation. Upon successful compilation, a patch with new shared object files and changes necessary to the main unit are applied to the running executable.

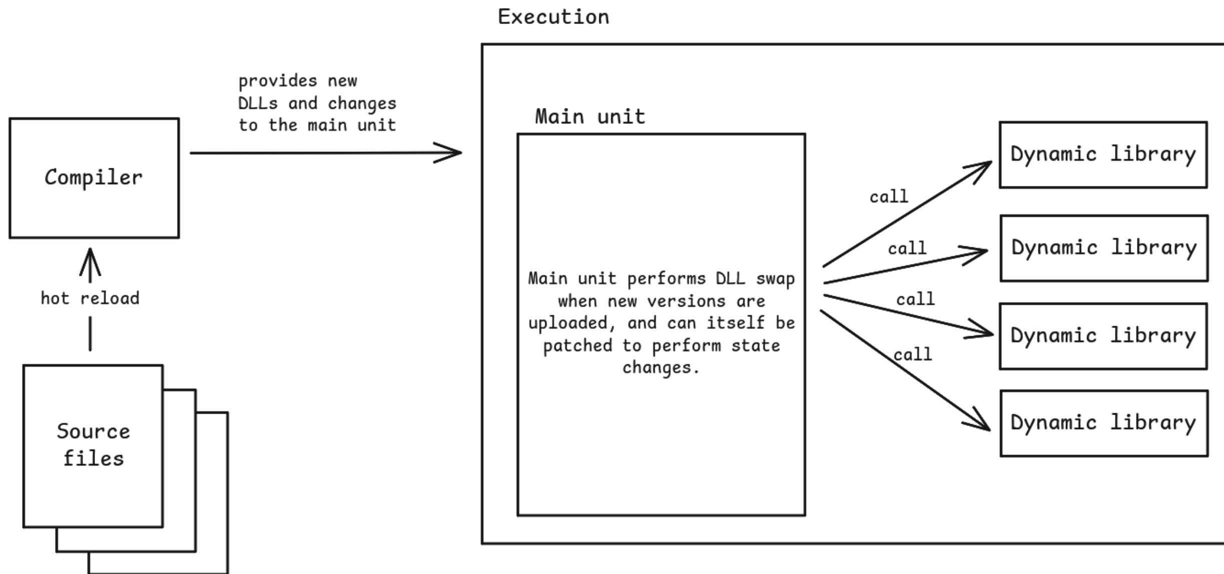

Fig. 1. Proposed compiler architecture

Prototype implementation

To test how two approaches can be used together, we implemented a simple proof of concept project. The source code is available on GitHub at [27], and general representation of the project workflow is presented at Figure 2. The main loop iterates through calls to simple mathematical functions, one of which is loaded from a dynamic library via DLR module, and accessed through a function pointer, and another one is compiled in a static module and directly called from the main loop. Both functions can be modified, and after a call to the compilation system the changes will be injected into the main loop, either by swapping the shared object for a newer version, or by providing a byte sequence to the RCP module, which is directly injected into the running program memory. A compilation system in this context is a standard C++ compiler, combined with a number of simple bash scripts, needed to perform file substitution and pass build parameters to the compiler. This project also shows that DLR is capable of loading completely new functions, while RCP cannot do that. RCP on the other hand allows much more flexibility, but requires more preparation and is harder to implement. Benchmarking of this project showed that patching time varies for the DLR approach, with bigger batches taking more time to load. With RCP, bytes can be loaded independently of the main program running, so the time for which execution is interrupted is limited to inserting the jump instruction to the code, and does not grow as patch gets bigger, as can be observed in Table 1.

