

The Impact of Optional Type Information on JIT Compilation of Dynamically Typed Languages

Mason Chang^{*†} Bernd Mathiske[†] Edwin Smith[†] Avik Chaudhuri[†] Andreas Gal[§]
Michael Bebenita^{*} Christian Wimmer^{*} Michael Franz^{*}

^{*}University of California, Irvine [†]Adobe Systems [§]Mozilla Corporation
{changm, mbebenit, cwimmer, franz}@uci.edu {bmathisk, edwsmith, achaudhu}@adobe.com gal@mozilla.com

Abstract

Optionally typed languages enable direct performance comparisons between untyped and type annotated source code. We present a comprehensive performance evaluation of two different JIT compilers in the context of ActionScript, a production-quality optionally typed language. One JIT compiler is optimized for quick compilation rather than JIT compiled code performance. The second JIT compiler is a more aggressively optimizing compiler, performing both high-level and low-level optimizations.

We evaluate both JIT compilers directly on the same benchmark suite, measuring their performance changes across fully typed, partially typed, and untyped code. Such evaluations are especially relevant to dynamically typed languages such as JavaScript, which are currently evaluating the idea of adding optional type annotations. We demonstrate that low-level optimizations rarely accelerate the program enough to pay back the investment into performing them in an optionally typed language. Our experiments and data demonstrate that high-level optimizations are required to improve performance by any significant amount.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors - Compilers

General Terms Design, Languages, Performance

Keywords ActionScript, JavaScript, Type Speculation, Type Inference, Tamarin, Dynamic Compilation, Optionally Typed Languages

1. Introduction

Developing a high performance virtual machine for a dynamically typed programming language is an open field of research. Dramatic performance improvements in dynamically typed languages occur because the virtual machines powering them speculatively compile dynamically typed code as statically typed code using runtime profiling mechanisms. Adding type information greatly improves the runtime performance of a virtual machine. Once dynamically typed code is approximated as statically typed code, well known static compilation techniques can be applied to improve performance by an even greater margin. However, two questions arise:

1. What is the performance improvement as an application developer adds type annotations to their program?
2. Which compiler optimizations improve performance?

We investigate answering these questions with a comprehensive performance evaluation of Tamarin, the virtual machine (VM) shipping in Adobe's Flash Player. Tamarin executes ActionScript which, like JavaScript, is a dialect of ECMAScript. Optionally typed languages, like ActionScript, enable language designers to directly measure the performance impact of adding type annotations to a dynamically typed language. We compare Tamarin's performance with two different JIT compilers, leaving all other modules in the VM unmodified. We compare NanoJIT, a compiler that performs only a limited set of optimizations, with our new experimental Type Enriched Static Single Assignment (TESSA) system, a higher level and more aggressively optimizing JIT compiler that uses LLVM as the back end for aggressive low-level optimizations. We evaluate both JIT compilers across two different benchmark suites, all modified to be fully type annotated, partially type annotated, or untyped. First, our system enables us to determine how much does adding type information to a program improve runtime performance. Second, we evaluate the performance impact of an aggressive optimizing compiler versus a JIT compiler that is designed to compile quickly.

Our paper contributes the following:

- We present a comprehensive performance evaluation of two benchmark suites when the source code is fully type annotated, partially typed, and untyped. (Section 4.1).
- We compare the performance of JIT compiled code generated by our aggressively optimizing compiler TESSA, with the lightweight NanoJIT compiler across two benchmark suites in all type configurations.
- We demonstrate that low-level optimizations from an aggressively optimizing back end rarely produce significant performance increases that justify their compilation time and high-level optimizations are required for significant performance increases. (Section 4.5).

2. System Architecture

We implement and perform our evaluation on the open source Tamarin virtual machine [10], which currently ships in Adobe's Flash player. Tamarin executes ActionScript [1], a dialect of ECMAScript [8]. JavaScript is the most common dialect of ECMAScript. ActionScript builds upon ECMAScript, adding language features such as optional static typing and classes. Most JavaScript can be executed without modifications on Tamarin.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DL'S'11, October 24, 2011, Portland, Oregon, USA.

Copyright © 2011 ACM 978-1-4503-0939-4/11/10...\$10.00

First, ActionScript source code is compiled into the ActionScript bytecode (ABC) file, which is the conceptual equivalent of a Java .class file. Adobe’s Flex SDK [6] compiles ActionScript source code into an ABC file.

Tamarin takes an ABC file as the input and begins execution from the global method via bytecode interpretation. When a method is called, Tamarin generates machine code for ActionScript methods via just-in-time (JIT) compilation. An ActionScript method is JIT compiled just before its first invocation. Only the global method is interpreted.

2.1 High and Low Level Optimizations

We define high and low optimizations by their conceptual semantic level. A high level optimization is aware of both the runtime behavior of the program and the language semantics. High level concepts include ActionScript types, such as arrays, objects and the *Any* type. Our definition also requires knowledge of ActionScript methods, which includes how to inline other ActionScript methods.

Low level optimizations are language agnostic because they do not take into account the semantics of the language or the runtime behavior of the running program. Low level optimizations also express operations that are closer to the physical machine, such as a memory location, or that an integer requires 4 bytes. Physical register allocation, machine instruction scheduling, redundant load and store elimination are examples of low level optimizations.

Consider the ActionScript *Any* type which can represent any other ActionScript type. High-level optimizations such as type inference and type speculation try to disambiguate and remove the *Any* type wherever possible because they incur a large runtime performance penalty. Low-level optimizations must define the *Any* type as a machine type, such as a 32 bit signed integer with the bottom three least significant bits as a type tag. Converting between the *Any* type and another primitive machine type can occur with bit manipulations. Optimizing the bit manipulations are low-level optimizations. Removing the *Any* type completely is a high level optimization.

2.2 ABC File Format

The Tamarin virtual machine is a stack machine that executes bytecodes contained in the ABC. Each bytecode either pushes, pops, or modifies values in one of two virtual stacks. The operand stack contains values that are used as input operands for the bytecodes. The scope stack contains objects that are used for property look up and resolution. Local variable types are either the developer’s explicitly annotated type, or the *Any* type if no explicit type annotation is given. Operand stack types, which represent temporary values, are defined by the bytecode. All the available type information that is presented in our evaluation originate from the ABC file.

2.3 NanoJIT Architecture

Adobe’s Flash and Mozilla’s Firefox web browser both use the shared NanoJIT JIT [5] compiler. NanoJIT compiles code extremely quickly, sacrificing code quality in exchange for a fast startup and compilation time. It performs only a limited number of optimizations. During JIT compilation, ABC bytecodes are translated into low-level intermediate representation (LIR) [3].

NanoJIT’s LIR models both the ActionScript operand and scope stack. Each ActionScript bytecode is translated to a series of LIR instructions that explicitly load or store values onto either stack. LIR also contains type coercions for values as needed. For example, calling an untyped method requires all arguments to be boxed into the ActionScript *Any* type. The logic for complex type coercions, such as *Any* to *Integer*, which must resolve the runtime type of the input value, become calls to C++ methods. Building

LIR instructions instead of a C++ call requires too many LIR instructions.

In select performance critical cases, such as virtual dispatch look up, NanoJIT does inline the fast path of a C++ runtime method. The inlining is done manually and the VM developer must hand create NanoJIT LIR to implement the equivalent C++ runtime functionality. We detect and evaluate the performance critical cases through a manual performance tuning process. NanoJIT does not collect any runtime profiling information and does not inline any ActionScript methods.

NanoJIT performs a limited number of optimizations during LIR creation via a pipeline of filters [25]. Each filter examines an incoming LIR instruction and performs a specific task. For example, the common subexpression elimination (CSE) filter first examines if the LIR instruction is pure and does not cause side effects. Then, the filter looks for instructions that have the same LIR opcode and are defined prior to the current instruction. If a previously defined instruction contains equivalent operands as the current instruction, the current LIR instruction is eliminated.

Another optimization pass through the LIR instructions eliminates redundant loads and stores. A linear scan register allocator [37] assigns each LIR instruction a physical machine register or stack slot as needed. Finally, NanoJIT assembles machine instructions.

