Duplo: A Framework for OCaml Post-Link Optimisation

NANDOR LICKER, University of Cambridge, United Kingdom
TIMOTHY M. JONES, University of Cambridge, United Kingdom

We present a novel framework, *Duplo*, for the low-level post-link optimisation of OCaml programs, achieving a speedup of 7% and a reduction of at least 15% of the code size of widely-used OCaml applications. Unlike existing post-link optimisers, which typically operate on target-specific machine code, our framework operates on a Low-Level Intermediate Representation (LLIR) capable of representing both the OCaml programs and any C dependencies they invoke through the foreign-function interface (FFI). LLIR is analysed, transformed and lowered to machine code by our post-link optimiser, LLIR-OPT. Most importantly, LLIR allows the optimiser to cross the OCaml-C language boundary, mitigating the overhead incurred by the FFI and enabling analyses and transformations in a previously unavailable context. The optimised IR is then lowered to `amd64` machine code through the existing target-specific code generator of LLVM, modified to handle garbage collection just as effectively as the native OCaml backend. We equip our optimiser with a suite of SSA-based transformations and points-to analyses capable of capturing the semantics and representing the memory models of both languages, along with a cross-language inliner to embed C methods into OCaml callers. We evaluate the gains of our framework, which can be attributed to both our optimiser and the more sophisticated `amd64` backend of LLVM, on a wide-range of widely-used OCaml applications, as well as an existing suite of micro- and macro-benchmarks used to track the performance of the OCaml compiler.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Imperative languages*; *Functional languages*.

Additional Key Words and Phrases: OCaml, C, post-link, optimisation, inlining, LLVM

ACM Reference Format:
Nandor Licker and Timothy M. Jones. 2020. Duplo: A Framework for OCaml Post-Link Optimisation. *Proc. ACM Program. Lang.* 4, ICFP, Article 98 (August 2020), 29 pages. https://doi.org/10.1145/3408980

1 INTRODUCTION

OCaml and C are two seemingly unrelated programming languages, yet all OCaml programs contain a significant number of C methods. These implement vital features, including the runtime system and performance-sensitive operations, invoked through OCaml's foreign-function interface (FFI), which cannot be expressed effectively in OCaml. While most of the functionality implemented in C is transparent to OCaml users, such functions are prevalent and vital to performance, raising interest in optimising them. Unfortunately, the compilation model of both languages and the FFI mechanism raises a difficult barrier: intertwined functions originating from C and OCaml only meet in the linker, in the form of machine code, which is well known to be difficult to optimise.

Typically, this issue would be addressed by link-time, post-link or dynamic runtime optimisers operating on machine code. Existing optimisers, such as DynamoRIO [Bruening et al. 2003], BOLT [Panchenko et al. 2019] or PLTO [Schwarz et al. 2001], are aimed towards binaries originating from C and C++ for good reason: modifying hardware instructions while preserving the semantics

Authors' addresses: Nandor Licker, nl364@cl.cam.ac.uk, University of Cambridge, United Kingdom; Timothy M. Jones, tmj32@cl.cam.ac.uk, University of Cambridge, United Kingdom.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/8-ART98
https://doi.org/10.1145/3408980

imposed by the garbage collector (GC) and adjusting the required metadata is a non-trivial challenge. This motivated us to address the problem at a higher level, prior to assembly-code generation, yet still on a representation containing whole-program information, which we call LLIR. Since at such a stage an imperative representation is more suitable, we tackle the challenge of optimising functional programs expressed in imperative form through our framework, Duplo.

1.1 Motivation

To highlight the complexity inherent in using the FFI, consider the problem of creating a data structure to represent complex numbers in OCaml with a pair of 32-bit floats. Where a loss in precision is tolerable, using the single-precision floating-point representation could lead to a significant reduction in memory usage, yet on a 64-bit platform OCaml only supports the double-precision variant. In a low-level systems language, such as C or Rust, packing a pair of floats into a 64-bit structure is a trivial task. However, the OCaml heap and data layout is not flexible enough to allow for a straightforward and effective implementation. The following definition, similar to that provided by the OCaml standard library, is brief, but inefficient;

```
type Complex = float * float
let make_complex r i = (r, i)
let real (r, _) = r
let imag (_, i) = i
```

Such an implementation of complex numbers requires a pair of pointers to boxed doubles, each occupying a header and a word, for a total of seven words, or 56 bytes, spread across three heap blocks created using up to three invocations to the allocator in the worst case. This is neither compact, nor fast: accessing an element requires chasing two pointers. A step towards a more efficient, albeit less readable, solution would involve the use of an array of two real elements:

```
type Complex = float array
let make_complex r i =
  let arr = Array.make 0 in
  Array.set arr 0 r;
  Array.set arr 1 i;
  arr
let real arr = Array.get arr 0
let imag arr = Array.get arr 1
```

This approach is more compact, at three words for a total of 24 bytes, yet it is not optimal due to the unavoidable use of double-precision floats. The implementation relies on the polymorphic `caml_make_vect` runtime method, which decides the kind of array to allocate (array of values or a specialised array of unboxed double-precision floats) based on the filler element provided. Given that in this use case only two elements are allocated at a time, even simple array allocation results in significant overhead. Furthermore, performance comes at the cost of increased complexity: specialised arrays demand intrusive changes to the compiler. Extending this mechanism to support fixed-point reals or other floating point representations, such as the recently popular `bfloat16` [Tagliavini et al. 2018], would further burden the compiler. A solution, optimal from the perspective of memory usage, that does not require modifications to the compiler has to rely on C and the FFI:

```
typedef struct { float i; float r; } complex_t;
#define Complex_val(block) ((complex_t *)block)
CAMLprim value complex_make(value i, value r) {
  CAMLparam2(i, r);
  value block = caml_alloc_small(2, 255);
  Complex_val(block)->i = Double_val(i);
  Complex_val(block)->r = Double_val(r);
```

```
  CAMLreturn(block);
}
CAMLprim value complex_imag(value c) {
  CAMLparam1(c);
  CAMLreturn(caml_copy_double(Complex_val(c)->i));
}
```

While this approach is succinct and compact, it is far from optimal in terms of performance. All external calls allocate on the heap, requiring the compiler to emit expensive trampolines that handle the context switching required by the garbage collector. Given the language barrier, the OCaml compiler cannot inline any of the trivial field accessors, which will often redundantly box and unbox floats. To illustrate the extent of the problem, we consider a typical short method that operates on this data type and can be expressed purely in OCaml:

```
let magnitude_squared c =
  let i = imag c in
  let r = real c in
  i * i + r * r
```

The execution of this method invokes two trampolines before reaching the implementation of the accessors, which box the flat fields into the doubles required by OCaml. The blocks resulting from these allocations are immediately unboxed and discarded by the OCaml computation, which proceeds by computing the magnitude and boxing it through another heap allocation. Such a brief method, which would lower to two loads and a few arithmetic instructions if implemented purely in C, requires a significant number of heap allocations and a corresponding number of loads and stores to memory, incurring an unacceptable overhead that would prohibit the use of this numerical datatype in any long-running computation.

The aim of our post-link optimisation framework, Duplo, is to provide a platform for analyses and transformations crossing the language boundary, ameliorating or eliminating the overhead imposed by the redundant, but unavoidable allocations. As a starting point, we enable inlining C methods into OCaml callers across the trampolines, while preserving all the information required for the proper functioning of the garbage collector. Besides removing a potentially expensive indirect call, the true value of inlining lies in the fact that it opens up opportunities for improvement.

A number of transformations need to follow before the allocations are eliminated. The type-agnostic representation of OCaml natively supports only pointers and 63-bit integers. Other primitive types, such as floats, are boxed into heap-allocated blocks, out of reach for simple constant propagation. In this context, propagating primitives involves forwarding writes to memory locations (boxing) to loads using them (unboxing): we achieve this through the use of a local points-to analysis. Replacing loads with values does not yet leave the blocks unused: in some cases, such as the one illustrated in Figure 9, the heap pointer is registered with a chain of local roots to allow the garbage collector to keep track of roots in C code. While not yet implemented, we discuss the analyses required by future transformations to eliminate allocations that are never used.

Improvements achievable by the framework we propose are not only limited to the efficient implementation of specific numeric types and arithmetic operations: zero-cost FFIs are crucial for a language to effectively interface with widely-used and well-tested libraries, such as `zlib`, `gmp` or `libpng`, as well as core POSIX methods. Foreign methods presently need to unbox non-integer values they receive from OCaml, boxing any returned structures, pairing them with methods to hash and finalize them. The frequent use of such features suggests that eliminating this overhead is valuable, yet existing toolchains cannot perform the required transformations.

While we show significant improvement on existing applications, we believe our optimiser has the potential to encourage the use of language features and constructs that are underused due to

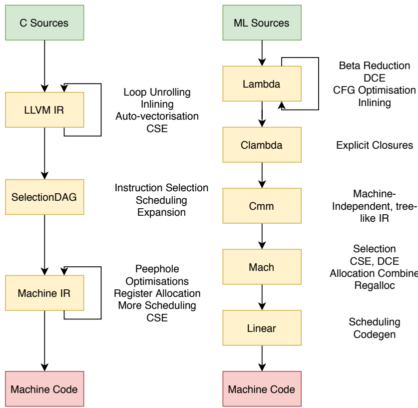


Fig. 1. Intermediate forms in C and OCaml

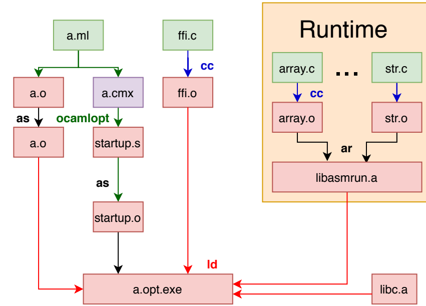


Fig. 2. Compilation of an OCaml executable

their high cost. A functional language such as OCaml undoubtedly has its benefits, however there are constructs which limit the utility of the language, mainly due to the rigidity of the memory model. Even though invoking C through the FFI is trivial and such a low-level language is well suited towards providing these missing features, the overhead is often unacceptable, warranting the specialisation of such data types inside the compiler, as is the case with `floatarray`. Through our cross-language post-link optimiser, we aim to reduce the need for such extensions to the compiler, instead encouraging the use of the FFI, which now should incur less of an overhead.

1.2 LLIR

Our platform is built around LLIR, the common representation for both C and OCaml. The challenge in defining such a representation lies in the fact that OCaml and C take distinct routes while lowering their sources to machine code, using intermediate forms that share little to no similarities.

Figure 1 illustrates the intermediate forms used in the compilation of both C and OCaml: `clang` lowers C to LLVM IR, applying high-level optimisations such as inlining and loop unrolling. The IR is then lowered to *Machine IR* by the instruction selector and scheduler through a *SelectionDAG*, enabling a series of low-level target-specific peephole transformations, among other target-agnostic passes running on machine code. On the other hand, OCaml compilation involves a significant number of intermediate steps that progressively lower the high-level language to machine code. The type-checked AST is lowered to `Lambda`, a high-level IR that supports closures and several optimisations such as inlining and dead-code elimination. `Clambda` follows, exposing the construction of closure environments and adding them as an explicit argument to closures. This IR is then lowered to the tree-like imperative and target-independent `Cmm` representation: at this stage, the preserved type information only indicates whether a value is an integer, a pointer into the OCaml heap or a naked address outside it. Before emitting machine code, instruction selection, scheduling and allocation rely on two more representations, `Mach` and `Linear`, which explicitly reference hardware registers. The former is tree-like, while the latter consists of a linear sequence of abstract instructions that have direct assembly equivalents. When enabled, the `flambda` representation is built between `Lambda` and `Clambda`, enabling more aggressive inlining across module boundaries.

While this choice of intermediate representations might be optimal for individual compilers, it poses a barrier towards interoperability. Figure 2 illustrates the steps involved in building application developed in both languages that relies on the FFI: the compiled sources originating from different compilers meet at the latest stage of the pipeline, when the object files containing machine code built from them are combined into an executable by the linker. At this stage, on this low-level

representation, only trivial transformations can be applied to eliminate the overhead incurred by the abstraction of modularity: linkers treat functions as strings of bytes, without the possibility of inlining or unreachable-code elimination. Traditional Link-Time Optimisations (LTO), which rely on compilers passing the IRs to the linker alongside the object files, cannot be used since an efficient representation that can be generated from both languages does not yet exist.

