Using LLVM for Optimized Lightweight Binary Re-Writing at Runtime

Alexis Engelke, Josef Weidendorfer
Department of Informatics
Technical University of Munich
Munich, Germany
EMail: engelke@in.tum.de, weidendo@in.tum.de

***Abstract*—Providing new parallel programming models/abstractions as a set of library functions has the huge advantage that it allows for an relatively easy incremental porting path for legacy HPC applications, in contrast to the huge effort needed when novel concepts are only provided in new programming languages or language extensions. However, performance issues are to be expected with fine granular usage of library functions. In previous work, we argued that binary rewriting can bridge the gap by tightly coupling application and library functions at runtime. We showed that runtime specialization at the binary level, starting from a compiled, generic stencil code can help in approaching performance of manually written, statically compiled version.**

In this paper, we analyze the benefits of post-processing the re-written binary code using standard compiler optimizations as provided by LLVM. To this end, we present our approach for efficiently converting x86-64 binary code to LLVM-IR. Using the mentioned generic code for arbitrary 2d stencils, we present performance numbers with and without LLVM post-processing. We find that we can now achieve the performance of variants specialized by hand.

***Keywords*-High Performance Computing; Dynamic Code Generation; Dynamic Optimization; Binary Transformation**

I. Introduction

In High Performance Computing (HPC), the usage of programming languages which get compiled to native code such as Fortran or C++ is common, as well as the usage of low level libraries which are relatively thin abstractions over hardware. An example for the latter is MPI [1]. The reason for favoring compiled languages and low level libraries is the need for predictable and stable runtime behavior, which is essential in being able for parallel application codes to scale well even when using thousands of processors. On the one hand, this low level of programming allows for sophisticated optimizations tuned for specific hardware features. But on the other hand, productivity is heavily limited compared to software development in other IT domains. For example, HPC programmers do not take advantage of memory safety as provided by managed environments (Java, .NET).

Due to the low productivity, strategies are desperately needed which allow both for higher abstraction and yet keep the ability for tuning of details. Promising concepts such as PGAS (partitioned global address space) were proposed. However, they often come with new programming languages or language extensions [2]–[4]. Compiler support is very helpful in reducing any overhead of provided abstractions. But with new languages, porting of existing application code is required, being a high burden for adoption with old code bases. Therefore, to provide abstractions such as PGAS for legacy codes, APIs implemented as libraries are proposed [5], [6]. Libraries have the benefit that they easily can be composed and can stay small and focused. However, libraries come with the caveat that fine granular usage of library functions in inner kernels will severely limit compiler optimizations such as vectorization and thus, may heavily reduce performance.

To this end, in previous work [7], we proposed a technique for lightweight code generation by re-combining existing binary code. We presented our open-source prototype DBrew (Dynamic Binary REWriting)[1]. The main features are (1) tight coupling of separately compiled functions (e.g. from application code and/or different libraries) by aggressive inlining and (2) specialization of generic code with information known at runtime. For example, variable function parameters can be configured to have known values, enabling constant propagation, dead-code elimination (due to runtime knowledge), and full loop unrolling. Specialization is useful for minimizing the runtime overhead of abstractions: how to best handle different runtime properties (input data, exact target architecture with e.g. varying cache sizes, specific features of I/O devices) can be covered in generic code. This gets specialized into a concrete implementation when executed by the proposed rewriting technique[2]. Rewriting at the binary level has several advantages. We can inline code from existing libraries (even commercial) without the need for source code, which makes the technique easy to deploy in different HPC environments. Furthermore, for the supported code transformations, staying at the binary level of a given ISA is fine: either instructions can be copied over into the newly generated code, partly replacing operands with known constants, or they simply disappear if all input parameters are known. We showed that the performance of the resulting code almost matched the performance of the variant with the specialization done statically by hand.

In this paper, we study the performance implications of

[1] Available at https://github.com/lrr-tum/dbrew

[2] C++ templates allow for variants, but may have code explosion issues.

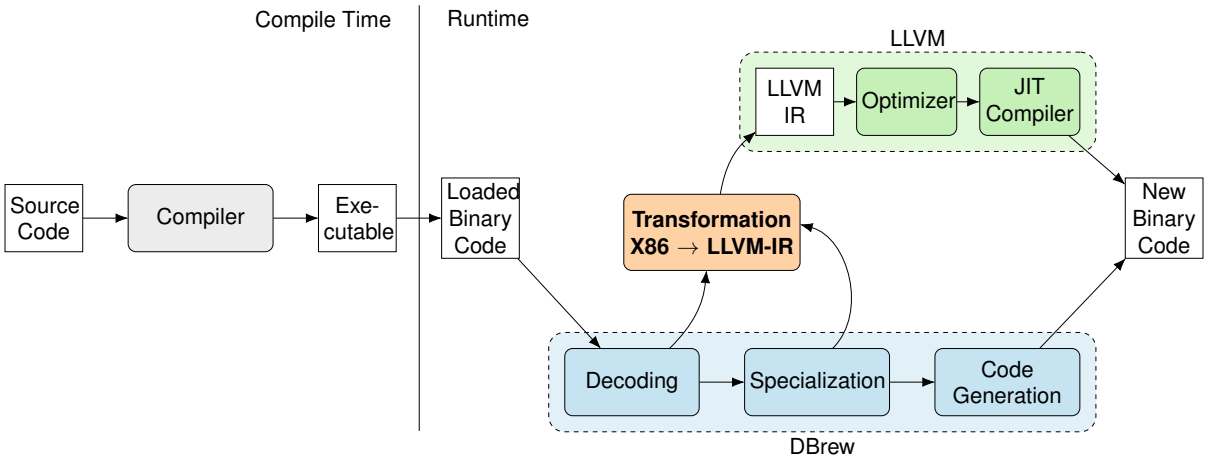


Figure 1: Overview on the process of binary re-writing in combination with LLVM. The decoded and optionally specialized binary code is transformed to LLVM-IR before standard compiler optimizations of LLVM can be applied.