2.4 TESSA System Architecture

In order to perform high-level optimizations, we translate ABC bytecodes into our TESSA IR. Each high-level optimization is implemented with the visitor design pattern. Currently, we implement type inference, ActionScript method inlining, and dead code elimination. Once we perform all high-level TESSA optimizations, we translate the TESSA IR into a low-level IR.

Once TESSA is lowered into a low-level IR, we perform low-level optimizations such as instruction scheduling. The current TESSA system implementation uses the LLVM intermediate representation [4] (LLVM IR) as the target low-level IR. LLVM’s back end chooses and performs all low-level optimizations. Finally, LLVM’s back end generates machine code [32].

2.5 TESSA Intermediate Representation

Our Type Enriched Static Single Assignment (TESSA) IR is an experimental high-level IR designed to enable more optimizations and expose more type information than NanoJIT’s LIR currently allows. Like previous well known virtual machine architectures [14] [18], which contain both high-level and low-level IRs, TESSA is our high-level IR. We perform high-level optimizations such as method inlining and type inference with this IR.

The TESSA IR uses SSA form [23], with the design of the instructions influenced by both LLVM and the Java HotSpot™ Client Compiler [31]. TESSA has two unique representations: a *Value* and an *Instruction*. A *Value* represents an ActionScript value, but cannot perform any operations. For example, all literal constants in ActionScript source are represented as a *Value*.

An *Instruction* performs a specific operation using multiple input *Value* operands, producing multiple output *Values*. Every *Value* has a *Type* that represents a high-level ActionScript type. We have a unique TESSA *Instruction* for each ABC bytecode that does not push a constant onto the stack.

2.6 Architecture Comparison

Figure 1 compares NanoJIT’s and TESSA’s architecture. NanoJIT takes as input ABC files, translates ABC bytecodes into NanoJIT’s LIR, and then quickly assembles machine code. The TESSA system also takes the same input ABC file and creates TESSA IR. We perform high-level optimizations with the TESSA IR, then lower

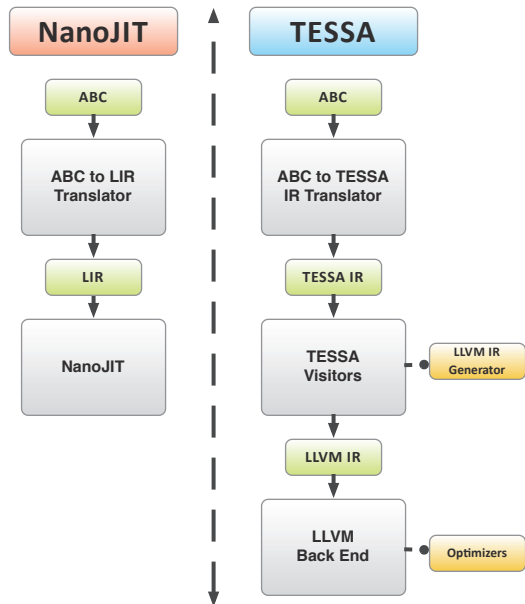


Figure 1. Architecture comparison of NanoJIT and TESSA.

the TESSA IR into LLVM IR. The LLVM back end then performs multiple low-level optimizations, finally assembling machine code.

We recognize the trade-offs required with each design. NanoJIT enables extremely fast compilation time while sacrificing performance of the JIT compiled code. The TESSA system requires more compilation time to produce faster JIT compiled code. The TESSA system has more compilation overhead because optimizations are performed on both the TESSA IR and LLVM IR.

3. Implementation Details

We implement TESSA as the IR in the C++ Tamarin virtual machine. Each ABC bytecode corresponds to at least one TESSA instruction. Each TESSA *Instruction* inherits from a base *Instruction* class. Unique instructions such as a *BinaryInstruction* (used for addition, subtraction, etc), contain pointers to their operands. A method contains multiple basic blocks, with each block containing a list of instructions. Each *Value* has a *Type* object associated with it. Each type defined in the ActionScript language has a unique C++ *Type* object. For example, the *Integer* type represents a 32 bit signed integer. The *Any* type is used when no type information is available for a *Value*. The following section details how we implement some IR operations.

3.1 Method Inlining

We have two heuristics to determine if an ActionScript method is inlined. First, the method must be short, having a bytecode length of less than 40 bytes, which is roughly 20-25 ABC bytecodes. Second, a method cannot inline more than four other methods to limit JIT compiled code size.

We perform method inlining by including the TESSA IR of the callee method into the caller. If a call has a virtual dispatch, we inline the base class' method. The inlined method is potentially incorrect as the vtable look up resolves a different method. To solve this, we insert a runtime check to ensure that the inlined method is the correctly dispatched vtable look up method. If the inlined method is not the correct method, we call the correct method instead of executing the inlined method.

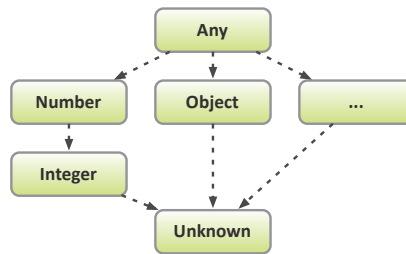


Figure 2. A subset of the type lattice modeling the type system in ActionScript. The *Any* type is the top of the lattice and the *Unknown* type is the bottom of the lattice.

We only inline a method if we can statically bind at least one method at a given call site. The ActionScript language specification states that the VM is allowed to statically bind a method name if both the dispatched method and the receiver type are known at compile time, which occur when the application programmer type annotates the receiver object and wraps the dispatched method in the ActionScript *package* keyword.

3.2 Type Inference

Our type inference algorithm relies on well-known type inference techniques [24][34][36] and is greatly influenced by Click et al. [22]. We model type inference as an iterative data flow problem. The current implementation only considers the inference of ActionScript primitive types. Furthermore, type inference is limited to local variables. In particular, we do not infer types for function parameters. Modeling the heap and global variables is future work.

Our type inference problem is encoded as a standard data flow analysis. The goal of the analysis is to infer, where possible, more precise types for local variables that are annotated with type *Any*. We first initialize the output type of each instruction to the bottom (\perp) of the type lattice (which is the most precise type, but unsound).

Next, we perform a standard forward iterative data-flow analysis that stops once a fixed point is reached. We compute a new type for the output of each instruction based on the type of its inputs. In particular, the instructions may be ActionScript primitive operations, function calls, assignments, or Φ , among others. The types of the inputs of an instruction must be subtypes of the expected types of the inputs of that instruction. We monotonically move up the type lattice, finding decreasingly precise types for the output of each instruction. The algorithm completes when the type of the output of every instruction has not changed (we have reached a fixed point). The inferred types are always sound. Furthermore, the algorithm is guaranteed to terminate because the worst-case scenario is that every instruction's type becomes the *Any* type, which is the top (\top) of the type lattice.

Our algorithm is formally described with a type lattice. Figure 2 demonstrates a subset of the ActionScript type system. The top (\top) of the type lattice is the least precise type, which represents the ActionScript *Any* type. The bottom (\perp) of the type lattice is the most precise type.

We assume that signatures for the ActionScript primitive operators are available. Since operators may be overloaded, we assume standard monotonicity conditions for their signatures: if $T_1, \dots, T_n \rightarrow T$ and $T'_1, \dots, T'_n \rightarrow T'$ are signatures for some operator, then

1. There is also a signature of the form $T_1 \cap T'_1, \dots, T_n \cap T'_n \rightarrow T''$ for that operator;
2. If $T_1 \subseteq T'_1, \dots, T_n \subseteq T'_n$, then $T \subseteq T'$.

ActionScript primitive operations satisfy these conditions. If after some iteration, the inferred types of the inputs of an ActionScript primitive operation are S_1, \dots, S_n and there is a signature $T_1, \dots, T_n \rightarrow T$ for that operator such that $S_1 \subseteq T_1, \dots, S_n \subseteq T_n$, then the inferred type of the output of the instruction becomes T . The monotonicity conditions ensure that this choice is deterministic, and the type of the output cannot decrease if the types of the inputs increase. Likewise, if after some iteration, the inferred types of the inputs of a Φ are S_1, \dots, S_n then the inferred type of the output of the instruction becomes $S_1 \cup \dots \cup S_n$.

