A Case for Shipping ALL Software Using Virtual Instruction Sets: The ALLVM Project

Vikram Adve and Will Dietz

With: Sean Bartell, Tom Chen, Sandeep Dasgupta, Theodoros Kasampalis, Maria Kotsifakou and Hashim Sharif

University of Illinois at Urbana-Champaign

Alumni: Chris Lattner, John Criswell, Swarup Sahoo

Supported by: ONR, NSF, SRC, DARPA
Virtual Instruction Set Computing

Key: Virtual ISA can enable far richer analyses, transforms than native ISA
Informal Definition:

\[ \text{\textbf{VISc}} \equiv \text{Software ISA differs from Target Machine ISA} \]

Key: Virtual ISA can enable far richer analyses, transforms than native ISA
**Popular Native Code Systems (Not VISC)**

“**VISC**” == Ship code as Virtual ISA (e.g., JVM, PTX)

Native code is pervasive for two broad classes of software

<table>
<thead>
<tr>
<th>High performance software is <em>largely</em> shipped as native code</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC applications</td>
</tr>
<tr>
<td>Media, Gaming, Finance, CAD, …</td>
</tr>
<tr>
<td>Web browsers</td>
</tr>
<tr>
<td>Database systems</td>
</tr>
<tr>
<td>Libraries galore</td>
</tr>
</tbody>
</table>
Static Compilation is NOT Enough

Modern software architectures
- Install-time configurations, software environments
- User-installed extensions, dynamically loaded libraries, layering

Modern hardware architectures
- Diverse vector hardware, GPUs, accelerators in SoCs

Modern security challenges due to untrusted code
- Browser extensions, mobile app markets, BYOD

Need rich analyses and transformations on end-user systems
All future software should “ship” using Virtual ISAs.
NOTE: Different systems can use different Virtual ISAs.

• The security benefits are strong

• There are no inherent performance penalties (and novel performance benefits are possible)

• It is technically feasible and commercially acceptable
Myth: Virtual ISA Threatens IP

Fact: *Binary code can be reverse engineered effectively using interactive tools + manual analysis*

- Better Solution #1: Encryption + Code Signing
- Better Solution #2: Obfuscation tools (must not interfere with program analyses)
ALLVM: Ship All Software as Virtual ISAs

Key: Virtual ISAs enable far richer analyses, transforms than native ISA
**LLVM Virtual Instruction Set and IR**

/* C Source Code */
int SumArray(int A[], int Num)
{
    int i, sum = 0;
    for (i = 0; i < Num; ++i)
        sum += A[i];
    return sum;
}

/** LLVM Code */
int %SumArray(int* %A, int %Num)
{
    bb1:
    %cond = icmp sgt i32 %Num, 0
    br i1 %cond, label %bb2, label %bb3
    bb2:
    %sum0 = phi i32 [ %t10, %bb2 ], [ 0, %bb1 ]
    %iv   = phi i64 [ %inc, %bb2 ], [ 0, %bb1 ]
    %t2   = getelementptr inbounds i32* %A, i64 %t7
    %t3   = load i32* %t2 , align 4
    %t4   = add nsw i32 %t3 , %sum0
    %inc  = add nuw i64 %iv , 1
    %t5   = trunc i64 %iv to i32
    %exitcond = icmp eq i32 %inc , %Num
    br i1 %exitcond , label %bb3 , label %bb2
    bb3:
    %sum1  = phi i32 [ 0, %bb1 ], [ %t4 , %bb2 ]
    ret int %sum1
}

- Simple, 3-address IR
- Architecture-neutral
- Language-neutral
- Explicit CFG
- Always in SSA form
- Typed memory, regs

LLVM enables sophisticated program analyses and transformations
Why LLVM IR for ALLVM? (1 of 2)

1. Fully executable virtual ISA

- Language-neutral; hardware-neutral; and a rich IR
- Extensive production-quality infrastructure and tools
- Widely used: Apple, Google, Intel, QCOM, ARM, ...
- Numerous front-ends: C, C++, (Fortran), .NET, Swift, Python, Ruby, Haskell, ...

Available at: llvm.org
First release: October 2003
## 2. Emerging adoption as a Virtual ISA

<table>
<thead>
<tr>
<th></th>
<th>Compile-time</th>
<th>Link-time</th>
<th>Install-time</th>
<th>Load/Run-time</th>
<th>Idle-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple, Sony, Intel, QCOM, ...</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Apple) tvOS, watchOS, iOS</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MacOS OpenGL</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenCL SPIR</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Renderscript</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Google) PNaCl</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

“For iOS™ apps, bitcode is the default, but optional. For watchOS™ and tvOS™ apps, bitcode is required.”

