Portability and Performance of Applications in the Manycore Era

Erven Rohou, ALF research group

Technion, חיפה

March 2011

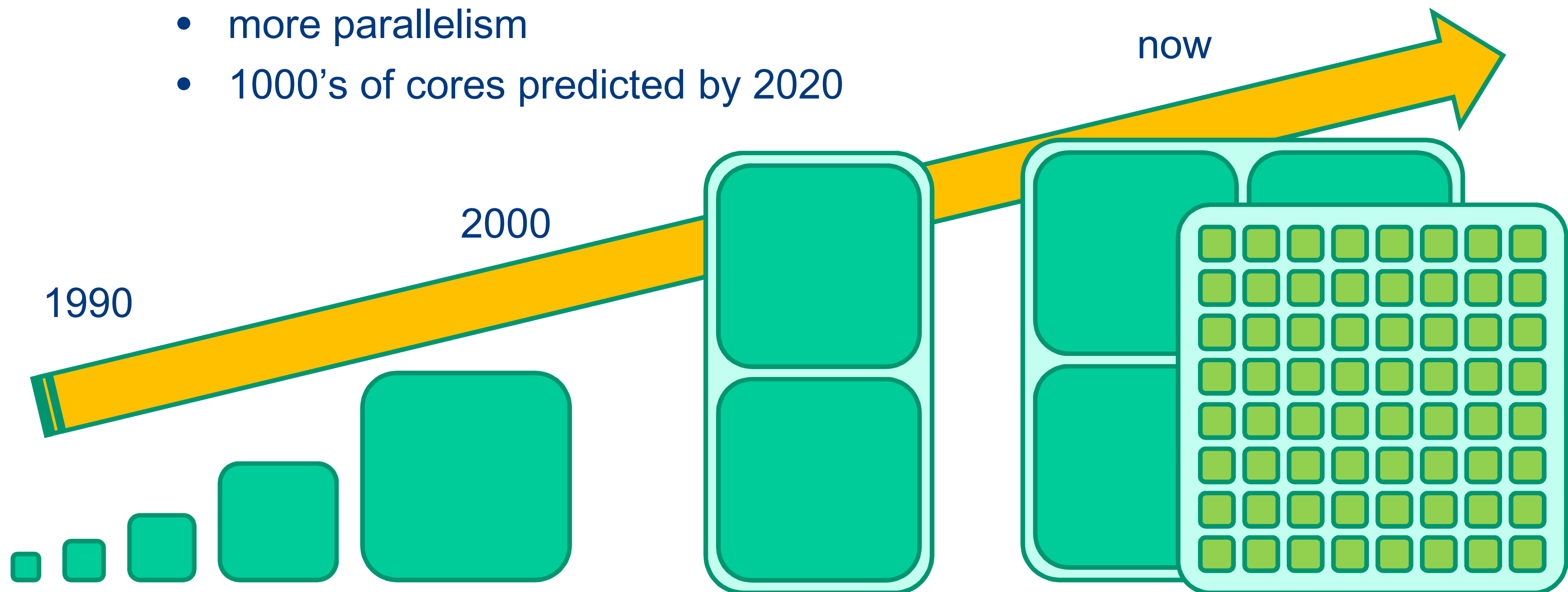
Evolution of Processor Architectures

More transistors used to mean

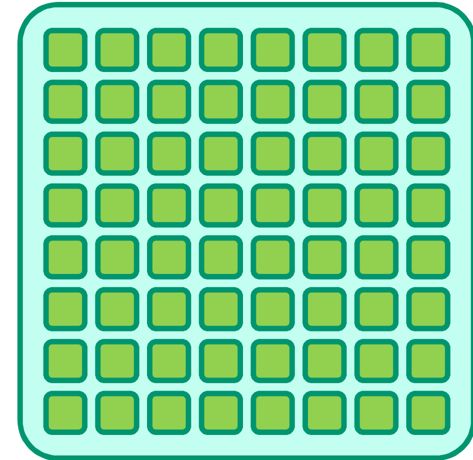
- improved μ-architecture
- faster clock

