

# A Case for VISC: Virtual Instruction Set Computers

**Vikram Adve**

**Computer Science Department**

**University of Illinois at Urbana-Champaign**

**with faculty...**

*Sanjay Patel*

*Craig Zilles*

*Roy Campbell*

**graduate students ...**

*Robert Bocchino*

*Michael Brukman*

*Alkis Evlogimenos*

*Brian Gaeke*

*Chris Lattner*

*Patrick Meredith*

*Andrew Newharth*

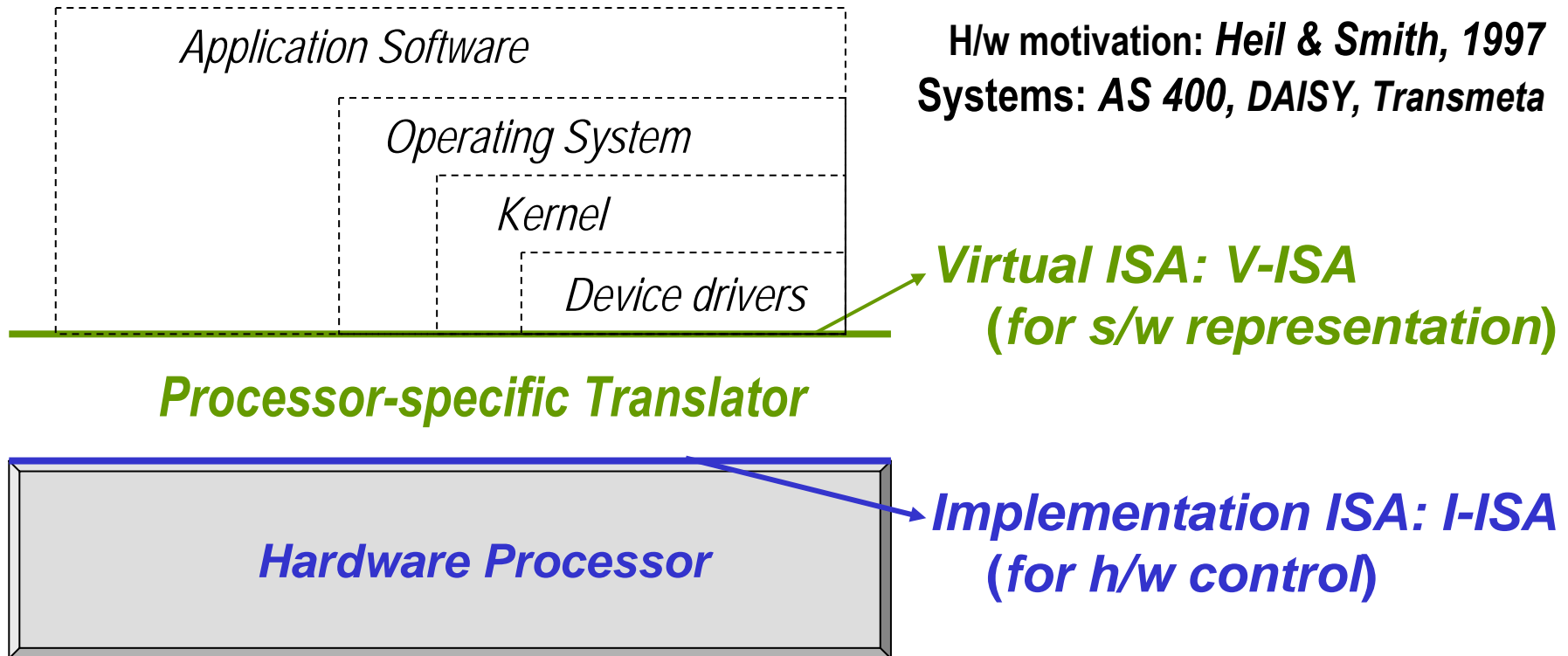
*Pierre Salverda*

**... and programmer**

*John Criswell*

***Supported by NSF (CAREER, NGS00, CPA04),  
Marco/DARPA, IBM, Motorola***

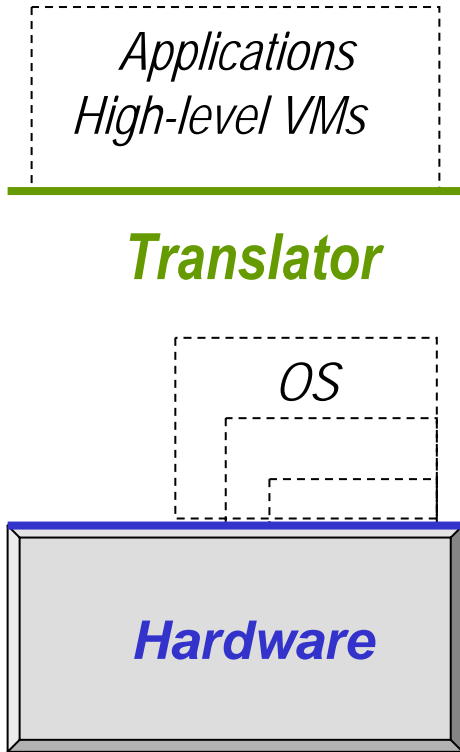
# VISC: Virtual Instruction Set Computers



3 fundamental  
benefits:

1. V-ISA can be much richer than an I-ISA can be
2. Translator and processor can be co-designed, and so truly cooperative
3. All software is executed via a layer of indirection

