The Case for Binary Rewriting at Runtime for Efficient Implementation of High-Level Programming Models in HPC

Josef Weidendorfer, Jens Breitbart
Department of Informatics
Technical University of Munich
Munich, Germany
EMail: {Josef.Weidendorfer, j.breitbart}@tum.de

***Abstract*—Implementations of Parallel Programming Models are provided either as language extensions, completely new languages or as a library. The first two options often provides high productivity, but requires the porting of codes. In contrast, calls to new libraries can be added more easily, however the use of abstractions in such programming model implementations can have high runtime overhead. In both cases, the mentioned drawbacks often hinder the adaptation of novel programming models for large existing codes.**

To combine the advantages of compiler analysis with the composability of pure libraries towards more efficient programming model implementations, in this paper, we propose a low level API for programmer controlled binary rewriting at runtime. This can be used by programming models provided as libraries to efficiently integrate their abstractions with application code. This enables incremental adoption for existing codes as well as favoring input-dependent optimization strategies yet providing similar performance as language extension approaches. We show first promising experiences.

***Keywords*-High Performance Computing; Parallel Programming Models; Dynamic Optimizations;**

I. Introduction

Software development in High Performance Computing is different from program development in other fields due high requirements in application scalability with computations, that typically require regular synchronization and communication. As a result, people invest a lot of effort in optimizing their applications and thus, they expect parallel programming models to allow tuning also on the lower levels. Furthermore, languages allowing for compiling to native code are preferred (e.g. C++ or Fortran) in contrast to languages using managed environments (such as Python or Java). Even though it may provide productivity advantages, the overhead of just-in-time compilation and a managed runtime is expected to contradict the needs of HPC applications. Another aspect driving the characteristics of HPC programming environments is that applications often have a legacy burden. Quite some simulation codes still are based on Fortran-77 code from the 80-ties. New application-level features are added incrementally. The result is that the Message Passing Interface (MPI) [1] is the most-used parallel programming model in HPC and is both low-level and easy to compose with 3rd-party libraries. Using message passing as paradigm, programmers have to manage data distribution themselves which is especially complex if load balancing has to be done.

Proposals for HPC programming models providing higher productivity typically take the form of either language extensions, completely new languages or libraries. For the rest of this paper we treat new languages identically to language extensions, as we do not expect there to be any important differences for this work. Figure 1 visualizes the two approaches. The language approach requires porting. New abstractions have to be added as extensions to the language. In contrast, multiple new abstractions introduced in libraries usually can be added incrementally to existing code, often resulting in a higher number of abstractions compared to the language approach. In this paper we propose a low level API for programmer controlled binary rewriting at runtime to reduce the overhead of these abstractions and therefor mostly focus on the library approach for the rest of this paper.

With our low level API, the programmer can re-generate a new version of any function by providing its function pointer at runtime. A pointer to the new function is returned which can be used as drop-in replacement of the original function. In contrast to the original function, however, the new function typically is optimized based on the available runtime information. For example, a generic stencil implementation could be optimized for a specific stencil type. This way, the developer using the stencil function can parse any arbitrary stencil pattern at runtime, yet the computation provides (almost) the same performance of a function manually optimized for a specific pattern. For configuration of the rewriting process, we rely on the ABI (Application Binary Interface) of the system. We present a prototype able to decode and rewrite a subset of the 64-bit Intel x86 architecture. To the best of our knowledge, a minimalistic flexible API for doing transformations at the binary level was not proposed before. All existing approaches involving code generation (see Section VII) expose higher level interfaces towards specific needs, if usable within application code at all. They do not provide code generation itself as a feature to be used by the programmer.

