Università degli Studi di Pisa

Facoltá di Ingegneria

Corso di Laurea Triennale in
Ingegneria Informatica

An efficient ActionScript 3.0 Just-In-Time compiler implementation

Candidato:

Alessandro Pignotti

Relatori:

Prof. Graziano Frosini

Dott. Giuseppe Lettieri

Anno Accademico 2008/09

Abstract

Adobe Flash: the current *de facto* standard for rich content web applications is powered by an *ECMAScript* derived language called *ActionScript*. The bytecode for the language is designed to run on a stack based virtual machine. We introduce a Just in Time compiler and runtime environment for such bytecode. The *LLVM* framework is used to generate optimized native assembly from an intermediate representation, generated from the bytecode while optimizing stack traffic, local variable accesses and exploiting implicit type information.

Sommario

Adobe Flash rappresenta l'attuale standard *de facto* per le applicazioni web interattive e multimediali. Tale tecnologia include *ActionScript*: un linguaggio di programmazione derivato da *ECMAScript*. Esso viene compilato in una forma binaria intermedia, pensata per essere eseguita su una macchina virtuale basata su stack. Introduciamo un compilatore *Just In Time* e un ambiente di esecuzione per tale forma intermedia. Il sistema *LLVM* è utilizzato come *back end* per generare codice nativo ottimizzato da una rappresentazione intermedia, costruita sfruttando le informazioni implicite sui tipi e ottimizzando gli accessi allo stack e alle variabili locali.

Contents

1 Introduction

This work makes extensive use of some non trivial concepts regarding the stack machine computational model and compilers internals, such as SSA intermediate representation. Moreover, even if the optimization techniques described are mostly generic, emphasis is put on the current target of the implementation: the ActionScript 3.0 language and bytecode. The next few sections introduce some concepts that will be useful for the reader to easily understand this work. First of all the stack machine and single static assignment (SSA) models will be described, then ActionScript and the tightly tied Flash framework will be briefly introduced.

1.1 Introduction to stack machines

A stack machine is a computational model where operands to instructions are always placed in a LIFO data structure, or a *stack*. The biggest advantage of stack machines is the high density of the code, as operands selection is not encoded in the instructions stream, but they are implicitly gathered from the top of the stack. The taxonomy of these kind of machines is defined by the number of stacks and the purposes of each of them. As an example we can cite the popular *Forth* programming language that was designed to have two stacks, one for subroutines control flow and local variables and another for temporary values [3].

The stack machine model is quite natural and it was first defined in the '60. It is indeed very handy to describe expression evaluation without any need for operator precedences rules. Consider the following simple arithmetic computation $5 - ((1 + 2) \cdot 4) + 3$. It can be easily rewritten in *Reverse Polish Notation* [8], starting from the deepest parenthesis, as $1, 2, +, 4, *, 5, -, 3, +$. This expression can now be evaluated from left to right with the following rules:

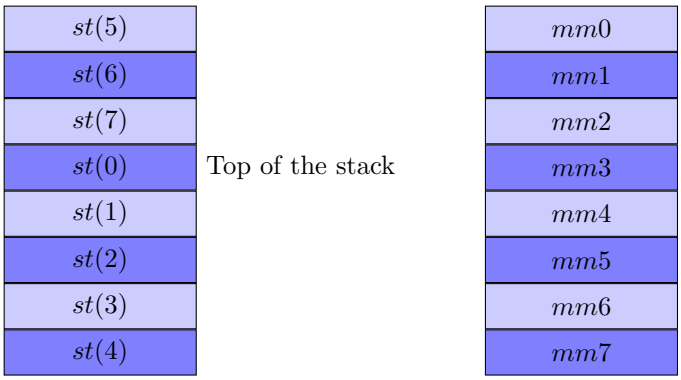
- Numeric values are pushed on the stack
- Operators pops the arguments from the stack and pushes the result back. In this case all operators are binary, but the same approach is extensible to n-ary functions.

Even if the features of such an architecture were deeply investigated during the '80 [26], and several physical platforms were implemented on this model, current mainstream CPUs are register based, even if a certain degree of stack support is common.

Execution step	1	2	3	4	5	6	7	8	9
Stack contents	1	1,2	3	3,4	12	12,5	17	17,3	14

Table 1: Stack contents during evaluation of the expression $1, 2, +, 4, *, 5, -, 3, +$ (written in *Reverse Polish Notation*).

For example the popular x86 architecture reserves one of its few general purposes register and several instructions for stack management. This kind of hybrid model is extremely widespread and it's even expected by the *GCC* compiler [29]. A notable exception included in the usual x86 ISA paradigm is the x87 instruction set, which was the first implementation of *IEEE-754* [6] floating point arithmetics for Intel's processor, now largely superseded by *SSE* and following extensions. The x87 instruction set was originally implemented in a separate co-processor, and was designed around a *register stack*. Even if the x87 co-processor has been integrated in the main CPU since the *Intel 486DX* processor, it still as to be managed has an asynchronous independent unit, which can be considered a limited, special purpose stack machine. This model is still used only for historical reasons, it should be noted that the same registers used for the x87 stacks are also aliased as an array to be used by the *MMX* and other vectorial instruction set extensions.



Registers from the point of view of the x87 instruction set, register indexing starts from the current top and wraps around if needed

Registers from the point of view of the MMX instruction set, register indexing is fixed

Figure 1: The x87 register stack,

Stack machine code	x86 code
`push 1`	`mov $1,%eax`
`push 2`	`add $2,%eax`
`add`	

Figure 2: Typical code example for a generic stack machine and x86

Even if the stack machine model is not really being implemented in current hardware devices, several modern programming language, for example *Java* and the *.NET* family of languages, have been recently designed to run in a stack based virtual machine. This layer of indirection is convenient to ensure portability and to restrict not trusted application to a limited and secure environment. The trade off for such advantages is the need for an *interpreter*, a trusted component that translates instructions from the Virtual Machine instruction set to the native one. The interpreter checks the sanity of the code and virtualize the access to the resources of the underlying physical machine. The interpreter should be able to translate the instructions so that the execution can be efficient on the host. The current mainstream architectures are quite different from the stack model and are usually based on a limited set of high speed memory register and several layers of associative caches to speed up memory access. Usually instructions can efficiently reference only data stored in registers and, on typical RICS architectures, memory access are delegated to special instructions. So a lot clever transformation has to be done to adapt the model to the host architecture.