The intermediate steps used in the compilation of OCaml are tailored for a high-level functional language, while LLVM is aimed for optimising imperative languages without a garbage collector. While technically possible, compiling LLVM IR to any of OCaml's representations from `Cmm` onwards would be counterproductive: `Cmm` has a limited set of data types and its lowering involves only two optimisations, dead-code elimination and common-subexpression elimination, along with a fairly basic instruction selector. On the other hand, LLVM is a production-grade compiler with an effective instruction selector and an extensive battery of target-specific optimisation passes, which we intend to exploit by lowering one of the OCaml IRs to a form that can be plugged into the LLVM compilation pipeline. Thus, the search for a common point for further transformation starts at the `Clambda` level, which is the first to turn closures into functions that could be represented in an imperative form: at this stage, free variables are explicitly loaded from an environment, allowing all functions to be hoisted to the global scope. Yet again, it is technically possible to lower `Clambda` to LLVM IR, with some caveats: LLVM's handling of exceptions and heap roots would incur a severe overhead, as confirmed by existing attempts and analyses [Leroy 2009; Scherer 2015]. Unfortunately, GC support in LLVM requires roots to be located on a shadow stack, whereas OCaml allows registers to be used across call sites, an optimisation crucial for performance [Lattner 2020a].

To address these shortcomings, we introduce a new intermediate representation, LLIR, described in detail in Section 3.2. LLIR represents both OCaml and C efficiently, including exceptions and GC roots. LLIR sits between the LLVM IR and the *SelectionDAG* used to lower it to machine code: it can be generated from LLVM's IR, but it can also be lowered to *SelectionDAG* in order to generate machine code from it using LLVM's existing target-specific backends. From a high-level perspective, LLIR is almost machine code, except it provides explicit and compact call instructions, exception landing pads, GC annotations and operates on virtual registers. While this IR could be generated from `Clambda`, to avoid duplication we rely on information computed at the `Mach` level: this is the first stage where locations storing heap roots are clearly identified. We also opt to reuse OCaml's existing facilities and drop one level, generating LLIR using a custom backend from `Linear` form.

1.3 LLIR-OPT

While the long-term goal is to mitigate, and potentially eliminate, the overhead of foreign-function calls between OCaml and C through the FFI by exploiting optimisation opportunities opened by LLIR, this paper is focused on introducing a framework to enable such optimisations, called Duplo. Besides the post-link optimiser operating on LLIR, the framework also includes backends for LLVM (described in Section 3.3.1) and the OCaml compiler (described in Section 3.3.2) to generate LLIR.

A crucial task for the post-link optimiser is machine-code generation: since the optimiser fits into the OCaml toolchain right before executables are expected to be produced, the post-link optimiser emits object files, lowering LLIR to machine code. We chose LLVM to perform this lowering: even though the code generator is not explicitly exposed, nor intended for external use, we managed to build a mapping from LLIR to *SelectionDAG* and adapt transformations operating on the *Machine IR* generated from the DAG to respect the additional semantics imposed by the presence of GC roots. Given that the optimiser links to LLVM, we decided to reuse as many facilities as possible, shaping the design of the in-memory representation of LLIR. The similarities between the data structures representing LLIR and LLVM IR/*Machine IR* allow for the reuse of several analyses and transformations, including complex ones such as dominator-tree construction. The peculiarities

of this mapping, distinct from the challenges encountered by previous projects attempting to tie LLVM to OCaml, are described in Section 3.5. As shown by our evaluation in Section 5 and in contrast with previous attempts, we report a significant improvement in performance.

Presently, the post-link optimiser is equipped with a set of fairly generic transformations typically found in SSA-based optimisers, described in Section 4.4. While these passes exploit some opportunities opened by the cross-language context, their main goal is to shape LLIR into a form that can be optimally lowered by the code generator. Due to the presence of trampolines between OCaml call sites and C callees, inlining is not straightforward, leading to the discussion in Section 4.1. Opening up the possibility of inlining only solves part of the problem: given that LLIR represents functional programs, indirect calls and heap-allocated objects containing closure environments are prevalent, posing difficulties for further transformations. Points-to analyses are vital towards understanding such programs, hence why we present a global analysis in Section 4.2, which we use for unreachable-code elimination, along with a local analysis in Section 4.3, allowing the removal of some redundant loads and stores. Future work will invest heavily in improving such information.

1.4 Contributions

The contributions outlined in this paper are:

- A novel Low-Level Intermediate Representation (LLIR) capturing both OCaml and C semantics
- A post-link optimiser framework, Duplo, along with a complete toolchain to generate LLIR from OCaml and C and emit machine code from LLIR, enabling cross-language optimisation
- A local and a global points-to analysis, tailored for the C and OCaml memory models, presently used for dead-store elimination, write propagation and unreachable-code elimination
- A comprehensive evaluation of the post-link optimiser on highly representative benchmarks, including a comparison to an experimental OCaml LTO extension at the `flambda` level

Our implementation is available under the MIT license at https://github.com/nandor/llir-opt.

2 BACKGROUND AND RELATED WORK

The difficulty of cross-language optimisation varies depending on the design implementation of the languages involved. In order to exploit opportunities available across this boundary, all participating sources need to be lowered to a common representation capable of representing the semantics of all languages, which is then linked, analysed and optimised before emitting byte code or machine code. In the context of certain JIT compilers, this is trivial and free: the Java Virtual Machine (JVM) [Lindholm et al. 2014] or the Common Language Infrastructure (CLI) [ISO 23271:2012(E) 2012] optimise byte code merged from multiple source languages, such as Java, Scala or Groovy on the JVM and C# , F# or Visual Basic on the CLI. Similarly, compiled languages supported by the LLVM IR and infrastructure [Lattner 2008] also often benefit from Link-Time Optimisation (LTO) provided by LLVM to optimise across language boundaries, albeit with occasional difficulty: while all languages in an environment such as JVM or CLI share the same memory model and object representation, such homogeneity is not guaranteed on the LLVM IR, reducing the effectiveness of certain transformations. In our case of OCaml and C, which do not share a common representation other than machine code, optimising across the language boundary proves to be more difficult.

Most recently, cross-language optimisations were enabled with positive results between Rust [Matsakis and Klock 2014], compiled using the `rustc` compiler, and C/C++, compiled using `clang` [Lattner 2008]. Since `rustc` targets LLVM in order to generate machine code, crossing this boundary required only minor adjustments [Mozilla 2019]. The `rustc` frontend was equipped with a flag instructing LLVM IR to be emitted from individual compilation units, which were then linked into a module containing LLVM IR originating from both C++ and Rust. Such modules can be

optimised using the existing LTO linker plugin before machine-code generation. These changes to the Rust compiler allowed cross-language optimisations to be enabled in the Firefox browser, which already relied on LTO for C++ modules, achieving positive results. While the improvements were not explicitly quantified, cross-language optimisations reduced the overhead of method calls between the two languages, allowing future features to be implemented in Rust. The possibility of cross-language inlining also led to reduced code duplication, allowing several Rust methods wrapping or re-implementing functionality already provided by C++ to be removed.

Another interesting pair of languages that both use the LLVM IR as a common representation at a late stage in the compilation pipeline are Haskell and C [Terei and Chakravarty 2010]. Despite the fact that Haskell is a high-level, lazy and garbage-collected language, it can be fairly easily lowered to LLVM IR from its `Cmm` form. Unlike OCaml, the Glasgow Haskell Compiler (GHC) does not require any additional metadata to describe root pointers. Instead, all live blocks are reachable from a stack and a set of STG (one of the GHC IRs) registers [Jones 1992]. By adding an additional calling convention to pin STG registers to hardware registers, Haskell can be efficiently lowered to LLVM IR, resulting in improvements on certain workloads heavily reliant on numeric computation. Since context switches between Haskell and C are simpler than with OCaml and avoid the use of trampolines, the boundary is open to some optimisations provided by LLVM. Following the implementation of the LLVM backend, cross-language LTO was also experimentally enabled, but no quantifiable benefits were measured [Simmons 2019]. This could be attributed to the complexity of the constructs prevalent in `Cmm`, especially those implementing laziness, which cannot be readily optimised by existing LLVM transformations. Since our target is eagerly evaluated and relies on a simpler object representation, it is expected to be more amenable towards optimisation.

In the context of OCaml [Leroy et al. 2014], we are not aware of any work aiming to optimise across FFI calls, but attempts have been made to cross this boundary at a different level. The construction of type-safe wrappers over the FFI on the OCaml side is an active area of research [Yallop et al. 2016] and type systems were also developed in order to verify the correctness of C callees [Furr and Foster 2005]. Such prior work is encouraging as it shows that a common ground exists and the memory models and object representations of the two languages can be bridged by analyses. Similarly to Haskell, LTO between OCaml modules was also experimentally implemented, employing the use of the `flambda` optimiser to operate on modules containing whole programs [Chambart 2016]. Even though the LTO fork is no longer maintained, we fixed bugs in order to evaluate it and compare it to our cross-language post-link optimiser in Section 5. As previously mentioned, compiling OCaml to LLVM IR proved to be problematic in the past, justifying our decision to pursue an alternative approach towards finding a common representation for optimisations [Leroy 2009; Scherer 2015].

From the family of ML-like languages, OCaml is the optimal candidate for post-link optimisations across the FFI boundary. While other ML implementations, such as MLton [Weeks 2006], implement whole-program optimisations and type-directed compilation, they expose a very limited interface to C programs. This limitation is the caveat of type-based representations: while individual structures are encoded more efficiently in MLton, exposing them to C through a convenient interface is difficult. In contrast, the object representation of OCaml is type-agnostic, relying on tagged blocks that contain a fixed number of fields, either integers or pointers to other blocks. Such a simple implementation allows C to easily interface with OCaml objects, leading to a powerful FFI that can be further improved through cross-language optimisations.

OCaml and C already have a common representation, namely target-specific machine code, and tools exist to optimise at this level. Unfortunately, such tools rarely support multiple targets and limit optimisations to peephole or local transformations, performed either statically or at runtime. BOLT is an example of a static optimiser that transforms the binary before runtime [Panchenko et al. 2019] and DynamoRIO is a dynamic rewriter capable of transforming executables based on

valuable profile information gathered at runtime [Bruening et al. 2003]. Although these tools achieve speedups on C/C++ applications, they are not suited to OCaml since preserving the semantics of root pointers and adjusting the metadata required by the garbage collector at the binary level poses significant difficulty, especially in the context of a dynamic rewriter where the improvement in running time needs to also offset the cost of analyses. In contrast, the additional information required to correctly analyse and transform machine code originating from OCaml can be easily generated by compiler backends and embedded into our LLIR.

Given the requirement for OCaml to box primitives into heap-allocated wrappers, we identify the need for accurate points-to analyses in order to enable transformations. While such analyses have been studied for a long time [Andersen 1994], the need to balance accuracy with performance ensures this is still an active area of research. Based on context, points-to analyses try to account for various parameters, such as flow-, field- and context-sensitivity, along with tweaking algorithms for specific constructs [Johnson et al. 2017; Khedker et al. 2012; Lattner and Adve 2003; Pearce et al. 2007; Sui et al. 2018; Wilhelm et al. 2000]. While most analyses are fine-tuned for specific languages, algorithms were developed to operate at the assembly level, overcoming the lack of type information [Guo et al. 2005]. The analyses we provide operate on a low-level representation, adapting well-established methods for solving flow constraints [Hardekopf and Lin 2007a].

3 LLIR AND LLIR-OPT

Our method for cross-language optimisations is built around a novel Low-Level Intermediate Representation (LLIR) that is a common form suitable for representing both C and OCaml programs. As hinted by its name and the stage at which it is integrated into the compilation chain, LLIR is an imperative, low-level, assembly-like representation. An LLIR program consists of a number of data segments that define constants using syntax similar to typical assemblers, along with a text segment split into functions, basic blocks and sequences of instructions operating on virtual registers. Similarities with assembly are not coincidental: LLIR was designed to be easily generated through target-specific backends in LLVM (Section 3.3.1) and OCaml (Section 3.3.2), which are already equipped for emitting assembly.