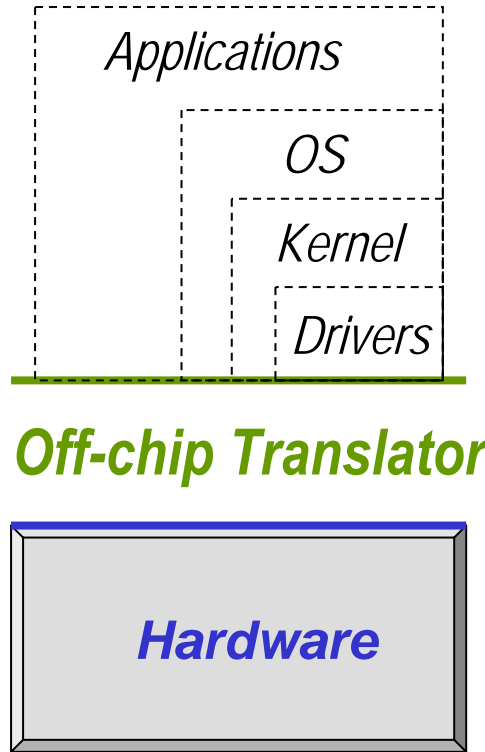
# 3 Models for Deploying VISC



*Compiler, VM benefits*

---

Example: LLVM



*Compiler, VM benefits*

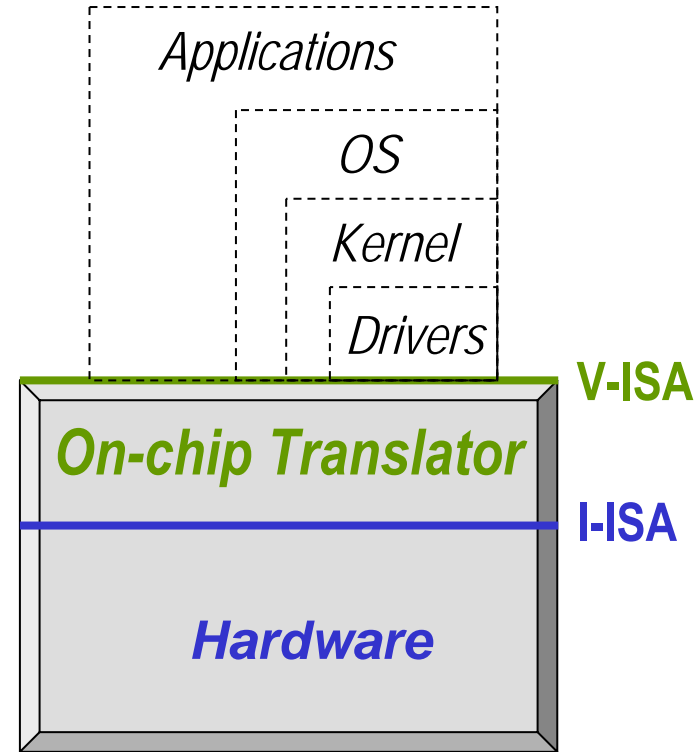
---

*OS benefits*

---

Approx: AS/400

Example: LLVM+LLVA-OS



*Compiler, VM benefits*

---

*OS benefits*

---

*Hardware benefits*

---

Approx: Transmeta, DAISY

Example: LLVA

# LLVA : A Rich Virtual ISA

**Typed assembly language +  $\infty$  SSA register set**

## Low-level, machine-independent semantics

- RISC-like, 3-address instructions
- Infinite virtual register set
- Load-store instructions via typed pointers
- Distinguish stack, heap, globals

## High-level information

- Explicit Control Flow Graph (**CFG**)
- Explicit dataflow: **SSA** registers
- Explicit types: **scalars + pointers, structures, arrays, functions**

# LLVA Instruction Set

<i>Class</i>	<i>Name</i>
arithmetic	<b>add, sub, mul, div, rem</b>
bitwise	<b>and, or, xor, shl, shr</b>
comparison	<b>seteq, setne, setlt, setgt, setle, setge</b>
control-flow	<b>ret, br, mbr, invoke, unwind</b>
memory	<b>load, store, alloca</b>
other	<b>cast, getelementptr, call, select, phi</b>

*Only 28 LLVA instructions (6 of which are comparisons)*

- Most are overloaded
- Few redundancies

# Semantic Information + Language Independence

**Not a universal I.R.**

**Capture operational behavior of high-level languages**

Simple, low-level operations: an assembly language!

Simple type system: ***pointer, struct, array, function***

Primitive exception-handling mechanisms: ***invoke, unwind***

Not all exceptions need to be precise

Transparent runtime system with no new semantics

- Unlike JVM, CLI, SmallTalk, ...

# Is This V-ISA Any Good?

## 1. Information content:

Rich, language-independent optimizations

Aggressive, static optimization

Flexible code reordering

⇒ *Enables sophisticated code generation*

## 2. Code Size:

LLVA / native code: 1.16 (x86), 0.85 (Sparc)

⇒ *Little penalty for extra information*

## 3. Semantic Gap:

Instruction ratio: 2.6 (x86), 3.2 (Sparc)

⇒ *Clear performance relation*

# OS-Independent Offline Translation

## *Define a small OS-independent API*

### Strictly optional...

- OS can choose whether or not to implement this API
- Operations can *fail* for any reason

### Offline caching

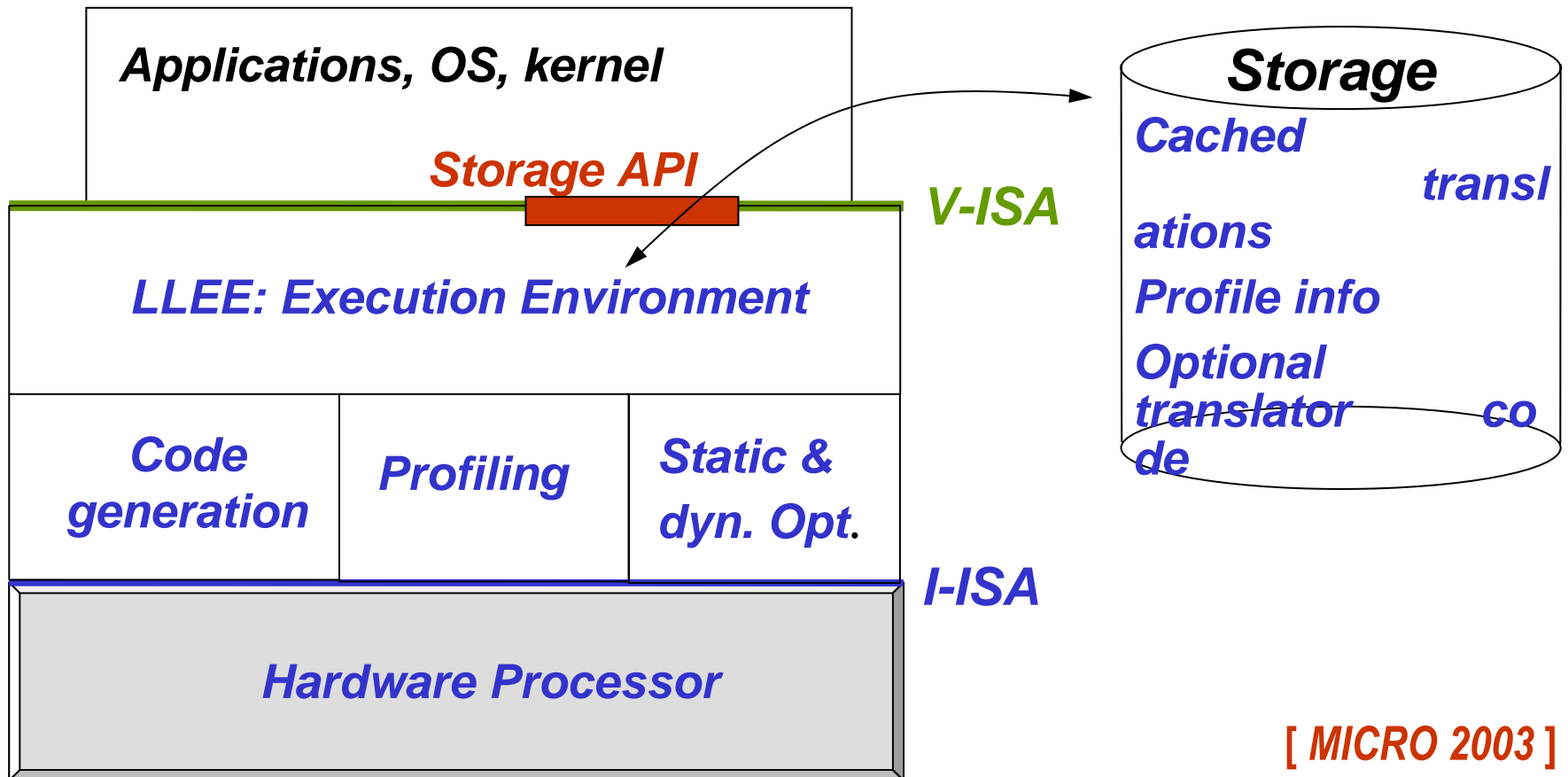
- Read, Write, GetAttributes [an array of bytes]
- *Example:* **void\* ReadArray( char[ ] Key, int\* numRead )**