our binary re-combination/specialization technique in more detail. We want to understand if there are inherent overheads within the resulting code which makes it impossible to approach statically compiled, manually specialized code. To this end, we forward code from DBrew into standard optimization passes of an established compiler backend, as shown in Fig. 1. For this, we use the C-API and JIT (Just-In-Time) compilation features of LLVM [8]. We use the same example as in our previous work (a generic 2d Jacobi code). We compare the performance of various variants and the effort required. While having the option to use LLVM optimization passes is nice in the scope of DBrew, we also did this experiment for two other reasons. On the one hand, we now have an infrastructure to understand for given use cases, which of the optimization passes are essential and which can be neglected. This will guide us to further improve the backend of DBrew with simple forms of essential optimization passes without the heavy LLVM resource overhead (both in space and time). On the other hand, we wanted to understand the importance of meta information for standard compiler optimization passes, as this is not readily available in the binary code from DBrew.

The paper is structured as follows: first, we shortly revisit the DBrew prototype in the next section. Then we describe in detail the required steps for converting x86-64 binary code into LLVM-IR. In Sec. IV, we describe specific aspects for the conversion in the context of specialization. Afterwards, results are presented for the 2d Jacobi case. Here, we also show excerpts of x86 code as produced by DBrew as well as the code when post-processed by LLVM. We discuss our findings, and after mentioning related work, we conclude the paper with ideas for future extensions.

II. DBrew: Dynamic Binary Re-Writing

DBrew is our prototype for the proposed lightweight code generation technique consisting of re-writing and re-combining pieces of compiled binary code. It currently works with the x86-64 ISA and Linux. The basic approach is to generate drop-in replacements of existing functions. A request for re-writing returns a function pointer with exactly the same function signature as the original code. We expect that re-writing may fail: each of the internal steps "decoding", "emulation" (if all input to an instruction is known), and "encoding" may not be covered for a given instruction in the instruction stream. This will trigger an internal error. DBrew has a default error handler which simply will return the original function to ensure correctness. However, the user can provide a custom error handler, which for example may iteratively enlarge the buffer space available for the generated code and restart the re-writing. Fig. 2 shows basic DBrew usage.

```
int func(int a, int b) { ... }
typedef int (*func_t)(int,int);
func_t newfunc;

int main() {
  // call original function
  int x = func(1,2);
  // new rewriter config for func
  dbrew_rewriter* r = dbrew_new(func);
  ... // configure rewriter
  newfunc = (func_t) dbrew_rewrite(r);
  // call rewritten version
  int x2 = (*newfunc)(1,2);
}
```

Figure 2: Basic usage of DBrew.

There exist different configuration options to influence the behavior of rewriting the binary code of a compiled function. On the one hand, the amount of resources available to rewriting can be limited (e.g. for decoding, duplicated

```
// set config and rewrite func
r = dbrew_new(func);
dbrew_setpar(r,1,42);
dbrew_setmem(r,start,end);
newfunc = (func_t) dbrew_rewrite(r);
// par 1: uses 42 instead of 1
int x2 = (*newfunc)(1,2);
```

Figure 3: Declaring known values for specialization.

variant generation, depth of allowed inlining, or space for generated code). On the other hand, for specialization, one can configure that some values which get used in a function should be fixed to a given constant. To specify which values should be assumed to have a fixed value, DBrew allows to specify memory ranges. The values stored within such regions are assumed to be fixed. Apart from that, DBrew configuration relates to function parameters. The user can specify fixed values for parameters of the function to be rewritten, or fixed values which are referenced through function parameters (by specifying that a parameter is a pointer to fixed values). Fig. 3 shows the specification of a specialization configuration. In this example, parameter 1 is fixed to value 42, and all values in a given memory range are assumed to be fixed. The latter replaces memory references by immediates in the generated instruction stream.

All configuration options of a DBrew rewriter object rely on the C ABI (Application Binary Interface) of the target ISA and operating system. An ABI specifies in detail how function parameters are passed (in registers or on the stack) and how register content is preserved over function invocations (i.e. whether the caller or callee has to save registers on the stack). Only with all compilers adhering to an ABI, object files generated by different compilers actually can be linked together. Thus, regarding DBrew, we assume that any function to be rewritten (or for whose inlining a parameter fixation is specified) also will adhere to the platform ABI. Only this allows DBrew to map parameter numbers given in the configuration to the actual register or stack space used[3].

For this paper, we additionally allow to configure the code generation backend. This backend does the new LLVM-IR transformation, triggers LLVM optimization passes, and uses the JIT code generator.

III. Converting x86 to LLVM-IR

The transformation of x86-64 assembly to LLVM-IR is designed to work on the function level. To perform the transformation with minimal overhead of the resulting code, we make some assumptions, mostly inferred from the assembly emitted by current C compilers (GCC or Clang). For example, we assume the Linux ABI (System V) and for floating point support, we only cover SSE instructions for now.

[3]For x86-64, a parameter slot typically uses 64 bit. However, the ABI allows parameters to extend to multiple 64 bit slots. Thus, the mapping from parameter number to the index of the 64 bit slot (either register or stack space) is not always 1:1.

A. Functions

As functions are a central construct in the LLVM-IR, the signature of the function and the calling convention has to be known. This information is required to construct a mapping of the function parameter in the LLVM-IR to the register or stack address where the function expects the argument. Likewise, when the function returns, the register containing the return value has to be determined.