An unoptimized and naive approach to support the stack machine model would generate a huge amount of *stack traffic*, and with the currently limited memory bandwidth this would be a serious bottleneck.

The stack machine model needs three insertions and two deletions on the stack. This means five memory accesses for a simple add. The same operation can be done using immediate operands on a typical x86 platform. It should be noted that modern processor architectures [20] have special stack handling units that can optimize stack accesses, but this is not widespread nor guaranteed.

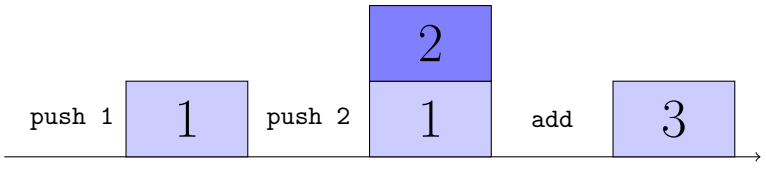


Figure 3: Stack status during execution of the example

C code	Pseudocode in SSA form
`int var=2;`	$var \leftarrow 2;$
`var+=3;`	$var1 \leftarrow var + 3;$
`var*=2;`	$var2 \leftarrow var1 * 2;$

Figure 4: Conversion between imperative and SSA form language. Variables assigned multiple times are renamed

1.2 The SSA (Single Static Assignment) Model and LLVM

The Single Static Assignment model is based on the assumption that any variable is defined only once, though it's actual value may change if the code is executed more than once, as in loops. When using this representation each variable can be considered a reference to the instruction returning it and not an actual value. This model is used as an internal representation by most commercial and open-source compilers [9]. It is quite easy to translate linear code from an imperative language (such as C) to SSA form, as shown in Figure 4.

For each variable in the imperative representation that is assigned more than once we can generate SSA form by renaming the variable to indexed versions. This is of course only possible on linear code.

To handle control flow we introduce the ϕ function. The value returned from ϕ depends on the code path executed previously. In the example of Figure 5, ϕ would yield var if coming from the if block or $var1$ otherwise. The ϕ function can be considered just a fictional notation as its purpose is only to link variable definitions to usage during code optimization.

It is simpler to run optimization passes over the *SSA* representation of the code, as it is easy to build a directed graph representing the instruction dependency. Several optimization tasks such us instruction reordering, duplicate and dead code elimination and constant propagation becomes trivial using the obtained dependency graph.

C code	Pseudocode in SSA form

`int var;`
`if(condition)`
`  var=2;`
`else`
`  var=3;`
`res=var;`
	$if(condition)then$
	$\quad var \leftarrow 2;$
	$else$
	$\quad var1 \leftarrow 3;$
	$endif$
	$res \leftarrow \phi(var, var1);$

Figure 5: Conversion between imperative and SSA form language. The ϕ function choose the right result when control flow merges

LLVM [35] is a compiler framework designed to enable effective optimization at compile time, link time (whole program optimizations) and at run-time. *LLVM* accepts an Intermediate Representation (the *IR*) that can be loaded from a textual assembly-like language or built using the *LLVM API* at run-time. The *IR* is strictly typed and portable. Several targets are supported, such as the x86 family (both 32 and 64 bits wide processors), PowerPC, MIPS and ARM. For most of those platforms a Just in Time compiler is available. The *IR* is based on the SSA model, although for simplicity it is also allowed to allocate stack memory using the *alloca* instruction. Memory allocated in this way can be promoted to registers during optimization using an appropriate pass (described later). Each assigned variable can be considered a *virtual register*. During compilation LLVM will take care of virtual to physical register mapping, and of *spilling*: saving register data to memory when no more free registers are available.

LLVM is recently gaining importance as *Clang*, the C and C++ compilers being developed on such framework, matures. In time we may expect *Clang* to reach and even exceed GNU Compiler Collection [31] capabilities. At the time of this writing the major advantages of *Clang* over GCC are:

- Clang code base is simple and designed as an API, allowing easy interaction and integration with other projects
- Clang is much faster and uses much less memory than GCC [33]
- Clang has very good support for expressive diagnostic of compilation errors and highlights the exact code location which causes problems.

It should be noted, however, that compiler support for C++ is still far from complete, while the GCC front-end is mature and well tested.

The LLVM framework provides to Clang and other front-ends a huge set of optimization passes that can be executed over the intermediate representation.

Pseudocode in SSA form

$var1 \leftarrow 7$
$var2 \leftarrow 2$
$var3 \leftarrow var1 + var2$
$var4 \leftarrow 3$
$var5 \leftarrow var3/var4$

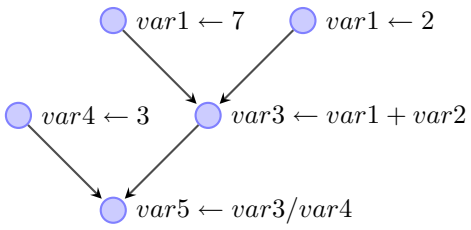


Figure 6: Generating the dependency graph from SSA code

This means that the back-end does most of the common optimization work. The front-end only has to worry about language specific optimization possibilities.

Some examples of the optimization passes provided by LLVM are:

Dead code elimination Aggressively purges code that is not reachable

Condition propagation Propagates information about conditional expressions to eliminate conditional branches in some cases

Merge duplicate constant Merges duplicated constant into a single shared one

Constant propagation For each assignment as $var = C$ with C constant the value is propagated in each use of the variable.