Now

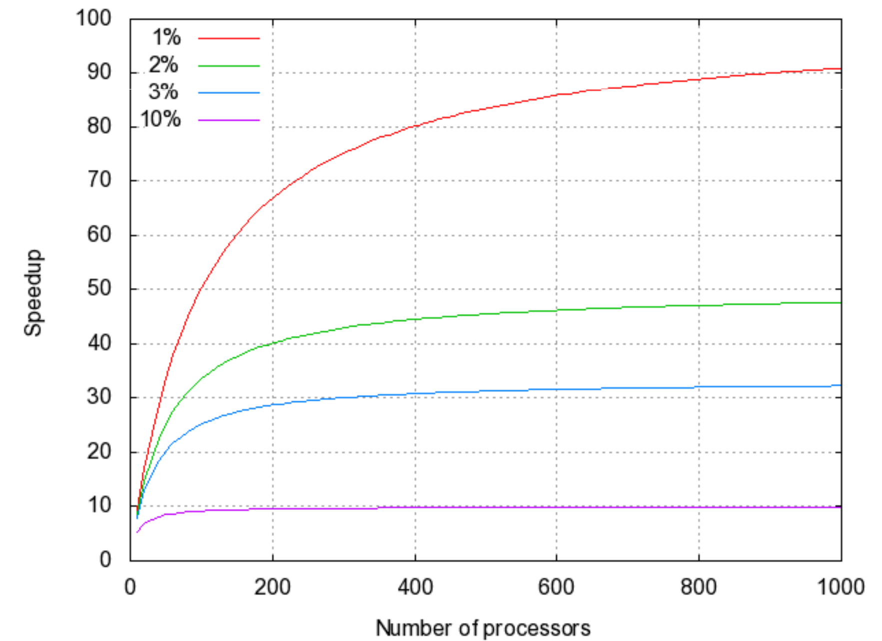
- more parallelism
- 1000's of cores predicted by 2020