B. Basic Blocks

A function in LLVM-IR must be split up into basic blocks, where each basic block is a sequence of LLVM-IR instructions which do not modify the control flow and ends with a branch to other basic blocks in the same function or a return instruction. The original x86 function code also can be split in basic blocks of machine instructions, where each basic block ends with an instruction which changes the control flow. Such kind of instructions include a jump, a conditional jump, a call or a return instruction.

An x86 `call` instruction is translated to a `call` instruction in the LLVM-IR. This has the assumption that the target of the call is actually a real function. However, standard compilers should only emit a call if this is the case. While this approach requires the called function to be at least declared with an appropriate signature in the LLVM-IR as the calling convention has to be applied, this approach leaves the decision on inlining to the LLVM optimizer. For consistency, an x86 `ret` instruction is transformed to a return instruction in the LLVM-IR. This assumes that the function does behave well and uses the return for a proper function return, which however should be the case for compiler generated code.

In case of a jump, a basic block can have at most two exits: one *branch* exit, which is taken on a jump or conditional jump whenever the condition is fulfilled. The other exit is the *fall-through* exit, which is the basic block which immediately follows the basic block. Jumps are in general transformed to branch instructions in the LLVM-IR. Indirect jumps are currently not supported as the jump target might be unknown at the time of the transformation.

When decoding a function, we can ensure that each instruction in the original function is part of exactly one basic block. When a jump points to an instruction which is not the first instruction of a basic block, the corresponding basic block is split up. The de-duplication of instructions allows for better optimization as the existing LLVM passes might not be able to identify parts of basic blocks as identical, resulting in larger and potentially less efficient code.

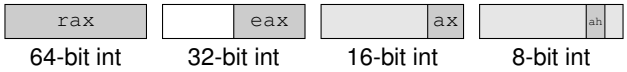
C. Registers

The x86-64 architecture has 16 general purpose registers with a length of 64-bits each, an instruction pointer and, depending on the available extensions, up to 32 vector registers with the AVX-512 extensions. Currently, only the SSE vector registers can be used, though.

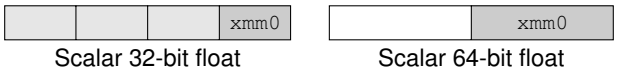
Instructions may access a register in different *facets*, as depicted in Figure 4. For example, an instruction can access a whole 64-bit general purpose register, but also the lowest 16-bit of the same register. Furthermore, a general purpose register can also be used as integer or pointer. The `addsd` instruction requires the lowest 64-bits of a vector register as a single 64-bit floating-point value, whereas the `addps` requires the lowest 128-bits of the register as vector of 32-bit floating-point values. In contrast, the `paddq` instruction requires the same lowest 128-bits of the register as a vector of 64-bit integers. Therefore, when an instruction accesses the contents of a register, it also has to specify the required facet of the register. A register is modeled in LLVM-IR by an integer of the appropriate length, i.e. an `i64` for each general purpose registers as well as the instruction pointer and an `i128` for each SSE vector register. This bitwise representation is a logical consequence of the way the registers are stored in hardware. Each register is a reference to the corresponding value (SSA variable [8]) in the LLVM-IR, or `undef` if the register has not been used. The mapping of the register to the corresponding value is stored for each basic block separately. It turned out that the LLVM optimizer is not able to eliminate the casts between the accessed facets and the integer representation, leading to a high overhead for more complex codes. Therefore, we additionally cache the values of the facets as produced by the instructions. To ensure that all registers have the same value as in the original program, each basic block has a set of Φ-nodes at the beginning, where the values of the registers in all facets of the predecessors are merged.[4] We note that each basic block has as significant amount of Φ-nodes, which are mostly unused. These unused nodes will be removed by the optimizer.

This schema of representing registers has various advantages: it is straight-forward and it allows to access the different facets of the registers with a minimal amount of casts. As a side-effect, this also allows to store both a pointer and an integer facet of the same register. This is important as pointer arithmetic and integer arithmetic are handled differently in the LLVM-IR but similar at machine code level. The LLVM documentation recommends to construct memory addresses using `getelementptr` (GEP) instructions instead of `add` instructions where possible to improve the pointer alias analysis [9]. Instructions which can be used for pointer and integer arithmetic, e.g. `add` or `lea`, can set both facets using the appropriate LLVM arithmetic, allowing for more optimizations as a consequence of better analysis.

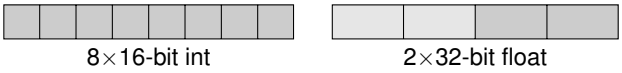
[4]Refer to [8] for a definition of Φ-nodes in the LLVM-IR



(a) Single Element facets (dark gray) of General Purpose registers can be accessed via a `trunc` instruction. For the high-byte registers (e.g. `ah`), an additional shift operation is required. For a write-back of 32-bit facets, the upper half of the register is zeroed (white), while for 8-bit or 16-bit facets the untouched part (light gray) has to be preserved via bit masking.



(b) Single Element facets (dark gray) of the SSE vector registers are extracted with an `extractelement` instruction from an appropriate vector type. On a write-back, the untouched part is in general preserved. However, some instruction (e.g. `movq`) may also zero the untouched part.



(c) Vector facets are constructed using a cast and, if necessary, extracted using a `shufflevector` instruction. When a vector covering the full register is written back, a simple cast is sufficient, otherwise up to two `shufflevector` instructions are necessary. Note that only the SSE registers have vector facets.

Figure 4: Registers can be accessed either as a single element or as a vector. The actual instruction defines the facet in which the register is accessed and may also specify whether untouched parts of the registers are preserved or zeroed.