Consider the ActionScript code `for (var i = 0; i < 100; i++)`. The Φ for the variable `i` is and the addition of `i++` are both initialized to unknown (\perp). The analysis reaches the Φ which has an integer (`const 0`) as input and unknown. The union of the types moves up the lattice from Unknown to Integer. Next, the addition (`i++`) adds an integer (`const 1`) and integer (Φ), which results in a number because we do not overflow check. Finally, we analyze the Φ again, which has inputs integer and number, and we move the type of the Φ up the lattice to the number type.

We initialize all values to \perp because of loops and Φ functions. Because we iterate in forward order, we assign types to loop Φ functions before their uses. If we initialized all values to \top , all uses of the Φ also resolve to \top .

3.3 LLVM IR Generation

Once an ActionScript method is in the TESSA high-level IR, we lower it to LLVM IR [4]. LLVM IR is designed to be platform independent, extensible, and express different structural and primitive types. LLVM IR is also in SSA form. Once we generate LLVM IR from TESSA IR, LLVM’s back end performs its low-level optimizations and generates machine code.

The `LLVMIRGenerator` iterates over each TESSA instruction, performing a straightforward translation of the semantics of each TESSA instruction into LLVM IR. For example, a TESSA *ConditionalBranch* corresponds to a LLVM `br` instruction. Each basic block in TESSA IR corresponds to a basic block in LLVM IR.

Our biggest challenge occurs in converting TESSA’s type system to LLVM IR’s type system. All values in Tamarin must either be a primitive type, (e.g., signed integer, double, object) or the `Atom` type, which represents the abstract `Any` type.

In many cases, LLVM’s type system is more strict than TESSA’s type system. For example, a TESSA boolean value must be represented as one bit in LLVM IR. LLVM’s type system also dictates that booleans must only be compared with other booleans. Thus, our implementation must implicitly convert primitive types. For example, when comparing a TESSA integer with a floating point value, we must convert the integer to the floating point type in LLVM IR. We insert these internal LLVM conversions when merging TESSA types with LLVM primitive types.

Since LLVM does not have an `Any` type, we have to emit explicit coercions from LLVM types to the `Atom` type in LLVM IR. The implicit type boxing and unboxing with the `Atom` type becomes explicit in LLVM IR. An `Atom` type is defined as a 32 bit LLVM signed integer, with the 3 least significant bits used as a type tag. Coercing between the `Atom` type and other primitive types require a call into a VM C++ method, such as `doubleToAtom` that coerces a primitive double to an `Atom` type. In such cases, we emit a LLVM call instruction to the VM C++ method.

3.4 Example Execution

This section gives a detailed example of actual runtime execution of a small ActionScript program. Consider the ActionScript source in table 3 and compiled ABC in table 1. ActionScript bytecodes are instructions for a VM with a stack architecture, where instruction operands are pushed onto and popped off an operand stack.

```
var sum : Number = 0;
for (var i : int = 0; i < 100; i++) {
    sum += i;
}
```

Figure 3. The ActionScript source used in this example program execution.

ID	ABC Opcode	Local Variables	Operand Stack
0	pushint 0	[[A] [A]]	int
1	convert_num	[[A] [A]]	Number
2	setlocal1 (\$sum)	[Number *[A]]	empty
3	pushint 0	[Number *[A]]	int
4	convert_int	[Number *[A]]	int
5	setlocal2 (\$i)	[Number int]	empty
6	jump BB 11	Loop Body	
7	getlocal2 (\$i)	[Number int]	int
8	pushint 10	[Number int]	int int
9	if less than, jump B11	Loop Body	
10	return		
11	getlocal1 (\$sum)	[Number int]	Number
12	getlocal2 (\$i)	[Number int]	Number int
13	add	[Number int]	Number
14	convert_num	[Number int]	Number
15	setlocal1 (\$sum)	[Number int]	empty
16	getlocal2 (\$i)	[Number int]	int
17	increment_int	[Number int]	int
18	convert_int	[Number int]	int
19	setlocal2 (\$i)	[Number int]	empty
20	jump BB 7		

Table 1. Example of Flex’s generated ABC bytecode given the ActionScript source.

manages local variables by allocating dedicated slots where references are loaded and stored to data objects. All values that are stored into local variable slots or pushed onto the operand stack are typed, with the *any* `[A]` type as the least precise type.

Consider the first ABC bytecode `pushint 0` (*id 0*). The opcode pushes a constant integer #0 onto the operand stack. The next ABC bytecode `convert_num` (*1*), converts the top operand on the operand stack into the ActionScript `number` type. Next, `setlocal1` (*2*) pops the top operand from the operand stack and stores it into local slot 1.

Control flow operations such as `jump` (*6*) and `if less than` (*9*) change program control flow to begin execution at a given ABC bytecode location. Thus, `jump` instructions represent the end of a basic block. The targets of control flow instructions, such as `if less than` (*9*) represent the beginning of a basic block. The ABC verifier ensures that the operand stack is consistent at all control flow merge points.

3.4.1 Sample NanoJIT Execution

Table 2 demonstrates the final optimized NanoJIT LIR that is created from the example ABC. A code generator analyzes each ABC bytecode and generates a snippet of NanoJIT LIR. NanoJIT LIR precisely models the ABC local variable array, scope stack, and operand stack, including their type signatures. All three ABC data structures are stack allocated in each JIT frame.

Consider the LIR instruction `store_double` (*0*), which represents the ABC bytecodes `pushint` (*0*), `convert_num` (*1*), `setlocal1` (*2*). The LIR instruction has the immediate 0 as an operand must store the constant as a double, into local variable 1. We also see that the same control flow graph represented by ABC is kept intact. Jump

ID	LIR Instruction	Comment
0	store_double local1 = #0	
1	store_int local2 = #0	
2	j → B24	jump loop header
B14		loop body
3	ldd1 = load_double local1	
4	ldi1 = load_int local2	
5	i2d1 = i2d ldi1	int to double
6	addd1 = addd ldd1, i2d1	double add
7	store_double local1 = addd1	
8	addi1 = addi ldi1, #1	integer add
9	store_integer local2 = addi1	
B24		loop header
10	ldi2 = load_int local2	
11	lti1 = lti ldi2, #10	int less than
12	jt lti1 → B14	jump true

Table 2. Example of NanoJIT’s generated LIR from the example ABC bytecodes.

ID	TESSA Instruction
1	ConstantValueInstruction #0
2	ConvertInstruction (1) → number
3	ConstantValueInstruction #0
4	UnconditionalBranch BB1
BB1	Loop Header
5	PhiInstruction [(BB0, 3), (BB2, 12)] (int)
6	PhiInstruction [(BB0, 1), (BB2, 10)] (number)
7	ConstantValueInstruction #100
8	ConditionInstruction LESS_THAN 5 7 (boolean)
9	ConditionalBranchInstruction 8 [BB2, BB3]
BB2	Loop Body
10	BinaryInstruction ADD 6 5 (number)
11	ConstantValueInstruction #1
12	BinaryInstruction ADD 5 11 (int)
13	UnconditionalBranch BB1
BB3	Return block
14	ConstantValueInstruction Undefined
15	Return 14 (Any)

Table 3. Example of the TESSA IR for the ActionScript source example.

instructions, such as `j → B24` (2), are block terminators at the same program location as shown in ABC. NanoJIT does not schedule or move instructions into different basic blocks.

Finally, since LIR is not in SSA, all ABC loads and stores also exist in NanoJIT LIR. Values are not merged into a single value using Φ instructions.

3.4.2 TESSA Execution

The same ActionScript source code in TESSA IR is shown in table 3. Initially, each ActionScript basic block becomes one BasicBlock in TESSA IR. Because TESSA IR is in SSA form, we have Φ instructions, which eliminates the explicit loads and stores in ABC bytecode. Each Φ contains a list of (Block, Value) pairs, denoting the value of the Φ depending on the incoming control flow edge. Each TESSA instruction has a high level TESSA type, representing the primitive ActionScript types.