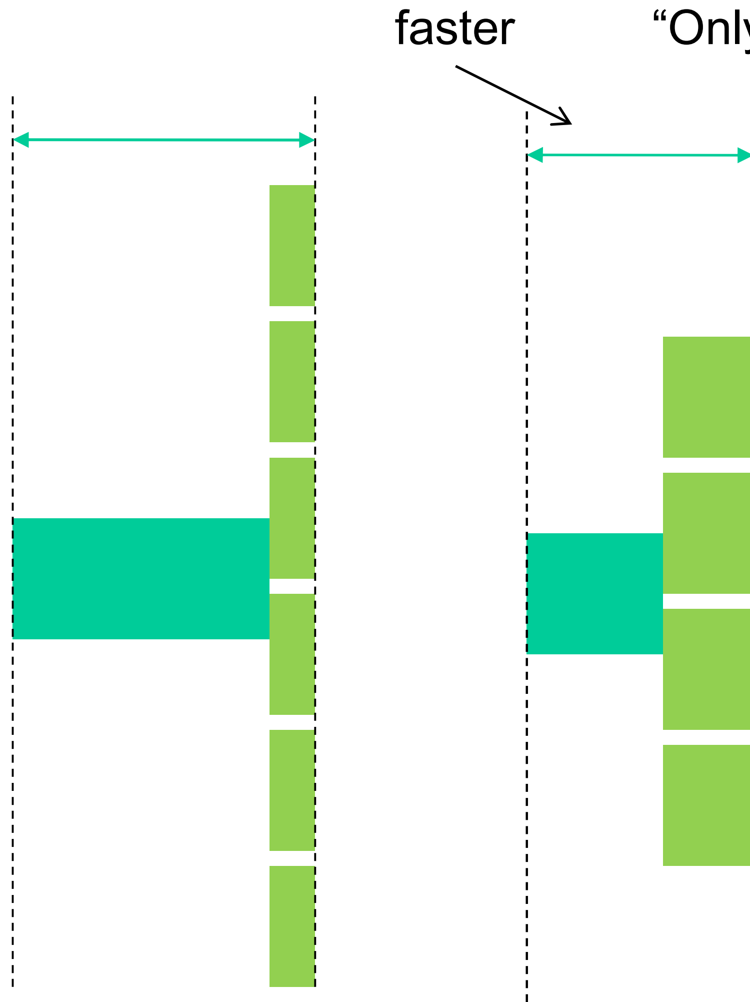
Amdahl's Law

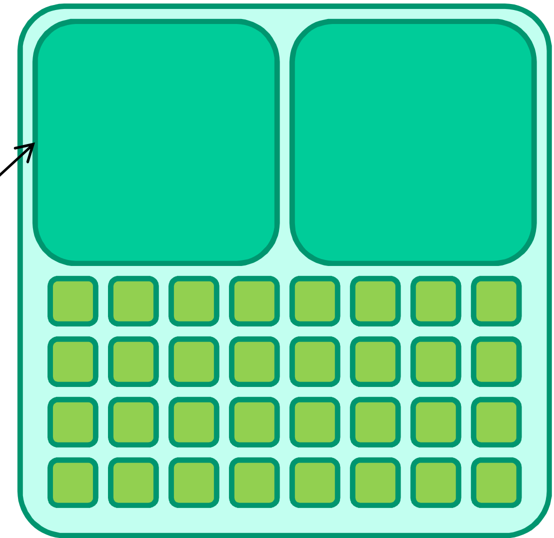


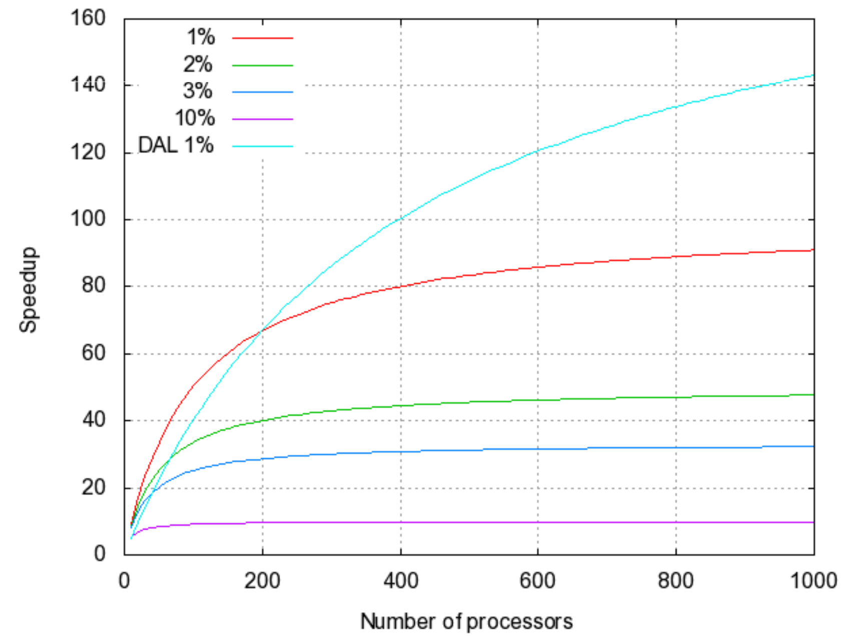
"Cannot run faster that sequential part"