1) Loading a Register: Single element facets from general purpose (Figure 4a) registers are extracted via a `trunc` instruction. The 8-bit "high" registers (`ah`, etc.) require a logical shift first. Single element facets from the SSE registers (Figure 4b) are handled using an `extractelement` instruction on a casted vector as the SSE registers are by design vector registers. This has the advantage that the LLVM optimizer is able to figure out the source of the value within the vector for further optimizations. If a `trunc` were applied, the cast from a vector to an integer which gets truncated would not be optimized. When accessing vector facets from the SSE registers (Figure 4c), the appropriate elements are extracted using a `shufflevector` instruction. This instruction is handled well by the optimizer and in many cases eliminated throughout the optimization process.

2) Storing to a Register: When a value is written in a general purpose register, the length of the register determines whether the higher part of the register gets zeroed due to the implementation in the processors. When a 64-bit value is stored, all parts are overwritten. A 32-bit value causes the upper half to be zeroed, this is modeled with a `zext` instruction. For 8-bit and 16-bit registers the upper part has

```
                sub rax, 1

%rax.1 = sub i64 %rax.0, 1
```

```
             mov eax, [rbp - 0xc]

%ptr1 = inttoptr i64 %rbp to i32*
%ptr2 = getelementptr i32, i32* %ptr1, i64 -3
%eax = load i32, i32* %ptr2, align 4
%rax = zext i32 %eax to i64
```

```
               addsd xmm0, xmm1

%v.0 = bitcast i128 %xmm0 to <2 x double>
%el.0 = extractelement <2 x double> %v.0, i32 0
%v.1 = bitcast i128 %xmm1 to <2 x double>
%el.1 = extractelement <2 x double> %v.1, i32 0
%add = fadd double %el.0, %el.1
%v.2 = bitcast i128 %xmm0 to <2 x double>
%ins = insertelement <2 x double> %v.2, double %add, i64 0
%xmm0.1 = bitcast <2 x double> %ins to i128
```

Figure 5: Examples of transforming individual x86-64 instructions to LLVM-IR. Some instructions translate straightforward, others require eight or more LLVM-IR instructions. Introduced overhead often is removed at a later stage.

to be preserved via bit masking. For some 8-bit registers (e.g. `ah`) an additional shift is required.

When a value is stored in a vector register, we can distinguish two cases: First, most SSE instructions preserve the untouched part of the register. When inserting into a vector, we can model the store with two `shufflevector` instructions: the first instruction enlarges the vector to insert to the vector length of the register to make it suitable for the second instruction. The other `shufflevector` instruction merges the two vectors together, with respect to the original size of the first vector. If the value to insert is a single value, we can simply use an `insertelement` instruction, because there is only one value to insert. Second, a few SSE instructions (e.g. `movq`) set the upper, untouched part to zero. This can be modeled by a `shufflevector` instruction or an `insertelement` instruction with a `zeroinitializer`. This is advantageous over a simple zero-extension as the LLVM optimizer has problems handling mixed integer and vector operations on the same value.

D. Flags

The six status flags of the flags register which are frequently used by applications are modeled as an `i1` each, similar to the way registers were stored. This is a contrary approach to the modeling in hardware, where multiple flags are stored in one `rflags` register. As string operations are currently not supported, the direction flag is not modeled yet, but can be added later. As we do not intend to support handling of system software, the system flags are ignored. The zero flag, the sign flag and the carry flag can be

```
mov rax, rdi
cmp rdi, rsi
cmovlt rax, rsi
ret
```

(a) Original Code

```
%cmp = sub i64 %rdi, %rsi
%sf = icmp slt i64 %cmp, 0
%of.tmp.1 = xor i64 %cmp, %rdi
%of.tmp.2 = xor i64 %rsi, %rdi
%of.tmp.3 = and i64 %of.tmp.1, %of.tmp.2
%of = icmp slt i64 %of.tmp.3, 0
%lt = xor i1 %sf, %of
%rax = select i1 %lt, i64 %rsi, i64 %rdi
ret i64 %rax
```

(b) Optimized LLVM-IR generated without Flag Cache

```
%lt = icmp slt i64 %rdi, %rsi
%rax = select i1 %lt, i64 %rsi, i64 %rdi
ret i64 %rax
```

(c) Optimized LLVM-IR generated with Flag Cache

Figure 6: Effect of a Flag Cache preserving the semantics of a `cmp` instruction on a code computing the maximum of two registers. Without a flag cache, the bitwise operations on the flags are not replaced with an equivalent comparison, leading to missed optimizations and less efficient code.

computed using basic integer comparisons. The overflow flag can be either computed via bitwise operations or via the LLVM intrinsics for overflow handling. The latter, however, is discouraged as the optimizer does not analyze intrinsics well [9]. The parity flag employs the `llvm.ctpop.i8` intrinsic to count the set bits in the lowest byte of the result. For this flag an intrinsic is involved, but as the parity flag is rarely used this will likely get removed during the optimization. The auxiliary carry flag is computed using bitwise operations, but is also rarely used.

In the x86 architecture, signed integer comparisons like *less-than* or *greater-or-equal* are performed using binary operations on the flags. However, it turned out that LLVM is not able to reduce them to the correct logical comparison, resulting in less efficient code and missed optimizations. Therefore, a *flag cache* which stores the operands of the latest `cmp` instruction has been implemented. If a signed comparison is made after such an instruction and the flag cache is valid, the appropriate comparison predicate is used. The effect of the flag cache turned out to be significant, as shown in Figure 6. Obviously, if the flags are modified by other instructions, the flag cache is invalidated.

E. Memory Operands

A memory address in the x86 architecture is a summation of up to three different components: a base register, a scaled index register where the value is multiplied with 1, 2, 4, or

8, and a constant offset. For the register operands of an address generation, the pointer facet is used when available. Otherwise, an `inttoptr` instruction has to be used at the cost of less optimizable code. We note that this should occur rarely with compiler generated code. The operands are connected using one or more GEP instructions. Constant addresses are reduced to a global base pointer instead of using `inttoptr` to improve pointer analysis, as recommended in the LLVM documentation [9]. The base pointer is set to the first constant address found. The x86-64 architecture supports segment overrides for the `gs` or `fs` segments, which are used for thread-local storage and system data structures. These are handled by constructing the pointer in the LLVM address spaces 256 and 257, respectively, which correspond to both segments by definition.

