Stitching - Patching of Call Sites for Better Dynamic Linking Performance

Daniele Vettorel <dv300>

Abstract—Virtually every application relies on a number of software libraries in order to provide their functionalities. Such libraries can be either statically or dinamically linked to the main application. While static linking ensures the dependency on a single version of the library that is embedded in the final binary, dynamic linking is often preferred for its convenience and the smaller overhead in size. Dynamic linking, however, comes with a runtime performance cost, as the operating system's dynamic loader has to lookup the symbols in the dynamic libraries. This mechanism is implemented with the help of the linker, which inserts trampolines in place of function calls. A trampoline requires the processor to jump twice and access an in-memory table to retrieve the actual function's address.
Building on previous work that proposed a hardware implementation of a trampoline-skipping mechanism, I investigate software solutions instead. I propose a number of implementations at different levels in the pipeline that aim to remove the overhead of function trampolines by replacing indirect calls with direct calls at either link time or run time. I show the effects of this technique, which I call *stitching*, on different applications, including a fully-fledged Apache web server.

Index Terms—Software performance, Software systems, Compilers, Dynamic linking, Dynamic code generation

1 Introduction

Software libraries are commonly used to build applications as they make it possible to access a set of functionalities via an external interface that hides the internal implementation. Code reuse and the convenience of relying on thoroughly-tested code bases are the major factors that push software developers to work with existing libraries instead of writing their own anew. When it comes to integrating (*linking*) an external library into an application, developers are faced with the choice of whether to link it statically or dynamically. Static linking works by copying the library's code and data sections into the final application binary, while dynamic linking works by keeping a reference to the dynamically-linked libraries in a special section of the binary, so that they can be loaded at run time. It is clear that both approaches have their own advantages and shortcomings. Static linking ensures no external dependency, but at the cost of an increased file size. In addition, since the library is copied into the application, it is not possible to simply update the library without linking the whole application again. On the other hand, dynamic linking makes it possible to update the external library independently from the main application (as long as the interface has not changed), doesn't result in a bigger binary, and allows for the sharing of memory pages in case the same library is being used by multiple applications at the same time. The drawbacks of dynamic linking, however, include the so-called *dependency hell*, whenever a specific version of the library is required in order to correctly run the application [1]. Moreover, dynamic linking results in degraded run time performance as the libraries need to be loaded on the spot, and calls to functions defined in shared libraries require the use of a *trampoline* function that looks up the actual function's address and then calls it. Despite this, dynamic linking is the method of choice for the majority of applications for its convenience. A proposal to implement a hardware mechanism to speed up function trampolines was put forward by Agrawal et al. [2], and a software-based simulation was explored as well. In this article, I analyse the overhead induced by function trampolines in an UNIX environment and propose different software solutions to minimize the run time cost of dynamic linking, all based on the idea of substituting indirect trampolines with direct calls whenever possible, technique which I call *stitching*. I explore different implementations at the compiler level and at the linker level by extending the LLVM [3] codebase, and reason about their benefits, disadvantages and implications in the overall picture of a production-level software system. I provide a set of benchmarks to gauge the trade-offs of every solution, which are tested against a micro-benchmark, a simple multi-threaded neural network workload, and a production-level multi-process server [4].

This article has been produced as coursework for the L25 Modern Compiler Design 2017-2018 module at the University of Cambridge, part of the MPhil in Advanced Computer Science.

2 Overview of the dynamic linking mechanism

In this section I present an overview of the dynamic linking mechanism used by many UNIX-based operating systems, including most Linux distributions. I assume an x86-64 architecture and that the file format for object code, shared libraries and executables is the ELF file format, according to the System V AMD64 ABI [5]. The overall mechanism relies on two main actors to work: the linker, usually part of the compiler tool-chain, and the dynamic linker, part of the operating system. The linker, while processing relocations found in object files, determines whether the symbol being relocated comes from a shared (dynamically-linked) library.

```
#include <stdio.h>
int main()
{
    printf("Hello world!");
    return 0;
}
```

```
pltheader:
    PUSH QWORD PTR [rip+0xe52] # Push second GOT entry
    JMP  QWORD PTR [rip+0xe54] # Transfer control to dynamic linker
    NOP  DWORD PTR [rax+0x0]   # 5-byte NOP
printf@plt:
    JMP  QWORD PTR [rip+0xe52] # Jump to function's GOT entry
    PUSH 0x0                   # Push relocation index
    JMP  pltheader             # Transfer control to PLT header
main:
    ...
    CALL printf@plt            # Call trampoline
    ...
```

Fig. 1: PLT trampoline in a C *Hello World* program

If that is the case and if the symbol represents a function, a trampoline function is created in a special section of the resulting ELF file, the *Procedure Linkage Table* (PLT) section. The linker also allocates an 8-byte space (the size of an address) in the *Global Offset Table* (GOT) section.

