DBrew – A Library for Dynamic Binary Rewriting

ROME 2016, August 23, 2016

Josef Weidendorfer

Chair for Computer Architecture
Department of Informatics, Technische Universität München

DBrew – A Library for Dynamic Binary Rewriting

ROME 2016, August 23, 2016

Josef Weidendorfer

Chair for Computer Architecture
Department of Informatics, Technische Universität München

Work together

with colleagues

Jens Breitbart, Tilman Küstner

with student

Alexis Engelke

About me

Computer Architecture Chair at TUM with focus on HPC

Interested in

- Performance Analysis Tools & Optimization Strategies (cache simulation Callgrind, recently multi-core NUMA)
- parallel programming models (e.g. PGAS)

- optimization techniques involving code generation

Code Generation

- in Valgrind (or Pin): Dynamic Binary Instrumentation
  - original binaries, instrumentation drives simulation
- project with ABB: improve performance in evaluation of large expression trees
  - interpreting bytecode vs. LLVM usage vs. manual generation
- performance optimizations SpMV, > 2GB CSR matrix
  - medical imaging (MLEM algorithm for PET): random structure
  - transform SpMV in 4GB linear code, code generator hand-tuned
  - do code generation & execution on-chip, sustained 8 GB/s
  - improvements > x2 (no indirection, no loop overhead)

Code Generation: Lessons Learned

Powerful technique if

- best performance depends on dynamic input data
- problem specific, hand-tuned generator is feasible
- programmer-controllable (algorithm/tuning knowledge)

Large Benefits from

- specialized code vs. generic code
  (similar to “compiled vs. interpreted”)
- code without lots of prologs/epilogs/loop overhead

Less-Manual Code Generation: Alternatives

Dispatch into statically generated variants

- using C++ templates (pre-processor macros with C)
- often too many variants (code explosion)

Generic JIT techniques

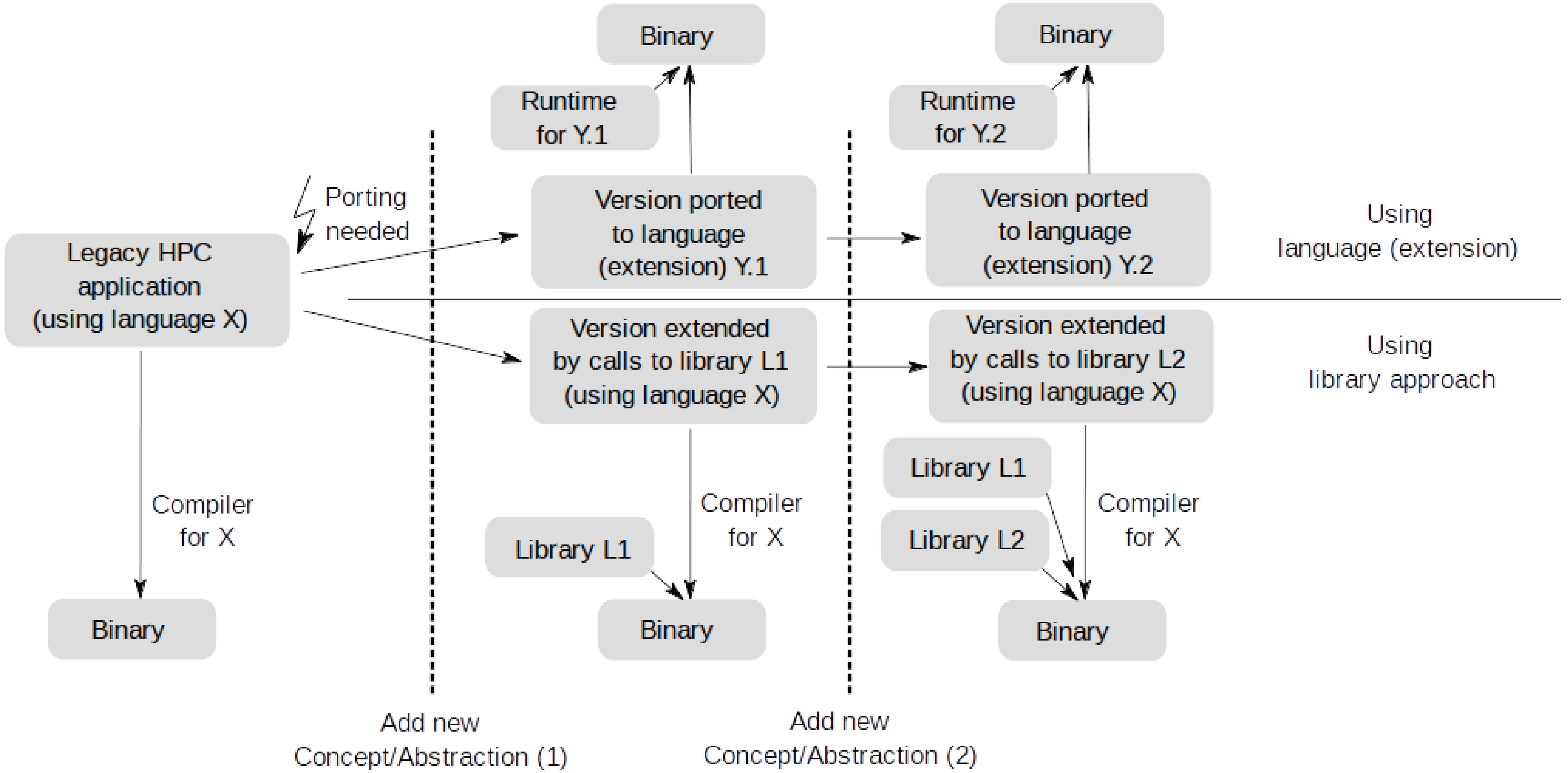
- generate LLVM-IR, use JIT at runtime / JS with V8
- not easy to control variant generation