All loads and stores are marked as non-volatile in the LLVM-IR, implying that reordering or elimination of these instructions may occur. Support for volatile memory operations may be added in future, but requires a more complex API as it is not possible to extract information from the assembly code.

F. Stack

One limitation of the abstraction provided by LLVM consists in the lack of direct access to the stack. As a consequence, a virtual stack has to be allocated via an `alloca` instruction in the entry basic block. However, as the virtual stack cannot grow dynamically, the size of the used part of the stack may not exceed a user-specified limit. The required contents of the stack, e.g. function parameters which are not passed in registers, have to be copied into the virtual stack as LLVM abstracts from the calling convention. When the stack pointer is changed using the `push` or `pop` instructions, a GEP instruction is employed.

IV. Specifics for Binary Specialization

The goal of the transformation of x86 binary code to LLVM-IR is to replace the DBrew code generator with a more advanced optimization and code generation backend. Thus, after the binary code specialized by DBrew is transformed to LLVM-IR, the standard optimization pipeline with level 3, similar to the `-O3` compiler option, is applied. The optimizations are also necessary to remove the overhead introduced by the transformation. Optionally, floating-point optimizations can be enabled similar to the `-ffast-math` compiler flag. The optimized LLVM-IR is compiled to new binary code using the JIT compiler of LLVM.

However, for some functions, the specialization via parameter fixation may take place at the level of LLVM-IR directly. To achieve this, the original, unmodified function is transformed to LLVM-IR first. Then, a new function which calls the original function with the fixed parameter is created. By marking the original function as *always-inline*, the function will be inlined as part of the LLVM optimization pipeline as long as the function is not recursive. The specialization is also done using the optimization passes as the constant parameter will be propagated through the now-inlined function. To make this approach work for memory regions, some additional effort is needed. As it is not possible to specify a given pointer as constant in the LLVM-IR, the content of the constant memory region has to be copied into the LLVM module as a global constant. Currently, the size of the constant memory area has to be specified explicitly. Furthermore, as the data type of the values in the memory region is not known, nested pointers will not be marked as constant and therefore, in contrast to DBrew, no further specialization can take place.

```
#define SZ 649 // matrix side length

// flat version
struct FP { double f; int dx, dy; };
struct FS { int ps; struct FP p[]; };

struct FS s4 = {4, {{-1,0,.25},
 {1,0,.25},{0,-1,.25},{0,1,.25}}};

void apply_flat(struct FS* s, double *m1,
            double* m2, int index) {
  double v = 0.0;
  for(int i=0; i<s->ps; i++) {
     struct FP* p = s->p + i;
     v += p->f * m1[index+p->dx+SZ*p->dy];
  }
  m2[index] = v;
}

// sorted version
struct SP { int dx, dy; };
struct SG { double f;
         int ps; struct SP p[]; }
struct SS { int gs; struct SG p[]; };
...
```

Figure 7: Generic 2d stencil computation code (element kernel) with the stencil given as a data structure.

V. Case study: Specializing a Generic Stencil

To evaluate the effects of LLVM integration, we use the same example as in our previous work [7]. We use DBrew to specialize a generic 2d stencil computation. The resulting code is to be used in a loop over the matrix cells. The code in Figure 7 shows the definition of a 4-point stencil using a (1) *flat* data structure and corresponding generic code, as well as a (2) *sorted* data structure which groups stencil points by coefficient (the corresponding generic code involves two nested loops; left out due to length).

We configure the rewriter to assume the stencil to be fixed. For this, we mark the first parameter to be a pointer to known fixed data (this applies recursively if pointers would have been used in `struct FS` or `struct SS`, respectively).

```
pxor xmm1,xmm1
pxor xmm1,xmm1
mov rax, -1
add rax, rcx
movsd xmm0, [rsi + 8 * rax]
mov rax, 1
add rax, rcx
addsd xmm0, [rsi + 8 * rax]
// ...
mulsd xmm0, [0x14c47d8]
addsd xmm1, xmm0
movsd [rdx + rcx * 8], xmm1
ret
```

```
movsd xmm0, [rsi + 8 * rcx + 8]
addsd xmm0, [rsi + 8 * rcx - 8]
addsd xmm0, [rsi + 8 * rcx - 8 * 649]
addsd xmm0, [rsi + 8 * rcx + 8 * 649]
mov rax, 0x14c47d8
mulsd xmm0, [rax]
movsd [rdx + 8 * rcx], xmm0
ret
```

Figure 8: Comparison of codes generated by plain DBrew (top) and after LLVM optimization (bottom).

We study two different types of codes: for the *element kernel* we rewrite the stencil function, and call the rewritten code in the inner loop over the matrix elements. For the *line kernel*, we wrap the kernel call into a loop over one line of the matrix, and rewrite the complete loop. This ensures that the stencil kernel gets inlined as loop body.

VI. Results

We transform different compiled codes into LLVM-IR, apply optimizations with the standard pipeline at optimization level 3 with floating-point optimizations enabled, and measure the running time of the original and the modified code. For the code which is used as input to the transformation we can also use the code which was produced by an optimization using DBrew. If DBrew is applied on the line kernel, the actual computation of an element is moved to a separate function which is inlined by DBrew to prevent loop unrolling of the loop over the elements of a line. Furthermore, we can use the parameter fixation as described in Section IV. This, however, does not make sense in combination with DBrew as a specialization on a parameter is only needed one time. In total, we have the following modes:

- Original: unmodified, as produced by the compiler.
- LLVM transformation: the code is transformed into the LLVM-IR, optimized and compiled back to assembly code. This is basically an identity transformation, which does not change the behavior of the code and is ideally as fast as the original code.
- LLVM transformation with fixation: the code is transformed into LLVM-IR and back to assembly code as above with the difference that the parameter which specifies the generic stencil is fixed.
- DBrew: specialized by rewriting through DBrew.
- DBrew combined with LLVM transformation: the assembly code specialized by DBrew is transformed into LLVM-IR, optimized and then compiled back into assembly code.