The trampoline function is a three-instruction procedure where the first instruction loads and jumps to the address pointed to by the GOT entry, the second pushes a *relocation index* to the stack, and the third jumps to a special PLT header function. The PLT header function is common to all functions that need dynamic linking and is also a three-instruction procedure that transfers control to the dynamic linker, which then proceeds with the symbol lookup. The call to the dynamic linker is executed in two instructions, where the first one pushes the second entry in the GOT section containing information about the current executable, and the second one jumps to the dynamic linker's lookup procedure. The third instruction is just a 5-byte NOP to align the PLT header function to a 16-byte boundary.

When the executable is loaded into memory for execution, the GOT entry initially holds the address of the second instruction of the PLT trampoline. Therefore, when the PLT trampoline is executed for the first time, the jump to the address in the GOT entry will simply transfer control to the next instruction of the trampoline. The relocation index is then pushed onto the stack and will be used later by the dynamic linker to identify which GOT entry to modify. When control is then transferred to the PLT header function, the second GOT entry is pushed on the stack to be used as identifying information by the dynamic linker, which is then invoked. The symbol is looked up and the GOT entry that refers to it is then patched with the function's actual address. Finally, the dynamic linker jumps to the function that has been looked up, which will execute and return to the instruction after the original call site. When the PLT trampoline is executed after the first time, the GOT entry will contain the desired function's address and therefore control will be transferred directly to it without going through the PLT header function and the dynamic linker again.

The dynamic linker's lookup function initially saves all registers used to pass parameters, according to the System V x86-64 calling convention (§3.2.3 of [5]), and then restores them before transferring control to the target function. This is to ensure the trampoline mechanism does not tamper with any of the parameters that the main application intended to pass to the shared library's function. Figure 1 shows an excerpt from a disassembled *Hello World* program that contains a trampoline for the `printf` function, which is defined in the dynamically-linked `libc` library.

3 Analysis of Trampoline Overhead

A trampoline mechanism as described in the previous section introduces performance overhead. While generally low, such overhead is not zero, and can be analysed with respect to a direct function call. I do not consider the dynamic linker's overhead in this analysis, but only the cost of going through the PLT and the GOT.

According to Agner Fog's Instruction Table [6] for the Intel Haswell processor (the one used throughout this article), which contains performance measurements at instruction level for the majority of processors in the market, a jump to an address whose value is fetched from memory has a cost that is half of that of a call instruction. This is without considering the additional cost of jumping to the trampoline function and accessing the GOT entry, which may or may not be in the processor's instruction and data cache at the time the trampoline is executed.

Let $\overline{C}_d$ and $\overline{C}_i$ be respectively the events of a data cache miss and an instruction cache miss. Then the cost of a trampolined function can be approximated as

$$\begin{aligned}\text{cost}(\text{trampoline}) \approx\ & \text{cost}(\text{jump}) + \text{cost}(\text{call}) \\ +\ & \text{cost}(\overline{C}_d) \times p_{\bar{C}_d} + \text{cost}(\overline{C}_i) \times p_{\bar{C}_i}\end{aligned}$$

where p is the probability of a cache miss event. On the other hand, the cost a direct function call is simply

$$\begin{aligned}\text{cost}(\text{direct}) \approx\ & \text{cost}(\text{call}) \\ \approx\ & \text{cost}(\text{jump}) \times 2\end{aligned}$$

and therefore the overhead can be calculated as

$$\begin{aligned}\text{overhead}(\text{trampoline}) &= \frac{\text{cost}(\text{trampoline})}{\text{cost}(\text{direct})} \\ &\approx \frac{3}{2} + \frac{\text{cost}(\overline{C}_d) \times p_{\bar{C}_d} + \text{cost}(\overline{C}_i) \times p_{\bar{C}_i}}{\text{cost}(\text{jump}) \times 2}.\end{aligned}$$

The above relation indicates that a trampoline is at least one and a half times more expensive than a regular function call. It is increasingly difficult to predict the impact of a cache miss in modern processors, as instruction pipelines become more and more complex.