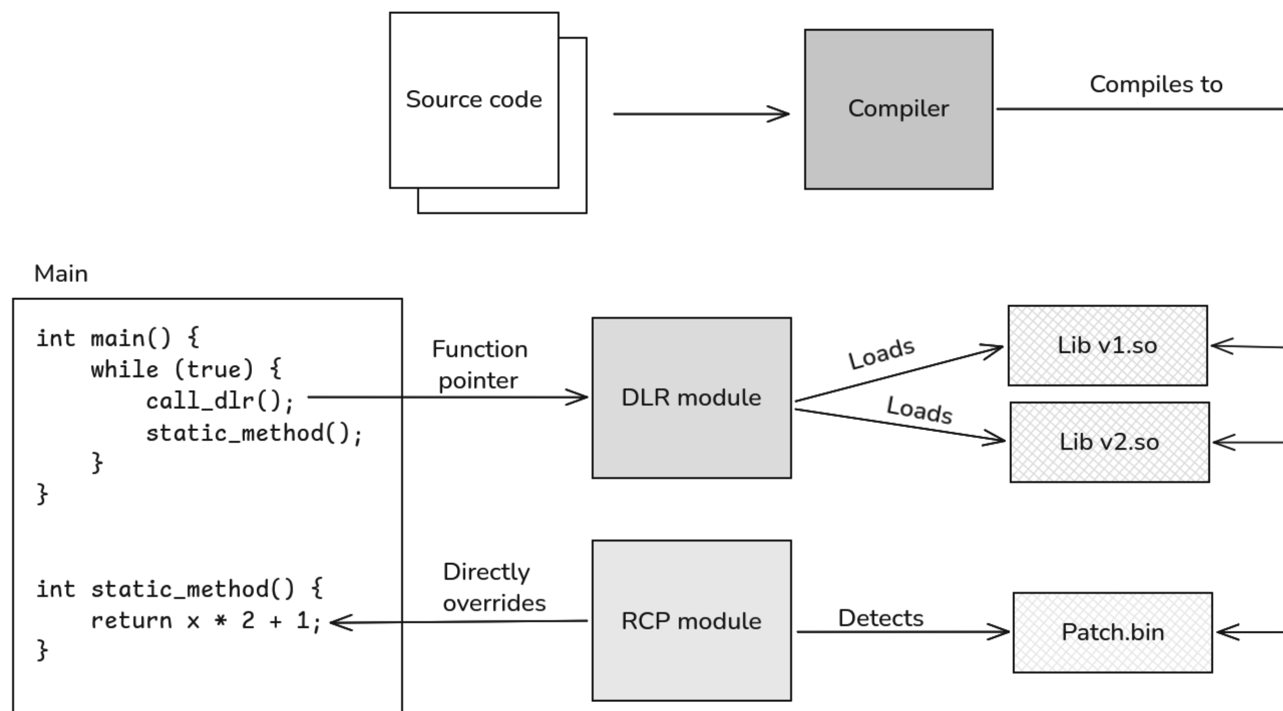

Fig. 2. Implementation of combined DLR and RCP method

Table 1.

Benchmarking results for different hot reload approaches

Patching mechanism	Mean patch time (ns)	Min patch time (ns)
RCP, ~40 bytes	2073	1607
RCP, ~300 bytes	1993	1594
DLR, 1 symbol	278	267
DLR, 1000 symbols	29565	27985

Conclusion

In this paper we explored the history of the hot reloading feature in different programming languages, starting with the first examples, which came from interpreted languages. We then discussed approaches to hot reloading in the most popular compiled languages, ranging from academic results, such as DCEVM for Java, to proprietary solutions like Live++ for C++. We also considered implementations and proposals thereof in less popular languages. In later sections, we discussed the importance of the hot reload feature for modern and future programming languages, and proposed an approach of incorporating this feature early in the development of future programming languages, to ease its implementation and usage. Lastly, we introduced a proposal for compiler architecture, which allows for non-restrictive and fast hot reloading by combining the Dynamic Library Reload with Runtime Code Patching. The proof of concept implementation, provided in the last section, confirmed the benefits of using both approaches, and provided a foundation for a future full-scale compiler, built around the idea of hot reloading.

References

1. Appavoo, J., et al. (2003) ‘Enabling autonomic behavior in systems software with hot swapping’, in *IBM Systems Journal*, vol. 42, no. 1. Armonk, NY, USA: International Business Machines Corporation, pp. 60–76.