The design of the representation is also influenced by the toolchains in which it integrates. Figure 3 illustrates the tools and intermediate files involved in the compilation of an OCaml executable with our post-link optimiser: in order to be seamlessly embedded into existing build systems and tools, LLIR files need to behave like target-specific object files. To achieve this, we provide a linker (`llir-ld`) and an archiving tool (`llir-ar`) to link executables and produce static libraries: they accept the same command-line arguments as the GNU binutils equivalents, except our linker operates on LLIR files and invokes the post-link optimiser, LLIR-OPT, to emit executable machine-code. To compile existing projects, we rely on cross-compilation and we treat LLIR as a distinct target configured identically to the machine for which assembly will be emitted, providing an additional LLIR implementation for any components implemented in assembly. This effort also aids portability: while the OCaml compiler requires ~700 lines of assembly for each supported platform, we provide the same amount of LLIR once and compile it to any target. Duplo-optimised binaries can also link to non-optimised libraries.

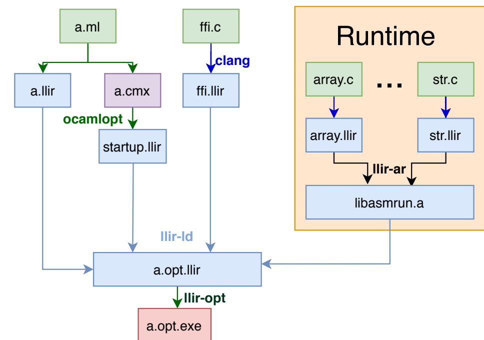


Fig. 3. Compilation using the Duplo framework

```
type point = { x: float; y: float }
let norm pt =
  sqrt (pt.x *. pt.x +. pt.y *. pt.y)

camlTest__norm:
  .args                0, i64
  .call                caml
  arg.i64              $0, 0 @caml_value
  mov.i64              $1, 8
  add.i64              $2, $0, $1
  ld.8.f64             $3, [$2]
  mul.f64              $4, $3, $3
  ld.8.f64             $5, [$0]
  mul.f64              $6, $5, $5
  add.f64              $7, $6, $4
  mov.i64              $8, sqrt
  call.f64.c           $9, $8, $7
  mov.i64              $10, caml_alloc1
  call.i64.caml_alloc  $11, $10 @caml_frame @caml_value
  mov.i64              $12, 0x4fd
  mov.i64              $13, -8
  add.i64              $14, $11, $13
  st.8                 [$14], $12
  st.8                 [$11], $9
  ret                  $11
```

```
typedef struct { float x; float y; } point_t;
float norm(point_t p) {
  return sqrt(p.x * p.x + p.y * p.y);
}
norm:
  .args     0, i64
  .call     c
  arg.i64   $0, 0
  ld.f32    $1, [0]
  mov.i64   $2, 4
  add.i64   $3, $0, $2
  ld.f32    $4, [$3]
  mul.f32   $5, $1, $1
  mul.f32   $6, $4, $4
  add.f32   $7, $5, $6
  mov.i64   $8, sqrt
  call.f32  $9, $8, $7
  ret.f32   $9
```

Fig. 4. LLIR representation of OCaml and C methods

Figure 4 shows LLIR representing the OCaml and C implementation of the same function. Functions are defined in the text segment using a simple label, followed by directives to specify their attributes, such as the types of arguments, the size and alignment of stack-allocated objects, the calling convention and inlining restrictions. While the two methods implement the same function, the OCaml version highlights the additional features required to represent a garbage-collected language: an allocation boxing a primitive is present, along with annotations carrying information required to emit GC metadata. Section 3.2 provides further details on the semantics of instructions and the memory organisation of programs expressed in LLIR.

LLIR is an SSA-based representation. While imperative SSA representations are largely equivalent to the lambda calculus often used in the compilers of functional languages [Appel 1998], we chose the former as it more closely resembles the low-level instructions and control-flow constructs present at this stage of compilation. An imperative intermediate form also greatly simplifies the translation to machine-code through LLVM, since the *Machine IR* is also in SSA form.

Besides the textual representation, our framework provides a flexible in-memory representation for LLIR in the post-link optimiser, as discussed in Section 3.4. The optimiser tool is also a code generator: LLIR is lowered to machine-code through LLVM's *SelectionDAG* and existing *Machine IR* pipeline. Since our representation is both generated from and lowered to *SelectionDAG* nodes, there are close similarities between a large subset of the instructions and the DAG. However, in order to implement garbage collection efficiently, we have extended the DAG with additional nodes and the *Machine IR* representation with custom pseudo-instructions, as described in Section 3.5.

3.1 OCaml Runtime and Object Representation

Even though Duplo is a framework for OCaml and C cross-language optimisations, it implements few OCaml-specific features. Most notably, the framework does not provide a runtime system, a garbage collector or an OCaml-specific heap representation: all these components are implemented by the standard runtime library bundled with the OCaml compiler, indicated in Figure 3, which is compiled from C to LLIR and optimised along with the target OCaml executables.

In order to simplify the analyses required to enable optimisations, Duplo reasonably assumes that the runtime implements garbage collection correctly, preserving an invariant: at any point in

the program, heap roots are assumed to be pointing to valid objects. The transformation of code sequences that do not contain any function calls does not require any OCaml-specific knowledge: the preservation of all side effects, including writes to memory, ensures that the heap is correct at any program point. Even though we presently target single-core OCaml 4.07.1, multi-core OCaml [Sivaramakrishnan et al. 2020] will be supported by emitting the appropriate atomic instructions or barriers and preserving their semantics. The assumption of correctness is exploited across function calls: while heap roots might be altered, by heap compaction for example, they must point to an object the same shape as before the function call. Using the annotation mechanism presented in Section 3.2.3 to identify roots, transformations can hoist, remove or reorder them across call sites.

The changes to the OCaml runtime are intended to be minimal in order to mitigate incompatibilities between versions of OCaml and allow Duplo to be a drop-in replacement. The object representation of OCaml is not altered, as such a change would break the entire ecosystem due to the presence of a large number of libraries that rely on the FFI and assume a specific representation. Furthermore, layout choices are more appropriate at earlier stages in the pipeline. More precisely, Duplo is orthogonal to high-level language features: the only requirement from an object representation is for pointers to be stored at correctly aligned locations, enforced on some hardware and crucial for performance on others. The optimiser preserves the semantics of instructions that build or inspect objects conforming to any representation respecting this criteria. Such a minimal assumption is sufficient to enable transformations and greatly simplifies the optimiser, since passes can handle both languages without requiring specific knowledge of either of them.

While a more complex representation could benefit OCaml, it would be transparent to Duplo, which achieves gains independently at a lower level. A change to the object model might severely limit the capabilities of the FFI by preventing C methods from easily allocating on the OCaml heap, passing non-trivial types as argument or trivially serialising and de-serialising objects, as is the case with Haskell[1] and MLton[2]. We consider the type-agnostic representation of OCaml to be a strength of the language, as it enables objects to be trivially accessed, traversed and serialised, while more complex representations require additional information to identify fields and heap pointers.

By design, our optimiser does not carry or exploit type information specific solely to a single input language since a compiler or a link-time optimiser specialised for that language should have already exhaustively optimised acting on it. Instead, LLIR targets the low-level common ground between languages to find new opportunities beyond the limitations of prior stages.

3.2 Low-Level Intermediate Representation

3.2.1 Types. Each virtual register used in LLIR has a static type, expressed on the instruction that defines it. With the exception of 128-bit wide types, which are broken down into 64-bit components, virtual registers map directly to physical register classes available on modern hardware:

- `i8` and `u8` are used for condition codes and shift widths, besides byte-wide arithmetic
- `i16`, `u16`, `i32`, `u32`, `i64` `u64`, `i128` and `u128` represent integers
- `f32` and `f64` are generic floating-point types
- `f80` is required by x87 FPU instructions

The type system is simple, reflecting the constraints of the underlying hardware. While most instructions expect all their operands to be of the same type or to match the bit width of the hardware equivalent, some exceptions exist. For example, `mov`, `sext`, `zext` and `fext` are used to cast types of different size or to convert floats to integrals; otherwise signed and unsigned types can be used interchangeably. Comparisons are allowed to produce an integral flag of any width and

[1] https://wiki.haskell.org/Foreign_Function_Interface#Foreign_types
[2] http://mlton.org/ForeignFunctionInterfaceTypes

Table 1. LLIR instructions

Instruction	Description	amd64 Lowering

`call.T.n.conv $r, args...`
`tcall.T.n args...`
`invoke.T.n $r, args..., ` L_{cont}, L_{throw}
`tinvoke.T.n args..., ` L_{cont}, L_{throw}
	Calls, tail calls and vararg calls with explicit control flow	`callq` or `jmpq` after argument setup

`ret $r`	Return value from function	`mov` to `%rax`, `ret`
`trap`	Unreachable point	`ud2`
`arg.T $r, n`	Virtual register for argument	Register, live on entry
`frame.i64 $r, obj, off`	Index into the stack	`$rbp`/`$rsp` offset
`alloca.i64 $r, $s`	Dynamic allocation on stack	`$rsp` adjustment
`vastart $s`	Vararg structure setup	register spills
`cmp.T.cc $r, $lhs, $rhs`	Comparison	`test`, `cmp`
`jmp` L_{block}	Unconditional jump	`jmp`
`jcc $cond, ` L_{true}, L_{false}	Conditional jump	`jeq`, `jne`, ...
`ji $target`	Indirect jump to block	`jmpq $reg`
`switch $v, ` L_i	Jump to branch *v*	`jmpq` with jump table
`select $r, $cond, $true, $false`	Ternary operator	`cmov` or branches
`ld.n.T $r, $m`	Load *n* bytes and extend	`mov`, `movsx`, `movzx`, ...
`st.n $m, $v`	Truncate to *n* bytes and store	`movq`, `movl`, ...
`xchg.T $r, $m, $v`	Atomic exchange	`lock xchg`

`add.i64 $r, $lhs, $rhs`
`sub.i64 $r, $lhs, $rhs`
`mul.i64 $r, $lhs, $rhs`
`neg.i64 $r, $arg`
`...`
	Arithmetic instructions	Target equivalent

`rdtsc $r`
`clz $r, $v`
`popcnt $r, $v`
	Target-specifc instructions	Target equivalent

`mov $r, $reg` `set $reg, $v`	Read hardware reg Write hardware reg	`mov` or `lea`
`undef.T $r`	Undefined value	nothing
`phi.T $r, ` $Block_i$, $vreg_i$	SSA pseudo-instruction	Replaced with copies
`mov.T $r, value`	Introduce a constant value	Folded into users

the shift or rotate amount can be an arbitrary integer as well. Even though functions specify types for their arguments, which must be consistent with the `arg` instruction, mismatches are tolerated at call sites, in accordance with the semantics of C. Memory operations, such as `ld`, `st`, `xchg`, `frame`, `vararg` and `alloca`, use or define pointers, expecting to operate on an integral type that has equal bits to the address width of the target.

3.2.2 Instructions. LLIR instructions are closely related to the instructions of the target architecture, with the exception of some pseudo-instructions that either expand to complex sequences or have no assembly equivalents. Table 1 summarises them and explains their use and lowering on `amd64`.

A typical LLIR instruction, such as `ld.8.i64`, consists of an opcode (`ld`), a type (`i64`) and an optional size (8). Additionally, calls also encode a calling convention. The value defined by an instruction is referenced through a virtual register, the first in the list. Based on their kind,

instructions accept multiple virtual registers, constants or labels as arguments. The size is used by calls to determined the number of fixed arguments (additional arguments are prepared for a vararg call) and by loads and stores to determine the number of bytes to transfer, extending them or truncating them to the argument or return type if necessary. LLIR is close to machine-code: unlike LLVM, instead of relying on intrinsics and inline assembly, all relevant instructions from the target machine are exposed. Even though some hardware instructions are fixed to use specific registers, all arguments and defined values are referenced through virtual registers, with the code generator determining the optimal allocation to hardware registers. Certain registers (such as `$rsp` and `$rip` on `amd64`) and special locations (return address, frame pointer) can be directly retrieved using the `mov` pseudo-instruction, since they are required to effectively implement exception handling.