Test	Median	Interquartile range
Trampoline	1.502	0.014
with instruction cache miss	6.757	0.505
with data cache miss	3.049	0.281
with both cache misses	7.258	0.111

TABLE 1: Trampoline overhead analysis

In order to verify this theoretical result, I built a simple benchmark program that compares the performance of calling an empty function, that is a function containing only the return instruction, through a direct call and through a PLT trampoline. Care was taken to ensure the function was not inlined or removed altogether. To simulate a cache miss, I used the `CLFLUSH` instruction to flush the PLT trampoline's cache line from the instruction cache and the GOT entry from the data cache. Table 1 shows the ratio between the execution time of a trampolined function over that of a direct call, obtained with 500 million runs of the benchmark program on an Intel i7-4710HQ Haswell CPU. In case of no cache miss, the results match the theoretical analysis, that predicted a 50% overhead. An instruction cache miss is extremely expensive, bringing the overhead of a trampoline to more than 670%, while data cache misses are less expensive but still significant. A combination of both cache misses results in even higher overhead. It is not possible to easily predict or measure the impact of bringing the trampoline's function and the GOT entry in the cache, as they could obtain a cache line and force another to be evicted, but it is safe to assume it plays a role in degrading the overall performance of a program.

4 Stitching - an overview

In order to reduce the overhead of dynamic linking, caused by the introduction of function trampolines, I introduce the concept of stitching, that is replacing a trampoline with a direct call once the address is known. This is similar to the optimization technique of de-virtualization in use by many compilers such as LLVM and the Java JIT compiler [7], where a call to a virtual function is replaced to a direct call to the most likely target along with some fall-back code in case the target differs from the predicted one. Stitching is fundamentally different in that it is always possible to replace a trampolined call to a dynamic library with a direct call, as the function's address is not going to change for the entire lifetime of the process. Moreover, no fall-back mechanism is needed.

There are, however, some architectural limitations that prevent converting a trampolined call to a fast, relative call that does not access memory. As documented in the Intel Instruction Set Reference [8] page 3-122, the x86-64 architecture allows for both relative and absolute calls, where the target of an absolute call must be fetched either from a register or from memory. The displacement of a relative call, instead, can be supplied as a 32-bit immediate, and can thus refer to a callee that is in a 32-bit neighbourhood of the caller. This means that the virtual address of dynamic library's function that is target of a call needs to be within 4GiB of the virtual address of the instruction following the call. This is in contrast with the convention of Linux-based operating systems of placing executables at a virtual address in memory that is much lower than those of dynamic libraries, even though this default behaviour can be changed.

I present different implementations of the stitching mechanism. The first one is entirely in the compiler and therefore does not require changes to the linker or the operating system. The second one is a variation of the first one but aims to further reduce the overhead of stitching at the cost of reduced coverage of call sites. The third one, very similar to the one presented by Agrawal et al. [2], is fully implemented in the linker. All implementations are based on the LLVM Compiler Infrastructure (version 6.0.0), which provides both a C and C++ compiler (`clang`) and a fully-fledged linker (`lld`).

5 Compiler-based stitching

I modified the compiler to add a new transformation pass just before the final assembly code is emitted. This new pass identifies every call to a function external to the module being compiled, i.e. the translation unit, and adds an additional call following the original one to a special stitching function. This function is defined in an external library, the stitcher, which contains the stitching logic.

In identifying the calls that need to be stitched, it is not possible to discern whether the external call will ultimately be to a dynamic library or is instead defined in another translation unit. While the stitcher contains a fall-back mechanism to handle every case, I also identified a solution that makes it possible to distinguish the two cases. I use Clang to generate a parsable AST of all the translation units that will be linked together, from which I can retrieve all functions that are defined and not only declared. I then exclude calls to those functions from the ones that will be stitched.

The call to the stitching function is added by a `MachineFunctionPass` and not at Intermediate Representation (IR) level since it is important that it is added immediately after the call instruction to the shared library's function. Guaranteeing so at IR level is not trivial since many optimization passes add, remove and move instructions around, and at the time the assembly code is emitted, there could be some instructions between the two calls. As I will show, in that case it would be impossible for stitching to work.

Figure 2 shows what the resulting assembly looks like after a program has been compiled and linked. The example shows two translation units, where only the `printf` function has been identified as external, while `myPrint` has been identified as internal.