The running time is measured by computing multiple iterations of a Jacobi approximation, where the computation of the stencil is performed using the codes obtained by applying the methods described above. The matrix has a size of 9×9 with 80 interlines, resulting in a matrix of size 649×649, which requires 3.2 MB of memory. In total, two matrices are used for the Jacobi iteration. To achieve significant running times, we perform 50,000 iterations. Beside the actual computation of the values, the measured running time also includes the loop which is used to iterate over the matrix and the overhead of the function call for the computation. In addition to the running time, we also measure the time needed to perform the optimization, because the transformation and specialization is designed to happen at runtime.

The performance is measured on a computer with an Intel Xeon E3-1270 v3 processor (Haswell) clocked at 3.5 GHz (3.9 GHz turbo) with 8MiB L3 cache. The machine is running Linux Mint 17 (64-bit) with Linux kernel 4.2.0. The benchmark code was compiled with GCC 5.4.1 with the `-O3 -march=native -mno-avx` options and linked statically against LLVM 3.7.1 and DBrew[5]. We disable AVX for the original code and for the LLVM generated code as this extensions are not supported yet by the transformation.

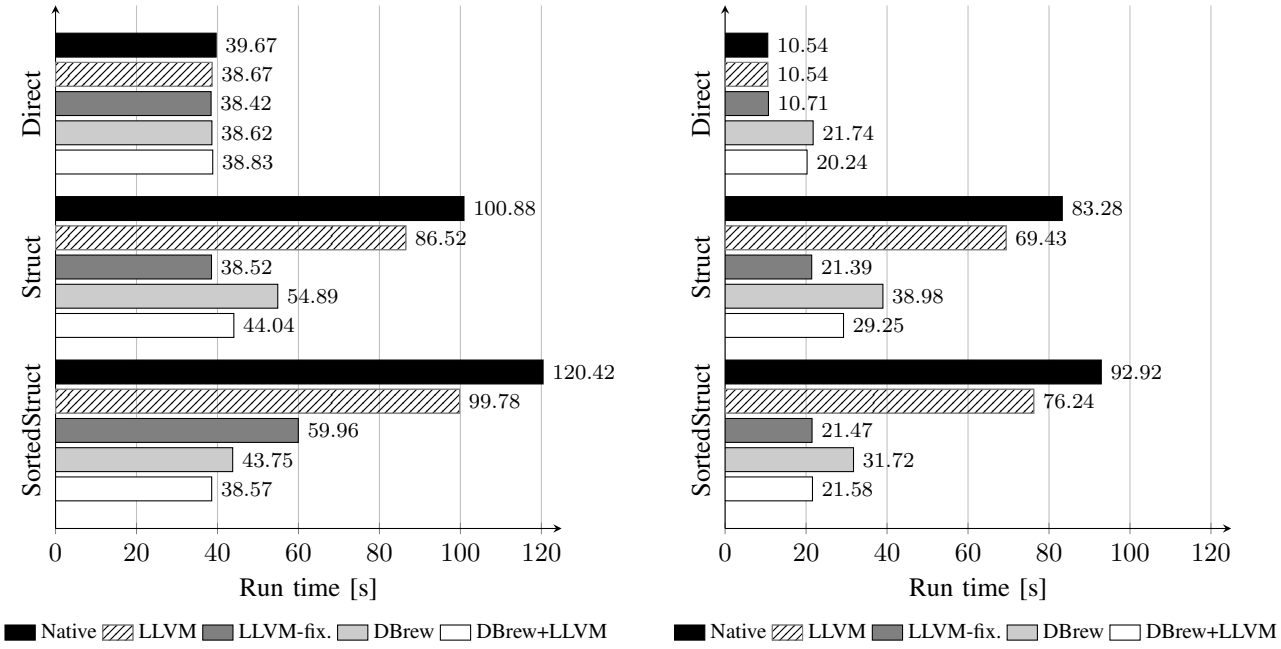
A. Runtime Element Kernel

The runtimes of the element kernel is shown in Figure 9a.

Direct: For the variant with the hard-coded stencil, we can observe no major differences between the different modes. This also implies that the transformation of the simple code to LLVM-IR does not involve any overhead.

Flat Structure: Using a generic structure for the stencil obviously leads to slower code than the hard-coded stencil. Applying the LLVM identity transformation on this code leads to slightly faster code. This is unexpected and implies that the original code produced by GCC is not optimal. The only difference is that GCC uses multiple `lea` instructions whereas LLVM employs a single `imul` instruction for the multiplication in the index computation (see above). However, we can also observe that the transformation to LLVM-IR does not involve any overhead in this case. The parameter fixation at the level of LLVM-IR leads to the same

[5]Commit e99c2c81 at github.com/lrr-tum/dbrew branch "llvm-hips17"



(a) Running times with the element kernel.

(b) Running times with the line kernel.

Figure 9: Comparison of the running times of the codes produced by the LLVM transformation (with and without fixation), DBrew specialization and DBrew with LLVM code generation back-end.

performance as the hard-coded stencil. The DBrew specialization has some overhead as no advanced combination of instructions or floating-point optimizations are performed, see Figure 8. Also, the combination of DBrew and LLVM has some overhead because the information about constant multiplication factors is not forwarded.