New language (feature)

- difficult to sell to HPC community, no incremental use possible to improve existing legacy code
- we are not compiler guys

Use Cases for HPC

How about we add another layer of abstraction?



[from our HIPS16 paper on DBrew]

Use Cases for HPC

- Code generation can remove dynamic overhead of abstractions/generalizations in programming models

Use Cases for HPC

Introduce abstractions which enable optimizations

- concrete implementation not statically decided but only at runtime
  - examples: traversal orders, data layouts, partitioning
  - depending on target architecture, input data/intermediary results
- generic application code gets specialized for dynamic decisions at runtime
- application code decoupled from tuning heuristics

Others

- enable high-performance MPI data types

Dynamic Code Transformation for Programmers?

- incremental usage with existing HPC software stacks (C/C++/Fortran)
  - API / library to be linked to binary
  - machine code level (enables use with 3rd party compiled code)
- low-level machinery to generic code generation
  - for use by higher-level library layers / (compiler) runtimes
  - ISA-agnostic interface
  - transformation of existing code
- can be best-effort
  - only performance-relevant: if transformation fails, use original (enough to cover ISA instructions used in hot paths)
- failed transformations must not be catastrophic
- no additional complexity for debugging

Dynamic Code Transformation for Programmers?

Existing tools: not directly usable

- dynamic x86 assembler libraries: too low-level
- LLVM
  - needs lot of meta information to be usable
    (to be provided by programmer/reconstructed by analysis)
  - large dependency
- Valgrind/Pin/DynamoRIO
  - use decoder/IR manipulation/generation, but not exposed
  - to observe binaries from outside, not to be used inside
- DynInst
  - to observe binaries from outside, not to be used inside

➔ do our own library (may use existing tools internally)

What is DBrew?

- API to transform native, compiled code at runtime
- generate new variants of already existing functions
- provides drop-in replacements of original functions

Example (in C)

```
#include "dbrew.h"
…
typedef int (*f_t)(int,int);

…
dbrew_set_func(f);
f_t ff = dbrew_rewrite(x1,x2);
…
```

```
a = f(p1, p2);
```

→

```
a = (*ff)(p1, p2);
```

What is DBrew?