-- iOS App Distribution Guide, Apple
But Many Unanswered Questions

Not enough research on benefits of a Virtual ISA
For software in static languages (C, C++, Fortran, OpenMP, ...)

Uses to date are limited, ad hoc, and haphazard

- What are the benefits for performance?
- What are the benefits for security?
- What are the benefits for software reliability?
**ALLVM Toolchain**

- **Self-hosting**: all tools ship in `.allexe` format
- **Developer site**: `allbin` → `Clang` → `bc2allvm` → `alltogether`
- **User site**: `allbin` → `Clang` → `bc2allvm` → `alltogether`

- **Clang**: C, C++ → `Linker + IP Optimizer`
- **Linker + IP Optimizer**: LLVM → `bc2allvm` → `alltogether`
- **bc2allvm**: LLVM → `alltogether`
- **alltogether**: LLVM → `allout` → `allready`
- **allout**: AOT opt + code gen → `allready`
- **allready**: Native object code cache
- **alley**: `Execution YEngine`

---

**Notes**:
- `allbin` is a tool that compiles C, C++ code into `.bin` format.
- `bc2allvm` converts LLVM bitcode into `.allexe` format.
- `alltogether` is the final stage where all tools interact.
- `allout` performs additional optimizations and generates code in `.allexe` format.
- `allready` stores native object code.
- `alley` is the execution engine for `.allexe` files.
Self-bootstrap: **Clang (C++)**: bash + cmake → make + clang → bc2allvm → altogether → allout / cache → alley

Substantial userspace software, tools work in ALLVM:
- xterm, libX11, vim, spidermonkey
- openssl, openssh
- (apache) httpd, nginx, redis, memcached, postgresql
- subversion, git
- binutils, coreutils, bash, zsh, tcsh
- lua, perl, python, ocaml (the C-based bits anyway)

Substantial capabilities for userspace:
- Runs on top of existing Linux OS, or in Docker
- **Binary cache**: Local and remote (trusted)
- **Nix package manager**: Atomic software upgrades

Adding more packages is “easy” if build system is somewhat sane
ALLVM Research Goals: What are the Benefits?

Security

- Secure Virtual Architecture
  (John Criswell; PhD '14)

  Runner-up, ACM Doctoral Dissertation Award
ALLVM Research Goals: What are the Benefits?

**Security**

- Secure Virtual Architecture
  (*John Criswell*; PhD '14)

**Software Reliability**

- Automated fault localization
  (*Swarup Sahoo*; PhD '12)

- Distributed system fault diagnosis
  (*Sean Bartell*)

- Verified codegen: Increasing trust in shipping virtual ISAs
  (*Theodoros Kasampalis*)
ALLVM Research Goals: What are the Benefits?

**Security**
- Secure Virtual Architecture  
  *(John Criswell; PhD '14)*

**Software Reliability**
- Automated fault localization  
  *(Swarup Sahoo; PhD '12)*
- Distributed system fault diagnosis  
  *(Sean Bartell)*
- Verified codegen: Increasing trust in shipping virtual ISAs  
  *(Theodoros Kasampalis)*

**Performance**
- Whole-system optimization; deduplication  
  *(Will Dietz)*
- Software specialization and debloating  
  *(Hashim Sharif)*
- *allready*: Binary-to-LLVM  
  *(Sandeep Dasgupta)*
- Autotuning: install-time search  
  *(Yishen Chen)*
- HPVM: Heterogeneous parallel systems  
  *(Maria Kotsifakou)*
Outline: Applications of ALLVM

• Code deduplication with *software multiplexing*
  
• Debloating via program customization

• Binary translation to LLVM IR
Sources of Code Duplication

- Duplicated libraries across applications
- Multiple versions of libraries or applications on a system
- Duplicated functions or code fragments within / across applications

*Software multiplexing is a framework to address all three issues (current system addresses first two)*
Code Duplication Across Software Versions

Multiple versions of a tool or library often co-exist on the same machine
=> extensive duplication
Multiple tools in a package often share extensive code

Nodes are functions (as hash value)
Edges mark equivalence; colored regions are dense with edges
## Example: Code Deduplication with *Allmux*