The stitching function does not accept arguments and does not return anything. It is responsible for the stitching mechanism, which is achieved by the means of self-modifying code. This requires the text section to be writable, and that can be achieved by either calling `mprotect(2)` to change individual memory pages' permission or to directly modify the ELF binary to make the entire text section writable. In this implementation I have opted for the second option as it is faster. The main logic is as follows:

1) The stitching function gets called and the return address is pushed onto the stack.

```
#include <stdio.h>
void myPrint(const char* str);
int main()
{
    myPrint("Hello...");
    printf("... world!");
    return 0;
}
```

```
#include <stdio.h>
void myPrint(const char* str);
    printf(str);
}
```

```
main:
    ...
    CALL myPrint        # Call to internal function
    ...
    CALL printf@plt     # Call to external function
    CALL __stitch@plt   # Call to stitching function
    ...
myPrint:
    ...
    CALL printf@plt     # Call to external function
    CALL __stitch@plt   # Call to stitching function
    ...
```

Fig. 2: Inserting calls to the stitching function after every external call

2) The stitcher retrieves the address from the stack and subtracts from it the amount of bytes of a call instruction, identifying the call site that needs to be stitched.
3) The stitcher follows the PLT trampoline and locates the GOT entry associated with that function. The GOT entry will contain the virtual address of the shared library's function.
4) If the function's address is within 4GiB of the call site's address, the stitcher replaces the call to the trampoline with a call to the actual function.
5) Finally, the call to the stitching function is replaced with a 5-byte `NOP`.

The stitching function is written in both x86-64 assembly and C. The `RAX` register is saved and restored as it could contain the return value of the previously called function. `RDX` and the vector registers `XMM0` and `XMM1`, also used as return values, need not be saved as they are not clobbered by the stitcher. Other registers need not be saved either as, from the compiler perspective, the call to the stitching function clobbers the same registers as the previous call, per the calling convention as set out by the System V ABI [5]. In other words, the stitching function call is "sheltered" by the previous call. This is why it is important for the two calls to be contiguous. If they were not, the stitcher could not deterministically retrieve the original call site's address, and it would risk clobbering some registers that the compiler had considered to be live.

The 5-byte `NOP` instruction replaces the call to the stitching function so that it is called only once. Moreover, in case the compiler mispredicted whether a call would be external where in reality it is internal (and therefore does not go through a PLT trampoline), the stitcher will simply do nothing and replace the call to itself with a `NOP`. The choice of a multi-byte instruction instead of a sequence of five one-byte `NOP` instructions has been made by following Intel's guidelines on the subject (Table 4-12 of [9]). While it is guaranteed not to perform any operation nor memory access, it is unclear whether the 5-byte `NOP DWORD ptr [EAX + EAX * 1 + 00H]` carries a false dependency on the `EAX` register, which would be unfortunate as it is very likely for it to be referenced after the call in case the function returns an integer value.

The stitching logic resides in an external, dynamically-linked library so that it is easier to switch between different implementations if need be. For example, stitching could be disabled and enabled altogether without the need of recompiling the executables that contain calls to stitching instructions. Moreover, as suggested by the Intel Optimization Manual [10], "Software should avoid writing to a code page in the same 1-KByte subpage that is being executed or fetching code in the same 2-KByte subpage of that is being written" (§3.6.9.1).

The advantages of this compiler-based approach is that it is extremely feasible in that the compiler can choose which calls to stitch, having in-depth knowledge of the program's structure. In this implementation, all external calls are stitched, but a more complex implementation could use static analysis to stitch only those calls that are in tight loops or are guaranteed to be called a large number of times. In fact, stitching a function used only once at initialization or at finalization has no benefit and instead causes unnecessary performance degradation due to the stitching mechanism. Such optimizations could be implemented with the aid of helpers such as LLVM's `LoopPass`. As a proof of concept, I will show a real-world example of the effects of such optimization in section 9.

As a drawback, while `NOP` instructions are cheap, they are not free. Therefore, the cost of executing those no operations is added to the one-time cost of modifying the code. Moreover, each call to the same dynamic library's function needs to be patched once per call site, because the stitcher has no knowledge about the whole program. In order to patch all call sites at once, it would require a data structure similar to an inverse relocation table, mapping external symbols to all their call sites.