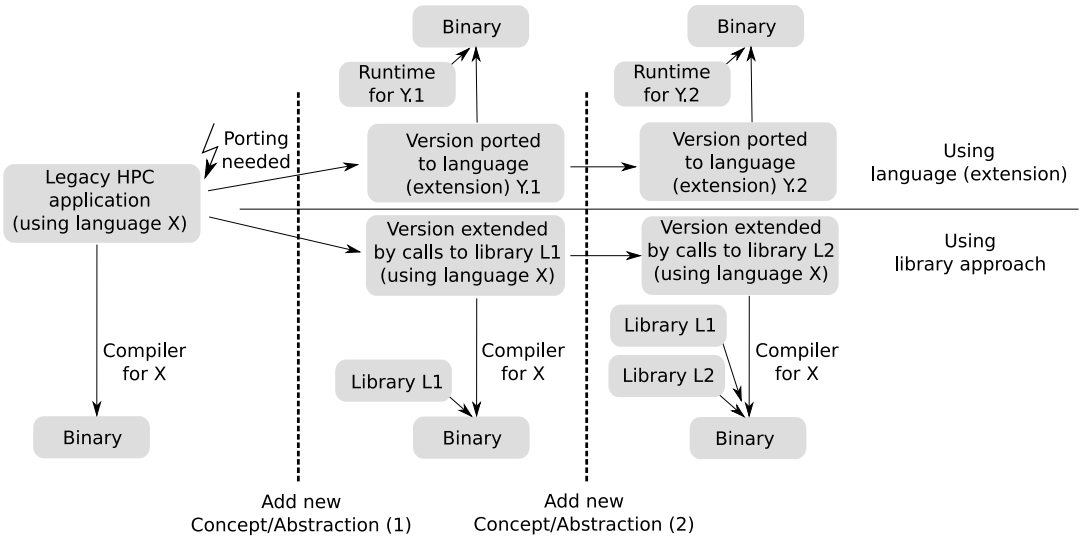

Figure 1. Different ways to add a new concept/abstraction to the programming model used. Top: Language approach needing porting, all concepts must be added to one language. Bottom: Library approach allowing composition without porting, however often with slower performance.

The rest of the paper is structured as follows. First, we revisit existing efficient solutions used to integrate new programming models in existing languages in the next section. In Section III, we describe our requirements for the rewriting API and the API itself together with the rewriting strategy being used. This includes a discussion of handling jumps. Section IV shortly describes our current prototype. In Section V, we show our first experiences with the prototype using a 2D stencil code. In this example, we start with a generic stencil computation taking stencil parameters (number of points, offsets, coefficients) as input. We compare the performance of a rewritten version specialized for a given stencil form with a manual implementation of that computation. Section **??** give a brief description on how binary rewriting could also be beneficial for new programming languages. After presenting related work, we finish with a conclusion of our findings.

II. Efficient Solutions Integrating New Programming Models as Libraries

In this section we describe generic techniques that can be used to optimize the performance of programming models that are implemented as libraries.

In C++, a library can be provided in header files to be inlined by the compiler and also make heavy use of the C++ template features. An important technique here are so-called Expression Templates [2]: Application data types are wrapped by classes (often called proxy classes) which do not directly execute operations on the data but collect the operations into expressions at runtime for later execution in an optimized fashion. This can be especially beneficial to minimize memory accesses by e.g. doing multiple operations on one element while it is still in a register. For example, Blitz++ [3] and Boost uBLAS[1] are libraries using this feature. While this technique is restricted to C++, programmers only are allowed to use operations predefined by the library which has to be available at source-code level in header files. As such, vendor-optimized commercial math libraries are out of reach (such as Intel MKL [4]). Furthermore, proxy classes cannot be 100% identical to existing types, which can result in unexpected compiler errors.

CUDA [5] and OpenCL [6] also rely on a library approach, but defer part of the compilation to runtime. This has another benefit: program input can be taken into account, allowing generating and running code specialized for given data. However, being quite low level and specific to the accelerator hardware, tuning on that level is done rarely by application programmers. An interesting approach was taken by Intel ABB (Array Building Blocks) [7], originally developed in academia and passed on to a startup (Rapid Mind) later bought by Intel. ABB provides a second version of various C++ keywords to allow programmers to write code which captures specific instances of generic code at runtime for later execution. However, ABB did not trigger enough interest due to being difficult to learn. It got abandoned by Intel as commercial product. Techniques involving re-compilation at runtime may simply invoke a regular compiler. For example, the compilation suite LLVM [8] can be linked as library and be called for generating specialized code. This works especially well for application specific generators [9], but also can be done generically. Depending on optimization passes to be used, one either has to preserve source code or intermediate code (e.g. using LLVM-IR). Again, vendor-optimized commercial math libraries cannot

[1]http://www.boost.org/doc/libs/1_60_0/libs/numeric/ublas/doc/index.html

be integrated in this approach. Further, even for open-source projects, libraries may be readily available at a compute center cite only in binary form, making it cumbersome for programmers to use techniques requiring source.

Due to the stated drawbacks, we came up with the solution we propose here.

III. Binary Code Transformation

A. Expected Benefits

Our low-level rewriter API allows programming model implementations as libraries. To get rid of abstraction overhead such as huge amounts of function calls and indirections, rewriting can be used. Further, rewriting can be used to deduce values that are constant during runtime, e.g. data structure indices that depend on data distribution are not known at compiletime if the data distribution depends on runtime information such as the number of systems used for the computation. However, once the application has started it is possible to optimize such index computations.

As the result of a rewriting step itself can be used as input for further rewriting, this approach is composable. For example, a generic code generator may take pieces of binary code as input. These pieces themselves can be the outcome of a rewriting step, but also the code generator can be rewritten to be specialized for given input. Thus, we expect this approach to be usable for making implementations of (parallel) programming models easier composable.