2. Kay, A.C. (1996) ‘The early history of Smalltalk’, in History of Programming Languages---II. New York, NY, USA: Association for Computing Machinery, pp. 511–598.
3. Steele Jr, G. L. (1990). *Common Lisp the Language* (2nd ed.). Digital Press.
4. Python Software Foundation (2025), *importlib — The implementation of import.* Available at: https://docs.python.org/3/library/importlib.html (Accessed: 15 October 2025)
5. Godbolt, M. (2020) ‘Optimizations in C++ compilers’, *Commun. ACM*, 63(2), pp. 41–49.
6. Oracle Corporation (2025) *HotSwap*. Available at: https://wiki.openjdk.org/spaces/mlvm/pages/7897093/HotSwap (Accessed: 15 November 2025)
7. Microsoft (2025) *Edit and continue for C#.* Available at: https://learn.microsoft.com/en-us/visualstudio/debugger/edit-and-continue-visual-cshar (Accessed: 30 October 2025)
8. Würthinger, T., Wimmer, C. and Stadler, L. (2010) ‘Dynamic code evolution for Java’, in *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*. New York, NY, USA: Association for Computing Machinery (PPPJ ’10), pp. 10–19.
9. Andersson, J. and Ritzau, T. (05 2000) ‘Dynamic Code Update in JDRUMS’.
10. Subramanian, S., Hicks, M. and McKinley, K. (06 2009) ‘Dynamic Software Updates: A VM-centric Approach’, in, pp. 1–12.
11. Lyalin, D. (2021) *Introducing the .NET Hot Reload experience for editing code at runtime*. Available at: https://devblogs.microsoft.com/dotnet/introducing-net-hot-reload (Accessed: 01 November 2025)
12. Microsoft (2025) *Write and debug running code with Hot Reload in Visual Studio.* Available at: https://learn.microsoft.com/en-us/visualstudio/debugger/hot-reload (Accessed: 01 November 2025)
13. Hayden, C.M. et al. (2012) ‘Kitsune: efficient, general-purpose dynamic software updating for C’, in Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. New York, NY, USA: Association for Computing Machinery (OOPSLA ’12), pp. 249–264.
14. Zylinski K. (2023) Hot Reload Gameplay Code: What, why, limitations and examples. Available at: https://zylinski.se/posts/hot-reload-gameplay-code (Accessed: 1 November 2025)
15. Buck, B. and Hollingsworth, J.K. (2000) ‘An API for Runtime Code Patching’, in *The International Journal of High Performance Computing Applications*, vol. 14, no. 4. Thousand Oaks, CA, USA: SAGE Publications, pp. 317–329.
16. Molecular Matters GmbH (2025) *Live++ for Windows - Documentation.* Available at: https://liveplusplus.tech/docs/documentation.html (Accessed: 30 October 2025)
17. Neamtiu, I. et al. (2006) ‘Practical dynamic software updating for C’, in Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery (PLDI ’06), pp. 72–83.
18. Neamtiu, I. and Hicks, M. (2009) ‘Safe and timely updates to multi-threaded programs’, in Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery (PLDI ’09), pp. 13–24.
19. Perkin, L. (2024) *Hot-reloading with Raylib.* Available at: https://zig.news/perky/hot-reloading-with-raylib-4bf9 (Accessed: 26 October 2025)
20. Konka, J. (2022) *Hot-code reloading on macOS/arm64 with Zig.* Available at: https://www.jakubkonka.com/2022/03/16/hcs-zig.html (Accessed: 8 November 2025)
21. Armstrong, J. (2007) ‘A history of Erlang’, in Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. New York, NY, USA: Association for Computing Machinery (HOPL III), pp. 6-1-6–26.
22. The Rust Foundation (2025) Dynamic_reload crate documentation. Available at: https://crates.io/crates/dynamic_reload (Accessed: 26 October 2025)
23. The Rust Foundation (2025) hot_lib_reloader crate documentation. Available at: https://crates.io/crates/hot-lib-reloader (Accessed: 26 October 2025)
24. Nguyen, H. (2024) *Guide for Implementing Live Reload Using Golang Air.* Available at: https://reliasoftware.com/blog/live-reload-using-golang-air (Accessed: 14 November 2025)
25. Mun language documentation. Available at: https://docs.mun-lang.org/ (Accessed: 15 November 2025)

26. Cassel, D. (2022), *NASA Programmer Remembers Debugging Lisp in Deep Space.* Available at: https://thenewstack.io/nasa-programmer-remembers-debugging-lisp-in-deep-space (Accessed: 27 October 2025)
27. Implementation of the prototype described in the paper. Available at: https://github.com/DShcherbak/hot-reload-poc (Accessed: 23 March 2026)

Дата першого надходження до видання: 26.03.2026
Внутрішня рецензія отримана: 14.04.2026
Зовнішня рецензія отримана: 20.04.2026
Дата прийняття статті до друку:05.06.2026
Дата публікації: 29.06.2026

Про авторів:

Денис Щербак,
аспірант
Denys Shcherbak
post-graduate student
https://orcid.org/0009-0006-7842-565X

Костянтин Жереб,
кандидат фізико-математичних наук, асистент кафедри
Kostyantyn Zhereb
Ph.D., (physical and mathematical sciences), assistant professor
https://orcid.org/0000-0003-0881-2284

Місце роботи авторів:

Київський національний університет імені Тараса Шевченка,
факультет комп’ютерних наук та кібернетики
Kyiv Taras Shevchenko National University,
Faculty of computer science and cybernetics
тел. +38(044) 521-32-74
E-mail: csc@knu.ua
https://csc.knu.ua