3.2.3 Annotations. Some instructions are tagged with annotations, required to correctly implement garbage collection. `@caml_value` is used to identify all virtual registers that refer to heap-allocated OCaml values, requiring transformations to carefully preserve this annotation when operating on instructions producing them. The value annotation is used to identify set of heap pointers live out of call sites annotated with `@caml_frame`: the OCaml garbage collector requires the return address to be mapped to a block of metadata indicating the location of the live values, whether in registers or on the stack. This system of annotations is more flexible than LLVM's `llvm.gcroot` intrinsic since it allows virtual registers storing heap roots to be mapped to physical registers, instead of forcing them to be spilled to a shadow stack.

Even though annotations are currently limited to OCaml, the system can be extended to other languages and collectors. If the OCaml runtime were to change fundamentally, introducing a vastly different collector, the annotation mechanism would have to be adjusted, although such a scenario is unlikely since we cover the design space that does not require sweeping changes to the ecosystem.

3.2.4 Stack. The hardware stack is used by both C and OCaml, although only C accesses it explicitly. Figure 5 highlights the downwards-growing stack layout, typical for the `amd64` target, along with LLIR registers addressing specific locations: `$sp` and `$fp` delimit the top of the stack and the bottom of the frame, respectively. `$sp` always maps to `$rsp`, but the frame pointer is usually omitted if no dynamic adjustments are present, allowing `$fp` to be derived from `$rsp`. The `$ret_addr` pseudo-register follows `$fp` to access the function's return address. Arithmetic on these pointers is banned: they can only be used to unwind the stack while searching for GC roots and to locate a handler and restore a previous stack frame while throwing an exception. While the stack layout is transparent to C, the pseudo-registers are vital for the efficient implementation of garbage collection and exception handling in OCaml. `$rsp` can be restored to the same frame as long as it points into the dynamically growing region, a feature required by C99 variable-length arrays [ISO/IEC 9899:1999, 1999]. While not yet implemented, LLIR could support `push`/`pop` instructions analogous to their `amd64` counterparts.

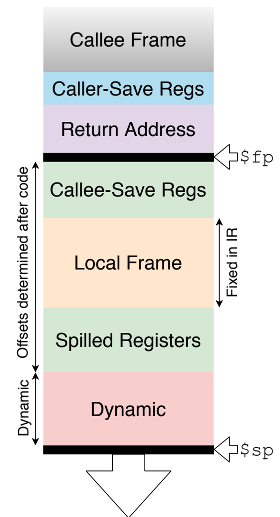


Fig. 5. Stack-frame layout

Local stack-allocated objects are specified using the `.stack_object id, size, align` directive on each function. The `frame.i64 $r, id, offset` instruction derives a pointer into a particular object, relative from `$rsp` or `$rbp`. The semantics of local pointers are inherited from C since their use in OCaml is restricted to allocating storage for exception handlers. Arithmetic is allowed as long as the pointer is at most 1 byte past the end of the object and pointers can be arbitrarily tested for equality and compared to null. However, ordering is only defined for pointers into the same object, as the LLVM backend is allowed to freely

reorder them. The `StackSlotColoring` pass on *Machine IR* is allowed to coalesce objects with disjoint live ranges.

3.2.5 Static Data. The static-data sections describe the raw and type-less initial contents of global data items from OCaml and C executables using simple labels and directives. LLIR directives are largely identical to those used in typical assemblers, such as *as*, allowing the reuse of existing backends to emit data sections. Besides the standard `data`, `rodata` and `bss` sections, LLIR supports a `caml` section used for OCaml objects with permanent lifetime. Objects are built up using directives specifying a few bytes of data at a time: `.byte`, `.word`, `.long`, `.quad` and `.ascii`. In addition, `.align` introduces padding to align the following data item to a specific boundary and `.space` reserves an arbitrary number of zero-initialised bytes, primarily for use in the `.bss` section.

While such a representation for static objects discards almost all type information from the source languages, it does represent the common denominator between OCaml and C and captures all the relevant contents. As an example, Figure 6 shows a typical declaration from an OCaml object, defined using a header with a tag preceding a label and all fields following it. The `.end` delimiter carries useful information, delimiting the boundaries between objects: according to the semantics of both C and OCaml, given a pointer into a block, arithmetic can only derive pointers within the bounds of that block.

```
  .data caml
  .align 8
  .quad 3072
camlTest__a:
  .quad 3
  .quad 5
  .quad b
  .end
```

Fig. 6. `let a = (1, 2, b)`

The end delimiter is used by points-to analyses and various transformations; unlike existing binary optimisers, which need to infer this information, our optimiser accurately captures it from source. Unfortunately, our representation loses fine-grained read-only attributes, since the GC colour bits in the headers of all OCaml objects, including immutable ones, are modified during collection, so all objects are placed in a readable and writeable segment. Similarly, the lowering of C loses the `const` attribute that might be present on individual fields. This places a burden on the optimiser, as analyses need to prove whether individual addresses are mutated.

3.2.6 Heap. LLIR does not explicitly represent the heap and it does not enforce any constraints beyond assuming that the addresses of blocks allocated are aligned to at least an 8-byte boundary (on `amd64`). It is the responsibility of individual analyses to understand calls that represent allocation sites, such as `malloc` and `caml_alloc1`. The models employed and the additional assumptions made by the points-to analyses provided with the framework are detailed in Sections 4.2 and 4.3.

3.3 IR Generation

3.3.1 LLVM. LLIR is emitted from C using a custom backend mapping LLVM IR to LLIR through a straightforward and well-documented process [Lattner 2020b]. Even though the code generator targets a new architecture, the `clang` frontend reuses the headers, builtin methods, ABI and configuration of the target machine (e.g. `amd64`), which is the final output of the framework. Unlike other targets, the LLIR backend skips register allocation, using an infinite number of virtual registers instead. The final assembly contains information in addition to that emitted for a typical target, attaching the calling convention, argument types and stack-object layout through directives on each function. Given that LLIR expects all virtual registers to be defined, the `undef` pseudo-instruction is provided to introduce such values, which would be typically omitted on other platforms.

3.3.2 OCaml. The LLIR backend in the optimising `ocamlopt` compiler maps instructions from the `Linear` representation to LLIR instructions, skipping register allocation altogether. While most instructions involve a simple one-to-many mapping without pattern matching or scheduling, calls require additional work because of exceptions. If exception handlers are present, LLIR imposes the

use of the `invoke` and `tinvoke` instructions. Unlike calls, these variants are basic-block terminators and require an explicit continuation block and an exception handler to be specified, information that is not readily available in `Linear` form. Our backend scans the `Linear` representation to find the exception handler for each call and emit the appropriate `invoke` instruction where necessary. As LLIR supports an infinite number of virtual registers, register allocation between the `Mach` and `Linear` stages is skipped: LLIR reuses the unique numbers `Mach` already assigns to values.

3.4 Optimiser Framework

The post-link optimiser is an executable invoked by the linker that reads an LLIR program expressing an entire OCaml application and outputs an object file containing optimised machine-code. It links to LLVM, reusing its code generator and several other facilities. The optimiser applies a number of passes, detailed in Section 4, which are enabled based on the selected optimisation level:

- **O0** does not enable any passes
- **O1** enables the bulk of all transformations, enumerated in Section 4.4
- **O2** in addition to **O1**, redundant load/store elimination is enabled, described in Section 4.3
- **O3** in addition to **O2**, global unreachable-code elimination is enabled, described in Section 4.2

Similarly to LLVM, passes can either apply transformations or perform analyses and expose their results to downstream passes. An individual pass mutates a program, which contains all data segments and functions. Functions are split into blocks, consisting of optional `phi` nodes followed by multiple instructions and a single terminator, which can be a jump, return, tail call, invoke or trap instruction. The types representing them implement the graph traits of LLVM, allowing facilities such as dominator-tree construction and post-order traversals to be reused.

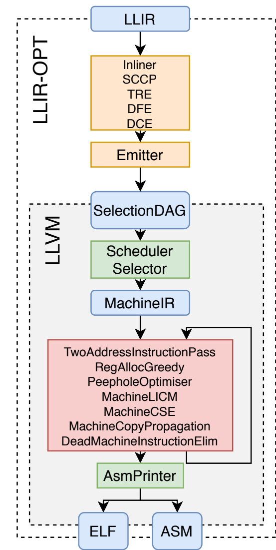


Fig. 7. LLIR-OPT pipeline

3.5 Machine-Code Generation

Duplo currently supports only an `amd64` backend. Machine-code is generated from LLIR by first converting instructions to LLVM *SelectionDAG* nodes and then passing the resulting *Machine IR* through the a slightly modified version of the target-specific backend of LLVM. Even though LLIR aims to be generic, several instructions (such as `clz`) and pseudo-registers (such as `$sp`) expose details specific to the target hardware: code can be generated through LLIR for multiple targets, but the IR for each target must be separately compiled. At the moment, only an `amd64` lowering is implemented. While most of the LLIR instructions map easily to existing DAG nodes, we extended *Machine IR* and associated passes with an additional instruction, `GC_FRAME`, to compute the metadata required by the garbage collector. The GC root mechanism, including the changes it imposes on the passes highlighted in Figure 7, could be upstreamed, but the patch exposing the SelectionDAG and target-specific backends require a fork.

Across each call site, the OCaml garbage collector requires a descriptor identified through the program counter to describe the set of locations that are live out of the call and contain heap roots that must be rewritten after compaction or promotion to the major heap. In LLIR, these values are identified by the `@caml_value` annotation on the instructions producing them: while lowering to selection DAG, we use live-variable analysis to identify those that should be added to the descriptor attached to each call site [Boissinot et al. 2008; Havlak 1997] and generate a `GC_FRAME` instruction referencing all the *Machine IR* virtual registers to which they are mapped. The register allocator

replaces the virtual registers with the actual physical location where the roots will be stored across a call, either on the stack in a spill slot or in an architectural register. A custom pass emits the required descriptor for each pseudo-instruction after all transformations have been applied. The register allocator and several passes were modified to respect the semantics of heap pointers and to avoid incorrectly hoisting, altering or removing instructions operating on them.

By avoiding the use of LLVM IR, our lowering manages to efficiently handle heap roots while adjusting only a small number of target-specific passes, instead of having to modify every single high-level LLVM transformation. While we lose out on some high-level optimisations provided by LLVM, because of the peculiarities of OCaml it is highly unlikely that they would provide any benefit: the points-to analyses of LLVM are designed to run fast on a C-style memory layout, sacrificing accuracy, without providing the information required to de-structure OCaml constructs. Additionally, support for OCaml heap-root semantics would incur extensive changes to numerous passes, such as Loop Invariant Code Motion. We chose to re-implement these passes at the LLIR level instead, in addition to transformations that specifically exploit the new cross-language opportunities. We alter only *Machine IR* passes, which are less numerous than the LLVM IR ones. The code generator and the low-level target-specific passes that are reused provide a significant boost to performance on their own, compared to the `amd64` backend of OCaml, as we observe and discuss in Section 5.

Presently, the performance of the LLVM backend is problematic on a very large method, `caml_program`, responsible for calling methods initialising other modules. In some of the largest OCaml applications, `caml_program` can contain basic blocks of 50,000 instructions, even before inlining, which is problematic for LLVM passes that run on basic blocks in worse than linear time. Since this method is generated in `Cmm` form at link time, Duplo is the first and only tool in the pipeline to optimise it, exposing the performance bottleneck. This warrants further investigation into the handling of blocks and functions of unusual size in LLVM, as methods of similar scale could be produced by automatically generated C/C++ code or created by LLVM's own LTO implementation.