- currently x86-64 only
- github.com/lrr-tum/dbrew
- prototyping state
- examples should work
- any feedback welcome

DBrew Design

Configuration

- based on ABI
  (application binary interface: calling convention)
- information about values (esp. function parameters)
- control over transformation (inlining, loop unrolling)
- used resources (buffer sizes, code buffer)
- failure handling

Failures

- decode error, buffer overflow, …
- robust: on failure, may return/branch to original code
- other failure handling: enlarge buffers, restrict inlining,…

DBrew Configuration: Information on Values

Values

- function parameters (identified via ABI)
- global variables (via address)
- reachable via pointers being function parameters

Possible information (e.g. for int value “x”)

- known to be constant (“x = 5”)
  - enables evaluation of all operations with known values
- fulfilling conditions (“x > 5”)
  - restricts possible execution paths
- being most likely a given value (“x often is 5”)
  - influences inlining (needs a guard)

DBrew Configuration: Control of Transformation

Inline or call into given functions?

- functions are specified via their function address (the symbol name resolves to the address)
- on call instruction
  - call original function, or
  - trigger rewriting and redirect call to rewritten function
- black list / white list of functions allowed to be inlined
- restriction on call depths for inlining

Transformation: Spezialization using Known State

- maintain “known-ness” of registers / stack frame content
  - memory defaults to being unknown (unless configured)
- known values make transformed code more specialized
  - “known-ness” information can deliberately be thrown away
- same code to be transformed for different “known-ness” state may produce different results
  - may result in “run-away” traversals ➔ buffer overflow
  - automatically provides loop unrolling
  - restricted by migrating known to unknown state (by inserting “compensation code”) ➔ configuration: prohibit loop unrolling

Transformation: Traversal

Traverse all reachable execution paths

- non-branch instructions
  - only known operands: emulated, no resulting code generated (constant propagation)
  - otherwise: forward to resulting code, embed known values
- branch with known target
  - proceed unless configured otherwise (over calls: inlining)
- branch with unknown targets
  - generate new paths to traverse
  - start new block to transform
  - merge points for backward jumps (for same known-ness)
  - “ret”: finish path, forward “ret”, proceed with next path

Transformation: Example

C code and resulting compiled machine code (AT&T):

```
int foo(int i, int j)
{
  if (i == 5) return 0;
  return i+j;
}
```

```
<foo>:
    add    %edi,%esi
    mov    $0x0,%eax
    cmp    $0x5,%edi
    cmovne %esi,%eax
    ret
```

Request transformation specializing for 1st par set to 2:

```
dbrew_set_func(foo);
dbrew_set_staticpar(0); // 1st parameter known
foo_t f = (foo_t) dbrew_rewrite(2, 3);
```

Transformation: Example – Debug Output

Static State:
Registers: %rsp (R 0), %rdi (0x2), %rip = 400a40

```
<foo>:
    add    %edi,%esi
    mov    $0x0,%eax
    cmp    $0x5,%edi
    cmovne %esi,%eax
    ret
```

Transformation: Example – Debug Output

```
Static State:
  Registers: %rsp (R 0), %rdi (0x2), %rip = 400a40

Process '0x400a40: add %edi,%esi'
Capture 'add $0x2,%esi' (into 0x400a40|0 + 1)
Static State:
  Registers: %rsp (R 0), %rdi (0x2), %rip = 400a42
```

```
<foo>:
    add    %edi,%esi
    mov    $0x0,%eax
    cmp    $0x5,%edi
    cmovne %esi,%eax
    ret
```

Transformation: Example – Debug Output

```
Static State:
  Registers: %rsp (R 0), %rdi (0x2), %rip = 400a40

Process '0x400a40: add %edi,%esi'
Capture 'add $0x2,%esi' (into 0x400a40|0 + 1)
Static State:
  Registers: %rsp (R 0), %rdi (0x2), %rip = 400a42

Process '0x400a42: mov $0x0,%eax'
Static State:
  Registers: %rax (0x0), %rsp (R 0), %rdi (0x2), %rip = 400a47
```

```
<foo>:
    add     %edi,%esi
    mov     $0x0,%eax
    cmp     $0x5,%edi
    cmovne  %esi,%eax
    ret
```