B. Rewriting by Tracing

The rewriting process essentially emulates a call to the function. This requires that a parameter setting is provided. During emulation, a new version is generated ("captured"), taking rewriting configuration into account. For example, a given parameter used in the emulation can be specified to be assumed to be a known, constant value for the newly generated version. Then, all operations using this parameter are rewritten to use the constant instead. Operations only involving constants just are emulated without generating any instruction, resulting in automatic constant propagation. For such a configuration, our rewriter works as specializer, also called *partial evaluation*.

Other rewriting features/configurations may involve function inlining, configure loop unrolling, or injection of handler calls (which even can be inlined) when specific operations such as memory accesses are detected during emulation.

C. Configuration

For configuration of the rewriting process, we rely on the ABI (Application Binary Interface) of the system. Among others, the ABI specifies register use and parameter passing at function call boundaries. Every compiler producing native code has to comply to this specification for linking (also to shared library functions at runtime) to work. By relating rewriting configuration to actions at function boundaries, the abstractions of the enforced ABI calling convention can be used to make the rewriter configuration itself architecture independent.

As we describe below, the rewriting can visit multiple functions during the rewriting process. Thus, a rewriter configuration provides the options for functions given their start address (this includes the function to rewritten itself).

- Is a given parameter assumed to be a known, fixed value in the rewritten result (defaulting to unknown)?
- Is a given function to be inlined when called?
- Are values as results of operations to be marked as unknown (this avoids loop unrolling, as described later)?
- How many specialized versions of the same original piece of code in a given function should be generated?

This list may be extended in the future as needed.

D. Requirements

The main objective of the API is to allow applications to become faster by generating smaller and faster code as replacement for existing functions, using information available at runtime. To this end, both *inlining* as well as *specialization* or *partial evaluation* are effective techniques. The first removes the overhead of jumps and function prologues/epilogues which are required by the system ABI to be able to do calls in the first place, using a corresponding calling convention. The second allows to generate variants for later execution in specific contexts where input data to the function are known to have fixed values. For example, this allows to reduce parametrized generic versions to specific needs, replacing variables with their known values in operations, removing branches with known outcome, or removing indirections when references are known. Any computation using values specified as being known can be removed and pre-computed already when the new variant is generated. The new variant therefore should consist of much less code and execute faster.

Partial evaluation works when input data is known. This often may not be known at first, but statistical information can be collected by profiling. For example, it may be observed that a parameter to a function often is 42. In this case, a specific variant can be generated which is called after a check for the parameter actually being 42. Otherwise, the original function should be executed. This concept easily can be extended to cover various statistical knowledge of the dynamic program flow and should be supported. We note that with the stated requirements, we can start from functions with built-in profiling functionality, and variant generation can remove profiling from the generic version. However, we may not be able to influence the original functions, and thus, it is convenient to inject calls into own profiling functions e.g. at function begin or end. Other interesting points for callbacks include memory accesses. By generating function variants with redirected memory accesses, different layout

of data structures can be supported. Further, we may want to detect if a call goes to a function with behavior known to us, and replace a generic behavior with a specific one.

For the presentation in this paper, we restrict ourself to the first mentioned use cases: inlining and partial evaluation. We expect the further mentioned features to be straight-forward extensions.

E. Rewriting Strategy and Proposed Interface

New variants of a function should be usable as simply as possible. To this end, we keep the function signature the same as the original function, as this allows the new variant to be used as drop-in replacement, even though some parameters may never be accessed as result of partial evaluation. This way, if a programmer knows how to call a function, he also knows how to call its replacement.

In the following, shown code always is in C. Our proposed API uses the prefix `brew_` (for Binary REWriting). Now, let us assume a function `func` which takes two integers as parameters and returns an integer. We explicitly declare its function signature with a `typedef`. The configuration of the rewriter is maintained in a structure `rConf`. The generator API function (called `brew_rewrite`) takes as parameters the configuration, the function pointer of the original function, as well as all parameters of a original function. Generating a new variant of function `myFunc` is shown in the following:

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

main() {
  // call original function
  int x = func(1,2);
  // rewrite func
  ... (configure rewriter)
  newfunc = (func_t) brew_rewrite(
            rConf, func, 1, 2);
  // call rewritten version
  int x2 = (*newfunc)(1,2);  }
```

We do partial evaluation by tracing the execution of the original function instruction by instruction. In each step, either the original instruction, a modified version, or nothing may be passed on as the next instruction to be appended to the newly generated variant. For every variable value used during execution, we maintain a flag for whether this value is assumed to be known or unknown. If it is known, we maintain the actual value also during tracing by emulating every instruction touching the value. Now, if an instruction only uses unknown values, we pass the instruction unmodified. If a value is known, we can replace the corresponding operand of the instruction pointing to the location where the value resides with the actual value itself as immediate operand, and append this modified instruction to the generated code. However, if all input to the instruction is known and thus also its result, there is no need to pass the instruction on at all. Instead, we only have to emulate the instruction for obtaining the known result value. The actual value can be used as immediate constant whenever a following instruction accesses the value.

As one location for storing state (being a register or a memory location) can only hold one value, the known-state of that value can also be stored together with the location. For locations not used yet, we mark its content as invalid. At tracing start, we want to configure which input data is assumed to be known. Input data are either values passed as function parameters or values fetched from memory during execution. Thus, the user of the API should be able to specify the known-state of function parameters and memory locations. For both, we default to unknown state and provide API functions allowing to annotate function parameters and memory ranges to be assumed to be known. Further, if a value is used as pointer, we want to be able to annotate this pointer as pointing to known data. This way, complex data structures using indirections easily can be marked as being completely known.

To specify known state, we use the C enum values `BREW_KNOWN` (and as extension for pointers to known state `BREW_PTR_TOKNOWN`). In the following example code, rewriting is configured to assume parameter 1 to be known, as well as some memory range from `start` to `end`. If a parameter is specified as being known, the concrete values are taken from the parameters of `brew_rewrite` mirroring the original parameters. Here, the known parameter 1 is set to 42. When calling the rewritten function, the first parameter actually is ignored, as the new variant has replaced every access to parameter one with value 42.

```
// set config and rewrite func
brew_initConf(rConf);
brew_setpar(rConf, 1, BREW_KNOWN);
brew_setmem(rConf,
          start,end,BREW_KNOWN);
newfunc = (func_t) brew_rewrite(
          rConf, func, 42, 2);