4 ANALYSES AND TRANSFORMATION

In order to prove the utility and feasibility of transformations at the LLIR level, we implemented a number of optimisation passes within our post-link optimiser. LLIR transformations have two goals. First, they transform LLIR to eliminate redundancies in the lowering of OCaml or C that were introduced either by inefficiencies in the existing compilers or due to the requirements of targeting LLIR. The pipeline composed of such transformations shapes the intermediate representation into a form that can be lowered optimally using LLVM's code generator. Second, the passes exploit global information exposed in LLIR to apply transformations across the language boundary. While some improvement is achieved, the purpose of the passes outlined in this paper is to enable efficient code generation, hence we do not explore problems related to optimal ordering or tuning heuristics.

4.1 Inlining

While external calls that cannot trigger garbage collection can be inlined trivially on the common LLIR representation, calls that allocate memory are executed through a trampoline, requiring additional adjustment. The trampoline, shown in Figure 8 in LLIR form, saves the return address and the stack pointer. The return address points to the GC metadata attached to the bottommost frame, indicating the starting point of the traversal, which identifies and adjusts roots. Besides embedding the callee into the caller, inlining

```
caml_c_call:
  .call         caml_ext
  .args         0, i64
  mov.i64       $0, caml_last_return_address
  st.8          [$0], $ret_addr
  mov.i64       $1, caml_bottom_of_stack
  st.8          [$1], $fp
  arg.i64       $2, 0
  tcall.i64.c   $2
```

Fig. 8. OCaml-to-C trampoline

needs to ensure that the last return address and the bottom of the stack are valid. The trampoline is not explicitly present in the IR: calls with the `c` calling convention annotated with `@caml_frame` are instead re-routed by the code generator, which also emits the metadata.

In addition to the annotated heap roots present in LLIR originating from OCaml, the garbage collector keeps track of additional objects allocated and used in C methods by storing pointers to them in a linked list rooted at `caml_local_roots`. An individual node in the list describes a single frame, tracking incoming arguments and a fixed set of locals. Fortunately, this list is independent of the stack described through descriptors, allowing annotated roots to coexist with pointers tracked through `caml_local_roots`. Our inlining method exploits this: whenever a method is inlined at a call site annotated with `@caml_frame`, the annotation is propagated to the relevant calls. The additional annotations ensure that all heap roots are accounted for across the nested calls, either through the local root chain or by a newly created descriptor. An inlined call requires the annotation if it can trigger the garbage collector. To reduce the number of annotations and, implicitly, the number of calls executed through the trampoline, we identify the set of functions which allocate:

$$\text{TRAMPOLINE}(f) = \begin{cases} 1, \text{ if } f \text{ allocates or has indirect calls} \\ 0, \text{ if } f \text{ is a leaf method} \\ \bigvee_{c \in \text{callees}(f)} \text{TRAMPOLINE}(c) \end{cases}$$

Since identifying the targets of indirect calls requires a potentially expensive points-to analysis, for the purpose of inlining we over-approximate and assume that all indirect call sites allocate. The predicate is then computed by propagating information across the topologically sorted graph of the strongly connected components in the call graph restricted to C methods [Tarjan 1972].

Figure 9 illustrates a typical C method, which extracts the real component of a complex number represented using two single-precision floats. The method is inlined into an OCaml call site. Since the `caml_copy_double` method invokes the known `caml_alloc_small` allocation site on its own, it is flagged as one of the methods that requires a trampoline, thus the `@caml_frame` annotation is propagated to it. Any values that would have been live across the call in the original caller will now be recorded in the frame for the copy method. Additionally, after inlining, the local root management code is embedded in the caller, redundantly tracking the original argument to the function. While not yet implemented, a future transformation could identify the set of pointers reachable through the root chain and annotate them with `@caml_value`, eliminating the link.

4.2 Unreachable-Code Elimination

A typical C application relies on function pointers and indirect calls sparingly, yet these constructs are ubiquitous in OCaml programs. Indirect calls pose a problem for unreachable-code elimination: any function pointer in a data segment can potentially reach a call site, preventing the immediate removal of functions that are referenced from a data segment. In the context of the OCaml FFI, data-to-code references are quite common: when a wrapper is created for a C data type, methods that serialise, deserialise, hash and finalise the object are registered in a global object referenced by all boxed instances of the type. Since an OCaml program might not use all data types or use all methods attached to a type, the post-link optimiser is uniquely qualified to remove such unused callbacks. To achieve this goal, we provide a global unreachable-code elimination pass.

Global unreachable-code elimination removes functions that are never invoked directly, but are referenced from either code or data by having their addresses taken. If a pointer to such a function never reaches an indirect call site, it is never executed, thus its body can be eliminated. Even though

```
complex_re: # C method invoked from OCaml
 .stack      0, 64, 16
 .stack      1, 8, 16
 .args       0, i64
 .call       c
 arg.i64     $1, 0
 # Setup of local root list
 frame.i64   $14, 1, 0
 st.8        [$14], $1
 # Chain of local roots
 mov.i64     $2, caml_local_roots
 ld.8.i64    $3, [$2]
 frame.i64   $5, 0, 16
 st.8        [$5], 1
 frame.i64   $8, 0, 8
 st.8        [$8], 1
 frame.i64   $17, 0, 0
 st.8        [$2], $17
 st.8        [$17], $3
 frame.i64   $10, 0, 24
 frame.i64   $20, 1, 0
 st.8        [$10], $20
 # Actual complex_re implementation
 ld.8.f64    $11, [$1]
 # Result is wrapped into a double
 mov.i64     $12, caml_copy_double
 call.i64.c  $13, $12, $11
 # Unlink C scope from GC chain
 st.8        [$2], $3
 ret         $13
```

```
camlTest__entry: # Original Caller
 ...
 mov.i64    $3, complex_re
 call.i64.c $99, $3, $1 @caml_frame @caml_value
 ...
camlTest__entry: # After inlining complex_re
 ...
 .stack      5, 64, 16
 .stack      6, 8, 16
 ...
 mov.i64     $3, complex_re
 frame.i64   $34, 6, 0
 st.8        [$34], $1
 mov.i64     $2, caml_local_roots
 ld.8.i64    $23, [$2]
 frame.i64   $25, 5, 16
 st.8        [$25], 1
 frame.i64   $28, 5, 8
 st.8        [$28], 1
 frame.i64   $37, 0, 0
 st.8        [$2], $37
 st.8        [$37], $23
 frame.i64   $30, 5, 24
 frame.i64   $40, 1, 0
 st.8        [$30], $40
 ld.8.f64    $31, [$1]
 mov.i64     $32, caml_copy_double
 call.i64.c  $99, $32, $31 @caml_frame @caml_value
 st.8        [$2], $23
 ...
```

Fig. 9. Inlining of `complex_re`, which extracts the real part of a complex number. Red indicates the instruction that performs all the useful work in the function, while the green registers highlight the inputs and outputs

a function is never called, its address might be compared for equality. In order to preserve the behaviour of function-pointer comparisons, the function pointers themselves cannot be replaced with a null pointer; replacing the body of such a function with a single trap instruction suffices. To prove that a function is never indirectly invoked, an Andersen-style point-to analysis [Andersen 1994] is used to identify all the functions in the points-to sets of indirect-call targets.

A typical low-level pointer analysis treats the static data segments as a contiguous chunk of memory, relying on field-sensitivity to distinguish separate objects and data structures [Guo et al. 2005]. Instead of implementing field-sensitivity or offset-sensitivity, our analysis relies on `.end` markers emitted by the backends to delimit individual objects in the data segment and to avoid the use of a field- or offset-sensitive analysis. In a safe language such as OCaml, deriving a pointer outside the bounds of an object is illegal, whereas in C a pointer can be derived at most one byte past the end of the object. Dereferencing negative offsets or addresses beyond the end of the object results in undefined or illegal behaviour in both C and OCaml. Given a pointer `p` into an object A followed by B in memory, whenever an `add` instruction is encountered tbat offsets `p`, the semantics of the source languages ensure that the pointer derived from `p` does not spill over into B. If object boundaries were not known, a proof would be required to show that the offset from `p` does not exceed the size of A for the pointer derived from `p` to still be limited to A.

The implementation of the analysis is split into two components: a constraint builder and a constraint solver [Hardekopf and Lin 2007a]. The constraint builder first inspects the data segments and builds a set for each object, recording pointers between objects in the sets. The control flow graph of the program is then traversed starting at external entry points, typically `main` in an executable. For each instruction in a function, flow- and context-insensitive subset inclusion constraints are generated to model the flow of information through registers and memory. The traversal stops at indirect-call sites, which are recorded separately. Once all functions are explored and constraints are recorded, the solver is invoked to propagate information, identifying indirect-call-site targets.

The indirect-call sites are then expanded and the traversal continues with the potential targets and the process is repeated until the points-to sets of indirect calls converge.

We considered two different solvers, but the current implementation largely follows that of Hardekopf and Lin [2007a]. The constraint solver builds a graph with set nodes and dereference nodes: edges between set nodes represent subset inclusion, edges from dereference nodes indicate reads, while edges into dereference nodes indicate writes. Pointers are propagated along edges using a work queue and new edges are added when new values are propagated into sets that are dereferenced. Strongly connected components, which indicate equivalent points-to sets, are represented using a forest of disjoint trees and are collapsed using lazy cycle detection (LCD) and hybrid cycle detection (HCD), both implemented using a variation of Tarjan's algorithm that discards the frequent components with a single node [Nuutila and Soisalon-Soininen 1993]. Points-to sets are stored in sparse bitsets, which offer a reasonable balance between performance and memory usage. Despite the $O(N^4)$ complexity, the analysis runs in a reasonable amount of time on a wide range of applications: the order of magnitude is in line with that reported in other low-level analyses [Guo et al. 2005]. Methods relying on global variable naming (GVN) to further reduce the size of the graphs were also considered, however the cost of reducing the graphs outweighed any performance improvements on some benchmarks, since the running time of the simplification step itself is also $O(N^4)$ [Hardekopf and Lin 2007b]. The linear-time Steensgaard's method was also considered, replacing inclusion constraints with equality constraints [Steensgaard 1996]. Unfortunately, the results were not accurate enough to remove any function bodies.

4.3 Local Points-To Analysis

C methods return values to OCaml by boxing primitives into heap-allocated objects, so points-to information at the local level is crucial towards propagating values through memory, allowing redundant allocations to be removed. Since the global algorithm outlined in Section 4.2 sacrifices accuracy for performance, we outline another Andersen-style analysis whose scope is limited to individual functions and exploits the pointer semantics of both OCaml and C in order to enable field-sensitivity (more accurately, offset-sensitivity at the LLIR level). Table 2 shows the mapping from LLIR to constraints, including the `offset` and `range` predicates, which compute individual offsets or introduce imprecision by generalising a pointer to an entire object.

Both OCaml and C impose constraints on the alignment of heap-allocated blocks and the locations where pointers can be stored: both `malloc` and the OCaml allocator align objects to the word boundary and pointer values can be written to memory only at word-aligned addresses, specifically 8 bytes on `amd64`. OCaml respects this constraint implicitly since blocks are essentially arrays of 8-byte fields, while in C storing a value to an unaligned location results in undefined behaviour. These constraints allow heap objects to be modelled as a sequence of 8-byte sets of pointers: stores of pointers to unaligned locations are discarded. Unlike the global algorithm that only considers pointers to allocation sites, the local analysis models the offset as well, storing values to the appropriate field of a block. The model of individual blocks is truncated to 16 fields: this applies to 0.2% of all allocations that have no size and the 3% that exceed it. Offset computation by `add` and `sub` instructions is illustrated in Figure 10: green indicates known offsets into objects, while purple represents the set of offsets that exceed the limits of the model.