### Size and performance of LLVM *linux-x86-64* software release

<table>
<thead>
<tr>
<th>Build config</th>
<th>Binaries size</th>
<th>Libraries size</th>
<th>Total size</th>
<th>Performance</th>
<th>Startup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>590M</td>
<td>2.1M</td>
<td>592M</td>
<td>Better</td>
<td>Fast</td>
</tr>
<tr>
<td>Shared (libllvm)</td>
<td>231M</td>
<td>38M</td>
<td>269M</td>
<td>Worse</td>
<td>Slow</td>
</tr>
<tr>
<td>Shared (separate libs)</td>
<td>11M</td>
<td>93M</td>
<td>104M</td>
<td>Worst</td>
<td>Slowest</td>
</tr>
<tr>
<td><em>Allmux</em></td>
<td>85M</td>
<td>0M</td>
<td>85M</td>
<td>Best</td>
<td>Fastest</td>
</tr>
</tbody>
</table>
Total time for building Clang system with four LLVM versions: Allmux version faster than dynamically linked versions because lower startup cost.
E.g., 1-10 apps that use Qt toolkit

*Allmux* uses **2.5x** less RAM vs. static linking; **35%** less than dynamic

*Allmux* performance is **much better than** dynamic linking; comparable to static
1) Software Multiplexing: $N$ pgms + $K$ libs $\rightarrow$ 1 pgm + $K$ libs

*Exposes duplicated code between programs, libs*

```c
int main(int argc, char* argv[], char* envp[]) {
    if (!strcmp(argv[0], "program-name1")) main1(argc, ...);
    if (!strcmp(argv[0], "program-name2")) main2(argc, ...);
    if (!strcmp(argv[0], "program-name3")) main3(argc, ...);
    if (!strcmp(argv[0], "program-name4")) main4(argc, ...);
    ...
}
```

“Designed in” by a few packages, e.g., GCC

Affects the build system heavily $\Rightarrow$ hard to add manually today
1) Software Multiplexing: $N$ pgms + $K$ libs $\rightarrow$ 1 pgm + $K$ libs

*Exposes duplicated code between programs, libs*

**Key:** IR-level compiler pass adds multiplexing
2) Bitcode for *all* components including dynamic libraries enables **linking before code generation**

⇒ **static linking without rewriting build system!**
Next Steps on Deduplication with ALLMUX

- Identify equivalent functions #1: structural equivalence
- Identify equivalent functions #2: semantic equivalence
- Identify equivalent fragments: perhaps by hashing

Towards a Bitcode Database
The one repository to rule them all!
Outline: Applications of ALLVM

• Code deduplication with software multiplexing

• Debloating via customization to a configuration

• Binary translation to LLVM IR
Configuration-based Slimming

- Customize for user-defined program configuration
  - Generate specialized binaries
  - Reducing code bloat as a result of specialization
Specialization transforms

1. Identify code that parses input configuration
2. Fully unroll only the loop(s) that parse inputs
3. Mark config variables that hold constant values
4. Aggressive interprocedural const. propagation for marked variables only
5. Aggressive constant evaluation, specialization
Experiments

- **Goal**: Compare against existing state-of-the-art Partial Evaluation tool (Occam)

- **Benchmarks**:
  - 7 OpenWRT programs: *optimized for embedded systems*
  - 7 Commonly used Linux programs
  - Yices – SMT Solver

- **18.35% Geom. mean code reduction across 14 programs**
OpenWRT programs

![Bar chart showing binary size as percentage of baseline for different programs like ifstat, httping, tcptraceroute, knockd, aircrack-ng, aireplay-ng, airtun-ng, with three categories Baseline, PE, TRIMMER. The chart compares the size reduction achieved by TRIMMER against PE and Baseline for each program.](image-url)
Linux Programs

![Graph showing binary size as percentage of baseline for different programs]

- **Baseline**
- **PE**
- **TRIMMER**

Programs compared:
- curl
- gprof
- readelf
- objdump
- netstat
- memcached
- bzip2
Binary to LLVM Translation

Led by: Sandeep Dasgupta
with Ed Schwartz (CMU)
already: Binary-to-LLVM

Preference: Only a few components will be binary

Motivation

- Some software components will only be in binary format
- Existing tools inadequate: McSema, BAP, SecondWrite, Qemu

Goal

- Extract “rich” LLVM IR from binary code
- Enable full set of ALLVM optimizations on partial-binary programs
- Needs variable info, type info, per-procedure stack frames
IR extracted by McSema is executable but very “low-level”

- Models runtime process stack as unified flat array
- Machine registers mapped in flat memory, not SSA virtual registers
- No information about variables, types, call graph, exceptions, etc.