Sorted Structure: For the sorted structure, the LLVM identity transformation is also slightly faster than the original code as a consequence of non-optimal instruction selection by GCC. The parameter fixation at LLVM-IR level has a high overhead. This was expectable as the sorted structure has *nested pointers*, which are currently not handled. Similar to the findings described in [7], the DBrew specialization has a lower overhead as for the flat structure because the redundant multiplications are eliminated. Applying the LLVM optimizations on the top of the DBrew specialization again leads to code with the same performance as the hard-coded stencil.

B. Runtime Line Kernel

The runtimes of the line kernel are shown in Figure 9b.

Direct: For the hard-coded stencil GCC employs vectorization, where, depending on the alignment, the first or last element is computed separately. Applying the LLVM transformation leads to code with similar performance. The code produced by DBrew is significantly slower as the original code does not involve vectorization and unoptimized move instructions. Involving LLVM on the code produced by DBrew removes these move instructions, but does not lead to vectorized code.

Flat Structure: Similar to the element kernel, the LLVM identity transformation produces significantly slower code for reasons of missed optimizations across basic blocks. Specialization at LLVM-IR level improves the performance, but is still slower than the code with the hard-code stencil as vectorization is not performed. Involving LLVM on the code produced by DBrew leads to performance improvements, but does not reach the performance of the LLVM-IR specialization as information about constant memory regions is not preserved.

Sorted Structure: For the sorted structure, we can observe similar results to the flat data structure, with the difference that the LLVM transformation applied on the top of DBrew leads to the same performance as the specialization at LLVM-IR level.

We note that in any case with specialization no vectorized code is generated by LLVM as the loop analysis passes of LLVM consider vectorization as non-beneficial for this loop. However, LLVM performs vectorization on the original source code. Therefore, we assume that missing meta-information leads to this missed optimization. When forcing vectorization (via the `-force-vector-width=2` command line flag), we can observe that the loop vectorized by LLVM is only 23% slower than the loop vectorized by GCC at compile-time. The difference is caused by unaligned memory accesses: while GCC includes alignment checks

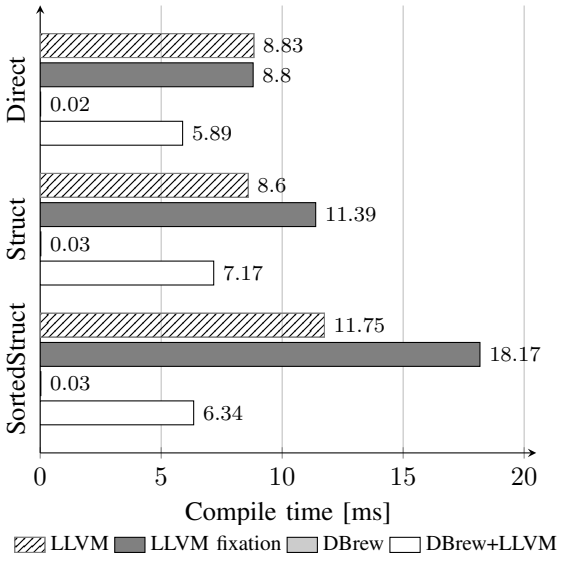


Figure 10: Average transformation times of the different modes when performing 1000 compiles on the line kernel. DBrew uses less than 0.05ms in any case while the time required by LLVM increases with the code complexity.

to perform aligned loads where possible, LLVM only uses unaligned accesses, which have a higher latency.

C. Compile Time

To analyze the compile times of the different code transformation modes, each transformation is applied 1000 times to achieve significant running times. The results are shown in Figure 10. We can observe that the time used for a single LLVM transformation is below 20 milliseconds for the more complex codes as more optimizations are performed. We can also observe that a standalone DBrew transformation requires significantly less time as less complex transformations are made. The additional overhead introduced by the LLVM transformation can, however, be considered as neglectable given the observed performance improvement.

D. Discussion

First, we note that with our approach, the semantic reconstruction of conditions in the transformation from x86-64 to LLVM-IR code works quite well. From our results, we find that the transformation has almost no overhead for simple codes. Unfortunately we could not convince the LLVM backend to generate vectorized code as the loop is considered as non-profitable. Most probably, this is caused by missing information about the actual type of the data stored in registers and memory. Another source of overhead are missing semantics, such as missing information about alignment of data or about constant memory regions.

Even though the LLVM transformation had no overhead in our example, we expect that some overhead might occur for more complex codes which stronger differ from the LLVM-IR generated by compilers, especially Clang. Support for other instructions and extensions like AVX can be added easily in future by simply extending the size of the vector registers and including appropriate facets.

VII. Related Work

Our proposal for lightweight code generation allows application programmers to use dynamic code generation in a controlled, explicit way. This is different from tools for observing the execution of another executable by inserting analysis code before [10] or during execution [11]–[13], but similar to functionality as found in [14] which proposes an extension of C using annotations. DyC allows parts of the C code to be transferred to runtime for allowing deferred specialization. DeGoal [15] is a recent proposal to integrate dynamic code generators at runtime. It provides programmers a specification language for controlling what the generator should do, including C code "compilettes" to be used by the generator as precompiled building blocks. Our approach actually tries to be a minimalisic version of this. However, we allow arbitrary compiled functions to be used as building blocks. Other language specific extensions to allow specialization at runtime are SEJITS [16] (replacing selected Python functions by dynamically generated native code at runtime) or Graal [17], a API proposed for Java to control dynamic compilation. In contrast to DBrew, all mentioned proposals do not directly work on binary level.