Redundant instructions elimination Eliminates assignments and memory loads that are redundant, as they do not change the destination variable value

Loop invariant code motion Tries to remove as much code as possible from the body of a loop and promotes memory locations used in the loop to registers, when possible

Loop unrolling Convert finite, short loops into linear code

Promote memory to registers Tries to keep frequently used stack memory locations in registers.

LLVM does not enforce a user to enable all those passes. A selection of the best sequence of passes should be made on a case by case basis, although a few of them are generally useful.

1.3 Introducing Adobe Flash

Adobe Flash (originally developed by *Macromedia*) is a platform for the development of rich, interactive and multimedia enabled web applications. The content is delivered to the client using a custom file format called *SWF* (pronounced *Swiff* and has to be interpreted using a custom browser plugin previously installed on the client. Adobe claims that his proprietary plugin is installed on around 90% of the machines in mature markets [12]. The Flash technology is considered a corner stone of the modern *World Wide Web*, enabling multimedia services as the popular *YouTube* video sharing platform [13].

The Flash technology success can be acknowledged to a number of factors.

Easy interface design Since the first versions of Flash, Macromedia (now Adobe) produced an easy to use design suite to develop interactive animations and applications.

Advanced graphics and interactivity capabilities Flash offered an impressive support for advanced graphics element, such as *alpha blending* (transparency), *Bzier splines*, *texturing* support with both images and several kind of color gradients. It should be noted that a somehow standard replacement for such capabilities can only be found in the not yet released *HTML5* standard [36]. Moreover, all those graphics elements could easily interact with user input from the keyboard and mouse.

Rendering consistency An highly attractive feature of the Flash player is the guaranteed consistency of the results for every client. This was quite an impressive feature in the past, when HTML was still plagued by the non standard features added during the so called "*Browser War*" [1]

Flexible scripting language Flash features a flexible scripting language: ActionScript (more detail on this in the following section). Being a dynamically typed language, similarly to JavaScript it allows for rapid application prototyping and it's easy and intuitive enough to be used by non professional programmers.

There are open-source efforts to develop an alternative to the proprietary Flash plugin. The most important of them is *Gnash* [32] which is heavily sponsored and considered a priority by the Free Software Foundation. The author of this work is currently working on a different project called *Lightspark* [27], which is designed to be very efficient by exploiting the features of modern platforms. This work is derived from the efforts on such project.

1.4 Introducing ActionScript 3.0

Most of the success of the Flash technology can be acknowledged to ActionScript, a simple though powerful scripting language embedded in *SWF* files, which allows for a great degree of control over the graphic elements and interactivity features of the movie. ActionScript was first introduced in version 2 of Macromedia Flash Player. At first only basic playback control such as *play*, *stop*, *getURL* and *gotoAndPlay* was offered, but over time new features were introduced as conditional branches, looping construct and class inheritance, and the language evolved to a general purpose though primitive one.

To overcame several limitations in the original design Adobe introduced ActionScript 3 in 2006 with the release of version 9 of the Flash player and development tools. The new design it's quite different from the previous versions and extremely expressive. The language is mostly based on ECMAScript 3 (which is also the base for the popular JavaScript language), but several new features are added to the standard, such as optional variable typing. The ActionScript specification defines a binary format and bytecode to encode the scripts so that no language parsing has to be done on the clients. Moreover a huge runtime library is provided, integrated with the graphics and multimedia features of the flash player. This work targets mostly the core part of the language and the implementation of the runtime component is considered for further development of the *Lightspark* project.

The ECMAScript and ActionScript model allows even primitive types as the *integer* and the *number* to be fully fledged generic objects. Moreover simple instructions, as the *add* are polymorphic and so the actual execution depends on the arguments types. This layer of indirection implies that even very simple operations has to be handled in an interpreted way, as we have no information on the nature of the data until runtime.

The ActionScript bytecode offers several kind of opcodes:

Arithmetic and logic operations such as *add*, *bitand*, *bitnot*, *increment* and *equals*. Those instructions are specified to convert input arguments to the *number* or *int* type, and to return a similar numerical type. This do not hold for *add* that is polymorphic and can add as well numbers and concatenate strings (The algorithm follows the specifications of ECMA-262 section 11.6 [21] and ECMA-357 section 11.4 [22])

Runtime type information and type conversion such as *astype*, *instanceof*,*coerce*, *convert_b* and *convert_i*. ActionScript supports to a great extend the handling and manipulation of object types at runtime, and a lot of polymorphic interfaces are offered in the runtime library.

Control flow such as *callproperty*, *returnvalue*, *ifeq*, *ifle*, *iftrue*, *jump* and *label*. The control flow instructions only allows for relative jumps and no absolute destination is encodable. Moreover the *label* instruction is actually a No-Op and it is used to point out the destination of backward jumps, this feature enables a single pass Just-In-Time compilation. As there are no indirect jumps, it's not possible to implement class polymorphism and interfaces using the classic virtual table approach and we have to use *callproperty* and similar instructions, as these invoke a function by name from objects properties which are overridable

Object manipulation such as *construct*, *newarray* and *newobject*. The root of the ActionScript class hierarchy is the *Object* class. Objects are no more than associative containers from names to other Object or Object subclasses instances. To build a new object you can use the *construct* instruction. Also the *newarray* instruction is provided as a shorthand for the *Array* constructor

Stack manipulation such as *dup*, *pushbyte*, *pushint*, *pushundefined* and *pop*. ActionScript is based on a stack machine, so several instructions are provided for stack manipulation. Beside the classic *push* for various types and *pop*, we can also see the *dup* and *swap* instruction. The former duplicate the last entry on the stack, the latter invert the positions of the last two entries.