Added stack deconstruction

- Recovers individual stack frames per function
- Distinguishes current vs. parent frame pointers
- Tested using McSema test suite; custom test cases
Stack Deconstruction

foo:
push %rbp
mov %rsp, %rbp
mov 16(%rbp), %rax
mov 24(%rbp), %r10

foo:
push %rbp
mov %rsp, %rbp
mov 8(%parent_stack_end), %rax
mov 16(%parent_stack_end), %r10
Ongoing Work

- Identify variables and promote them as symbols
- Represent every symbol in the IR with a meaningful type instead of the generic types provided by McSema

```c
unsigned int foo(char* buf) {
    unsigned alligned_len = 0;
    unsigned int c = strlen(buf);
    if(c%8 == 0 ) {
        return c;
    }
    alligned_len = 8* (c/8) + 8;
    return allign_len;
}
```

<table>
<thead>
<tr>
<th>Variable Names</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) buf</td>
<td>char*</td>
</tr>
<tr>
<td>2) c</td>
<td>unsigned int</td>
</tr>
<tr>
<td>3) alligned_len</td>
<td>unsigned int</td>
</tr>
</tbody>
</table>

1) and 2) inferred using `strlen` prototype
3) inferred using arithmetic operation
Proposal

All future software should ship as virtual ISAs

• The security benefits are strong

• There are no inherent performance penalties (and novel performance benefits are possible)

• It is technically feasible and commercially acceptable

http://allvm.org
## Summary and Implications

<table>
<thead>
<tr>
<th>Application / product areas</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LLVM</strong></td>
</tr>
<tr>
<td>Compilers; Mobile software; Security</td>
</tr>
<tr>
<td><strong>HPVM</strong></td>
</tr>
<tr>
<td>Mobile and embedded SoCs; Accelerators</td>
</tr>
<tr>
<td><strong>DLVM</strong></td>
</tr>
<tr>
<td>DNN toolkits and systems</td>
</tr>
<tr>
<td><strong>ALLVM</strong></td>
</tr>
<tr>
<td>Late-stage software customization; debloating; autotuning</td>
</tr>
</tbody>
</table>
Translation Validation for Increasing Trust in Compilation of Shipped Code

Led by: Theodoros Kasampalis
with Daejun Park and Prof. Grigore Rosu
Cross-Language Program Equivalence with Application to LLVM

Theodoros Kasampalis, PhD student

Joint work with Daejun Park, Vikram Adve, Grigore Rosu
Motivation

• Low trust in code generation process (bugs, undefined behaviors, etc.)

• Existing solutions are not practical for verified code generation with production-quality compilers (e.g. LLVM)

  ➢ Verified Compilers (e.g. CompCert) – built from scratch

  ➢ Translation Validation (e.g. LLVM-MD) – used primarily for same language transformations
TV prototype for LLVM ISel (IR to x86-64)

LLVM ISel

Sync Point Generator

Virtual X86 function

KEQ language-independent equivalence checker

\text{hint-gen}

\text{common.k}

\text{Vx86.k}

\text{llvm.k}

compiler hints

sync points
KEQ: K Equivalence Checker

- **Input:** a relation over symbolic states – called synchronization points
- **Output:** a bisimulation proof of program equivalence
  - Leverages our cut-bisimulation theory
- **Built on top of K Framework**
  - Leverages the K symbolic execution engine
- **Language-independent**
  - Parametrized with the input and output language semantics
  - Definitions defined in K
LLVM Instruction Selection Phase

• Translates LLVM IR into various target ISAs
  - primary language translation step beyond the front-end
  - 140,000 lines of C++ and TableGen code

• IR to Selection DAG for each basic block
  - Amenable to optimal pattern matching selection

• Output: Machine IR
  - Target ISA representation extended with some high-level features
  - Virtual x86: Machine IR for x86-64
## K Semantic Definitions