Some operations result in offsets that cannot be determined, such as `phi` nodes joining distinct pointers, requiring loads and stores to be over-approximated, storing to and loading from all fields. We implement this effectively through a pair of IN and OUT nodes, indicated in red: stores to an entire object are linked to the IN node, while loads originate from the OUT node. Both nodes transitively connect to all fields of the object; besides some offset calculations, the `range` function also generates them. Another source of uncertainty is the external node, representing information

Algorithm 1 Constraint solver with offsets

```
1:  procedure SolveConstraints(G)
2:      Q ← empty()
3:      for n ∈ G do enqueue(Q, n)
4:      while |Q| > 0 do
5:          from ← dequeue(Q)
6:          if d ← deref(s) then
7:              for ⟨A, index⟩ ∈ points-to(d) do Connect(d, field(A, index), field(A, index))
8:              for ⟨A⟩ ∈ points-to(d) do Connect(d, all-in(d), all-out(d))
9:      for to ∈ range-edges(from) do
10:         Propagate(Q, to, {⟨A⟩|⟨A⟩ ∈ points-to(from) ∨ ⟨A, i⟩ ∈ points-to(from)} )
11:     for to, off ∈ offset-edges(from) do
12:         Propagate(Q, to, {offset(A, i, off)|⟨A, i⟩ ∈ points-to(from)})
13:         Propagate(Q, to, {⟨A⟩|⟨A⟩ ∈ points-to(from)})
14:     for to ∈ set-edges(from) do Propagate(Q, to, points-to(from))
15: procedure Propagate(Q, to, set)
16:     if set ⊈ points-to(to) then
17:         points-to(to) ← points-to(to) ∪ set
18:         enqueue(Q, to)
19: procedure Connect(d, in, out)
20:     for store ∈ in-edges(d) do
21:         if in ∉ set-edges(store) then
22:             set-edges(store) ← set-edges(store) ∪ {in}
23:             enqueue(Q, store)
24:     if out-edges(d) ⊈ set-edges(out) then
25:         set-edges(out) ← set-edges(out) ∪ out-edges(d)
26:         enqueue(Q, out)
```

flowing in and out of the function and outside the scope of the analysis. Due to its link to a self-dereference node, any escaping pointer is expanded to its transitive closure, consistent with the conservative assumption that any derived pointer can be altered in an unknown context.

Algorithm 1 solves the constraints by propagating pointers and computing offsets in the corresponding graph and is an extension of the global analysis outlined earlier. Line 12 over-approximates all pointers in a set and propagates the range pointers, while line 14 applies the offset calculation. On line 15, over-approximated pointers are forwarded. The expansion of dereferences is handled on line 8 for known pointers, while line 10 adds edges to the IN and OUT nodes mentioned earlier.

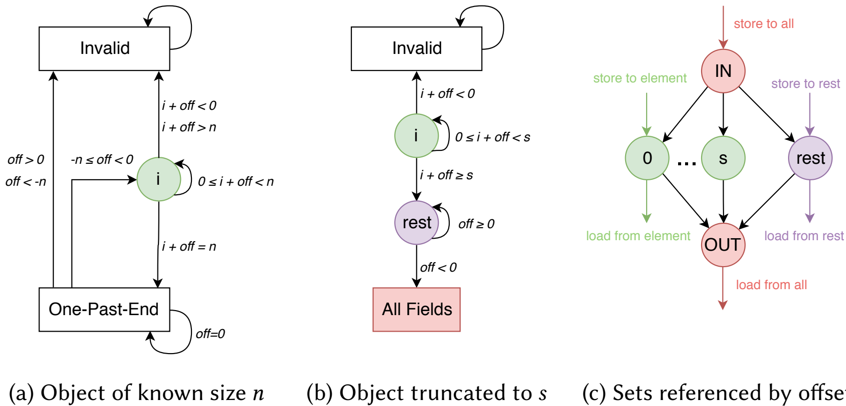
Presently, the use of the results of this analysis is limited to the elimination of redundant stores and the propagation of values through memory from stores to loads. Due to the simplicity of inlining heuristics, not enough opportunities are exposed for these optimisations to be broadly useful, resulting in only a small number of stores being removed in large applications.

4.4 Other Transformations

Our framework implements a number of typical transformations commonly used in compilers relying on SSA-based intermediate representations, including LLVM. While some transformations benefit from the global context in which the post-link optimiser operates, the main reason for their presence is to ameliorate inefficiencies present in the OCaml compiler, the LLIR backend or redundancies introduced when lowering from C, leaving the IR in a form that can be optimally lowered by LLVM's target-specific code generator. These passes complement the OCaml compiler by providing transformations that would be awkward to implement at any existing stage.

Table 2. Constraint generation

Instruction		Constraint	Graph
`frame.i64`	`$a, i`	$\langle \mathrm{F}, i\rangle \in a$	
`call.i64`	`$a, malloc, $s`	$\langle A, 0\rangle \in a, A = \mathrm{new}(s)$	
`mov.i64`	`$a, S`	extern $\subseteq a$	
`arg.i64`	`$a, i`		
`add.i64`	`$a, $b, n`	$a = \mathrm{offset}(b, n)$	
`sub.i64`	`$a, $b, n`	$a = \mathrm{offset}(b, -n)$	
`add.i64`	`$a, $b, $c`	$a = \mathrm{range}(b \cup c)$	
`or.i64`	`$a, $b, $c`		
`and.i64`	`$a, $b, $c`	$a = b \cup c$	
`select.i64`	`$a, $cond, $b, $c`		
`sub.i64`	`$a, $b, $c`	$a = \mathrm{range}(b)$	
`ld.8.i64`	`$a, [$b]`	$*b \subseteq a$	
`st.8`	`$a, [$b]`	$a \subseteq *b$	
`xchg.i64`	`$b, $a, [$c]`	$a \subseteq *c \wedge *c \subseteq b$	
`phi.i64`	`$a, [BB`$_0$`, b`$_0$`], ...`	$b_i \subseteq a$	
`call`	`f, $a`$_0$`, ...`	$a_i \subseteq$ extern	
`call.i64`	`$r, f, $a`$_0$`, ...`	$a_i \subseteq$ extern $\wedge$ extern $\subseteq r$	



(a) Object of known size n (b) Object truncated to s (c) Sets referenced by offset

Fig. 10. Calculation of $\mathrm{offset}(A, i, \mathit{off})$ and sets referenced through known or unknown offsets

Copy Propagation removes redundant copies introduced by the backends when they deconstruct SSA or assign virtual registers. While the `mov` instruction can appear anywhere, it is only relevant when it introduces annotations that are different to the operand's annotations.

Sparse Conditional Constant Propagation (SCCP) [Wegman and Zadeck 1991] is a typical pass for SSA-based IRs. Similarly to LLVM, our implementation also propagates relevant annotations and extends the lattice of values with an *undefined* value, which is propagated respecting the semantics of C. SCCP often folds redundant address calculations (replacing `add.i64 $0, ptr, 5` with `mov.i64 $0, ptr+5`) and simplifies conditional jump conditions.

Dead-Code Elimination is based on reverse dominance frontiers and removes instructions without side effects that do not produce results that escape the scope of the function through a store, return instruction or as an argument. This pass eliminates instructions redundantly emitted from OCaml's linear IR that could not be easily eliminated otherwise: since LLIR represents jumps to exception handlers from function calls explicitly through a `invoke` instruction, this pass manages to remove redundant exception landing pads.

CFG Simplification alters jump instructions and removes unreachable basic blocks. If the condition of a jump is known, it is turned into an unconditional jump, allowing the unused branch to be eliminated. This pass also performs jump threading by following and skipping sequences of basic blocks containing a single unconditional jump. Along with dead-code elimination, this pass manages to fully eliminate loops present in certain benchmarks.

Dead-Function Elimination automatically removes functions with no references, eliminating a large number of extern functions from C modules, along with a large number of C methods from OCaml libraries that are never invoked through the FFI. When points-to information is available, methods that never reach an indirect-call site are also eliminated.

Tail-Recursion Elimination turns tail-recursive methods into iterative loops, enabling our optimiser and LLVM to improve them. The lowering of loops in OCaml is not particularly effective, but its optimiser and code generator handle tail recursion well. The opposite is true of LLVM and our post-link optimiser, which are based on SSA: most optimisation passes target loops and prologue/epilogue insertion is not optimal on tail-recursive methods. In addition, transforming tail recursion into a loop aids register allocation, since arguments are no longer fixed to specific registers at the point of the backwards jump.

Higher-Order Specialisation identifies functions that take an argument and pass it directly to an indirect-call site, then identify the functions that are mapped to that argument across the whole program. A specialised version is created for each function, turning the indirect call into a direct one and allowing the inliner to embed the callee into the call site.

5 EVALUATION AND RESULTS

We evaluated our post-link optimiser on a wide range of executables from the *opam* repositories, along with a suite of open-source micro- and macro-benchmarks maintained by the OCaml community for the purpose of monitoring the performance of the OCaml compiler. We compare binaries produced by the post-link optimiser (built on LLVM 8) using the four different optimisation levels (`O0`, `O1`, `O2`, `O3`) to baselines produced by a slightly modified version of the default `amd64` backend of OCaml 4.07.1 (`O3` is only included in the discussion of code size since it does not affect performance). We discuss the impact on performance in Section 5.1, while Section 5.2 shows the code-size reduction achieved. Since the number or size of allocations is not altered by the post-link optimiser in its current form, there is no impact on memory usage to analyse.

We also consider the experimental high-level LTO implementation patched on top of OCaml[3], which stacks with our post-link optimiser, enabling comparisons of the `amd64` reference backend and all optimisation levels with and without the transformations applied at the `flambda` level. Since LTO mostly performs unreachable-code elimination and we found no statistically relevant difference in running times, we only report the effect on code size. The experimental LTO patch does not support shared libraries, thus Section 5.3 reports separately on the performance of the entire toolchain by benchmarking the build times and execution times of widely used OCaml applications that rely on them to compile or to run, comparing the reference to optimised toolchains.

In order to establish a fair baseline for comparison, some transformations that are not yet supported by the post-link optimiser have been disabled in the `amd64` OCaml backend:

- `%r14` does not cache the exception handler's address (`caml_exception_pointer`)
- `%r15` does not cache the allocation pointer (`caml_young_ptr`)
- Basic blocks that are never reached by a fall-through are not aligned

While these transformations could be performed in the post-link optimiser, the present implementation and evaluation focuses on optimisations that are beneficial at the post-link stage. In addition, to minimise differences, all variants were adjusted to build the runtime using the same version of `clang` and to produce static ELF executables linked to the `musl` [Felker 2019] standard C library, excluding any inline assembly from C dependencies.

5.1 Standard Benchmarks

Our first performance evaluation relies on standard tests collected by OCamlPro, *operf-micro* and *operf-macro* [4], which are commonly used to test the OCaml compiler and its variants [Sivaramakrishnan et al. 2020]. Although there is an overlap between the two sets of benchmarks, we present them separately in Sections 5.1.1 and 5.1.2 since we execute them using separate harnesses.

We repeated the experiment on two different systems, both running Ubuntu 18.04 on `amd64` processors: a server based on a single-socket Intel Xeon W-2195 with 256GB of RAM and a mini computer equipped with an Intel Pentium Silver J5005. Despite the fact that these two systems only cover the extremes of `amd64` implementations, we do not explicitly report results from mid-range desktop-grade processors as we found them to perform similarly to Xeons on single-threaded applications in terms of relative performance. Both microarchitectures are superscalar and out-of-order, but the server-grade W-2195 is significantly more complex, has more execution units and issues more instructions compared to the J5005, which also completely lacks an L3 cache.

Unlike the default harnesses provided with the benchmarks and maintained by the OCaml community, we account for measurement bias. Initially, we noticed that the running time of some benchmarks was dominated by a single short, hot loop. Manual inspection of the emitted code revealed that in most cases our backend chose better instructions (`movl $n, %eax` instead of `movq $n, %rax`, for example), but aligned loop entry points to a different offset inside a cache line. While loop alignment only slightly affected performance on the simpler J5005, we found measurements reported on the W-2195 to be unreliable. Optimal block layout is outside the scope of this paper: given that most benchmarks are dominated by such loops and neither OCaml, nor our backend lays them out based on informed decisions, we view them as opportunities for future improvement. To discard the effect of loop alignment, we follow a previously proposed mitigation and built ten different versions, disabling all code alignment and randomising the order of the functions [Mytkowicz et al. 2009]. Outside benchmarks, functions are aligned to a 16-byte boundary.