Object properties manipulation such as *findproperty*, *getlex* and *getslot*. Object properties can be accessed using these functions both by name and by index for better performance.

Resolution scope manipulation such as *pushscope* and *getscopeobject*. Several of the instruction that finds properties are designed to search first in objects on the scope stack, before searching the Global object. These instruction manipulate the scope stack.

Local variables manipulation such as *setlocal* and *getlocal*. Beside the stack ActionScript also provide a fast local storage, this instructions are used to access values stored there. From an high level point of view locals are a simple array of objects, and the values are accessed using indexes and not names.

2 Related work

ActionScript *per se* has not been subject of many studies. But as it is a very near relative of the JavaScript language it is interesting to show the approaches used by a current generation JavaScript engine: Tracemonkey, included in Mozilla derived web browsers, such as Firefox. Moreover the architecture of the *Tamarin* and *Gnash* engine will be described, being respectively the *Adobe* and *GNU* sponsored implementation of the language.

2.1 Tracemonkey

Tracemonkey is the JavaScript engine currently used in Firefox 3.5 and other Mozilla derived browser. The engine is an evolution of the preceding *Spidermonkey* enhanced using the *Trace Tree* technique, described in [18]. This approach is based on the assumption that most of the execution time is usually spent on loops, and so it's sensible to invest computation power and memory to compile only those critical, computational expensive code to native using the Just In Time compiler.

Sample C code

```
var i;
for(i=0;i < 100; i++)
{
    if(i < 10) {
        print("Low value");
    }
    else if(i < 60) {
        print("Mid value");
    }
    else {
        print("High value");
    }
}
```

Resulting hot trace

```
var i;
for(i=0;i < 100; i++)
{
  assert(!(i < 10));
  assert(  i < 60 );
  print("Mid value");
}
```

Figure 7: A JavaScript code snippet and it's representation as an hot trace. Each time the loop backward jump is executed a counter is incremented, when a threshold is reached the trace is recorded. In the example the threshold is 15 and the second case of the loop is recorded in the trace. The trace itself contains the two assertions that makes the trace valid. If an assertion fails a *side exit* happens and the execution is resumed using the interpreter. As side exits are expensive a counter is increment also for each of them, when one is executed enough times the corresponding code path is added to the trace.

Loops are discovered using a simple heuristic: a loop candidate starts at the target of a backward jump and ends when the same location is reached again. Code start being executed using an interpreter, during execution a counter for each backward jump target is maintained and incremented every time a branch to the target is taken. When a threshold is reached the branch target is considered an *anchor* for a *hot loop*. Following instructions are recorded, until the anchor is reached again, or a maximum length is reached. When a conditional branch is encountered only the currently taken branch is considered and recorded. For the other case a so called *side exit* is generated. When a side exit is executed the dynamic context used in the native code is synchronized to the one expected by the virtual machine, and interpretation is resumed. Of course if the hit count of a side exit becomes big enough, also that code path will be translated to native code. This event is called *trace extension*. When a trace is extended the preceding instructions are shared with the container trace, while following instruction are duplicated. This approach minimize the complexity of the control flow graph, so that compilation can be extremely fast. The speed of the compiler makes it possible to recompile the whole trace each time an extension happens, to take advantage of new optimizations opportunities that may be discovered by looking at the newly added code in the context of the surrounding trace.

2.2 Tamarin

In 2006 Adobe released his own ActionScript3 engine as an open-source project for inclusion in the *Mozilla* project. The name of this new project is *Tamarin*. The architecture of this project evolved over time, the main features of the different approaches are briefly described below.

2.2.1 Tamarin central

Tamarin central is the stable version of the engine. It is based on a very simple architecture and code is compiled on a *method-at-once* basis. The engine features both an interpreter and a JIT compiler engine (*NanoJIT*) [7], which accepts a custom intermediate representation (*LIR*) and targets several platform (x86, x86-64, PPC, ARM). A simple heuristic is used to choose between the two modes of execution: basically the initialization methods are run trough the interpreter, others methods are converted to native code.

When a method has to be compiled, his ActionScript bytecode is parsed and statically verified. From the bytecode the *LIR* representation is derived,

one opcode at a time. The resulting *LIR buffer* can then be passed to *NanoJIT*. Some optimization steps can be done on the buffer by using *filters*, although the engine is designed for simplicity and speed and not for heavyweight optimization. Compiled code can then be directly executed on the physical processor. To handle unexpected condition in the native code, *guards* are inserted. Guards are runtime checks which assert that the data used in the method is compatible with the assumptions made by the JIT compiler, for example numerical type can be checked against an allowed range of values.

2.2.2 Tamarin tracing

Tamarin tracing is the new development branch of the project. It implements the tracing approach described in the previous section 2.1. The most interesting aspect of the project is that *Forth* is used as a back-end for ActionScript bytecode compilation. So actually this engine does a double round of translation. ActionScript bytecode is first translated into a *Forth* intermediate representation, which is then translated to the *NanoJIT* intermediate representation, and in the end compiled. *Forth* is itself a stack based language, so the first translation is mostly natural, probably this design originated to leverage the experience with Forth compilers for the advantage of the new language. For a long time after the open-source release of Tamarin there were attempts to merge it as the JavaScript engine for Mozilla browsers, the project was called *ActionMonkey*. In the mean time the *SpiderMonkey* engine was being enhanced by *TraceMonkey*, and in 2008 the ActionMonkey project was abandoned, as Tamarin has not yet reached the stability and performance results of the current mature engine.

2.3 Gnash

Given the extremely high market penetration of the Flash technology, for a long time the creation of an open source Flash player was considered a priority by the Free Software Foundation. In the beginning the GPLFlash [4] project was primarily supported. Nowadays most of the developer of that project moved to *Gnash*, released under the GPL, which itself derives from the code base of *GameSWF*.