// ignores value 1
int x2 = (*newfunc)(1,2);
```

The above description of function tracing should be enough to understand the rewriting process for functions consisting of a series of instructions which do not change control flow. However, before we can pass the `ret` instruction to the newly generated function, we need to check if the state of the return value is set to be known. If so, we need to generate an explicit load of the return value location as specified by the ABI with the immediate value (for 64-bit x86, this is RAX for integers).

If a call to another function with a series of instructions is detected by tracing, currently we always assume that we should inline the function. To keep the call, we would need to know the effect of the function regarding changes of known-state and concrete values if known. If configuration asks to not inline a function, we currently simply signal a failure. This is left as future work[2]. For inlining, we simply can go on with the first instruction in the called function and return back to the calling site on a `ret` instruction. To this end, we maintain a shadow stack remembering traced call instructions and corresponding return addresses.

F. Jump Processing

In the following, we describe our tracing strategy when detecting jumps. For unconditional jumps, we can proceed as with calls without changes to the shadow stack. For jumps to multiple possible target addresses (both conditional jumps as well as unknown indirect jumps), we need to distinguish whether the jump condition (or jump target) is marked as being known or unknown. If known, the actual value of the condition (or jump target) can be checked, and we can proceed as with unconditional jumps. We note that we also maintain the known-state for the various condition flags (e.g. zero or carry flag), being set with most x86 instructions depending on their result value.

Without discussing unknown jump targets, a given class of loops now can already be handled. That is, loops starting from a known value with a loop condition only involving constant values. For example, for a loop goes from 1 to 42, let us assume the the loop counter is stored in some register. First, the known constant value 1 will be written to this register. Whenever the register with the loop counter is accessed by an instruction, the rewriting process will replace a reference to the register with the actual value 1. At some point, the loop counter will be incremented and following, a jump back to the beginning of the loop is done. Both the condition check being known and the jump back do not result in any instruction passed to generated code. Instead, the loop body will be traced a second time with the loop counter set to known value 2. This repeats until the loop counter is 42, resulting in complete unrolling of the loop. This behavior actually is nice for small loops. However, for larger loops huge functions get generated. Our current solution is for functions to be able to configure that every conditional jump (even if known) should be treated as being unknown. As we inline other functions, this configuration may change during tracing, but is restored when returning to the previous function.

Finally, what should happen on detecting jumps to unknown targets? As the result is unknown at rewriting time, we need to generate code for each possible execution path, preserving the jump instruction. We currently signal failure if we trace an indirect unknown jump. This is future work. To handle multiple paths yet to traced, we keep a queue of yet-to-be-rewritten basic blocks as well as a list of already generated blocks. When finishing a basic block with two possible following code paths, we are free to choose the next block to process. Preferring the path according to some prediction strategy is useful as the generated code for the basic block processed next will simply follow the code of the previous block, unless the basic block was already processed before.

The described strategy sounds straight-forward: if a path ends with a jump to an already processed block, we can stop and go to the next unprocessed block in the yet-to-be-rewritten queue. At some point, all reachable basic blocks of a function are processed and we finish. However, the correctness of our tracing strategy crucially depends on the known-state of values. Depending on whether a value is (1) marked as known or unknown and (2) its value if known, generated code will be different. For example, when a reference to a location with a known value is an operand in an instruction, we replace the reference with the value, generating code dependent on the known-state. Further, known values decide about control flow. Therefore, we also have to maintain known-state of values when processing basic blocks in arbitrary order. To this end, we need to add the facility to save and restore the state of all known-ness as well as the values themselves if known. We call this the *known-world state* in the following. We save this state when the code path at the end of a traced block is diverging, that is at conditional jumps with multiple possible following code paths. We relate the saved known-world state to the created yet-to-be-rewritten blocks. Whenever we process a new basic block, we first restore the corresponding known-world state before starting tracing.

After processing a basic block, the code path may go on at an already processed basic block. However, the previous processing may have used another known-world state for tracing that block. As mentioned before, tracing the some code with different known-world state can result in different code being generated. Therefore, we have to extend our strategy to identify yet-to-be-rewritten blocks not just by block start address, but also by an unique identifier for different known-world states. To overcome code explosion, we can produce compensation code for migrating between world states as long as there are only values changing from being known to unknown. For each such value, we have to generate code to load the corresponding locations with their known values.

The described processing cannot prohibit complete unrolling of loops even if branch conditions are unknown whenever the known-world state changes among loop interations. However, we can restrict the number of generated blocks from the same original block by reducing known-

[2]We think that we can do an isolated tracing of the called function to detect its effects regarding known values. If the effects become too complex, we always can signal failure.

world state to one of an already translated block. For that, if a given configuration threshold is reached, we search for possible migrations. If none is possible (no state with values changing from known to unknown), we search for states with same known-ness, but different known fixed values and migrate to a state where corresponding values become unknown (this algorithm always terminates at a state with all values unknown).

Yet, we want to have a simple way to avoid any unrolling of loops. For this case, we force every value created by an operation in a function to become unknown. This of course makes every branch condition be unknown (conditions are not passed as parameters). We note that called functions still get specialized, either due to constant values directly passed through as parameter, or by a configuration for the called function setting a parameter to be known.

G. Summary of Rewriting Steps

When rewriting of a function is requested, the following steps are done:

- Put the first basic block onto the yet-to-be-rewritten queue, together with a known-world state reflecting input parameter configuration.
- Take the next basic block to rewrite, decode it, load the corresponding known-world state and start tracing. Captured instructions are kept in decoded form. We trace over jumps and calls with known targets. Calls configured to not be inlined are kept, generating compensation code to make registers "unknown" which are parameters according to the ABI. For tracing after the call returns, we assume all caller-saved registers to be dead/unkwown, while all callee-save registers keep their known state.
- After tracing a basic block, succeeding basic blocks are determined. If not yet translated, they are added to the yet-to-be-rewritten queue. Basic blocks starting at same address are treated to be different when their known-world state differs. If a threshold for different variants of translations starting at same address is reached, we try to migrate to the known-world state of an existing translation. If this is not possible, we determine the translating with the smallest difference and set values known but different to be unknown. In all cases, compensation code for migration of the known-world state may be generated.
- When all blocks are rewritten, we run optimization passes over the newly generated, captured blocks.
- Determine the best order of generated blocks for the final rewritten code.
- Generate binary code from captured blocks. Unless we fall-through from the previously generated code, we leave some space between blocks to allow for required jump instructions at the end.
- Do relocation of all need jumps, given start addresses from the previous step.

At all times, it is possible that we reach a situation that cannot be handled. For example, when buffers run out of space (there is a configuration for maximum size), when instructions cannot be decoded, or when the machine code for an instruction to generate is unknown. This will result in a failure of the rewriting process, but it is *not* catastrophic. It simply means that the user of the rewriter API has to use the original version of the function he wanted to rewrite.

IV. Prototype

Our prototype is able to decode and rewrite a subset of the 64-bit Intel x86 architecture. It does not depend on other libraries. One can imagine that the rewriting step should employ many standard compiler optimizations, and thus, e.g. building on top of LLVM is winning goal (involving translation of binary code to LLVM-IR and back). We note that optimization passes in LLVM are written to be used in standard compilers. It may need modifications for them to be able to take programmer knowledge as rewriter configuration into account.

The prototype does all rewriting steps as described. However, there currently are no optimization passes implemented. Even without, the prototype is already quite capable as shown below. We do not expect that the rewriting step has to include a lot of standard compiler optimizations itself to be useful. We note that the rewriter already should take highly-optimized code as input. In this sense, it is a delayed step complementing static compilation to produce the final code for execution. The most important aspect is well working inlining, getting rid of standard prologues and epilogs enforced by the ABI. We note that this does not need a register re-allocation but only register renaming. However, avoiding register spills to the stack is important when free register spaces becomes available due to specialization. The mentioned optimizations are future work. Nevertheless, sophisticated optimization passes in the rewriter are useful: e.g. we plan to implement a simple greedy vectorization pass which may take programmer knowledge and runtime information provided via rewriter configuration into account, guiding the search for best replacement of scalar operations with vector instructions. We note that such optimization passes can be much simpler than corresponding compiler passes, as being tailored to specific cases supported by runtime information.

Once our prototype covers an acceptable portion of 64-bit x86 instructions, we will make it open-source on GitHub.

V. First Experiences

If we assume a programming model given as *library implementation*, there must be functions implementing the abstractions from the programming model resulting in code that does communication, computation and memory accesses

as needed. For useful abstractions and real code, we argue that there always have to be functions to be called in inner-most loops of programmer supplied kernels. If the model would have been given as language extensions, the compiler would optimize away corresponding overhead. However, this is not possible for a library implementation. For example, DASH [10] (a C++ library providing a PGAS programming model) must translate between global and local address space for every call to `operator[]` on distributed data structures. As a result, using this operator is not recommended in inner-most loops, even if the developers know the data is local to the calling node. The runtime checks if the data is actually local result in high overhead.

A. Specializing Generic Stencil Computation

To show our first experiences with programmer controlled binary rewriting, we present results from our prototype for specializing a generic 2d stencil computation. This has similar characteristics to the PGAS use case mentioned: the stencil application is used in the inner-most loop with the generic function having quite some overhead. The following code shows the definition of a 5-point stencil using a corresponding data structure. The stencil takes the average of neighboring points and subtracts the original value at a given position:

```
struct P { double f; int dx, dy; };
struct S { int ps; struct P p[]; };

struct S s5 = {5,{{0,0,-1.0},{-1,0,.25},
 {1,0,.25},{0,-1,.25},{0,1,.25}}};
```

Next, we show the generic code returning an updated value for applying an arbitrary stencil.

```
double apply(double *m, int xs,
             struct S* s) {
  double v = 0.0;
  for(int i=0; i<s->ps; i++) {
      struct P* p = s->p + i;
      v += p->f * m[p->dx + xs*p->dy];
  } return v; }
```

Finally, this is an example of a code doing a corresponding 2d matrix traversal, calling the generic stencil computation:

```
double m1[ys][xs], m2[ys][xs];
...
for(y=1;y<ys-1;y++)
 for(x=1;x<xs-1;x++)
  m2[y][x] = apply(&m1[y][x],xs,&s5);
```

Running 1000 iterations on a 500^2 matrix (on a laptop with an Intel Core i7-3740QM with 2.7 GHz and 6 MB L3 cache, using Ubuntu 15.10 with gcc 5.1 and "-O2" compile flag), this requires 2.00s (We note that the space traversed for the 2 matrices is 4 MB, fitting into L3). Now, in contrast, directly writing code for the stencil, the computation only takes 0.74s, ie. 37% of original runtime. This is expected and shows the benefit of information already known at compile time.

Specializing the `apply` function and using the rewritten version is quite simple:

```
typedef double
  (*apply_t)(double*,int,struct S*);
...
brew_setpar(rConf, 2, BREW_KNOWN);
brew_setpar(rConf, 3, BREW_PTR_TOKNOWN);
apply_t app2 = (apply_t) brew_rewrite(
  rConf, apply, 0, xs, &s5);
...
for(y=1;y<ys-1;y++)
 for(x=1;x<xs-1;x++)
  m2[y][x] = (*app2)(&m1[y][x],xs,&s5);
```

Here, we configure the rewriter to assume both the matrix side length and the stencil to be fixed. For the latter, we mark the third parameter to be a pointer to known fixed data (this applies recursively if pointers would have been used in `struct S`). The code for `app2`, generated at runtime with our rewriter, looks like this (nine instructions in the middle, similar to instructions `i-01`, `i-02`, and `i-03`, are left out).