The size overhead can be computed as $C \times N_E$, where C is the number of bytes of a call instruction and N_E is the number of call sites to external functions which have been stitched. This figure can increase or decrease depending on the alignment of functions affected by the stitching transformation, which are usually maintained by the compiler on a 16-byte boundary.

```
#include <stdio.h>
int main()
{
    puts("Hello World!");
    return 0;
}
```

```
main:
    ...
    CALL __stitch_puts # Call to helper function
    ...
__stitch_puts:
    ...
    CALL puts@plt      # Call to external function
    CALL __stitch@plt  # Call to stitching function
    ...
```

Fig. 3: Trampoline-based stitching

6 Trampolined stitching

In order to remove the overhead of `NOP` instructions, I implemented a sightly different version of the previous approach. Each call site to a dynamic library's function `f` is replaced with a call to a special helper function `__stitch_f`. This special function contains a call to the original function `f` and a call to the stitching function as described in the previous section. If multiple translation units contain a call to the same external function, the definition of the helper function will be repeated for each of them. This would violate the One Definition Rule as set out by the C++ standard (§3.2 of [11]), and therefore the special linkage type `LinkOnceODRLinkage` has been used to prevent the linker from complaining. This is the same mechanism used by `inline` functions in C++. Figure 3 shows the resulting assembly of an example program.

The stitcher has been modified to accommodate the changes in the overall logic. The call site address needs to be retrieved further down the stack, since the stitching trampoline will push its address onto the stack and an 8 byte value to align the stack pointer to 16 bytes. This alignment requirement is imposed by the System V ABI [5] (§3.2.2). If further stack adjustments are made, the stitcher cannot reliably retrieve the correct address from the stack. Those cases can be detected, since a different assembly from the normal case will be emitted, and stitching is disabled for that call site. Common examples are functions with a variable number of arguments, which often are passed on the stack. Therefore, this trampoline-based implementation reduces the number of call sites that can be successfully stitched.

This transformation pass is implemented both at IR level and at the `MachineFunctionPass` level. During IR transformation, for each external function, an helper function is defined. Later, before the instruction selection phase, all calls to the external functions are replaced with calls to the previously-defined helpers. The stitching function `__stitch` (inside the helper function) has its register mask modified so that the return registers are flagged as callee-saved. This prevents the compiler from emitting unnecessary instructions to save the value of certain registers. Tail calls are not replaced as they are often lowered to `JMP` instructions which do not push the return address to the stack, confusing the stitcher.

This implementation makes it possible to replace the call to the helper function with a direct call to the dynamic library's function, without the need of a `NOP` padding after it. In case of tight loops, this helps further improve performance in comparison to the previous implementation. Whenever stitching is not possible, the call is replaced with the PLT trampoline call. The average size overhead of this implementation is $H_S \times unique(N_E)$ where H_S is the average helper function's size (with alignment taken into account) and $unique(N_E)$ is the number of unique external calls.

Since this solution introduces an additional trampoline, it is interesting to reason whether it would be possible to exploit the PLT trampoline in the first place. While theoretically possible, there are some obstacles that prevent a PLT-based stitching implementation from being viable. First, the call to the stitching function would need to be in every of the PLT stubs and not in the PLT header function, since that is called only once per external function and then never again. The second and subsequent calls to an external function would not go through the PLT header, and therefore not allow the stitcher to perform any action. Therefore, the call to the stitching function would need to be in the PLT trampoline itself, before the first jump to the GOT. The stitcher would need to detect whether the GOT has been already patched (it will not be for the first call), and then replace the trampoline call with a direct call. This means stitching can be performed only the second time a call to an external function is made, which in itself is not a serious limitation and instead could be desirable sometimes.

However, the main problem with a PLT-based approach is that modifying the PLT definition would break the ABI, which requires PLT functions to be exactly 16 bytes (Figure 5.2 of [5]). While it is unclear whether such requirement is normative, as it appears in a comment in a figure and not in the main text body, I modified the LLVM linker's PLT generation logic to add a spurious `NOP` instruction in the trampoline, both at the beginning and at the end. Test programs worked correctly, i.e. they produced correct output, but the output of tools such as `objdump` was corrupted, indicating that they relied on the PLT functions being 16 bytes.

7 Linker-based stitching

In this third implementation, I modified the LLVM linker (`lld`) to avoid PLT trampolines altogether. While the implementation is slightly different, the final result is very similar to the software-based approach taken by Agrawal et al. [2].