Possible Future Design





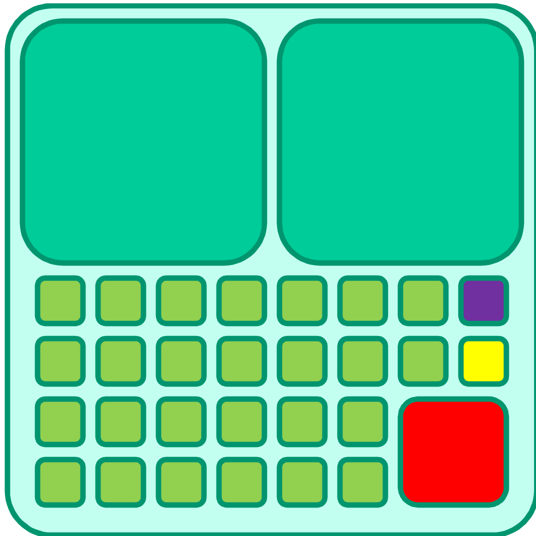


Heterogeneity by Design

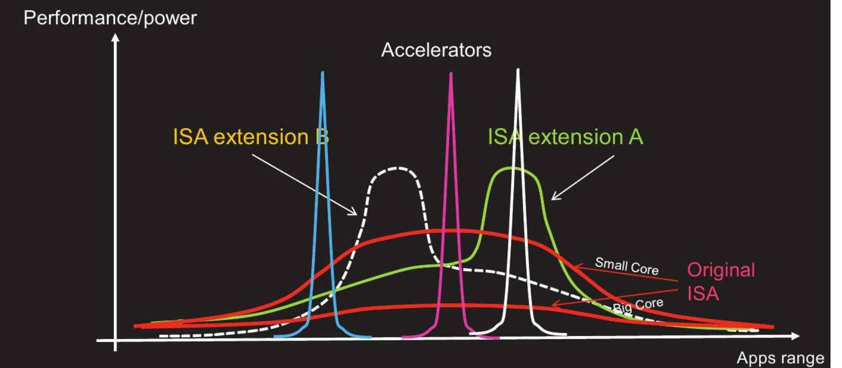
"Free" silicon

Prepare for various application domains

- GPU
- DSP
- …



[Prof. Uri Weiser's talk at INRIA]



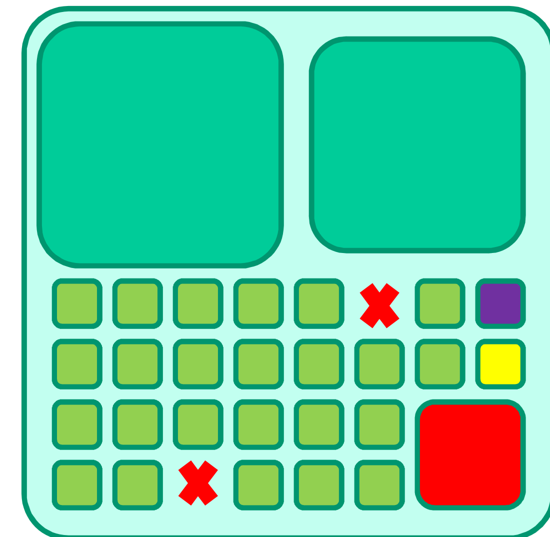
Heterogeneity by Necessity

Shrinking designs and increasing process variability

Larger chips

Business requires binning to maintain acceptable yield

- frequency ratings
- Cell's SPUs
- more diversity expected in future



Lifetime of Application ≫ Lifetime of Hardware

Consequences for programmers

- actual target unknown
- portability needed: bytecode format
- kind of parallelism unknown
  - data-level, thread-level, pipeline, ILP…
- amount of parallelism unknown

Portability and performance at risk

- most applications are still sequential
  - or, at least, have a residual sequential part
- many parallel applications have specific models of parallelism built-in