Transformation: Example – Debug Output

```
Static State:
  Registers: %rsp (R 0), %rdi (0x2), %rip = 400a40

Process '0x400a40: add %edi,%esi'
Capture 'add $0x2,%esi' (into 0x400a40|0 + 1)
Static State:
  Registers: %rsp (R 0), %rdi (0x2), %rip = 400a42

Process '0x400a42: mov $0x0,%eax'
Static State:
  Registers: %rax (0x0), %rsp (R 0), %rdi (0x2), %rip = 400a47

Process '0x400a47: cmp $0x5,%edi‘
Static State:
  Registers: %rax (0x0), %rsp (R 0), %rdi (0x2), %rip = 400a4a
  Flags: CF (1) ZF (0) SF (1) OF (0) PF (0)
```

```
<foo>:
    add     %edi,%esi
    mov     $0x0,%eax
    cmp     $0x5,%edi
    cmovne  %esi,%eax
    ret
```

Transformation: Example – Debug Output

Static State:
  Registers: %rsp (R 0), %rdi (0x2), %rip = 400a40

Process '0x400a40: add %edi,%esi'
Capture 'add $0x2,%esi' (into 0x400a40|0 + 1)
Static State:
  Registers: %rsp (R 0), %rdi (0x2), %rip = 400a42

Process '0x400a42: mov $0x0,%eax'
Static State:
  Registers: %rax (0x0), %rsp (R 0), %rdi (0x2), %rip = 400a47

Process '0x400a47: cmp $0x5,%edi'
Static State:
  Registers: %rax (0x0), %rsp (R 0), %rdi (0x2), %rip = 400a4a
  Flags: CF (1) ZF (0) SF (1) OF (0) PF (0)

Process '0x400a4a: cmovnz %esi,%eax'
Capture 'mov %esi,%eax' (into 0x400a40|0 + 2)
Static State:
  Registers: %rsp (R 0), %rdi (0x2), %rip = 400a4a, %rip = 400a4d
  Flags: …

<foo>:
    add   %edi,%esi
    mov $0x0,%eax
    cmp $0x5,%edi
    cmovne %esi,%eax
    ret

Transformation: Example – Result

```
<foo>:
    add     %edi,%esi
    mov     $0x0,%eax
    cmp     $0x5,%edi
    cmovne  %esi,%eax
    retq
```

→

```
<foo>:
    add     $0x2,%esi
    mov     %esi,%eax
    ret
```

First Results

Directly generate machine code after transformation

Optimizations missing yet

- register renaming after inlining
  (values in registers used for parameters often may get saved/restored)
- reduce stack spilling by using registers freed due to specialization
- …

Still should work already quite well

- we transform existing optimized machine code

First Results: Generic 2d stencils

```
Stencil s5 = {5, { { 0, 0, .4},
                   {-1, 0, .1},
                   { 1, 0, .1},
                   { 0,-1, .1},
                   { 0, 1, .1} }};

double apply(double *m, int xsize, Stencil* s)
{
    double res;
    int i;

    res = 0;
    for(i=0; i<s->points; i++) {
        StencilPoint* p = s->p + i;
        res += p->factor * m[p->xdiff + p->ydiff * xsize];
    }
    return res;
}
```