### Offline translation

- OS “executes” LLVA program in *translate-only mode*

# OS-Independent Translation Strategy

Offline code generation whenever possible,  
online code generation when necessary



# Processor Design Implications of VISC

## ***“Software-controlled” microarchitectures***

*(with Sanjay Patel and Craig Zilles)*

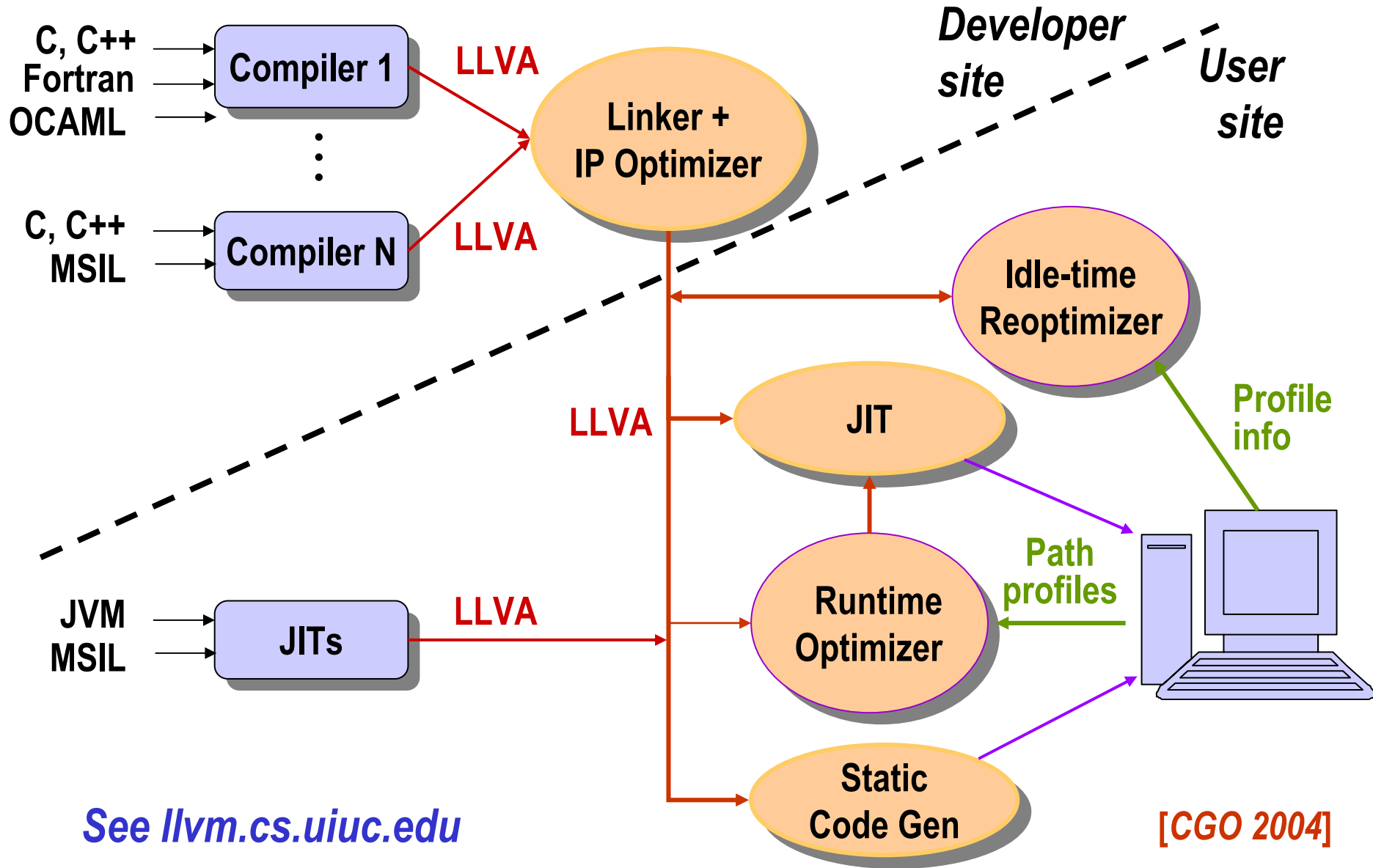
### **General processor design benefits**