A dynamic library's load address is fixed with the `prelink --reloc-only` utility, and all its symbols' addresses are then retrieved with the `nm` tool. A dictionary

```
#include <stdio.h>
int main()
{
    printf("Hello...");
    puts("... World!);
    return 0;
}
```

```
# libc.so.cache
...
printf 0x85120000
puts   0x84fa8000
...
```

```
main:
    ...
    CALL 0x85120000 # Direct call
    ...
    CALL 0x84fa8000 # Direct call
    ...
```

Fig. 4: Linker-based stitching with a dictionary containing the addresses of functions in a shared library

having as keys the functions' names and as values their addresses is then built and stored in a file to be later consumed by the linker. When loading shared libraries, the linker loads the cached dictionary, if present, and uses it when processing relocations. Any function call that would have gone through a PLT trampoline and that is found in the dictionary is replaced with a direct call to the dynamic library's function. Otherwise, if no dictionary exists for a given library, or if the function's address is outside 4GiB of the call site's address, the linker proceeds to generate PLT trampoline as usual. Figure 4 shows an example where `libc`'s functions `printf` and `puts` are stitched after they are retrieved from the cached dictionary.

An exception are functions marked as `ifunc`. An `ifunc` function is a special routine that dynamically queries the CPU for its capabilities (for example using the `CPUID` instruction) and returns the address of a function that is best suited for the underlying architecture. For example, `log` is an `ifunc` function that returns the address of `log_avx` on architectures that support the AVX instruction set. This mechanism is often employed to boost the performance of commonly-used routines such as mathematical ones. Due to the fact that they rely on runtime information, `ifunc` functions are not stitched by the linker.

It is important to note that this is usually the most efficient solution in terms of performance gain, but it is also the least flexible. While there is no runtime cost for patching the call sites, any change in the memory layout of a dynamic library whose cache information was used by the linker will render the executable unusable unless it is linked again. Therefore, protection mechanisms commonly in use today such as *Address Space Layout Randomization* (ASLR), which changes the load address of shared libraries every time they are used, need to be disabled. *Lazy loading*, i.e. loading a library only if actually used by the program, cannot be used in conjunction with this stitching approach.

This approach is similar to pre-linking but is not identical. Pre-linking fills the GOT with the correct entries avoiding the runtime overhead of looking up the symbols, but does not remove the PLT trampoline.

8 Benchmarks and Methodology

I put the three implementations presented in the previous sections to test, analysing the performance gain (or loss) induced by each of them on a set of benchmark programs, which consists of:

1) a micro benchmark where a single external function is repeatedly called in a loop;
2) a simple open-source neural network implementation [1] used to learn integer addition;
3) the Apache HTTPD web server [4] (version 2.2.34), with the `prefork` module.

8.1 Micro benchmark

The micro benchmark is a simple C program that contains a loop whose body consists in a single call to an external function such as `clock_gettime` or `expf`. The loop is executed 100 million times before the program exits. The performance measure used for this benchmark is the total execution time, including the time spent in loading the program into memory and exiting.

8.2 Neural network workload

I used the KANN framework, which provides some example neural network implementations. I used the `rnn-bit` program, which is a recurrent neural network implementation, to learn integer addition. The training dataset is a sequence of 30 thousands addition expressions containing the two operands and the result of the addition. The performance measure used for this benchmark is the same used for the micro benchmark, the total execution time including loading and exiting.

8.3 Apache HTTPD

I tested the Apache HTTPD web server using the `siege` [2] tool, which allows for benchmarking request processing

1. https://github.com/attractivechaos/kann/ (commit 09cd297)
2. https://www.joedog.org/siege-home/

Stitching implementation	Benchmark	Median	IQR	Statistically significant	Performance gain
No stitching	Micro benchmark (`expf`)	1.0252s	0.0092	*N/A*	*N/A*
	Micro benchmark (`atoi`)	1.8997s	0.0196	*N/A*	*N/A*
	Neural network (2 epochs)	16.0768s	0.0848	*N/A*	*N/A*
	Apache HTTPD (15s)	4869 trans/s	140	*N/A*	*N/A*
Compiler-based	Micro benchmark (`expf`)	0.9660s	0.0071	Yes	6.13%
	Micro benchmark (`atoi`)	1.8390s	0.0224	Yes	3.30%
	Neural network (2 epochs)	16.0853s	0.0599	No	-0.05%
	Neural network (2 epochs) only `expf`	15.9894s	0.1107	Yes	0.55%
	Apache HTTPD (15s)	4489 trans/s	84	Yes	-8.46%
Trampolined	Micro benchmark (`expf`)	0.9681s	0.0064	Yes	5.90%
	Micro benchmark (`atoi`)	1.8426s	0.0130	Yes	3.10%
	Neural network (2 epochs)	15.9946s	0.0882	Yes	0.51%
	Neural network (2 epochs) only `expf`	15.9767s	0.0626	Yes	0.63%
	Apache HTTPD (15s)	4443 trans/s	68	Yes	-9.61%
Linker-based	Micro benchmark (`expf`)	0.9672s	0.0083	Yes	6.00%
	Micro benchmark (`atoi`)	1.8373s	0.0133	Yes	3.40%
	Neural network (2 epochs)	15.9521s	0.0191	Yes	0.78%
	Apache HTTPD (15s)	4979 trans/s	85	Yes	2.20%

TABLE 2: Benchmarking results

time of a web server for a configurable number of seconds. The performance measure used for this benchmark is the number of transactions per second as reported by the tool. The page being served is a simple *Hello World* HTML page.