Gnash offers complete support for Flash version 7, and for some features of version 8 and 9. Flash video playback is supported through the *FFmpeg* or *GStreamer* libraries [2] [5].

This project includes a good support for the ActionScript 1.0 and 2.0 lan-

guages. For a long time an ActionScript 3.0 implementation has been under work, but it's still in early testing. Moreover, only the interpreter is supported as there is no JIT compiler ready. The *Open Media Now* foundation has founded a summer of code project to boost the work on the Virtual Machine, but right now any results is yet to be seen.

3 An efficient ActionScript JIT compiler

3.1 Motivations

During the last years of the '90 we've seen the conversion and explosion of the Web as a platform for multimedia contents distribution. The launch of video sharing platforms such as Youtube and Vimeo [14] preceded by a long time the availability of a standard for video content on the web, which actually was only proposed with the working draft of *HTML5* in 2008 [36]. Since 1998 every vendor has tried to push its own solution for streaming media, the main competitors were *RealPlayer*, Microsoft's *Windows Media* and Apple's *QuickTime*. Each one needed a custom plugin installed in the browser. In the meantime the first versions of Macromedia Flash were gaining popularity for interactive contents and animations. When in 2002 video playing capabilities were added to Flash it basically imposed itself as a new *de facto* standard. In 2006 Macromedia was acquired by Adobe and a major rework was done on the Flash scripting language, giving birth to ActionScript 3. This new versatile language, together with the multimedia and interactive features already provided by Flash are now a fundamental component of current web technologies. The dominant position of Flash has been recently questioned by Microsoft's *Silverlight* based on *.NET*, but nowadays is not yet clear if there will be a shift in the market share of those technologies. [15]

It has to be noted that the current position of Flash as a cornerstone of the modern web is totally out of any standardization. This poses the risk of a jeopardization of the usability of the web, in a similar way to what happened during the "browser war" when every vendor tried to push its own implementation by adding non overlapping sets of non standard features [23]. However Adobe seems to be interested in converting *de facto* to actual standards [11].

For example Adobe *PDF* (Portable Document Format) has been an open specification for a long time and in 2008 it became an ISO standard (ISO 32000). Even if Flash has not yet been proposed for a standardization procedure, it's complete specification has been available since 2009. As Flash is a key component of the Web the *Free Software Foundation* considers a priority to have an Open Source implementation of the Flash Player [17]. The *Gnash* project had reverse engineered most of the SWF (Flash binary format) prior to the official release of the specification, but it yet misses an efficient implementation of the new generation scripting language, ActionScript3 which powers most of the new applications.

Lightspark is a modern Open Source implementation of the Flash Player

[28], written from scratch and designed to be efficient exploiting the current multi-core architectures and programmable graphic cards. A primary feature of the project is the very good support for the new features introduced in ActionScript3, as class and interface inheritance. In this chapter we are going to discuss the implementation challenges and optimizations developed to achieve this results.

3.2 Definitions

First we define some keyword that will be used frequently

Just in Time compiler A system to transform high level source code or bytecode to native machine code at runtime, just before the actual invocation of the code. Just-In-Time (*JIT*) compilation is a mainstream technique currently implemented by all major Virtual Machines such as Microsoft's *.NET* and *Java* [30] [25]. Compiling code at runtime causes a performance penalty over static (off line) compilation, but brings many advantages such as the ability to optimize the code for the specific processor of the host and a chance to enforce security policies and type safety. Hot code paths can also be cached so that after an initial slow setup time most of the code will be already translated.

Basic Block A linear sequence of instructions ending on a branch (conditional or unconditional)

Trace A linear sequence of instruction ending with an unconditional branch. The trace concept is useful as most often conditional branches are short and may fall inside the trace itself, increasing optimizations possibilities

Method A collection of *Basic Blocks* linked by branches. Each method has a private stack and local data storage.

3.3 A naive approach: the interpreter

The simplest approach for executing a bytecode would be an *interpreter*: a system that decodes one instruction at a time and execute it by using routines written in a high level language. In ActionScript each method declares statically the maximum amount of stack and local data storage needed. So the interpreter should allocate for each method an *array* of pointers to *Objects* for the stack and the locals. The routines implementing the opcodes semantics would pop arguments and push return values from the stack array. This approach is very

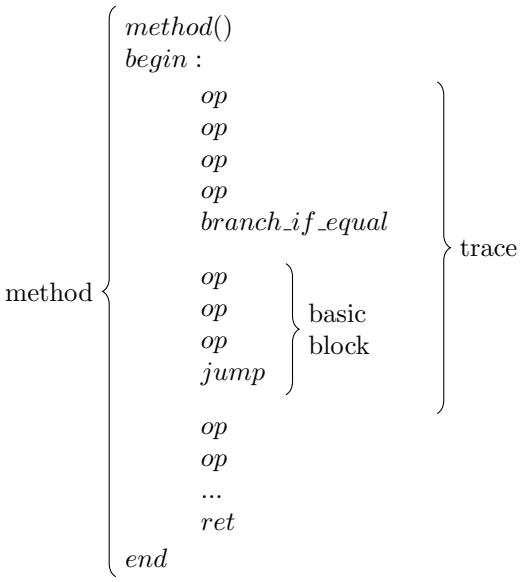


Figure 8: A *method* is composed by *basic blocks* linked by branches. A *trace* is a sequence o operations that ends on a unconditional branch

simple to implement and indeed very functional for code executed only once, as there is no need for code analysis and compilation. The major disadvantages are the huge amount of memory traffic involved even for simple operations, and the very frequent creation and destruction of object used for temporary objects, which causes a large overhead

3.4 Our approach

We introduce a Just-In-Time compiler for the ActionScript3 bytecode. *JIT* compilers can be designed to translate code at various granularities. A common choice is to translate *traces* or *methods* at once. We choose the latter approach. The ActionScript bytecode is loaded from the SWF file, if embedded in a Flash movie, or from standalone *ABC* files [10]. The bytecode for each method is parsed and split into *basic blocks*. In general this analysis should be done in two passes:

1. Detection of jumps targets, as those are the starting point for basic blocks. For conditional branch both the actual target and the fall-through instruction are considered as targets.