- ISA evolution
- Truly cooperative hardware/software
- Rich program information

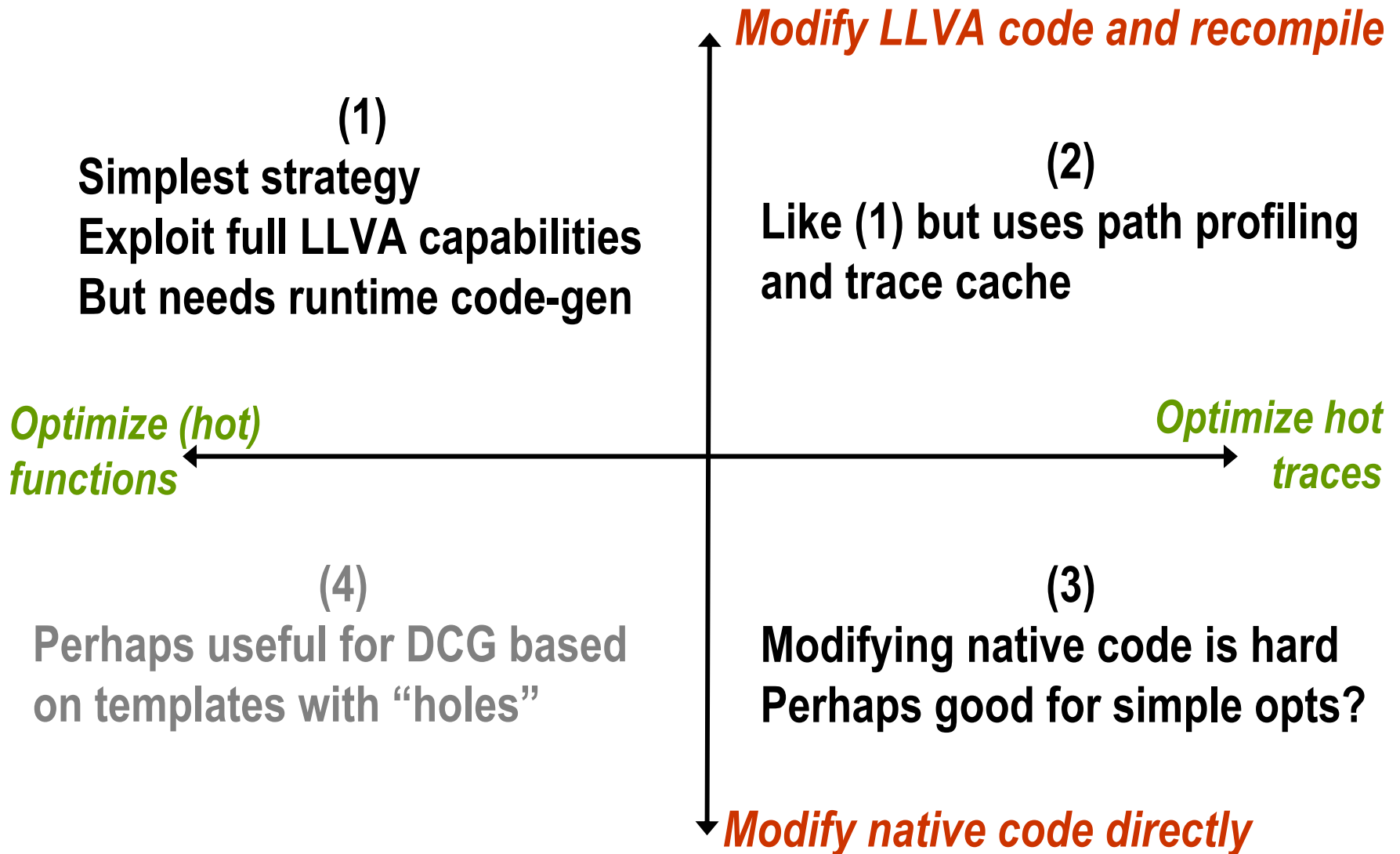
### **Example: String Microarchitectures**

- Translator groups instructions with predictable dataflow (“strings”)
- High ILP with simple, distributed hardware
- Less global traffic between clusters
- Speculation recovery can ignore many local values

# The LLVM Compiler System



# Runtime optimization framework



# Compiler Implications of VISC

## 1. *Back-end code generation is done once, and done well*

### Old Model

Each compiler has a back-end

Each VM has a back-end

Analysis, optimization capabilities  
vary widely

Back-ends have limited chip-  
specific information

### New Model

Single, system-wide back-end:

⇒ worth making powerful

⇒ “front-end” compilers focus on  
language-specific issues

Microarchitecture-aware code  
generation

Hardware support and cooperation  
(profiling, speculation,  
accelerating dynamic opts)

# Compiler Implications of VISC

## 2. “Lifelong” optimization model becomes natural: Direct consequence of rich, persistent representation

### Old Model

Persistent repr: *machine code*

Static languages:

- **Compile-time**
- **Link-time**
- **No end-user profiles**

“Dynamic” languages:

- **Mainly run-time**

### New Model

Persistent repr: *rich V-ISA*

Static *or* dynamic languages:

- **Compile-time (where legal)**
- **Link-time**
- **Install-time**
- **Run-time**
- **“Idle-time” between runs**

End-user profile information

# Compiler Implications of VISC

## 3. *Whole-system optimization becomes practical because of using a single, uniform representation*

### Old Model

*Many opaque boundaries:*

- **Application / OS**
- **Application / VM (partly)**
- **VM / Native**
- **VM / OS**

### New Model

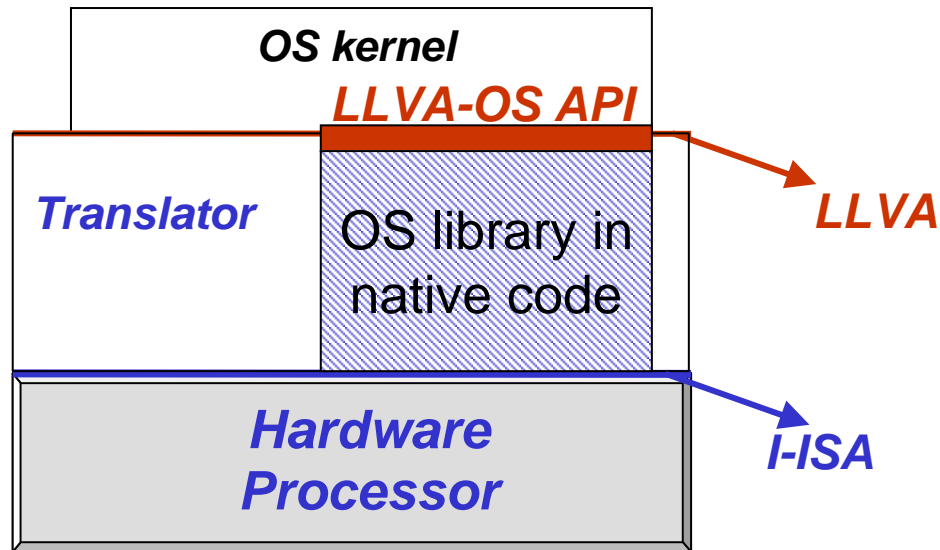
*Entire software stack is uniformly exposed to optimization...*