We lower TESSA IR into LLVM IR in a mostly one to one translation, as shown in table 4. LLVM IR allows only typed operations, such as addition, whereas TESSA IR is higher level, allowing typed and untyped additions. For example, `fadd double` performs a floating point addition, requiring both operands to be floating point

ID	LLVM IR
%22	add i32 0, 1
%23	sitofp i32 0 to double
%24	sitofp i32 0 to double
	br label BB1
BB1	Loop Header
%25	Φ i32 [0, %BB0], [%35, %BB2]
%26	Φ double [%24, %BB0], [%34, %BB2]
%28	icmp slt i32 %25, #100
	br i1 %28, label %BB2, label %BB3
BB3	
	ret i32 4
BB2	Loop Body
%33	sitofp i32 %25 to double
%34	fadd double %26, %33
%35	add i32 %25, 1
	br label %BB1

Table 4. Example of the LLVM IR that is generated after lowering TESSA IR.

operands. Thus, we insert the `sitofp` LLVM instruction, converting a Signed Integer to Floating Point value. Finally, LLVM IR inlines constants as an operand when available, such as instruction `%35 add i32, %25, 1`.

4. Evaluation

We evaluate the performance of JIT compiled code generated by both NanoJIT and TESSA with the LLVM 2.8 back end. We first examine pure performance with different amounts of type information provided by the programmer. Next, we investigate LLVM’s optimization configurations to determine the effect of low-level optimizations on performance. Finally, we characterize the overhead associated with the TESSA approach versus NanoJIT. All evaluations are performed Windows 7 x32, using an Intel Core 2 Duo E8400 3.0 GHz processor with 4 GByte of memory.

4.1 Performance

We evaluate performance using both the SunSpider [9](SS) and V8 benchmark suite [15]. Both are originally JavaScript benchmark suites designed to run in a web browser. Each benchmark suite is run in multiple configurations:

1. *NanoJIT*: The unmodified product version of Tamarin.
2. *TESSA Baseline*: A straight translation from ABC to TESSA IR, TESSA IR to LLVM IR, and LLVM IR to JIT compiled code. No TESSA optimizations are performed. All LLVM optimizations are enabled.
3. *TESSA Optimized*: Method inlining is performed, followed by type inference and then dead code elimination.

We evaluate the effect of both high-level and low-level optimizations across fully typed, partially typed, and untyped code. Fully typed code takes advantage of the optional manual type annotations available in ActionScript. We manually modified each benchmark to include type annotations. Partially typed code contains type annotations for method signatures, which include only method parameters and the method return types, as well as heap variables. All local variables remain untyped. Untyped code is unmodified benchmark code.

Both benchmark suites run extremely quickly, with the whole SunSpider benchmark executing in 500 ms on all modern browsers. We are more interested in actual peak performance of JIT compiled code, not compilation time. To accurately measure all three type

configurations, we manually modify each typed benchmark to have a total execution time of 30 seconds with NanoJIT by wrapping the benchmark in a loop. We then modify the partially typed and untyped benchmarks to execute the same number of loop iterations as the typed benchmarks. This allows us to test the impact of any optimizations that both TESSA and LLVM have compared to NanoJIT.

All benchmark graphs in the following sections are normalized against NanoJIT on fully typed code. A value of 100% means that the current benchmark and configuration executes as fast as NanoJIT on typed code. A value less than 100% means that the configuration is slower than NanoJIT. A value greater than 100% means the configuration is faster than NanoJIT.

We do not have benchmark results for `V8\earley -boyer` and the SunSpider benchmarks `regexp-dna`, `date-format-xparb`, `string-base64`, `string-tagcloud`, and `s3d-raytrace`. `V8\earley -boyer` has a known verifier bug that is fixed in the shipping Tamarin virtual machine, but the fix is not yet applied in our research implementation. The SunSpider benchmarks `regexp-dna` and `string-base64` have known front end ASC bugs that are fixed in the shipping Tamarin VM. The benchmarks `date-format-xparb` and `string -tagcloud` use `eval`, which is not supported in ActionScript.

4.1.1 Typed Code Performance

Figure 4 demonstrates the performance of TESSA with LLVM's back end and NanoJIT on fully type annotated ActionScript source code. The overall picture shows that despite the number of optimizations LLVM performs over NanoJIT, very little actually improves in terms of JIT compiled code performance. The geometric mean demonstrates that TESSA is equal to NanoJIT and 4% faster than NanoJIT with all TESSA optimizations enabled.

There are some clear cases where LLVM's optimizations dramatically outperform NanoJIT, such as `SS\s3d-cube` and `SS\bitops -bitwise -and`. However, in many cases, LLVM's back end generates slightly slower or equally fast code compared to NanoJIT. Once higher-level optimizations are enabled, TESSA outperforms NanoJIT. For example, `V8\richards` and `V8\deltablue` outperform NanoJIT due to method inlining.

`SS\bitops-bitwise-and` contains the biggest performance gain across all of the benchmarks, being 2.7x faster than NanoJIT. However, we see performance is equal regardless of any higher-level optimizations. TESSA outperforms NanoJIT because the benchmark contains one loop that executes very frequently and LLVM's register allocator keeps loop variables in a register, thus greatly increasing the performance. NanoJIT must load and store both values on and off the stack during each iteration of the loop.

TESSA outperforms NanoJIT on `V8\crypto` by a large margin, with the gap becoming wider with type inference. The baseline TESSA configuration outperforms NanoJIT not because of LLVM, but because we inline the conversion of an ActionScript floating point number to an integer instead of calling a C++ method. TESSA's type inference improves performance even more because the `crypto` benchmark performs numerous computations, creating many temporary values, before finally assigning a result to a local variable. Our type inference algorithm determines the types of the temporary values while NanoJIT does not.

Both `V8\richards` and `V8\deltablue` do not improve much from LLVM's optimizations, but do gain a considerable amount of performance improvement because of method inlining. Both benchmarks consist of many small methods, thus inlining those small methods in hot loops removes the overhead of each call instruction, gaining performance. The SunSpider benchmarks `s3d-cube`, `bitops-bits-in-byte`, and `access-nbody` are the

only SunSpider benchmarks where LLVM's lower-level optimizations outperform NanoJIT.

We report a few negative results too. In many cases, LLVM's back end actually performs worse than NanoJIT with `SS\control flow -recursive` being the worst performing benchmark. NanoJIT outperforms LLVM because NanoJIT does common subexpression elimination on pure function calls. The recursive method calls a C++ VM method to look up the definition of the recursive method multiple times. NanoJIT eliminates these definition look ups while LLVM does not. We will investigate ways to enable LLVM's CSE pass to eliminate pure VM C++ function calls as well.

Finally, we see that we lose performance when we enable type inference on `V8\splay`, and the SunSpider benchmarks `access-nbody`, `math-cordic`, `math-partial-sums`. In these test cases, we are able to obtain a more precise type for a variable than was declared. For example, an ActionScript variable that is declared as a floating point number is actually only assigned unsigned integer values. However, the variable is always compared to a signed integer. To produce the correct value, we convert both the signed integer and unsigned integer into a floating point type, using a floating compare to compare the two values. This is legal because both a signed and unsigned 32 bit integer value can fit into a 64 bit value. The conversion is done in a hot loop and incurs the current reduction in performance.

4.1.2 Partially Typed Code Performance

Figure 5 demonstrates the performance results for partially typed code. Partially typed code has the same type annotations as fully typed code, except that all function local variables are untyped. Heap variables and method signatures still retain all their type annotations. All benchmarks are normalized against NanoJIT on typed performance. Using the geometric mean, NanoJIT is 51%, TESSA 52%, and TESSA with all optimizations 30% slower than NanoJIT on typed code. The overall picture is the same as typed code performance. LLVM's back end has roughly the same performance as NanoJIT's without any high-level optimizations. In some cases, LLVM is much slower than NanoJIT.

Type inference enables us to approach fully typed code performance without all the manual type annotations. We see significant performance increases with type inference on the `SS\bitops-bit` test cases because we narrow down the types of each variable into a numeric type, but not necessarily the integer type. However, coercing between a numeric type and integer type still outperforms coercing between the any type and integer type. The same logic applies for the `V8\crypto` benchmark. Since the benchmark is computation intensive, we determine that some variables are numeric types, which results in a visible performance gain.