2. Building of basic blocks, which is now trivial as they start from a jump target and ends on a branch.

For ActionScript this analysis can be done in a single pass, as the targets of backward jumps are marked by a special opcode. During this phase we also find and prune out *dead code*: instructions that are not in any block. Moreover we extensively validate relative branches to guarantee that the targets are actually inside the function body. For a long time the official Adobe player missed this check and it was possible to jump to an arbitrary byte offset inside the file. It is actually very common for malicious Flash files to carry the exploit payload obfuscated in unsuspected data, such as compressed images. [16]

From each basic block we build an *LLVM* compatible SSA representation, using the *LLVM* API [34]. The resulting blocks are linked using branches. As ActionScript only allows relative jumps the linking can be completely done at compile time. For each method we invoke the *LLVM* JIT compiler to get a function pointer to native code, this pointer is then used to invoke the corresponding code. The generation of native code is done only once at the first invocation of the method, the resulting code pointer is cached so that subsequent calls incurs in no overhead. To maintain compatibility and interfacing with C++ code a small wrapper is used to invoke the generated code. The wrapper allocates the private stack and the local data storage. Conforming to ActionScript calling convention, arguments are copied onto the first elements of the local data array (See section 3.4.3).

Our implementation guarantees compatibility between routines written in C++ and compiled ActionScript code and it's possible to arbitrarily mix and nest both kind of code. This makes it really easy to implement ActionScript opcodes using high level routines written in C++, similarly to the interpreter approach. To write complex functionality such as variable lookup from associative arrays we make use of the convenient features of the *Standard Template Library*. High level routines are extremely inefficient for frequently used simple operations. As described later in section 3.4.2 several optimizations can be made by specializing the generic, high level operations to a simpler native version, exploiting implicit type information in the code.

3.4.1 Optimizing stack traffic

A first optimization on the ActionScript model is to purge as much traffic stack as possible. For each basic block we maintain a *static stack* and for each method a *runtime stack*. *Push* and *pop* operations are only done on the static stack at compile time. Code is translated so that each instruction that would have popped N arguments from the stack accept N arguments, and each instruction that would have pushed a value assigns the value to a variable, which

in SSA form can be considered a virtual register. The static stack keeps track of which virtual register holds the value that would be on the corresponding stack slot, so it's possible to resolve stack traffic at compile time inside each basic block. At the block boundary each stack value pushed in the block but not yet consumed is flushed to the runtime stack. *Pop* operations which cannot be served from the static stack are going to be resolved on the runtime one. In such case the popped value will reference a memory load.

The described optimization is extremely effective as most of the values pushed on the stack are extremely short lived, and often are consumed by the following instruction. Moreover looping constructs usually have no impact on the runtime stack as they consumes every value they push onto the stack. As the virtual registers have a short lifespan this do not increase register pressure on architectures which are register starved such as x86.

3.4.2 Exploiting implicit type information

ActionScript, being derived from *ECMAScript* reaches an extremely high level of abstraction, and embraces totally the object oriented model. Everything in the language is an instance of *Object*, even functions and primitive types, such as *integers* and *floating point* values. This is obtained by subclassing each class from the basic *Object*. This kind of abstraction allows for a great flexibility and a polymorphic behavior. The *ADD* instruction for example will do arithmetic addiction when applied to numerical typed values, and string concatenation if either argument is a *String*. The right operation is resolved at runtime.

This of course have a large overhead over the use of native data types. The overhead is caused by two major factors: first of all the runtime checks needed to determine the object type, but also by the high memory traffic on the heap. In the naive approach, the creation and destruction of short lived object would represent a large share of the execution time. A first optimization to solve the latter problem is to grab frequently used objects such as *Integer*s and *Number*s from a pool. Objects gathered from the pool are not going to destroy themselves when their reference count goes to zero, but will get back to their *manager*. It possible to verify that in real life scenarios, even a very small pool of around 10 object is enough to fulfill all the request for temporary objects. Figure 9 describes the relationship between the objects and their manager.

It also possible to exploit the strong typing information that is implicit in the semantic of several instructions. For example instructions that returns numerical constants gives us a very strong information about the type of the object. Moreover most arithmetical operators are specified to output a *Number* and

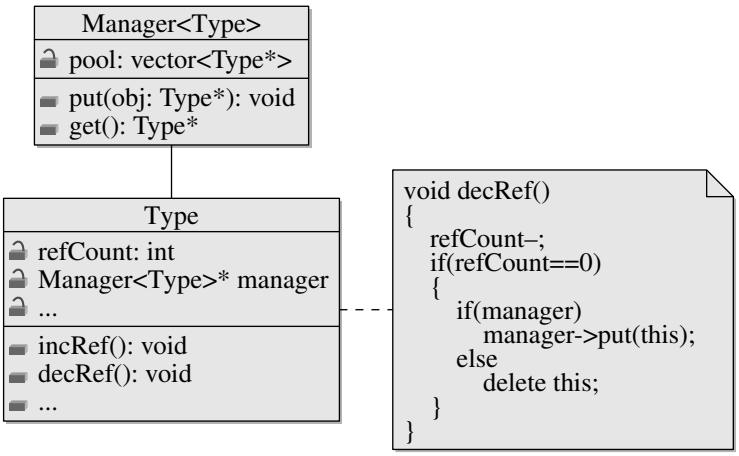


Figure 9: Relation between the object pool manager and the object instances. When reference count reaches zero unmanaged object are destroyed and managed objects go back to their pool

will convert their input values to *Integer* or *Number* type, using the conversion operators that most object offers. This kind of information, which is already known at compile time, can be used to convert the primitive data types to fully fledged object only lazily. This technique is called *Boxing* and *Unboxing* and it has been implemented first in *Java* [19]. As the static stack is not really tied to any memory, the value it references can be of any type. So we keep values as primitive types during the code parsing and compilation phases. For each value in the static stack we also store the data type known at compile time.