- **All components**
- **All interfaces**

*... All the time*

- **“lifelong” whole-system optimization**

# OS Strategy: An API for Kernel Mechanisms



(with Roy Campbell)

**Two kernels being ported:**

1. **Linux kernel**
2. **Choices nano-kernel**

## LLVA-OS API: Architectural mechanisms to support kernels

- Defined as a library of "intrinsic functions," i.e., an API
- Examples: *map\_page\_to\_frame*, *register\_interrupt*, *load\_fp\_state*
- Two forms of program state: Native (opaque) and LLVA (visible)

# OS Implications of VISC

## Portability

- OS is portable across hardware revisions

## Security

- Translator adds a layer of abstraction
- Translator can enforce memory safety [LCTES03, ACM TECS04]

## Hardware/Software Flexibility

- OS primitives are visible to architecture
- Different processor designs can choose to accelerate different mechanisms
  - Thread scheduling
  - Exception handling

# Summary

**VISC : decouple s/w representation from h/w control**

⇒ **Software-controlled microarchitectures**

⇒ **Single, hardware-specific, optimizing back-end**

⇒ **Lifelong, whole-system optimization**

⇒ **Portable, flexible operating systems**

[llvm.cs.uiuc.edu](http://llvm.cs.uiuc.edu)

# C and LLVA Code Example

```
struct pair {
    int X; float Y;
};
void Sum(float *, struct pair *P);

int Process(float *A, int N) {
    int i;
    struct pair P = {0,0};
    for (i = 0; i < N; ++i) {
        Sum(A, &P);
        A++;
    }
    return P.X;
}
```

```
%pair = type { int, float }
declare void @Sum(float*, %pair*)

@entry:
    %P = alloca %pair
    %tmp.0 = getelementptr %pair*, %P, long 0, ubyte 0
    store int 0, int* %tmp.0
    %tmp.1 = getelementptr %pair*, %P, long 0, ubyte 1
    store float 0.0, float* %tmp.1
    %tmp.3 = setlt int 0, %N
    br bool %tmp.3, label @loop, label @return

loop:
    %i.1 = phi int [ 0, @entry ], [%i.1, @loop]
    %A.1 = phi float* [ %A.0, @entry ], [%A.2, @loop]
    call void @Sum(float* %A.1, %pair* %P)
    %A.2 = getelementptr float*, %A.1, long 1
    %i.2 = add int %i.1, 1
    %tmp.4 = setlt int %i.2, %N
    br bool %tmp.4, label @loop, label @return

return:
    %tmp.5 = load int* %tmp.0
    ret int %tmp.5
}
```

**tmp.0 = &P[0].0**

**A.2 = &A.1[1]**

Typed pointer arithmetic for explicit access to memory

# Type System Details

## Simple language-independent type system:

- *Primitive types*: void, bool, float, double, [u]int x [1,2,4,8], opaque
- *Only 4 derived types*: pointer, array, structure, function

## Typed address arithmetic:

- `getelementptr %T* ptr, long idx1, long idx2, ...`
- crucial for sophisticated pointer, dependence analyses

## Language-independent like any microprocessor:

- No specific object model or language paradigm
- “cast” instruction: performs any meaningful conversion

# V-ISA Constraints on Translation

**Previous systems faced 3 major challenges**

[Transmeta, DAISY, Fx!32]

## Memory Disambiguation

- Typed V-ISA enables sophisticated pointer, dependence analysis

## Precise Exceptions

- Per-instruction flags: *ignore*, *non-precise*, *precise*

## Self-modifying Code (SMC), Self-extending Code (SEC)

- Translation model makes SEC straightforward, but not SMC

# LLVA Exception Specification

**Key: Requirements are *language-dependent***

## On/off bit per instruction

- *OFF*  $\Rightarrow$  all exceptions on the instruction are ignored
- *ON*  $\Rightarrow$  all applicable exceptions enabled

## All enabled exceptions are precise

- Imprecise exceptions are difficult to use anyway
- *External* compiler can decide which exceptions to enable

# LLVA: Self-modifying Code Specification

**Key: Function-level JIT code generation is *automatic***

## High performance, restricted(?) option:

- Only allowed to modify an *inactive* function (i.e., not on stack)
- Simply invalidate in-memory translation
- JIT will automatically re-translate ...

## Low performance option:

- Modify any instruction any time: Conservative translation, execution

# LLVM Exception Handling Support

## Provide *mechanisms* to implement exceptions

- Do not specify exception semantics (C vs C++ vs Java vs ...)

## LLVM provides two simple instructions:

- **unwind**: Unwind stack frames until reaching an 'invoke'
- **invoke** *fp arg1 ... argN with okLabel except exceptLabel*
  - Call function, but branch to '*exceptLabel*' if unwind
  - **invoke** has two successors (normal and exceptional)
  - Exception edges are explicit in the CFG!

We've implemented C++ and C (setjmp / longjmp) exceptions using this

# A simple C++ example:

**C++**

```
{  
  Class Object; // Has a dtor  
  func();      // Might throw  
  ...  
}
```

**LLVM**

```
      ; Allocate stack space  
%Object = alloca %Class  
      ; Construct object  
call %Class::Class(%Object)  
      ; Call function  
invoke func() to label L1 except label L2  
  
L1: ...  
    ...  
  
L2:      ; Destroy object and continue propagation  
call %Class::~~Class(%Object)  
unwind
```

# Machine Independence (with limits)

## No implementation-dependent features

- Infinite, typed registers
- **alloca**: no explicit stack frame layout
- **call, ret**: typed operands, no low-level calling conventions
- **getelementptr**: Typed address arithmetic