Other significant gains occur in the SunSpider `s3d-morph`, `s3d-cube`, and `access-fannkuch` benchmarks due to type inference in two critical areas. First, we determine that a few loop variables are the integer type. Second, we determine that some declared variables have the array type. Once we infer that a declared variable is an array, and that an element in the array is being accessed by an integer, we optimize untyped code to produce array access get and set code rather than a generic get and set property code, achieving significant performance improvements.

`V8\richards` and `V8\deltablue` are the only benchmarks that demonstrate performance gains because of method inlining and type inference. The combination of both method inlining and type inference enable up to 20% performance gains, demonstrating that type inference through an inlined method produces some benefits.

We also see some neutral or negative results with partially typed code. The `SS\string` benchmarks are constrained by the VM implementation of strings. Most of the time spent in these benchmarks stress string library code, not JIT compiled code. Therefore,

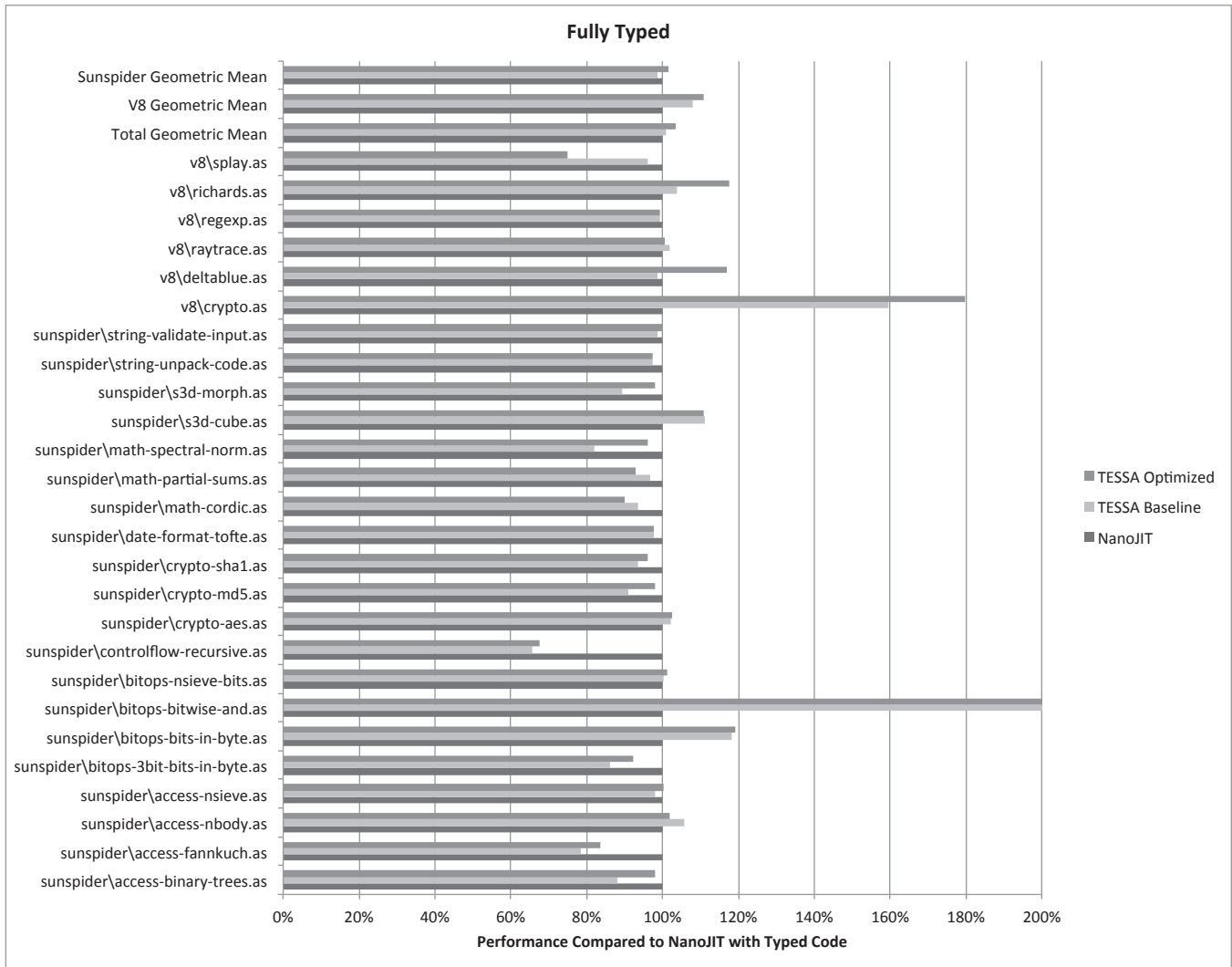


Figure 4. With Fully typed code, both NanoJIT and TESSA without optimizations have equivalent geometric mean performance, although the performance varies greatly between the benchmarks.

the performance of such benchmarks are not only equal across all configurations, but also close to typed code. `SS\access-nbody` is dominated by property look up code. Since we cannot early bind the types of objects, and therefore resolve properties, the runtime property look up overhead is the bottleneck, not JIT compiled code.

We actually lose performance with type inference in `SS\math-cordic` due to an increase in type information. In a hot loop, we determine that one variable is a floating point numeric type. Without type inference, the variable remains the `Any` type. However, the code within the loop changes from a comparison between two `Any` types, to a `Any` type and a `Numeric` type. This change forces us to box the numeric type into the `Any` type and perform the comparison with two `Any` types in every loop iteration. The resulting operation is more expensive than actually leaving one of the variables untyped. We need to develop some heuristics to detect and eliminate these performance reducing cases. Overall, we see that LLVM’s low-level optimizations produce JIT compiled code that is comparable to NanoJIT. We are able to achieve significant performance improvements with only type inference and method inlining.

4.1.3 Untyped Code Performance

Figure 6 demonstrates the performance results for untyped code. Using the geometric mean, NanoJIT is 61%, TESSA 62%, and TESSA with all optimizations 53% slower than NanoJIT on typed code. Overall, the same theme applies: LLVM has equal or lower performance than NanoJIT. We also see that the general performance gains and losses occur on the same benchmarks as partially typed code.

TESSA performs well with type inference on the `SS\bitops` benchmarks, as well as the `SS\s3d` benchmarks for the same reasons as partially typed code. With `bitops`, we are able to determine the type of key variables in the loops. With the `SS\s3d` and `SS\access-fannkuch` benchmarks, we determine the types of key local variables are integers and arrays, optimizing array access. Finally, TESSA with type inference gains significantly on `crypto-aes` because the benchmark accesses many arrays in loops. We are able to prove that a variable is an array, and that the index being accessed is an integer, resulting in the performance increase. We also see that we lose performance with type inference on `SS\access-nsieve`.