First Results: Generic 2d stencils

```
BB 0x7fc4c4b90000 (17 instructions):
      0x7fc4c4b90000:  c5 f9 57 c0           vxorpd   %xmm0,%xmm0,%xmm0
      0x7fc4c4b90004:  c5 fb 10 0f           vmovsd   (%rdi),%xmm1
      0x7fc4c4b90008:  c5 f3 59 0c 25 18 71  vmulsd   0x627118,%xmm1,%xmm1
      0x7fc4c4b9000f:  62 00
      0x7fc4c4b90011:  c5 fb 58 c1           vaddsd   %xmm1,%xmm0,%xmm0
      0x7fc4c4b90015:  c5 fb 10 4f f8        vmovsd   -0x8(%rdi),%xmm1
      0x7fc4c4b9001a:  c5 f3 59 0c 25 28 71  vmulsd   0x627128,%xmm1,%xmm1
      0x7fc4c4b90021:  62 00
      0x7fc4c4b90023:  c5 fb 58 c1           vaddsd   %xmm1,%xmm0,%xmm0
      0x7fc4c4b90027:  c5 fb 10 4f 08        vmovsd   0x8(%rdi),%xmm1
      0x7fc4c4b9002c:  c5 f3 59 0c 25 38 71  vmulsd   0x627138,%xmm1,%xmm1
      0x7fc4c4b90033:  62 00
      0x7fc4c4b90035:  c5 fb 58 c1           vaddsd   %xmm1,%xmm0,%xmm0
      0x7fc4c4b90039:  c5 fb 10 8f b0 e0 ff  vmovsd   -0x1f50(%rdi),%xmm1
      0x7fc4c4b90040:  ff
      0x7fc4c4b90041:  c5 f3 59 0c 25 48 71  vmulsd   0x627148,%xmm1,%xmm1
      0x7fc4c4b90048:  62 00
      0x7fc4c4b9004a:  c5 fb 58 c1           vaddsd   %xmm1,%xmm0,%xmm0
      0x7fc4c4b9004e:  c5 fb 10 8f 50 1f 00  vmovsd   0x1f50(%rdi),%xmm1
      0x7fc4c4b90055:  00
      0x7fc4c4b90056:  c5 f3 59 0c 25 58 71  vmulsd   0x627158,%xmm1,%xmm1
      0x7fc4c4b9005d:  62 00
      0x7fc4c4b9005f:  c5 fb 58 c1           vaddsd   %xmm1,%xmm0,%xmm0
      0x7fc4c4b90063:  c3                    ret
```

First Results: Generic 2d stencils

Matrix 1002^2 elements, 1000^2 updates, 1000 iterations
Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz

Generic version:	7.4 s
Rewritten:	3.5 s
Manual 5p version:	2.9 s

Grouped factors (nested loop, outer over factors)

Generic version:	8.5 s
Rewritten:	2.8 s

Always called via function pointers (!): no vectorization…

DBrew Snippets

Snippets

- short functions provided by DBrew
- semantic is known to DBrew (obviously)
- if called in code to be transformed, snippets can
  - specify DBrew configuration or meta information
  - may do different things depending on configuration
  - can be replaced with semantically identical code

Example

- mark an int value to become known on rewriting

```
int dbrew_mark_known(int i) { return i; }
```

(this is basically a NOP when inlined)

DBrew Vectorization

Transform a scalar kernel into a vectorized variant

- input parameter is marked to be a scalar FP
- generates a variant with the parameter being a vector
- all operations on the “to-be-vectorized” value will be replaced by element-wise vector operations
  - example (x86 AVX): vmulsd ➔ vmulpd

How to call vectorized variant?

- via DBrew snippet which adapts to expansion done (may depend on architecture: x2 for SSE, x4 for AVX)

DBrew Vectorization: Example

Kernel

```
double add_kernel(double v1, double v2)
{ return v1 + v2; }
```

Snippet

```
void dbrew_apply4_R8V8V8(dbrew_func_R8V8V8_t f,
                         double* ov, double* i1v, double* i2v)
{   ov[0] = (f)(i1v[0], i2v[0]);
    ov[1] = (f)(i1v[1], i2v[1]);
    ov[2] = (f)(i1v[2], i2v[2]);
    ov[3] = (f)(i1v[3], i2v[3]); }
```

Usage

```
void vadd(double* dst, double* src1, double* src2, int n)
{
    for(; n>0; n-=4, dst+=4, src1+=4, src2+=4)
        dbrew_apply4_R8V8V8(add_kernel, dst, src1, src2);
}
```

DBrew Vectorization: Example

Transfrom vadd to use vectorized add kernel (AVX)

```
dbrew_set_func(vadd);
dbrew_set_vectorsize(32);
vadd32 = (vadd_t) dbrew_rewrite(a, b, c, len);
```

Results for 20 iterations of vadd (10 mio elements)

- naïve (simple C loop): 0.40 s
- un-transformed snippet: 0.43 s
- rewritten-16 (SSE): 0.36 s
- rewritten-32 (AVX): 0.35 s