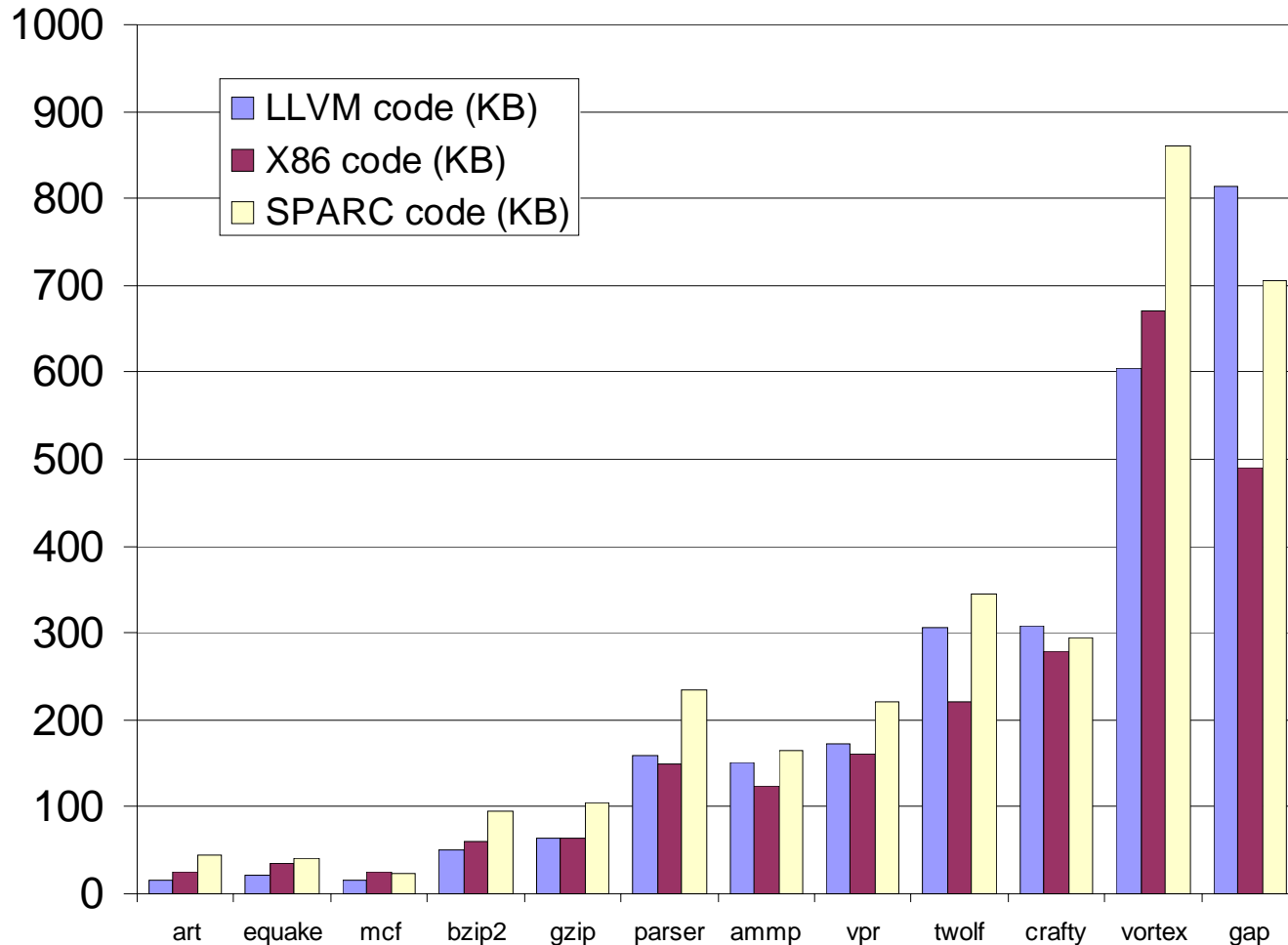
## Pointer-size, endianness

- Encoded in the representation
- Irrelevant for “type-safe” code

**Not a universal instruction set**

**Design the V-ISA for some (broad) family of implementations**

# Static Code Size

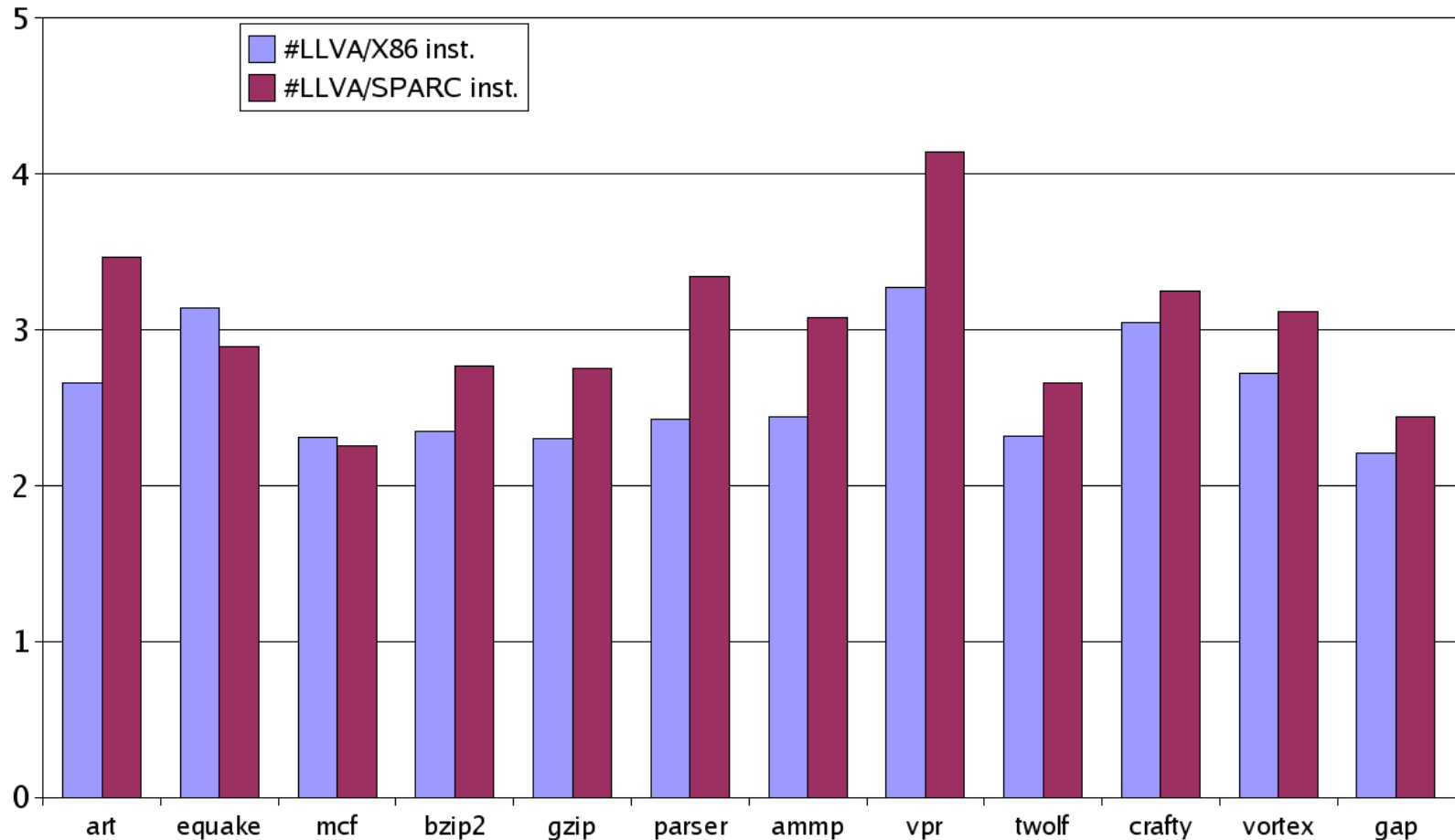


**Average for x86 vs. LLVA: About 1 : 1.16**

**Average for Sparc vs. LLVA: About 1 : 0.85**

**⇒ Little penalty for extra information**

# Ratio of static instructions



**Average for x86:** About **2.6** instructions per LLVA instruction

**Average for Sparc:** About **3.2** instructions per LLVA instruction

⇒ **Very small semantic gap ; clear performance relation**

# SPEC: Code generation time

SPEC benchmark	Translate (s)	Run (s)	Ratio
art	0.03	114.72	0
equake	0.03	18.01	0
mcf	0.02	24.52	0
bzip2	0.04	20.9	0
gzip	0.05	19.33	0
parser	0.16	4.72	0.03
ammp	0.11	58.76	0
vpr	0.14	7.92	0.02
twolf	0.02	9.68	0
crafty	0.45	15.41	0.03
vortex	0.78	6.75	0.12
gap	0.48	3.73	0.13

Typically  $\ll$  1-3% time spent in simple, JIT translation

# Link-time Interprocedural System

***Link-time*  $\equiv$  *Transparent, whole-program optimization***

## Data Structure Analysis (DSA) and Call Graph

- Context-sensitive, field-sensitive, flow-insensitive pointer analysis,