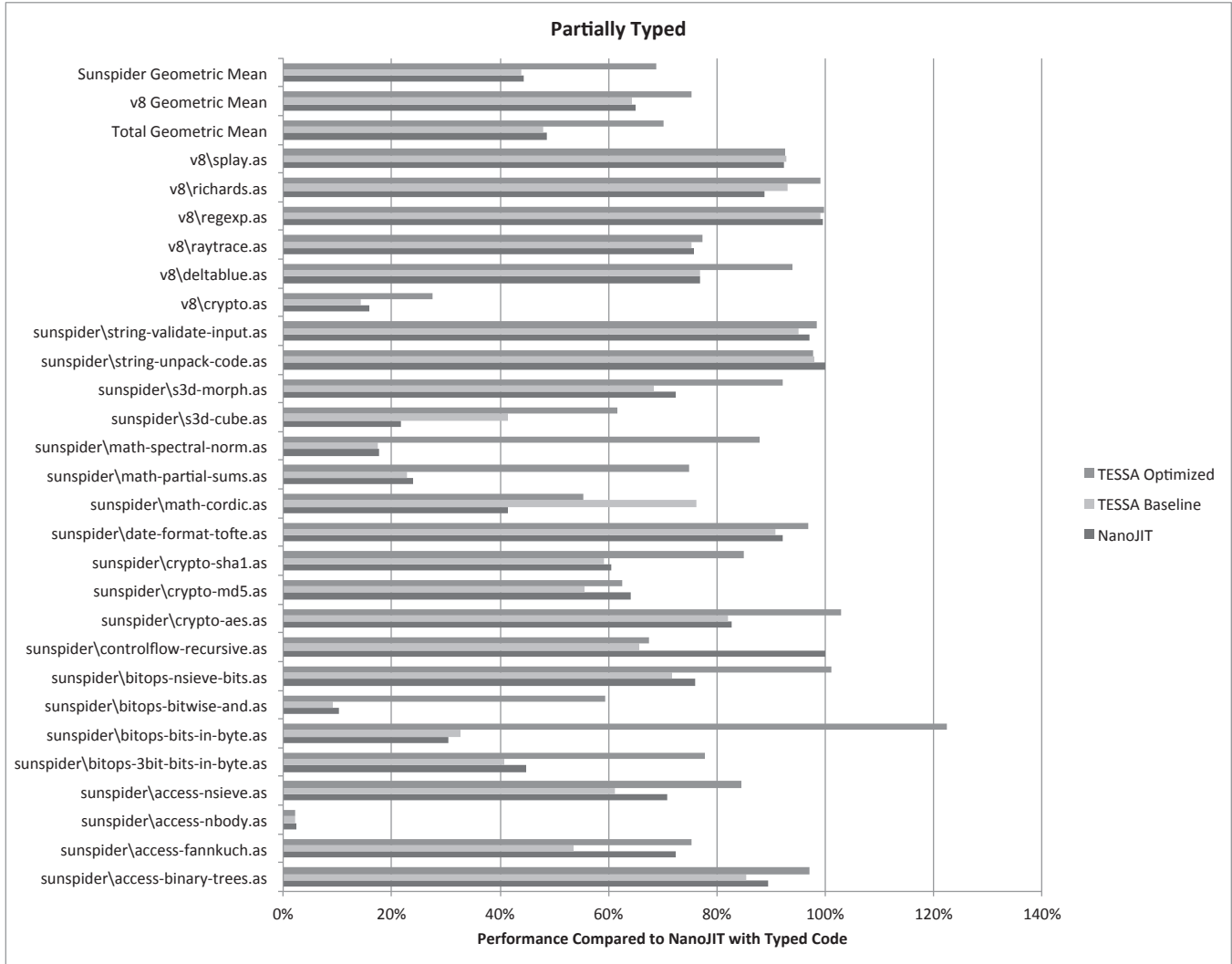


Figure 5. Partially typed code shows a performance degradation compared to fully typed code. However, type inference makes significant performance improvements.

We believe this is due to a bug in our type inference algorithm and will investigate this further.

However, we no longer have any performance gains on any of the V8 benchmark for two reasons. First, we cannot inline any methods as each method is dynamically bound by name at each call site. Because we cannot statically bind any method, inlining is not possible. Second, most variables are either global variables or properties on objects and exist in the heap. We do not currently propagate types through the heap and limit our algorithm to local variables in a method.

The untyped code performance illustrates the key point. Low-level optimizations do not produce significant performance advantages and type speculation is required to have untyped code performance approach typed performance. In simple cases, we are able to achieve 80% of typed performance with only type inference, but any complex program has significant performance degradation without a VM that supports type speculation.

4.2 LLVM Optimization Levels

LLVM has four different optimization levels that enable certain optimizations. The four levels roughly correlate to GCC's -O (Opti-

mization) flag. LLVM's four flags are: NONE (~ -O0), LESS (~ -O1), DEFAULT (~ -O2), and AGGRESSIVE (~ -O3). NONE optimizations create excessive loads and stores, with the only optimization being some amount of intelligent register allocation. The LESS setting performs optimizations that do not require any speed or space trade offs. The DEFAULT setting performs more aggressive instruction scheduling and a few other optimizations, but still do not require any speed or space trade offs. AGGRESSIVE turns on expensive optimizations and trades space for time.

The results for each benchmark with each LLVM optimization level are shown in Figure 7. Each data point is compared to LLVM's NONE optimization level. As expected, we see the overall JIT compiled performance increases as we enable more LLVM optimizations. However, the performance increases of low-level optimizations quickly taper off after the LESS level. Because we need to balance compilation time with runtime, the LESS optimization level was appropriate in our evaluation.

4.3 Startup Time

We define startup time as the amount of time required to initialize LLVM's subsystems. We measure LLVM's initialization time

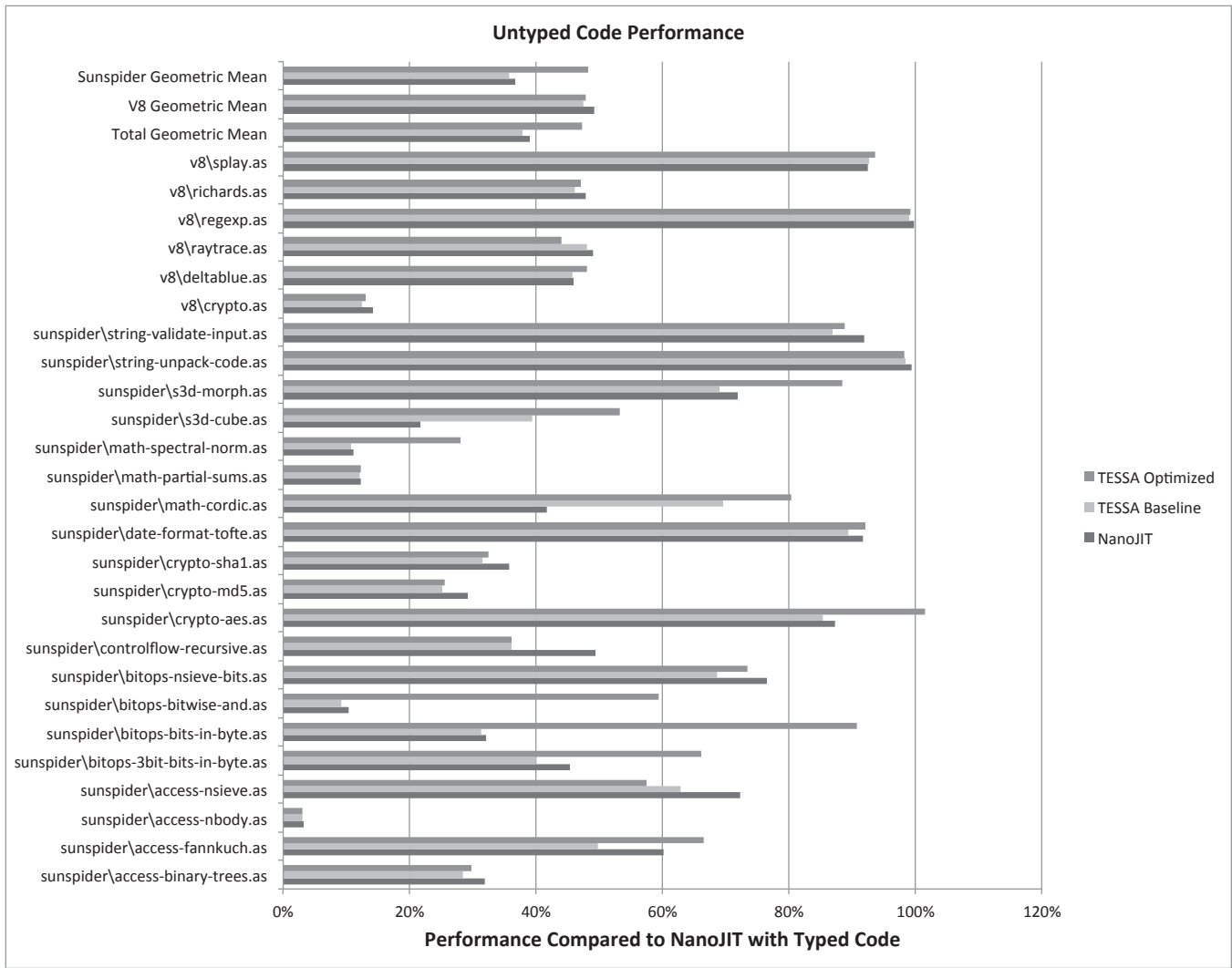


Figure 6. Untyped code performance. The same theme applies as in partially typed and fully typed code. Low-level optimizations provide little performance improvement.

with the Windows `QueryPerformanceCounter` API and take the average of ten measurements. LLVM’s back end takes on average of 514 microseconds to startup and is negligible compared to our compilation and execution times.