(AVX has unneeded prolog/epilog)

Experiments with LLVM

Experimental backend

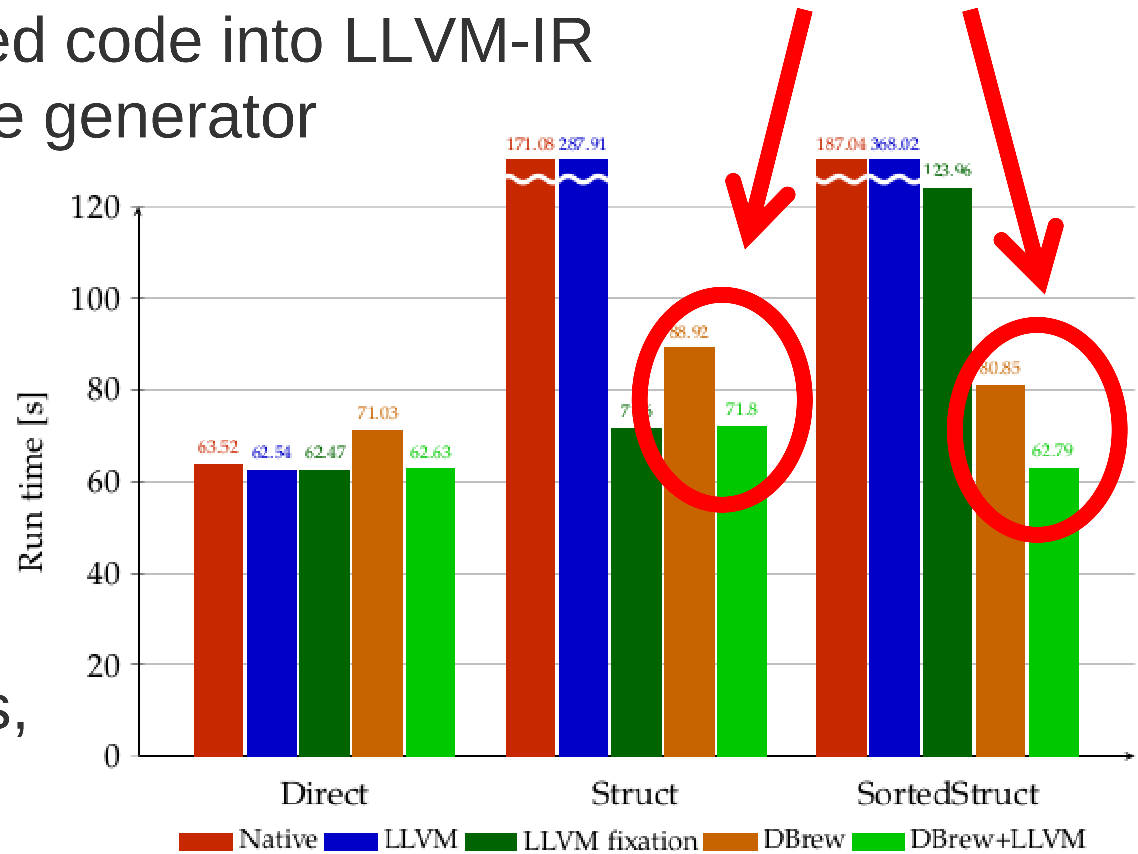
- translate transformed code into LLVM-IR
- use LLVMs JIT code generator

Results for
2d stencil variants

Experiences

- DBrew does well
- much meta info required (signatures, pointers vs. int)
- useful LLVM opts

Benefits



(a) Running times where a single element is computed in one step.

Future Work

Internals
- low-hanging optimizations in own generator backend
- use 3rd-party decoders/generators (e.g. Valgrind VEX)
- validation: transformations correct?

Usage
- minimize the overhead of dynamic data distributions
- abstractions for iteration spaces, dynamic data layout

Discussion
- other usages
- better user interface (in C++, …)

Thanks – Questions?

github.com/lrr-tum/dbrew

Josef.Weidendorfer@in.tum.de