[3] https://github.com/ocaml/ocaml/pull/608
[4] https://github.com/OCamlPro/operf-micro, https://github.com/OCamlPro/operf-macro,

5.1.1 operf-macro. The *operf-macro* suite of benchmarks contains 191 individual tests: a mix of small programs measuring the performance of methods from widely used libraries, including the OCaml `stdlib`, along with benchmarks of commonly-used applications executed on typical inputs, such as the *menhir* parser generator [Pottier and Régis-Gianas 2016] and *js_of_ocaml* compiler [Vouillon and Balat 2014]. The relative running times, reported by the `wait4` system call, are illustrated in Figure 11. Benchmarks were executed sequentially, pinned to a single core while all others were idle and, whenever possible, inputs were adjusted such that running times were above 1 second. The speedup reported on the y axis is computed by dividing the mean running time of 50 executions, measuring five runs of each randomised variant. All tests with no statistically significant difference were excluded, as identified by a Welch t-test with a *p-value* of 0.01. For brevity, each column represents the geometric mean of a group of tests: either the same executable invoked with different parameters or a single program testing different methods in a library.

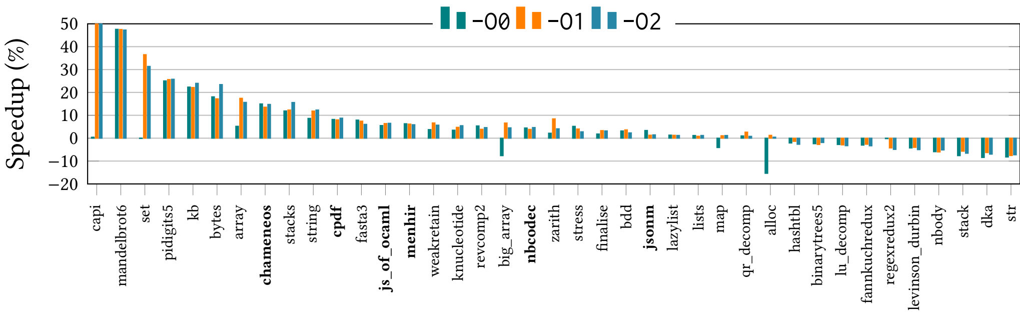
While the running time of large applications (indicated in bold) and some benchmarks is noticeably improved, by up to 7%, some smaller benchmarks suffer from degraded performance. Manual inspection of the generated machine code reveals the culprit to be LLVM: even though the DAG-based instruction selector consistently picks better instructions and finds improved schedules, the greedy register allocator sometimes fails to find the optimal spill or live-interval split points. These issues will be fixed over time by tweaking the heuristics used in the LLVM's greedy register allocator and implementing more aggressive instruction folding.

The `capi` benchmark, measuring the overhead of calls from OCaml to C, satisfies our expectations and proves the utility of optimising across the language boundary. The short external C methods, which may or may not allocate on the heap, are effectively inlined into OCaml closures, allowing further passes to reduce them to constants, sometimes eliminating entire loops, resulting in a speedup of up to 200% (clamped in the graph) starting at the `O1` level. While some closures originating from OCaml are not yet eliminated in this particular benchmark, six others were folded into constants and are excluded from the plot as their running time is zero.

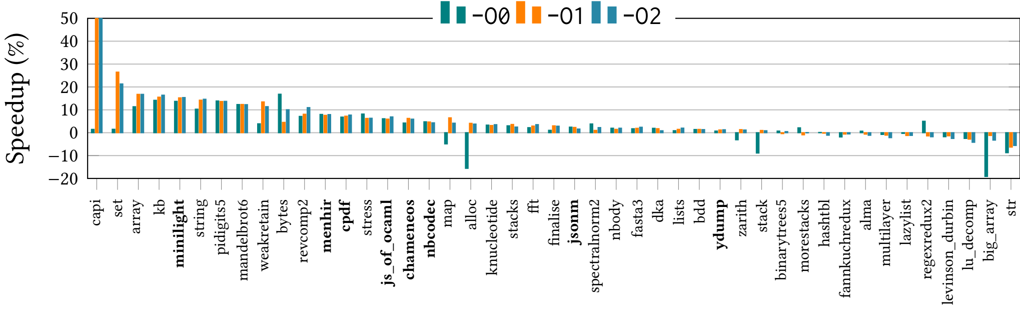
Tail-recursion elimination (TRE) is responsible for significant differences between `O0` and `O1` in benchmark groups such as *big_array* and *alloc*. LLVM's *Machine IR* does not handle tail recursion particularly well, including a costly prologue and epilogue in loops expressed as tail-recursive functions, a pattern OCaml programs heavily rely on. The `amd64` backend treats self-recursive methods separately, excluding the prologue and epilogue from the loop body. Our TRE achieves the same effect, turning tail-recursive methods into loops that lower to machine code similar to that generated by `ocamlopt`. As an added bonus, since in loop form variables are not pinned to particular argument registers and loops have a pre-header, the register allocator has more freedom to find a better assignment and the loop-invariant code-motion pass can hoist redundant computations.

5.1.2 operf-micro. Micro-benchmarks are executed using a simplified version of the *core-bench* framework. Unlike macro benchmarks, which rely on wall time, this test harness uses the Time Stamp Counter (TSC) provided by `amd64` processors and measures the running time of a varying number of iterations, yielding a noisy plot. We use RANSAC on the points correlating iteration counts with execution times to find the best robust parameters for $t = t_{iter} \times n + t_{overhead}$ and use t_{iter} as an estimate of the running time of a single iteration. Since the running time of these benchmarks is by design dominated by a single loop, we apply the mitigation mentioned previously and report the geometric mean of the ratios computed from ten executables with randomised layout.

Figure 12 reports the geometric mean of the ratios of running times of groups of similar benchmarks, excluding those that did not report a statistically significant change according to the Welch test. In general, our optimiser delivers significant improvements across the board. Given that there is no relevant difference in performance between the different compiler optimisation levels, and

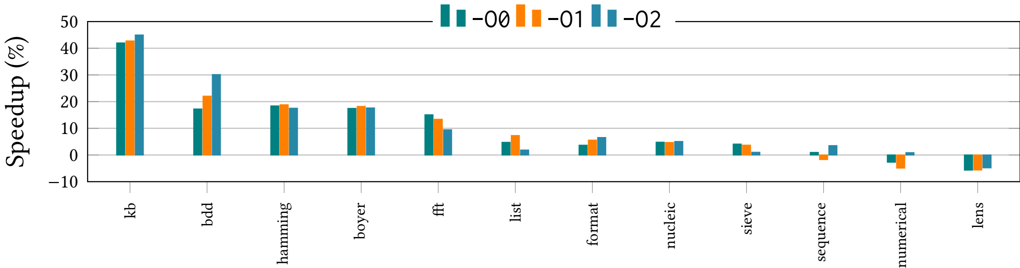


(a) Intel Xeon W-2195

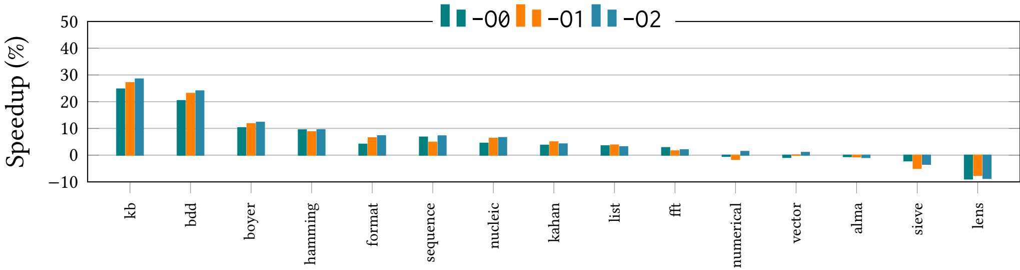


(b) Intel Pentium Silver J5005

Fig. 11. OCaml macro benchmarks



(a) Intel Xeon W-2195



(b) Intel Pentium Silver J5005

Fig. 12. OCaml micro benchmarks

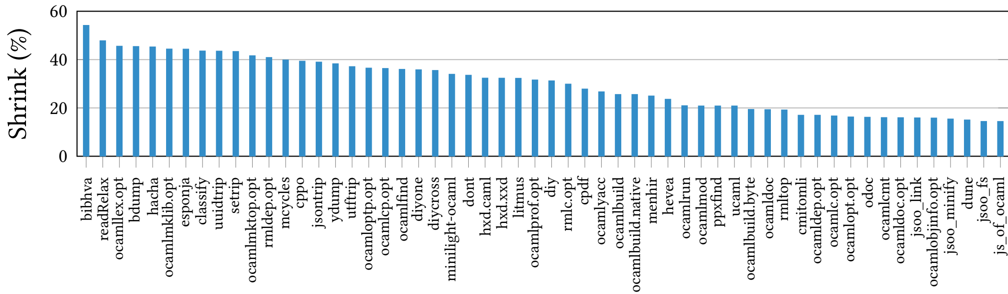


Fig. 13. Code size reduction with `-O3`

there are few opportunities for cross-language inlining on these benchmarks, the improvements can be attributed to better code generation achieved by the LLVM backend. Manual inspection of the disassembly of the *lens* test reveals only minor differences in the generated code. Due to the complexity of the target processor, selection and scheduling sometimes fail, but our benchmarks still show that on average LLVM outperforms the existing OCaml `amd64` backend.

5.2 Code Size

The post-link optimiser reduces the size of the text segment of binaries by 14% to 54%, as indicated in Figure 13, which shows the ratios of text segment sizes encoded in the ELF headers. Additionally, Figure 14b plots the mean and the distribution of improvements achieved at each optimisation level: while on average the unreachable-code elimination pass enabled by `-O3` reduces code size by around 8% on its own, a significant reduction of around 20% is noticeable even at `-O0`, which only enables a handful of passes and only minimally exploits the global and cross-language context on which the optimiser operates. A number of factors contribute to this improvement:

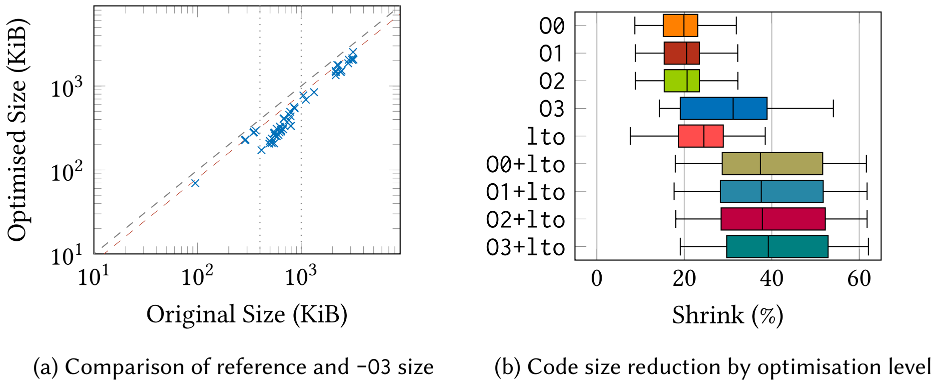
Amelioration of compiler inefficiencies The OCaml compiler often emits unreachable exception landing pads that are trivially eliminated by the control-flow-graph simplification.

Amelioration of linker deficiencies The dead-function elimination pass trivially removes C and OCaml symbols that are marked as extern in their modules, but have no actual uses and cannot be removed by the linker (without `-ffunction-sections`, at least).

Better instruction selection and register allocation by LLVM Even though the backend is instructed to optimise for performance, it often manages to choose more compact instructions, minimises stack spills and reloads, and merges spills and reloads into instructions that take memory operands. These choices allow for the better use of instruction caches.

The effect of other transformations is not as noticeable. Inlining, enabled at the `-O1` level, in its present form is quite conservative, yet responsible for a minimal increase in code size. `-O2` makes some use of local points-to information by removing a small number of unused load and store instructions, shrinking binaries by a few hundred bytes.