8.4 Test harness

The test harness is a Python script that is responsible for calling the program under test and measuring the total execution time in case of benchmark programs 1) and 2), or collecting the number of transaction per seconds from the standard output in case of 3). The total execution time is obtained by querying a performance counter via `time.perf_counter`, with a resolution of 1 nanosecond.

8.5 Methodology and statistical analysis

I consider the median over 22 runs of the same program under test, and an additional warm-up run is discarded and is not included in the results. 2 runs, the best and the worst performing ones, are discarded as to minimize the effect of outliers, leaving a sample of 20 runs. Along with the median, the interquartile range (IQR) is reported as a measure of variability. Since the data being collected cannot be trivially assumed to be normally distributed [12], I use the un-paired, non-parametric Wilcoxon Rank Sum Test [13] to determine whether two datasets are statistically different at 95% confidence.

8.6 Hardware and software environment

The tests have been run in a virtualized Linux enviroment (Ubuntu 14.04 LTS, kernel version 3.19.0-47-generic), supplied with 4GiB of main memory (DDR3 1600MHz) and reserved 4 cores of an 8-core *Intel Core i7-4710HQ* CPU (2.50GHz up to 3.50GHz).

The LLVM infrastructure has been built from source [3], along with the Clang compiler [4] and the lld linker [5]. All tools report version 6.0.0. All of the implementations, benchmark programs and results described in this article are open-source under a MIT license and available in the form of a Git repository at https://github.com/banex19/stitching.

9 RESULTS

Table 2 shows the results obtained by testing the three stitching implementations against the set of benchmarks presented in the previous section. The *Statistically significant* and *Performance gain* columns compare a stitched version of the benchmark with the relative non-stitched one.

9.1 Micro benchmark

I tested the micro benchmark program with two functions, `expf` and `atoi`. The first calculates the Euler's number raised to a given power, while the second parses a string interpreting its content as an integer.

All stitching implementations perform very well with respect to the `expf` benchmark, with the compiler-based one outperforming the non-stitched version of the program by more than 6%. The trampoline-based implementation is very close with a 5.9% performance gain, realistically hindered by the additional work the stitcher has to perform. The linker-based implementation performs equally well as expected.

The `atoi` benchmark shows that stitching has less of an impact (3.4% performance gain in the best case), probably due to the time spent performing work in the function itself. The `atoi` function is more expensive than `expf` in terms of CPU cycles, therefore shadowing the cost of the function call.

3. LLVM commit `6281d8cca4f8b1e9c1b3d79bfff1ce67e5a663fb`
4. Clang commit `5a52b9e6b0480ced82bf0bccbddd0f1d7804752d`
5. lld commit `771a6f30ccd525a2367c7c3f566b75ada10afe9b`

Implementation	Requires writable page	Requires 4GiB neighbourhood	Requires no ASLR	Can be disabled	Can be partial
Hardware	No	No	No	No	No
Compiler-based	Yes	Yes	No	Yes	Yes
Trampolined	Yes	Yes	No	Yes	Yes
Linker-based	No	Yes	Yes	No	No

TABLE 3: Comparison of different stitching implementations

9.2 Neural network workload

The KANN framework doesn't rely heavily on external libraries, but stitching proves to be effective nevertheless. The linker-based implementation achieves a 0.78% speedup compared to the non-stitched version, and the trampolined one is close with a 0.51% improvement. The compiler-based implementation's performance is actually worse than that of the original program, even though the slowdown is not statistically significant according to Wilcoxon's test. This is due to the added overhead of the `NOP` instructions, which the trampolined version is able to avoid.

To show the flexibility of implementations at the compiler level, I profiled the KANN program used for the benchmarks in order to find which functions were taking the most time. I used the Intel VTune Amplifier performance profiler [6] to conduct this analysis. As mentioned, KANN doesn't rely too much on external libraries, but almost 20% of the time is spent in the `expf` function, defined in the `libm` library. I then modified the compiler-based and the trampolined implementations to only stitch calls to the `expf` function. The results show that this optimization helped increase the overall performance gain of stitched programs, with the trampolined `expf`-only version coming closer to the linker-based implementation. The compiler-based implementation goes from slowing down the program to speeding it up by 0.55%.