```
i-00: pxor    %xmm0,%xmm0
i-01: movsdq  (%rdi),%mm1
i-02: mulsdq  0x612100,%mm1
i-03: addsdq  %mm1,%mm0
...
i-13: movsdq  0xfa0(%rdi),%mm1
i-14: mulsdq  0x612140,%mm1
i-15: addsdq  %mm1,%mm0
i-16: ret
```

It is easy to recognize the 5 points in the stencil in the output (not in the given truncated version). Coefficients from the stencil data structure are referenced directly, e.g. seen in `i-01` referencing a double value at address 0x612100. The known distance between rows (specified as second parameter) can be seen as constant displacement in `i-13`. Using the specialized version, our computation takes 0.88s, only 44% of the generic version and 18% slower than the manually written stencil.

B. Optimizations

Analysis of the generated code shows that some optimizations are missed. First, the rewriter does not see that four coefficients of the stencil are the same. A compiler would transform the expression to remove corresponding loads and do less multiplications. Second, as the `appy` function is specialized in isolation, optimizations such as caching values in registers when reused in neighborhood computations, and

similar important, vectorization of multiple stencil applications cannot be done.

Regarding the first missed optimization, we can rewrite the generic version of the stencil to group points with same coefficients, which needs to be reflected in the stencil structure. We note that this keeps the abstraction, providing arbitrary stencil forms at runtime. However, the code is written knowing the behavior of the rewriter. We think this is fine as we expect that often, the code subject for rewriting will not come the application programmer but from a library tuned for being used together with binary rewriting. We just show how the structure used in a "grouped" version looks like.

```
struct P { int dx, dy; };
struct G { double f;
           int ps; struct P p[]; }
struct S { int gs; struct G p[]; };
```

Now, the generic version has another loop going over the groups, and not surprisingly, using the generic version on the same 5-point stencil with the 1000 iterations on 500^2 matrices actually gets slowed down to 2.21s, around 10% slower than the original version. However, the rewritten version improved from 0.88s to 0.74s, exactly as fast as the manual version. Looking at the disassembly of the rewritten single stencil computation, it is even better than the manual version which did not incorporate the knowledge of the matrix side length.

The other missed optimizations actually take the calling site of the stencil computation into account. Reuse of values and vectorization is a huge improvement, and we actually cheated regarding to the manually written stencil up to now: the 0.74s come from a version which uses function pointers to call into each single stencil computation. If we avoid the function pointer, the runtime goes down from 0.74 to 0.48s. Thus, it seems to be beneficial to apply our rewriter to a complete matrix sweep (which still can use indirect function calls which they get removed by specialization). Our prototype can be configured to avoid complete loop unrolling which definitely is needed here. However, we currently miss optimization passes for the rewritten code to be able to get better. With controlled unrolling (such as four-times), we imagine that it should be quite simple to write optimization passes for straight-line code (ie. a basic block) which do (1) instruction reordering removing redundant loads, (2) vectorization by replacing scalar instruction with vector versions with same semantics. This is future work.

C. Failed Approaches to Avoid Loop Unrolling

Earlier in this paper, we mentioned our strategy from keeping the rewriter to do complete loop unrolling. We have a configuration per function which states that each newly created value should be treated as unknown (not touching values passed in as parameters). This is a brute force approach which actually destroys any possibilities for specialization. However, it does not remove chances for specialization for nested called functions which get inlined.

In the stencil code above, the loop for a matrix sweep starts at index 1, being a constant. The loop unrolling is enforced due to this constant value being incremented to become 2 for the second loop body, triggering another version to be created (the known world state changes). Our first approach focused on the idea that we just need to mark the loop counter value as being unknown. This can be done by introducing a function `int makeDynamic(int)` which just returns its parameter. The rewriter can detect the call to this specific function, and mark the value passed in to always become unknown. This way, the loop for a matrix sweep can be rewritten as follows:

```
for(y=makeDynamic(1);y<ys-1;y++)
 for(x=makeDynamic(1);x<xs-1;x++)
  m2[y][x] = apply(&m1[y][x],xs,&s5);
```

However, the compiler is allowed to do arbitrary transformations of the iterator space, as long as the order of calls to `apply` is observed, given that we force `apply` not to get inlined by the compiler in the first place. When we tried above version, we saw that the compiler created another loop count variable still starting at 0, and where the original loop count was required, it added the value returned from `makeDynamic` before. Thus, there still was a constant known value which changed in each iteration, resulting in complete unrolling again.

This effect is interesting for two aspects: we never should assume that generated binary code has much relation to the source, unless explicitly enforced by the ABI or language specification. By making a function visible to the linker and prohibiting its inlining in C, the compiler cannot assume specific semantics and has to keep the order of calling the function. The second aspect is related: calling non-inlined functions can actively prohibit optimizations which a compiler otherwise would do. This means that by making binary code better suited to be input to our rewriter, code quality will be degraded. Our rewriter needs to compensate this effect to be useful.

VI. Binary rewriting for new languages

So far we concentrated only on libraries providing an additional programming model, however our approach can also be useful for new languages as we think it is not possible to eliminate all abstractions at runtime. For example, the PGAS language Chapel [11] uses so called domain maps to describe the distribution of data among systems. The distribution is typically not changes during runtime or only at certain points (e.g. load balancing). Binary specialization can be used to optimize accesses using the domain map and a runtime system could trigger a new specialization whenever the domain map is changed. That way, such changes would

be transparent to the user. We expect that there are further possibilities to use our approach in new languages, but further research is required to identify these.

VII. Related Work

There are a lot of tools using code generation internally: just-in-time compilers for various languages and byte-code specifications (JVM, .NET), ISA translators for VMs such as QEmu [12], binary instrumentors such as Valgrind [13], Intel Pin [14], or DynInst [15]. However, all such tools expose a higher level interface towards specific needs. It should be possible for the mentioned projects to build a re-implementation on top of our binary rewriter. On the other hand, our rewriter is more than a code generation backend for a given architecture found in a compiler.

There are a few proposals for enabling dynamic code generation very similar to our approach. In [16], Grant et al. propose an extension of C using annotations (DyC) allowing parts of the C code to be transfered to runtime for allowing deferred specialization. However, DyC specializes C, not binary code. DeGoal [17] 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. Cling[3] is a C++ interpreter using the LLVM infrastructure. It could provide similar feature as our binary rewriter, however parsing C++ is time consuming and it is unclear if it could provide the same level of performance as the binary rewriter.

There are quite some language specific approaches to dynamic code generation. A lot of scripting languages allow execution of code built from strings at runtime, and being evaluated (often named `eval`). Examples are Python and JavaScript. The just-in-time compilation usually produces native code if executed often. Using this to include runtime information for better performance is called *Multi-Staging*, proposed e.g. for Scala in [18]. In [19], the authors propose AnyDSL, a simple extension of a systems language (syntax taken from RUST) allowing programmers to specify where code should be specialized for better performance. They use this for abstractions supporting easy programming of heterogeous systems. SEJITS [20] is a project from Berkeley proposing specialization for selected functions in Python, replacing these with dynamically generated native code automatically at runtime. This way, the projects want close the "productivity gap" for programming HPC systems. Graal [21] is an new API proposed for Java for controlling dynamic compilation. It allows programmers to specify meta information to guide low-level optimizations in an upcoming JVM. All the works mentioned are related to a programming language. Our approach is language agnostic, working on the binary level.

[3]https://root.cern.ch/cling

VIII. Conclusion

In this paper, we presented our idea of a low-level API for application programmers to transform binary code at runtime. We discussed the benefits of this technique and our vision for using it to compose higher-level programming models for HPC. First experiences with a prototype show that the incremental approach taken by the API is promising: when rewriting fails, using the original function always is possible. Further, rewriting makes sense only for performance sensitive hot code paths. Thus, the current prototype is already useful even without complete coverage of the x86 ISA and sophisticated optimization passes. Any extensions of the prototype can be guided by the requirements of performance critical hotspots in selected HPC applications as well as implementations of abstractions built on top of it.

As next step, we will implement register renaming for improved inlining of small functions and deep call chains. We plan to use our prototype for enabling optimizations in HPC codes using a library with PGAS abstractions such as iteration through global address space. We want to use our API to detect remote memory accesses in arbitrary code, triggering preloading from remote nodes per RDMA, and use a second rewritten version of the same code which redirects memory access to the local pre-loaded data.

References

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

[2] K. Iglberger, G. Hager, J. Treibig, and U. Rde, "Expression templates revisited: A performance analysis of current methodologies." *SIAM J. Scientific Computing*, vol. 34, no. 2, 2012.

[3] T. L. Veldhuizen, "Blitz++: The library that thinks it is a compiler," in *Advances in Software tools for scientific computing*. Springer, 2000, pp. 57–87.

[4] *Intel Math Kernel Library. Reference Manual*. Santa Clara, USA: Intel Corporation, 2009.

[5] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[6] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, 2010.

[7] C. J. Newburn, B. So, Z. Liu, M. D. McCool, A. M. Ghuloum, S. D. Toit, Z.-G. Wang, Z. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, "Intel's array building blocks: A retargetable, dynamic compiler and embedded language." in *CGO*. IEEE Computer Society, 2011, pp. 224–235.

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

[9] T. Müller, J. Weidendorfer, and A. Blaszczyk, "Expression tree evaluation by dynamic code generation - are accelerators up for the task?" in *Parallel Processing (ICPP), 2013 42nd International Conference on*, Oct 2013, pp. 230–239.

[10] 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.

[11] 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.

[12] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] T. Rompf and M. Odersky, "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls," *SIGPLAN Not.*, vol. 46, no. 2, pp. 127–136, Oct. 2010.

[19] R. Leißa, K. Boesche, S. Hack, R. Membarth, and P. Slusallek, "Shallow embedding of DSLs via online partial evaluation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2015. New York, NY, USA: ACM, 2015, pp. 11–20.

[20] 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.

[21] 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.