- Context-sensitive call graph

## Automatic Pool Allocation

- Segregate data structures into separate pools on the heap
- 25% to 2.2x improvement over malloc/free on heap-intensive

## Several other useful techniques

- Inlining; dead global elim; dead argument elim; tail call elim
- Unused exception handler elimination
- Static safety checking: array bounds; pointers; stack; heap

# Two-level Path Profiling Algorithm

## Light-weight, two-level profiling:

1. **Find a hot loop region:** simple counter on back edges
2. **Insert path profiling code:** within hot loop regions *and callees*
3. **If top K paths are “hot enough”:** extract, insert in trace cache

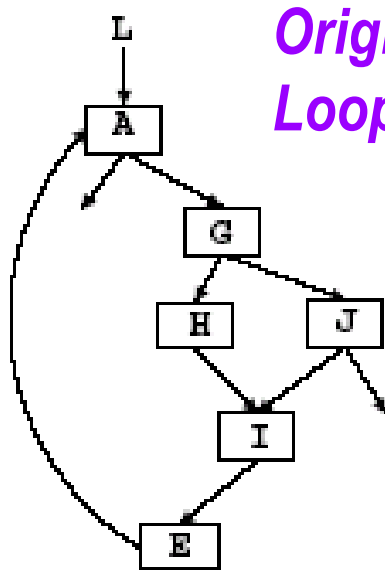
## Strengths

- Finds “**hot hyperblocks,**” not just “hot paths”
- Tracks paths **across procedure calls** (if callee has no loops)
- **Adaptive:** repeat (1) and (2) as often as necessary
- Net performance gain in most cases! : **2% average (9% to -7%)**

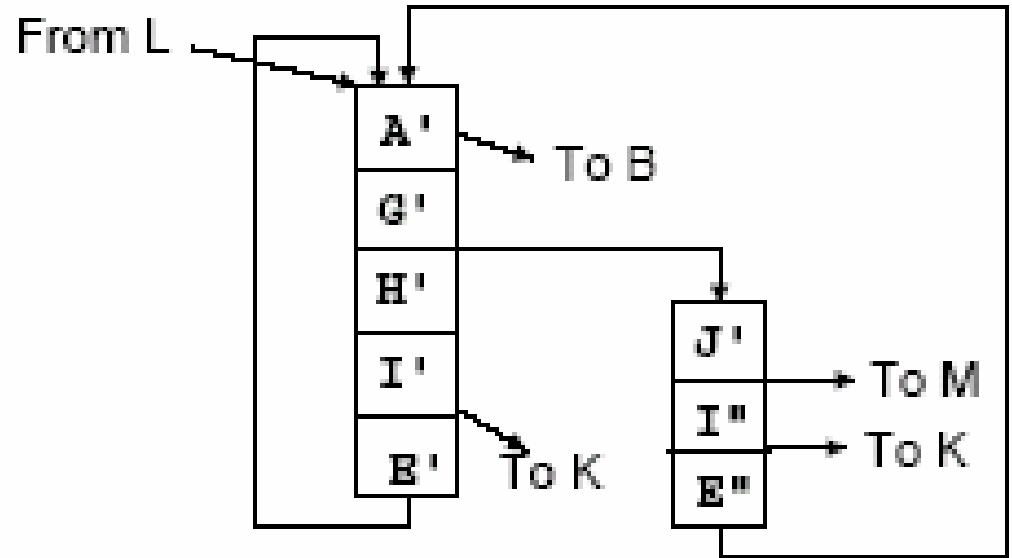
## Weaknesses

- Low coverage in some codes (we are working on this)