9.3 Apache HTTPD

In the case of the Apache web server, runtime stitching introduces a significant performance slowdown. This is because Apache forks multiple processes in order to process HTTP requests, and does so with the help of the `fork` system call. Under the hood, the `fork` implementation creates a new process that initially shares the same memory pages as the parent process, that get copied only if the child process writes to them. This mechanism, known as *Copy on Write* (COW), makes it so that stitching needs to be performed at least once per child, resulting in performance degradation and inefficient memory usage.

The COW mechanism is implemented by setting the `MAYWRITE` flag on the memory pages of the processes involved in the fork operation. Whenever a process writes to any of those memory pages, the operating system's page fault handler checks whether the COW flag is set. If it is, the pages are copied. This could be avoided by modifying the kernel so that pages used for the `text` section are never flagged as eligible for COW, since there is little benefit in them not being shared just like they are in case of intra-process threads. However, I did not explore this possibility further, as it is outside the scope of this project.

6. https://software.intel.com/en-us/intel-vtune-amplifier-xe

The linker-based implementation, which does not interfere with the COW mechanism, shows a 2.20% improvement in terms of the number of transactions per second. This is similar to the results presented by Agrawal et al. [2] and confirms that removing the overhead of trampolines can significantly improve performance of real-world applications such as web servers.

10 Comparison of implementations

Table 3 shows a comparison of the three implementations presented in this article and the hardware implementation proposed by Agrawal et al., which currently is not available in any modern processor. While there are drawbacks to a software implementation, namely the requirements to have the function's address in a 4GiB neighbourhood and to have a writable page for the `text` section, there are also advantages. The ability to switch off stitching if not needed, or to only stitch an expensive loop, can be useful if employed as a sort of profile-guided optimization.

A hardware implementation comes with a fixed cost that must be offset by the resulting performance gain. However, there are operating systems that do not use a trampoline-based mechanism for dynamic linking, such as Microsoft Windows. In Windows, the runtime linker patches the address of the dynamic library's function in the *Import Address Table* (IAT, equivalent to the GOT), and subsequent calls do not need to go through a trampoline as the main code directly calls the address found in the IAT. In other words, there is no persistent PLT. Nonetheless, a hardware implementation could help in other situations as well, such as virtual function calls in object oriented programs, but care must be taken to ensure it would be as general as possible.

11 Conclusion and further work

I presented three different implementations of stitching, a software-based solution to reduce the overhead of trampolines in dynamic linking. Benchmarks ranging from micro benchmarks to a fully-fledged web server show that runtime and link-time removal of trampolines results in increased overall performance, even when a program doesn't rely heavily on external libraries.

Further work could be directed towards the implementation of a static analysis pass to better decide which calls to stitch, or an extension to the profile-guided optimization module in LLVM to include analysis of call sites to external functions.

References

[1] J. Donald, "Improved portability of shared libraries," Jan. 2003.

[2] V. Agrawal, A. Dabral, T. Palit, Y. Shen, and M. Ferdman, "Architectural support for dynamic linking," *SIGPLAN Not.*, vol. 50, no. 4, pp. 691–702, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2775054.2694392

[3] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[4] The Apache Software Foundation, "Apache HTTP Server," https://httpd.apache.org/, 2017.

[5] M. Matz, J. Hubika, A. Jaeger, and M. Mitchell, *System V Application Binary Interface - AMD64 Architecture Processor Supplement*, Nov. 2014.

[6] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," http://www.agner.org/optimize/instruction_tables.pdf, May 2017.

[7] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, "A study of devirtualization techniques for a java just-in-time compiler," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '00. New York, NY, USA: ACM, 2000, pp. 294–310. [Online]. Available: http://doi.acm.org/10.1145/353171.353191

[8] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developers Manual*, Sep. 2016, vol. 2A.

[9] ——, *Intel® 64 and IA-32 Architectures Software Developers Manual*, Sep. 2016, vol. 2B.

[10] ——, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Jun. 2016.

[11] ISO/IEC, *ISO International Standard ISO/IEC 14882:2014(E) Programming Language C++*, 2014.

[12] L. David and I. Puaut, "Static determination of probabilistic execution times," in *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, June 2004, pp. 223–230.

[13] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, 03 1947. [Online]. Available: https://doi.org/10.1214/aoms/1177730491