It's possible to propagate type information for returned values using a single pass over the code. To take advantage of the implicit conversions on arguments we need a multi-pass approach. During the first iteration, for each value pushed on the stack, we record if it's going to be converted and what the final type is going to be. Then we try to generate code that proactively casts the value to their final type before pushing them on the stack. This approach propagates type information backwards in the instruction flow, allowing a very effective optimization, especially for object properties access. The dynamic nature of ActionScript Objects gives, in general, no guarantees over the type of the properties stored in the object, but if we find that a certain property will be casted before being used we can safely ask the object to return the property already converted to the final desired type. If the object internally stores the property as a primitive type it can be returned without boxing it, otherwise it will just be casted ahead of time.

This implicit data type information that we extracted from the code flow

can be used to specialize polymorphic operations at compile time. For example the previously discussed *ADD* opcode can be converted to the native processor operation when both arguments are known to be numbers or integers, if we do not have any type information for an argument we can call the generic implementation of the instruction.

3.4.3 Optimizing accesses to method local data

One of the major disadvantage of a canonical stack machine is that no access is allowed to values not on top of the stack, so it is difficult to store long lived values such as loop counter or global data. On mainstream architectures that supports the stack semantics (such as *x86*), this problem is resolve by allowing relative addressing from the top of the stack, usually referred by a special *stack pointer register*. This allows to store on the stack both temporary values and local data for methods.

ActionScript adhere strictly to the stack machine model and only to the top of the stack is accessible. But a different data storage is available for long lived and frequently used values. Those locations are called *Locals*. Each method has to statically declare how many locals will be used, and those are referred to by their index. The first few locals are reserved by the calling convention for the *this* pointer and the function arguments, as shown in Figure 10

Locals data can only be manipulated using the `setLocal` and `getLocal` operations, the former takes the value from the top of the stack and moves it to a local, the latter copy a value from a local to the stack. Moreover a special instruction called *kill* set a local to the *Undefined* state, effectively signaling the end of the variable life. As seen for the stack values, we also try to exploit implicit type information for locals. During Just in Time compilation we build a static locals data structure where we keep both a reference to the current value stored in each local and the corresponding type, in a similar way to the static stack structure. When translating the *setLocal*/*getLocal* operations we also propagate type information. Of course the type information can only easily handled on linear code. When we reach a branch we would have to synchronize our static_locals to the actual locals array in memory by creating a fully fledged object for values stored in native types. This can be evaded by doing *inter block*

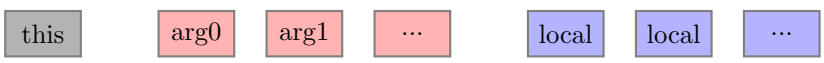


Figure 10: Locals data is used for long lived values. The first locations are reserved for the *this* pointer and the function arguments.

analysis.

An iterative approach is applied to propagate type informations from each block to each possible subsequent block. During this operations we also finds out when local values will no longer be used and so their value can be safely discarded.

For each method, before final code generation:

1. Set the types of the static locals for each block to *Object*, as we have not yet any information on the types.
2. For each block
   (a) We load the current known types of the static locals at the beginning of this block
   (b) If the code of this block effectively reset a local to a known value, we flag the local as *resetted*. A locals is considered resetted when a block invokes a *setLocal* or a *kill* before any *getLocal* on the index.
   (c) We parse the code and extract object type informations. When appropriate type information is propagated to the static locals. Each iteration will gather more information about the local variables' type and more operations will be specialized to native versions.
   (d) At the end of the block we save the current locals' type.
3. For each block: we check if local at index n is of the same type for each predecessor block. If so we propagate this information to the types at the beginning of this block.
4. If any of the known types at the beginning of a block changed we get back to step 2

At the end of this loop we are going to do the actual code generation, exploiting all the type informations that we gathered. At the end of each block a different epilogue code is generated for each successor, to cast object to the expected data types. Locals that are going to be *resetted* in a block are discarded during the epilogue.

4 Results

4.1 Testing infrastructure

To have faithful performance comparison against other engines we used the test suite from the *Tamarin* project. The actual tests are derived from the *Java Grande Benchmark* suite [24]. A version of the tests ported to ActionScript is included in the *Tamarin* source tree and can be compiled to ActionScript bytecode using Adobe's *ASC* compiler. The tests are run against Lightspark and the development branches of Tamarin, both the classic one and the *tracing* version.

The suite includes the following tests

Crypt Encrypts and decrypts random data using the *IDEA* symmetric algorithm. This test implies a great deal of integer and logic operations.

FFT Calculate the *Fast Fourier Transform* over random data. This test stress floating point calculation.

Series Calculate the first N coefficients of a Fourier series. This test makes a lot of calls to functions members of the Math class

SOR Calculate the solution of a linear system using the *Successive over-relaxation* method

SparseMatMult Execute matrix multiplication over a sparse matrix. This test makes heavy use of indirect memory accesses.

4.2 Testing results

From Figure 11 is clear that *Lightspark* is lagging slightly behind Tamarin, but the running times are actually on the same order of magnitude. Those results depends on several factors:

1. Tamarin is a mature project which has been developed by a professional team over several year, while Lightspark has not yet undergone the necessary tuning and performance profiling process
2. Moreover also the *LLVM* framework is very young and Lightspark will automatically take advantage of the improvements on that framework, both on compilation speed and code optimization.
3. ActionScript 3 introduced an optional strict typization for function local variables. Basically typed locals are declared inside an *activation object*, and accessed as normal properties. This feature is not yet completely supported by Lightspark.
4. Several frequently called functions, for example the ones in the Math class, could be linked at compile time, without resorting to a slow map lookup at runtime.