Regarding the work presented in this paper on x86 to LLVM-IR transformation, we know two similar projects: McSema [18] is a project that tries to reconstruct the semantics of binary code and transform it to LLVM-IR, meant for reverse-engineering and de-obfuscation of binary code. Fcd[6] has similar goals, but constructs C++ code for each instruction, which is lowered to LLVM-IR by Clang. In contrast to our approach, the existing systems do significantly less effort in optimizing the performance of the transformed code. The registers are only stored in the bitwise representation without type information, leading to missed optimizations for pointers and vector registers. Reconstructing this information and performing other optimizations (e.g. condition semantics) on the generated LLVM-IR would require significantly more effort. In general, we found that both, McSema and Fcd, are lacking the required flexibility and efficiency to produce performant LLVM-IR at runtime. We note that our x86-64 to LLVM-IR transformation can be used for reverse-engineering and de-obfuscation as well.

VIII. Conclusion

In this paper, we presented our work on transformation of x86-64 binary code to LLVM-IR. This is used to post-process the output of our previously proposed technique

[6]http://zneak.github.io/fcd/

for lightweight code generation at runtime (DBrew), by recombining and specializing pieces of code at the binary level. The extension presented in this paper had different goals: first, we wanted to understand how far we can get by post-processing the generated binary code using a state-of-the-art compiler backend. For our test case, a generic 2d stencil code, we found that we can approach the performance of specialized variants done by hand quite well. The second goal of our LLVM transformation was to come up with a way to understand which specific optimization passes are most essential for high performance. We did not actually show such results in this paper, but we can do this kind of studies now. We hope that can identify a small subset of optimizations we would like to implement as lightweight post-processing for DBrew without the heavy cost of LLVM. Finally, from the experiments shown in this paper we got important insights regarding lost meta-information at the binary level: a lot of LLVM optimization passes expect information such as type information which is not readily available from DBrew. To really take advantage of a lot of LLVM passes (such as vectorization), we need to have meta-information available. However, DBrew already has a lot of ways to control rewriting at the binary level – it seems to be more effective to provide explicit APIs, such as a way to transform scalar kernels into vectorized kernels.

REFERENCES

[1] M. P. I. Forum, "MPI: A Message-Passing Interface Standard Version 3.0," 2012.

[2] U. Consortium, "UPC language specifications, version 1.3."

[3] H. Zima, B. L. Chamberlain, and D. Callahan, "Parallel programmability and the Chapel language," *International Journal on HPC Applications, Special Issue on High Productivity Languages and Models*, vol. 21, no. 3, pp. 291–312, 2007.

[4] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, D. Grove, A. Shinnar, M. Takeuchi, O. Tardieu, P. M. I. S. Agarwal, B. Alpern, D. Bacon, R. Barik, B. Blainey, B. Bloom, P. Cheng, J. Dolby, S. Fink, R. Fuhrer, P. Gallop, C. Grothoff, H. Horii, K. Kawachiya, A. Kielstra, S. Ko, I. Peshansky, V. Sarkar, O. Solar-lezama, S. Alex, E. Spoon, S. Sur, T. Suzumura, C. V. Praun, L. Unnikrish, P. Varma, K. N, I. Venkata, J. Vitek, H. C. Wang, S. Zakirov, Y. Zibin, R. K. Shyamasundar, V. T. Rajan, F. Tip, A. Vaziri, and H. Xue, "X10 language specification version 2.5," 2014.

[5] T. Alrutz, J. Backhaus, T. Brandes *et al.*, "GASPI: A Partitioned Global Address Space programming interface," in *Facing the Multicore-Challenge III*, ser. Lecture notes in computer science, vol. 7686. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 135 – 136.

[6] K. Fürlinger, C. Glass, A. Knüpfer, J. Tao, D. Hünich, K. Idrees, M. Maiterth, Y. Mhedheb, and H. Zhou, "DASH: Data structures and algorithms with support for hierarchical locality," in *Euro-Par 2014 Workshops (Porto, Portugal)*, 2014.

[7] J. Weidendorfer and J. Breitbart, "The case for binary rewriting at runtime for efficient implementation of high-level programming models in HPC," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 376–385.

[8] LLVM-Project, "LLVM language reference manual, for LLVM version 3.8.1." 2016, http://llvm.org/releases/3.8.1/docs/LangRef.html, Accessed Feb 3, 2017.

[9] ——, "Performance tips for frontend authors, for LLVM version 3.8.1." 2016, http://llvm.org/releases/3.8.1/docs/Frontend/PerformanceTips.html, Accessed Feb 3, 2017.

[10] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, ser. PASTE '11. New York, NY, USA: ACM, 2011, pp. 9–16.

[11] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. . Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[13] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," *SIGPLAN Not.*, vol. 35, no. 5, May 2000.

[14] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers, "DyC: An Expressive Annotation-Directed Dynamic Compiler for C," in *Theoretical Computer Science*, 2000.

[15] H.-P. Charles, D. Courouss, V. Lomller, F. Endo, and R. Gauguey, "deGoal: a tool to embed dynamic code generators into applications," in *Compiler Construction*, ser. Lecture Notes in Computer Science, A. Cohen, Ed. Springer Berlin Heidelberg, 2014, vol. 8409, pp. 107–112.

[16] B. Catanzaro, S. A. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. A. Yelick, A. Fox, B. Catanzaro, S. Kamil, Y. Lee, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and O. Fox, "SEJITS: Getting productivity and performance with selective embedded JIT specialization," in *In First Workshop on Programming models for Emerging Architectures*, 2009.

[17] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck, "An intermediate representation for speculative optimizations in a dynamic compiler," in *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL '13. New York, NY, USA: ACM, 2013, pp. 1–10.

[18] A. Dinaburg and A. Ruef, "McSema: Static translation of x86 instructions to LLVM," 2014, talk at REcon 2014, https://www.trailofbits.com/resources/McSema.pdf, Accessed Feb 3, 2017. Montreal, Canada, June 2014.