Figure 14a correlates the original and optimised binary sizes on a logarithmic plot. The binaries situated on the red dashed $y = 0.8x$ line are those that consistently benefit from the previously mentioned optimisations, while those underneath are further reduced through unreachable-code elimination based on points-to analysis. The executables smaller than 400KiB are C programs included in the OCaml distribution (*ocamlyacc*, *ocamlrun*), which are transformed similarly to a traditional C LTO tool. Since higher-order functions and callbacks are sparsely used in C, points-to analysis does not provide any benefit on them. On the other hand, applications larger than 400KiB are effectively minimised using such information, although the effect tapers off as they grow larger, especially after 2MiB. In its present form, unreachable-code elimination eliminates



(a) Comparison of reference and -O3 size (b) Code size reduction by optimisation level

Fig. 14. Code size reduction

an OCaml module if none of its methods are ever indirectly invoked or used as a functor. As binaries grow larger and use more modules, the more likely it is that at least one function from each library (especially the standard library) is used. The transformation seems to be most effective on executables less than 1MiB in size, including *ocamlc*, *ocamllex* and *cpdf*. The `ocamlopt.opt` benchmark compares the compiler with the `amd64` and the LLIR backends, which differ by a few kilobytes. While this does not affect the size comparison, it prevents performance benchmarks. Because of difficulties with the bootstrapping process of OCaml, a Duplo-optimised compiler targeting `amd64` cannot be built for comparison.

Figure 14b also reports results achieved by LTO on its own and combined with all levels of the post-link optimiser. While LTO on its own outperforms the optimiser without unreachable-code elimination enabled, `-O3` obtains significantly better results. The largest reduction is achieved by running both optimisers, although `-O3` stacked with LTO does not lead to a significant improvement over `-O2` anymore. The distributions indicate that there is significant overlap between the functions targeted by unreachable-code elimination provided by `-O3` and functions eliminated by LTO, while other transformations performed by the optimiser, especially those that ameliorate inefficiencies present in OCaml's backend, exploit opportunities unavailable on the intermediate representations of OCaml. `-O3` still manages to remove functions that cannot be proven unreachable by `flambda`.

Enabling LLVM LTO while building the runtime broke the bytecode interpreter during our attempt, suggesting that extensive changes outside the scope of our project are required. Assuming LTO was enabled, dead-code elimination would not have been any more effective since functions can be removed if they have no references from OCaml and such information is available only after linking the runtime, with the OCaml program relying on it in the linker, at the machine code level.

5.3 OCaml Applications

We also evaluated performance on widely used OCaml applications that require shared libraries to build or run. Table 3 reports the build times of the projects that provide the executables, the sizes of their text sections reported by `readelf` and their running times on the longest-running test from *operf-macro* or large examples of our own design.

Build times represent the wall time while the build system was executing 24 jobs in parallel to better quantify the impact on the critical path, more relevant than total time spent on the CPU. A comparison at the `O3` level is excluded since builds are an order of magnitude slower, as expected from a global points-to analysis. Projects with few dependencies (`ocamlc`) suffer less of an impact: since Duplo is a post-link optimiser, it must lower and optimise all functions from libraries when building an executable, which are otherwise readily available in binary form to a traditional linker.

Table 3. Benchmark of large OCaml applications, including the build times of the projects, text section sizes of individual executables and running times averaged over 10 runs on large inputs, executed on an AMD Ryzen 3900X. The independence of the samples was confirmed by a Welch t-test with $p = 0.01$

Tool	Reference			-O0						-O2					
	build	size	exec	build		size		exec		build		size		exec	
ocamlc.opt	1m48s	1.97Mib	6.03s	2m28s	+37.0%	1.78Mib	-9.3%	5.78s	-4.1%	2m58s	+64.8%	1.92Mib	-2.4%	5.60s	-7.2%
coqc	1m36s	6.04Mib	6.02s	2m26s	+52.1%	5.49Mib	-9.0%	5.83s	-3.1%	2m53s	+80.2%	5.97Mib	-1.1%	5.75s	-4.5%
coqchk	1m36s	1.60Mib	13.46s	2m26s	+52.1%	1.43Mib	-10.6%	12.62s	-6.3%	2m53s	+80.2%	1.60Mib	-0.3%	12.34s	-8.3%
ccomp	2m48s	1.97Mib	12.68s	3m02s	+8.3%	1.68Mib	-14.4%	12.54s	-1.1%	3m06s	+10.7%	1.78Mib	-9.4%	12.18s	-3.9%
alt-ergo	0m04s	1.49Mib	2.37s	0m14s	+250.0%	1.31Mib	-12.5%	2.08s	-12.1%	0m17s	+325.0%	1.42Mib	-5.0%	2.06s	-13.0%
ocamlformat	0m04s	4.72Mib	12.79s	0m47s	+859.2%	4.27Mib	-9.5%	12.51s	-2.2%	3m57s	+4736.7%	5.28Mib	+11.8%	12.41s	-3.0%
js_of_ocaml	0m04s	2.71Mib	7.92s	0m45s	+878.3%	2.45Mib	-9.9%	7.30s	-7.8%	1m00s	+1204.3%	2.61Mib	-3.9%	7.16s	-9.7%

`ocamlformat` is an outlier since aggressive inlining produces functions that are large enough to hit various edge cases in LLVM; such issues can be mitigated.

Performance improvements are consistent, however the code size of `ocamlformat` is an outliner since the inliner increases it while improving performance. This application uses `base`, a standard library replacement: future work will consider reducing the impact of such duplication.

6 CONCLUSION AND FUTURE WORK

Duplo, the framework we describe in this paper, provides a common representation to enable post-link optimisations on OCaml applications using the FFI. We evaluated the performance of the optimiser and the new code generator on a wide range of OCaml applications and benchmarks, measuring both code size and running time on three modern `amd64` processors. While the bulk of the gains in performance can be attributed to the improved backend, the optimisation passes on LLIR deliver some improvements on their own. LLIR is a crucial step in the process: by comparing our post-link optimiser to existing attempts of lowering OCaml through LLVM, we establish that the use of the improved code generator would not be possible without our intermediate step.

In addition to the initial set of optimisations that yield the reported results, we also highlight redundant constructs exposed by certain transformations, such as inlining, that are not yet optimised away. Future iterations of the optimiser will target such redundancies in order to exploit information in the cross-language context and reduce the FFI overhead. We hope to enable novel use cases, such as the development of libraries introducing novel data types, especially numerical ones, implemented over the FFI without incurring an overhead for its use.

The future of the project is focused on understanding interactions between C and OCaml through the heap in order to further eliminate redundancies imposed by the FFI, exposed by the inliner in the form of unnecessary boxing and unboxing constructs. The simplification of such constructs will enable existing transformations to optimise further and reduce the FFI overhead.

ACKNOWLEDGEMENTS

We thank the ICFP reviewers for their valuable feedback. We thank Lukasz Dudziak for testing the accompanying artefact. This work was supported by the Cambridge Trust and the Engineering and Physical Sciences Research Council (EPSRC) through grant reference EP/P020011/1.

ADDITIONAL RESOURCES

Additional resources, including a virtual machine with instructions to reproduce all the results reported here, are available at https://doi.org/10.17863/CAM.52533.

REFERENCES

Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. University of Cophenhagen.

Andrew W Appel. 1998. SSA is functional programming. *ACM SIGPLAN Notices* 33, 4 (1998), 17–20.

Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoît Dupont de Dine hin, and Fabri e Rastello. 2008. Fast liveness checking for SSA-form programs. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization.* 35–44.

Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 265–275.

Pierre Chambart. 2016. PR #608: Whole program dead code elimination. https://github.com/ocaml/ocaml/pull/608.

Rich Felker. 2019. The musl C standard library. *Retrieved July* 30 (2019), 2019.

Michael Furr and Jeffrey S Foster. 2005. Checking type safety of foreign function calls. *ACM SIGPLAN Notices* 40, 6 (2005), 62–72.

Bolei Guo, Matthew J Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I August. 2005. Practical and accurate low-level pointer analysis. In *Proceedings of the international symposium on Code generation and optimization.* IEEE Computer Society, 291–302.

Ben Hardekopf and Calvin Lin. 2007a. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 290–299.

Ben Hardekopf and Calvin Lin. 2007b. Exploiting pointer and location equivalence to optimize pointer analysis. In *International Static Analysis Symposium.* Springer, 265–280.

Paul Havlak. 1997. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 4 (1997), 557–567.

ISO 23271:2012(E) 2012. *Information technology — Common Language Infrastructure (CLI).* Standard. International Organization for Standardization, Geneva, CH.

ISO/IEC 9899:1999, 1999. *Programming languages - C.* Standard. International Organization for Standardization, Geneva, CH.

Nick P Johnson, Jordan Fix, Stephen R Beard, Taewook Oh, Thomas B Jablin, and David I August. 2017. A collaborative dependence analysis framework. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization.* IEEE Press, 148–159.

Simon L Peyton Jones. 1992. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of functional programming* 2, 2 (1992), 127–202.

Uday P Khedker, Alan Mycroft, and Prashant Singh Rawat. 2012. Liveness-based pointer analysis. In *International Static Analysis Symposium.* Springer, 265–282.

Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5.

Chris Lattner. 2020a. Garbage Collection Safepoints in LLVM. https://llvm.org/docs/Statepoints.html. Accessed: 2020-02-11.

Chris Lattner. 2020b. Writing an LLVM Backend. https://llvm.org/docs/WritingAnLLVMBackend.html. Accessed: 2020-02-19.

Chris Lattner and Vikram Adve. 2003. *Data structure analysis: A fast and scalable context-sensitive heap analysis.* Technical Report. Citeseer.

Xavier Leroy. 2009. Google Summer of Code Proposal. https://inbox.ocaml.org/caml-list/49C79A54.5020406@inria.fr/ Accessed: 2020-02-14.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2014. The OCaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique* 54 (2014).

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java virtual machine specification.* Pearson Education.

Nicholas D Matsakis and Felix S Klock. 2014. The Rust Language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

Mozilla. 2019. Closing the gap: cross-language LTO between Rust and C/C++. http://blog.llvm.org/2019/09/closing-gap-cross-language-lto-between.html. Accessed: 2019-10-01.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. 2009. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices* 44, 3 (2009), 265–276.

Esko Nuutila and Eljas Soisalon-Soininen. 1993. *On finding the strong components in a directed graph.* Helsingin Teknillinen Korkeakoulu. Tietojenkäsittelytekniikan Laitos.

Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization.* IEEE Press, 2–14.

David J Pearce, Paul HJ Kelly, and Chris Hankin. 2007. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 1 (2007), 4.

François Pottier and Yann Régis-Gianas. 2016. Menhir reference manual. *Inria, Aug* (2016).

Gabriel Scherer. 2015. Native compiler for oCaml on System Z. https://inbox.ocaml.org/caml-list/CAPFanBEAN6BA2PhMJ00ybUZVHisu4aLHOwfqmrcH26oBsYW28g@mail.gmail.com/ Accessed: 2020-02-14.

Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*.

Brandon Simmons. 2019. GHC LLVM LTO Experiments Scratch Notes. http://brandon.si/code/ghc-llvm-lto-experiments-scratch-notes/ Accessed: 2020-02-17.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *ICFP* (2020).

Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 32–41.

Yulei Sui, Xiaokang Fan, Hao Zhou, and Jingling Xue. 2018. Loop-oriented pointer analysis for automatic simd vectorization. *ACM Transactions on Embedded Computing Systems (TECS)* 17, 2 (2018), 56.

Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benin. 2018. A transprecision floating-point platform for ultra-low power computing. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1051–1056.

Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

David A Terei and Manuel MT Chakravarty. 2010. An LLVM backend for GHC. In *ACM Sigplan Notices*, Vol. 45. ACM, 109–120.

Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience* 44, 8 (2014), 951–972.

Stephen Weeks. 2006. Whole-program compilation in MLton. *ML* 6 (2006), 1–1.

Mark N Wegman and F Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 2 (1991), 181–210.

Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. 2000. Shape analysis. In *International Conference on Compiler Construction*. Springer, 1–17.

Jeremy Yallop, David Sheets, and Anil Madhavapeddy. 2016. Declarative foreign function binding through generic programming. In *International Symposium on Functional and Logic Programming*. Springer, 198–214.