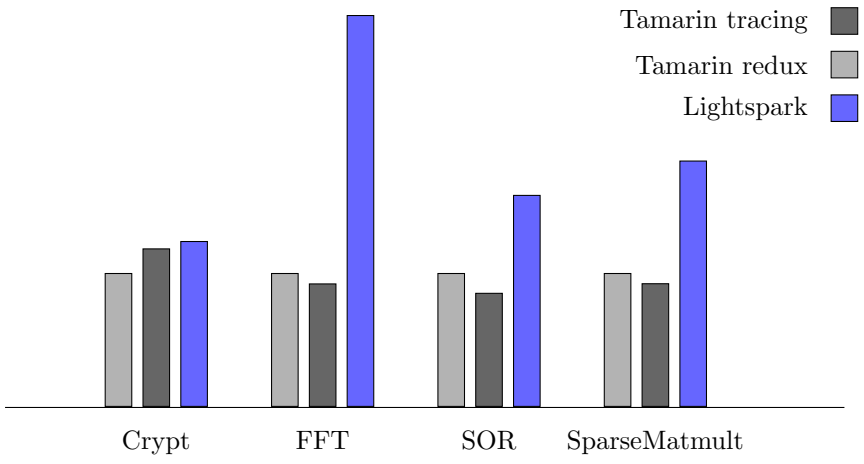


Figure 11: Normalized execution times for the test cases. The reference time is from *Tamarin redux*, which is the development branch of the stable version of *Tamarin*. Lightspark is not yet mature enough to outperform *Tamarin*, but the results in the Crypt test are very encouraging. The tests were run on a 64bit *Intel Pentium D* machine. It must be noted that the comparison with *Tamarin tracing* is not completely fair, as the *x86_64* platform is not yet supported, and so the 32bit version was used.

References

[1] *Browser War* article on wikipedia.
http://en.wikipedia.org/wiki/Browser_war.

[2] *FFmpeg* project homepage.
http://ffmpeg.org/.

[3] *Forth* language article on wikipedia.
http://en.wikipedia.org/wiki/Forth_(programming_language).

[4] *GPLFlash* project homepage.
http://gplflash.sourceforge.net/.

[5] *GStreamer* project homepage.
http://gstreamer.freedesktop.org/.

[6] *IEEE* standard for floating-point arithmetic (*IEEE 754*) article on wikipedia.
http://en.wikipedia.org/wiki/IEEE-754.

[7] *NanoJIT* project homepage.
https://developer.mozilla.org/En/Nanojit.

[8] *Reverse Polish Notation* article on wikipedia.
http://en.wikipedia.org/wiki/Reverse_Polish_notation.

[9] *Static Single Assignment* article on wikipedia.
http://en.wikipedia.org/wiki/Static_single_assignment_form#Compilers_using_SSA_form.

[10] Adobe. Actionscript virtual machine 2 overview.
http://www.adobe.com/devnet/actionscript/articles/avm2overview.pdf.

[11] Adobe. Adobe and industry standards.
http://www.adobe.com/enterprise/standards/.

[12] Adobe. Adobe flash player version penetration. "http://www.adobe.com/products/player_census/flashplayer/version_penetration.html".

[13] M. Cha, H. Kwak, P. Rodriguez, Y.Y. Ahn, and S. Moon. I Tube, You Tube, Everybody Tubes: Analyzing the Worlds Largest User Generated Content Video System.

[14] X. Cheng, C. Dale, and J. Liu. Understanding the Characteristics of Internet Short Video Sharing: YouTube as a Case Study. *ArXiv e-prints*, July 2007.

[15] Dragos-Paul. Introducing Microsoft Silverlight.

[16] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Analyzing and Detecting Malicious Flash Advertisements.

[17] Free Software Foundation. High priority free software projects. `http://www.fsf.org/campaigns/priority.html`.

[18] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical report, Citeseer, 2006.

[19] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java (TM) Language Specification, The (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[20] Intel. Intel 64 and ia-32 architectures software developer's manual. `http://www.intel.com/products/processor/manuals/`.

[21] Ecma International. The *ECMA-262* specification. `www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`.

[22] Ecma International. The *ECMA-357* specification. `www.ecma-international.org/publications/files/ECMA-ST/Ecma-357.pdf`.

[23] M. Jenkins, P. Liu, R. Matzkin, and D.L. McFadden. The browser war: Econometric analysis of Markov perfect equilibrium in markets with network effects. *Unpublished manuscript, University of California, Berkeley*, 2005.

[24] JA Mathew, PD Coddington, and KA Hawick. Analysis and Development of Java Grande Benchmarks.

[25] E. Meijer, WA Redmond, J. Gough, and A. Brisbane. Technical Overview of the Common Language Runtime. 29:7.

[26] Jr. Philip J. Koopman. *Stack Computers: the new wave*. Ellis Horwood, 1989.

[27] Alessandro Pignotti. *Lightspark* project homepage. `http://lightspark.sourceforge.net`.

[28] Alessandro Pignotti. Lightspark: An high performance flash player implementation.
http://lightspark.sourceforge.net.

[29] J. Singer. Gcc .neta feasibility study. *Journal of .NET Technologies*.

[30] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[31] The GCC team. Gnu compiler collection project homepage.
http://gcc.gnu.org/.

[32] The Gnash team. Gnash project homepage.
http://www.gnu.org/software/gnash/.

[33] The LLVM team. Clan vs other open source compilers.
http://clang.llvm.org/comparison.html.

[34] The LLVM team. Llvm api documentation.
http://www.llvm.org/doxygen/.

[35] The LLVM team. Llvm project homepage.
http://www.llvm.org/.

[36] W3C. A vocabulary and associated apis for html and xhtml.
http://www.w3.org/TR/html5/.