4.4 Compilation Time

We must examine how much compilation time our architecture requires compared to NanoJIT. Compilation time is defined as the total amount of time to translate ABC into TESSA, then TESSA to LLVM IR, and LLVM IR assembled to machine code. This definition of compilation time also includes the time LLVM requires to optimize LLVM IR and assemble machine code.

We use the WIN32 API `QueryPerformanceCounter` to retrieve the value of a high resolution performance counter. We then measure the compilation time ten times and calculate the mean. Startup time is disregarded as a factor in our compilation time measurements because NanoJIT and LLVM have equal startup times.

Figure 8 shows the compilation time results of our TESSA architecture versus NanoJIT. The left bar illustrates the compilation time of a basic TESSA configuration and LLVM’s back end. TESSA includes time creating TESSA IR from ABC and lower-

ing TESSA IR into LLVM IR. LLVM compilation time consists of optimizing LLVM IR and generating machine code. TESSA optimized and LLVM optimized measures both TESSA optimization time and LLVM’s back end compilation time.

We see that our whole architecture has a significant impact on compilation time, being up to 52X slower than NanoJIT. Some of this is expected due to the fact that both TESSA and LLVM are heavily object oriented programs while NanoJIT has been tuned to compile fast. We also see that up to 80% of the total compilation time is spent in LLVM and not in TESSA. In cases such as richards, LLVM spends more time compiling with TESSA optimized because more TESSA IR is created due to method inlining. However when looking at the absolute time, the compilation time with TESSA is still acceptable because NanoJIT compiles extremely fast. The fastest benchmark, richards, requires 0.035 seconds of execution time on NanoJIT and 0.111 seconds with LLVM.

While RegExp has the longest compilation time, the benchmark stresses VM regular expression library code and spends little execution time in JIT compiled code. Therefore, presenting an absolute time for the RegExp benchmark makes little sense as execution time dwarfs compilation time. The second longest compiling

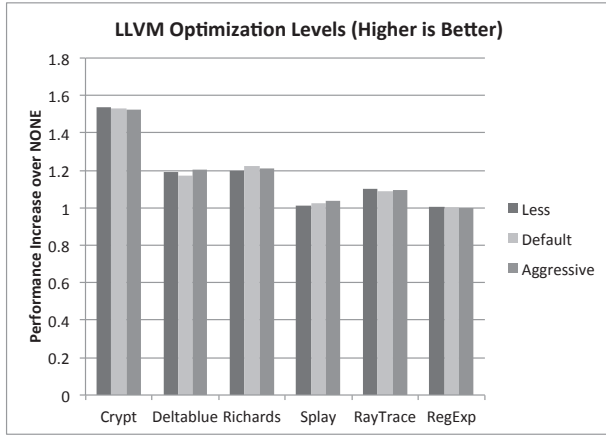


Figure 7. LLVM’s low-level optimizations increase performance at the LESS level, then quickly taper off providing little benefit.

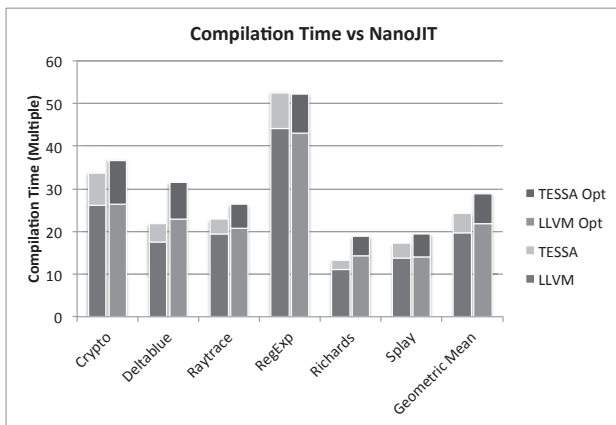


Figure 8. Across all benchmarks, LLVM spends significantly more time during the compilation phase than NanoJIT.

benchmark, Crypto, needs 0.078 seconds of execution time with NanoJIT and 0.577 seconds with LLVM.

4.5 Discussion

The data forces us to conclude that Tamarin is reaching a performance wall with fully typed JIT compiled code because both high-level and low-level optimizations do not greatly improve performance. Other parts of the virtual machine must be improved until JIT compiled code becomes the limiting factor for performance. Fully typed code also demonstrates that the low-level optimizations LLVM provides rarely accelerate the performance of JIT compiled code enough to warrant the compilation time required to perform the optimizations. Improving performance requires high-level optimizations. The data shows that type inference provides the biggest impact, followed by method inlining, then dead code elimination.

Untyped code shows dramatic performance decreases with both JIT compilers, with the most spectacular result being $SS_{\text{spectral-norm}}$, being more than 90% slower than the same benchmark on typed code. Overall, both NanoJIT and TESSA with only low-level optimizations are half as fast as typed code. High-level optimizations help mitigate the problem. However, performance still degrades considerably due to the lack of type information.

Garret et al. [27] demonstrates that most programs contain variables that do not change their type, or are type stable. A well known compilation strategy is to speculate the type of each variable based on runtime feedback, compiling the method as typed code rather than untyped code. In our case, speculating on untyped code with the information provided by runtime feedback doubles the speed of Flash. In select cases, such as untyped bitops-bitwise-and, speculation improves performance by over 10X. Untyped code requires speculative compilation to increase performance.

Partially typed code fills a niche in a compilation scheme. Fully typed code still outperforms both untyped and partially typed code by a wide margin. Untyped code and dynamically typed languages are popular for their flexibility, quickly enabling developers to prototype new features for their programs. However, developers must face significant performance degradation until they are willing to add type annotations to current code. Optionally typed code is a flexible middle way that shows considerable performance improvements over untyped code, allowing developers to add type annotations over time. In many cases, partially typed code with type inference delivers performance comparable to fully typed code.

In the worst case scenario, speculative untyped code has too many assumptions, greatly reducing any performance improvement because of time spent recovering from false assumptions. Partially typed code fits in the compilation scheme as a medium performance option in cases where a method compiled with type inference is fast enough. Another possibility is to use stable runtime type information, or variables that do not change their type in speculated code, and automatically achieve partially typed performance without the penalty of incorrect speculations.

Overall, the data shows which optimizations are beneficial for ActionScript and other dynamic languages. We have to implement speculative code execution for untyped code, and in the worst case, default to partially typed code with type inference to reduce the penalty of false speculation. Agesen et al. [17] has previously demonstrated that type inference and runtime type feedback are complementary techniques. In addition, NanoJIT performs most of the optimizations required. NanoJIT needs a better register allocator and a type inference optimization pass, which will be implemented in future work.

5. Related Work

The JavaScript VMs shipping in current web browsers are closely related to Tamarin. Mozilla’s TraceMonkey [26] uses a novel trace compilation technique that finds and type specializes “hot” code paths. Apple’s Nitro [11] contains a highly optimized JIT and uses with great success previous research on type specializing code. Google’s V8 [14] mimics a static language based on run-time type information and applies static code compilation techniques to create a fast JIT compiler. All three VMs tune performance based on runtime profiling information. In addition, there are proposals to add optional type annotations to JavaScript [7], thereby improving JIT compiled code performance in those virtual machines.

ActionScript, like many other languages, contain optional type annotations to reduce the dynamic nature of the language. There are two sides of the spectrum: adding dynamic capabilities to statically typed languages or adding type information to dynamic languages. C#, a statically typed language, introduces the dynamic keyword [2] that enables variables to bypass static type checking. Scala [35] is statically typed language without explicit type annotations and infers types based on their usage. Abadi et al. [16] explores the implication of adding the dynamic keyword to a statically typed language.

The other method to approach the problem is to improve the type system of the language itself either by creating a new type system or adding manual annotations. Both Thiemann [43] and

Anderson et al. [19] propose a type system for a subset of the JavaScript language. Anderson also contributes a type inference algorithm for their subset of JavaScript. Typed Scheme [45] is a dialect of PLT scheme that requires programmers to type annotate scheme programs. ActionScript approaches the problem by adding static type annotations to a dynamic language.