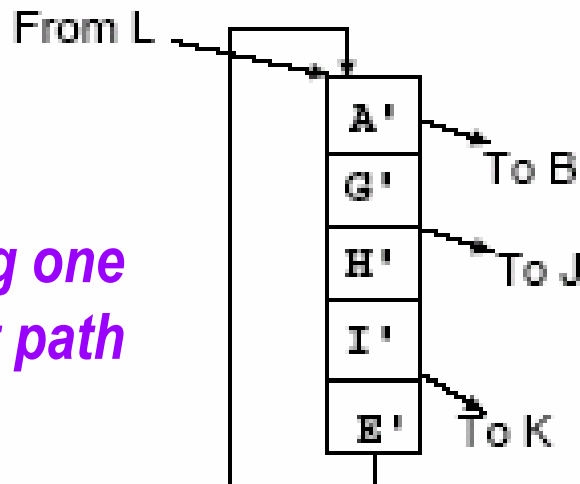
# A Hot Hyperblock vs. a hot Path



*Original  
Loop Region*

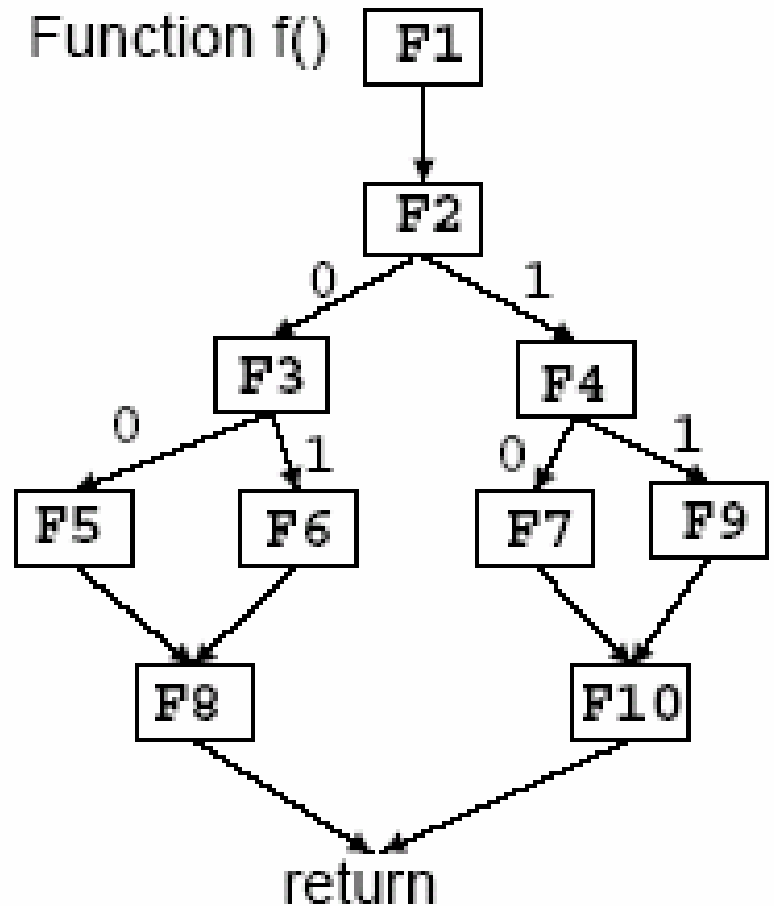
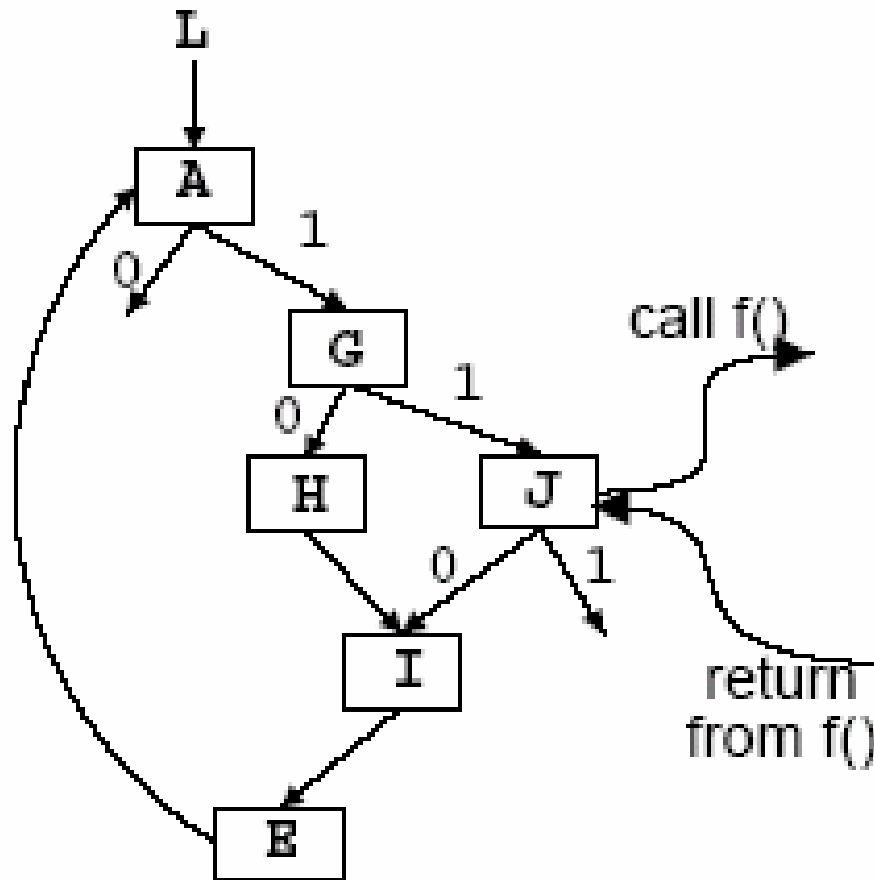


*Extracting one  
hot path*



*Extracting two  
hottest paths*

# Example Interprocedural Path



Path 11000

A → G → J(until call) → F1 → F2 → F3 → F5 → F8 → (return)J → I → E

# Ongoing and Future Work

## Microarchitecture designs that exploit VISC

String microarchitectures: cooperative partitioning of dataflow

## Explicitly parallel V-ISA:

Abstracting control parallelism: SMT, CMP

Abstracting data parallelism: vectors and streams

## Implementing language-level virtual machines

Shared mechanisms for optimization, GC, RTTI, exceptions, ...

## OS Implications

Exploring benefits of security, hardware/software flexibility

# LLVM Status

*Public releases in Oct 03, Dec 03, Mar 04, Aug 04.*

*1200+ downloads, many active users*

## Release includes:

- **Front ends:** C, C++
- **Back ends:** Sparc V9, x86, PPC (all offline or online), and C
- **JIT System:** Sparc, x86
- **Link-time interprocedural optimizer + many analyses, opts**

## Under development:

- **Trace-driven runtime optimizer**
- **Front-ends:** JVM, MSIL, OCAML, Scheme
- **Full Linux port to LLVA**