<table>
<thead>
<tr>
<th></th>
<th>LLVM IR Semantics</th>
<th>Virtual x86 Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Types</strong></td>
<td>• varied-width integer types</td>
<td>• unsigned integers</td>
</tr>
<tr>
<td></td>
<td>• composite array and struct types</td>
<td>• various flag bits</td>
</tr>
<tr>
<td></td>
<td>• the corresponding pointer types</td>
<td>• 64-bit addresses</td>
</tr>
<tr>
<td></td>
<td>• unsigned integers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• various flag bits</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 64-bit addresses</td>
<td></td>
</tr>
<tr>
<td><strong>Features</strong></td>
<td>• (un)signed integer arithmetic</td>
<td>• unsigned integer arithmetic</td>
</tr>
<tr>
<td></td>
<td>• Casts between ptrs/int</td>
<td>• (un)conditional jumps</td>
</tr>
<tr>
<td></td>
<td>• getelementptr</td>
<td>• eflags register</td>
</tr>
<tr>
<td></td>
<td>• (un)conditional branches</td>
<td>• various mov instructions</td>
</tr>
<tr>
<td></td>
<td>• call/ret</td>
<td>• call/ret</td>
</tr>
<tr>
<td></td>
<td>• alloca/load/store</td>
<td></td>
</tr>
<tr>
<td><strong>Memory abstraction</strong></td>
<td>map from symbolic addresses to memory objects represented as byte arrays</td>
<td></td>
</tr>
</tbody>
</table>
Synchronization Point Generator

• **Where?**
  - Beginning/end of each function
  - Before/after each callsite
  - Before each loop header

• **These points are a cut for each function**

• **Constraints over symbolic variables**
  - Describe what parts of the two states should be “the same”
## Synchronization Point Generator

<table>
<thead>
<tr>
<th>Sync Point Type</th>
<th>Constraint</th>
<th>How to generate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td>corresponding args</td>
<td>from calling conv</td>
</tr>
<tr>
<td>Exit</td>
<td>same return value</td>
<td>from calling conv</td>
</tr>
<tr>
<td>Before call</td>
<td>corresponding args, same callee</td>
<td>from calling conv</td>
</tr>
<tr>
<td>Loop header</td>
<td>corresponding live regs</td>
<td>hints + liveness analysis</td>
</tr>
<tr>
<td>After call</td>
<td>same return value, corresponding live regs</td>
<td>from calling conv (return value), hints + liveness analysis</td>
</tr>
</tbody>
</table>

- **Required Static Analysis**
  - Loop detection (natural loops)
  - Liveness analysis

- **Hints**
  - Virtual register correspondence
Example: The Collatz conjecture test

define i32 @collatz(i32 $n) {
  entry:
    ; p0
    br label while_end

  while_cond:
    ; p1, p2
    %x0 = phi i32 [1, entry], [%add1, %if.end]
    %n0 = phi i32 [%n, entry], [%add1, %if.end]
    %temp = icmp eq i32 %n0, 1
    br if %temp, label while.body, label while.end

  while.body:
    %add1 = add i32 %c, 1
    %temp = ures i32 %n, 2
    %temp1 = icmp ne i32 %temp, 0
    br if %temp1, label if.then, label if.else

  if.then:
    %mul1 = mul i32 %n, 3
    %add2 = add i32 %mul1, 1
    br label %if.end

  if.else:
    %div = udiv i32 %n, 2
    br label %if.end

  if.end:
    %x1 = phi i32 [%add0, %if.then], [%div, %if.else]
    br label while_end

  while.end:
    ; p3
    ret i32 %x0
}

define void @collatz() {
  entry:
    ; p0
    %temp = load i32 %collatz
    ; p1
}

define i32 @virtual_mips(
  entry:
    ; p0
    ret i32 %collatz
)

(a) LLVM IR (b) Virtual MIPS
Example: The Collatz conjecture test

<table>
<thead>
<tr>
<th>Sync Point</th>
<th>Prev BB (LLVM)</th>
<th>Prev BB (Vx86)</th>
<th>Equality Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0 (entry)</td>
<td>-</td>
<td>-</td>
<td>%n = edi</td>
</tr>
<tr>
<td>p1</td>
<td>%entry</td>
<td>.LBB0</td>
<td>%n = %vr6_32 1 = %vr7_32</td>
</tr>
<tr>
<td>p2</td>
<td>%if.end</td>
<td>.LBB5</td>
<td>%add1 = %vr2_32 %n.1 = %vr5_32</td>
</tr>
<tr>
<td>p3 (exit)</td>
<td>-</td>
<td>-</td>
<td>%c.0 = eax</td>
</tr>
</tbody>
</table>
Questions?
Example: Sgemm

- A single work item computes TILE_H elements of C
- TILE_M work items cooperate to load TILE_H x TILE_N elements of B in local memory
- Figure shows computation performed by one work group
SGEMM – Dataflow Graph Structure