Recent research has tried to create a combined type system that incorporates both dynamic and static type systems. Siek et al. proposes gradual types [38][39][40], which provides many of the benefits of static typing for the portions of the program that are typed. Untyped portions of the program must contain runtime type checks. When untyped code interacts with typed code, the untyped code is implicitly coerced into the declared type when safe. Tobin-Hochstadt et al. [44] introduces the notion of *blame*, where untyped code is blamed for program errors or blamed for performance degradation. We currently implement implicit type coercions for primitive ActionScript types only. Thatte proposes quasi static typing [41]. Bracha [20] proposes a pluggable type system, where multiple different type systems coexist. Previous work provides the theory and foundational principles behind our work. We contribute a performance evaluation of the same benchmark across typed, untyped, and partially typed code, demonstrating the viability and performance implications of a language with a hybrid type system.

Type inference has previously been used with great success in a variety of languages. Hindley-Milner [24][34] proposes a type inference algorithm, which lays the groundwork for many other type inference algorithms. Thatte introduces type inference with partial types [42], Gronski et al. [28] proposes hybrid checking, and Palsberg et al. [36] for object oriented languages. Click et al. [22], proposes an optimistic constant propagation algorithm that starts optimistically and recovers from false assumptions. Our type inference algorithm is greatly influenced by this work, as we also start type inference optimistically rather than pessimistically. How we finally evaluate and solve the type equations for each instruction are guided by these previous type inference algorithms.

Using runtime feedback and speculation to improve the performance of JIT compiled code, such as type specializing untyped code, significantly improves the performance of dynamically typed languages. Chambers et al. [21] introduces the idea of customization in SELF, where each method is compiled specifically based on the type signature of each method. Hölzle et al. proposes both polymorphic inline caches [29] and runtime feedback [30] to improve the performance of SELF code. Both techniques are speculative optimizations that improve untyped code performance. Our evaluation enables us to measure the upper bound of performance with speculatively typed code. Agesen et al. [17] also compares the performance of using type inference versus runtime type feedback, which guides our compilation heuristics when evaluating the performance of partially typed code.

Previous research demonstrates the effect of compiler optimizations in different contexts. Lee et al. [33] shows that few compiler optimizations increase performance in Java, a statically typed language. Unladen Swallow [12] is a Python virtual machine which uses LLVM as a backend JIT compiler replacement. In late 2010, Kleckner [13] demonstrated many of the same issues in Unladen Swallow. Dynamically typed languages require high-level knowledge of the program to increase JIT compiled code performance.

6. Conclusions and Future Work

We present a comprehensive performance evaluation of two different JIT compilers across three different type configurations with the same benchmarks. The data supports our conclusion that aggressive low-level optimizations do not significantly increase the performance of JIT compiled code. As we see with LLVM's op-

timization levels, and a comparison with NanoJIT, it is sufficient that a JIT compiler performs only a few select low-level optimizations. High impact performance gains only occur due to high-level optimizations such as method inlining and type inference.

Our future work looks to combine partially typed code with type inference as one compilation technique in our overall compilation strategy. We will use partial typing with type inference in combination with type speculation and runtime feedback to greatly improve the performance of untyped code. We will take our type inference algorithm and implement it in NanoJIT and continue to refine the algorithm to infer subtypes. We believe both type inference and runtime feedback are complimentary and will incorporate both techniques into the Tamarin VM.

Acknowledgments

We want to thank the anonymous reviewers who have given us helpful feedback to improve this work. We also express our gratitude to the LLVM contributors for their detailed and helpful documentation that made this work possible. Parts of this effort have been supported by generous gifts from Adobe, Mozilla, and by the National Science Foundation (NSF) under grant CNS-0905684. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

- [1] ActionScript 3.0 Overview — Adobe Developer Connection - http://www.adobe.com/devnet/actionscript/articles/actionscript3_overview.html.
- [2] dynamic (C# Reference) - <http://msdn.microsoft.com/en-us/library/dd264741.aspx#Y669>.
- [3] LIR - MDC - <https://developer.mozilla.org/en/Nanojit/LIR>.
- [4] LLVM Assembly Language Reference Manual - <http://llvm.org/docs/LangRef.html>.
- [5] NanoJIT - MDC - <https://developer.mozilla.org/En/Nanojit>.
- [6] Open Source Framework, Web Application Software Development — Flex - Adobe - <http://www.adobe.com/products/flex/>.
- [7] Proposed ECMAScript 4 Edition - Language Overview - <http://www.ecmascript.org/es4/spec/overview.pdf>.
- [8] Standard ECMA-262. 3rd Edition - December 1999. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-362.pdf>.
- [9] Sunspider JavaScript Benchmark - <http://www.webkit.org/perf/sunspider/sunspider.html>.
- [10] Tamarin - MDC - <https://developer.mozilla.org/en/Tamarin>.
- [11] The WebKit Open Source Project - <http://webkit.org/>.
- [12] unladen-swallow - Project Hosting on Google Code - <http://code.google.com/p/unladen-swallow/>.
- [13] Unladen Swallow Retrospective - QINSB is not a Software Blog - <http://qinsb.blogspot.com/2011/03/unladen-swallow-retrospective.html>.
- [14] V8 - Project Hosting on Google Code - <http://code.google.com/p/v8/>.
- [15] V8 Benchmark Suite - <http://v8.googlecode.com/svn/data/benchmarks/current/run.html>.
- [16] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.
- [17] O. Agesen and U. Hölzle. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Lan-

- guages. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–107. ACM, 1995.
- [18] B. Alpern, C. Attanasio, J. Barton, M. Burke, P. Cheng, J. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, et al. The Jalapeno Virtual Machine. *IBM Systems Journal*, 2000.
- [19] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 428–452. Springer, 2005.
- [20] G. Bracha. Pluggable Type Systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.
- [21] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the Symposium on Interpreters and Interpretive Techniques*, pages 146–160. ACM, 1989.
- [22] C. Click and K. Cooper. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, 1995.
- [23] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
- [24] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.
- [25] A. Gal, M. Bebenita, M. Chang, and M. Franz. Making the Compilation “Pipeline” Explicit: Dynamic Compilation Using Trace Tree Serialization. Technical Report CS-TR-07-12, 2008.
- [26] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghghat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-Based Just-In-Time Type Specialization for Dynamic Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 465–478. ACM, 2009.
- [27] C. Garrett, J. Dean, D. Grove, and C. Chambers. Measurement and Application of Dynamic Receiver Class Distributions. Technical Report CSE-TR-94-03-05, University of Washington, 1994.
- [28] J. Gronski, K. Knowles, A. Tomb, S. Freund, and C. Flanagan. Sage: Hybrid Checking for Flexible Specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.
- [29] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.
- [30] U. Hölzle and D. Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336. ACM, 1994.
- [31] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):1–32, 2008.
- [32] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–. Published by the IEEE Computer Society, 2004.
- [33] H. Lee, D. von Dincklage, A. Diwan, and J. Moss. Understanding the Behavior of Compiler Optimizations. *Software: Practice and Experience*, 36(8):835–844, 2006.
- [34] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
- [35] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, 2004.
- [36] J. Palsberg and M. Schwartzbach. Object-Oriented Type Inference. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, 1991.
- [37] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.
- [38] J. Siek and W. Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming*, pages 81–92, 2006.
- [39] J. Siek and W. Taha. Gradual Typing for Objects. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007.
- [40] J. Siek and M. Vachharajani. Gradual Typing with Unification-Based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*, pages 7:1–7:12. ACM, 2008.
- [41] S. Thatte. Quasi-Static Typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1989.
- [42] S. Thatte. Type Inference with Partial Types. In *Theoretical Computer Science*, pages 615–629. Elsevier, 1994.
- [43] P. Thiemann. Towards a Type System for Analyzing Javascript Programs. In *Programming Languages and Systems*, pages 408–422. Springer, 2005.
- [44] S. Tobin-Hochstadt and M. Felleisen. Interlanguage Migration: From Scripts to Programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 964–974. ACM, 2006.
- [45] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 395–406. ACM, 2008.