Portability and Performance Potpourri

Address future architectures

- heterogeneous
- manycore

Parallelism only makes sense for performance

Investigate several facets of performance

- parallel sections
- sequential sections
- trade-offs

This Talk

Recent work

Ongoing work

Share some thoughts for future work

Instruction Set Virtualization

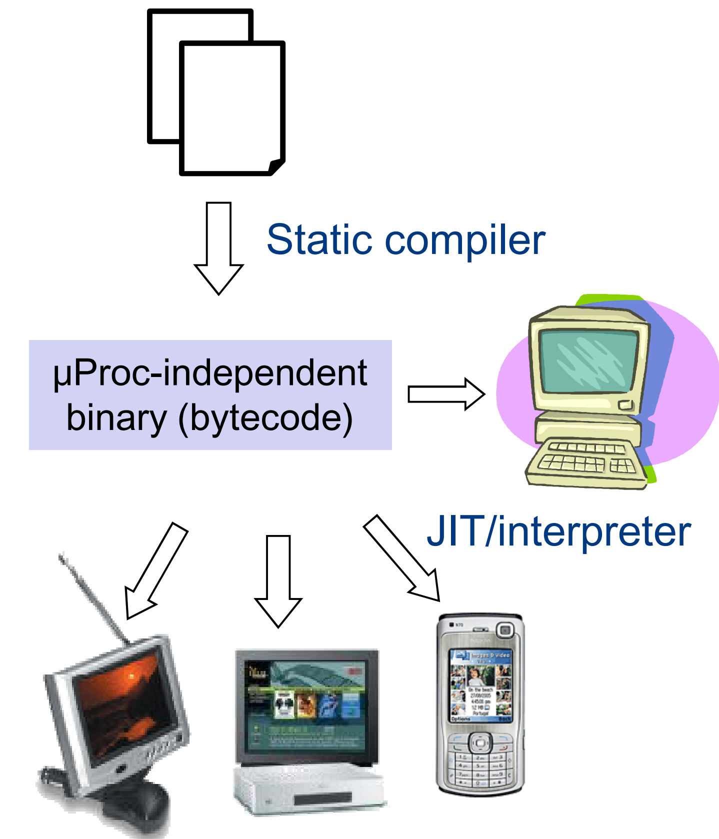
Abstract (virtualize) instruction set

Advantages of bytecode distribution

- simplifies development and deployment
- postpones code generation
  - better exploitation of hardware
    - even new features (e.g. FPU)
  - better mapping of parallelism

Execution

- interpreter
- JIT compiler for performance



Split-Compilation

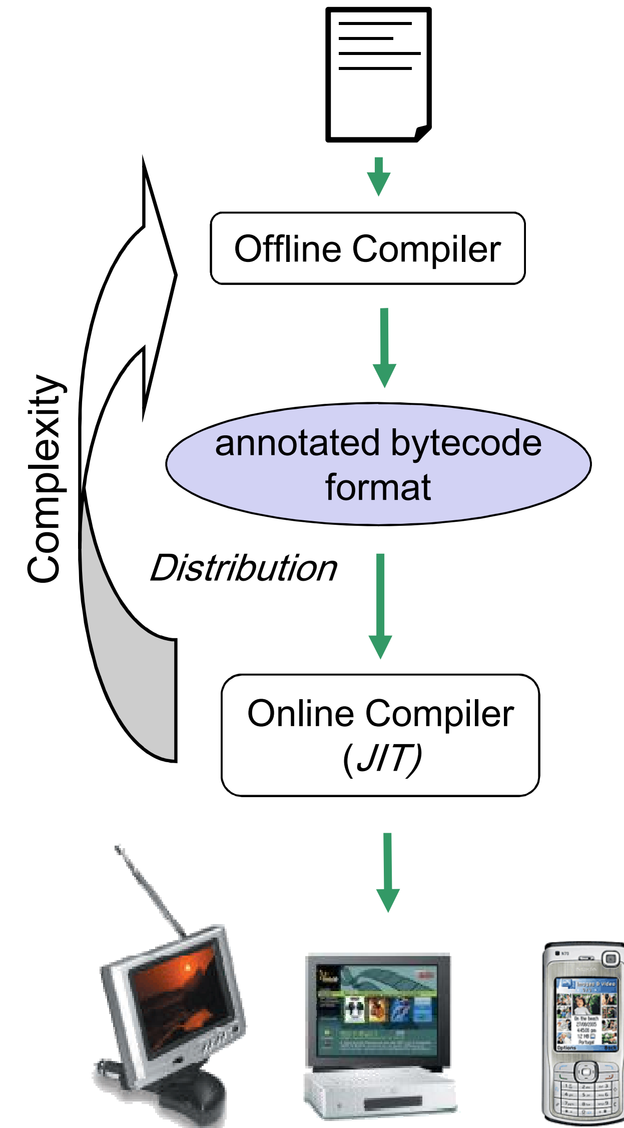
JIT compiler runs under severe constraints

- compile time is part of run time
- resources shared with application
- typically embedded device

Offload complex compiler analyses and transformations

Convey relevant information from offline to online stage

Make best use of existing technology



[Joint work with IBM: HiPEAC11, CGO11]

Split-Compilation applied to Vectorization

SIMD is key to performance

- but vectorization needs target-dependent information

Vectorization too complex for JIT compilers

➔ Apply split-compilation to reconcile contradictory trends

Vectorization at a Glance

Compiler optimization

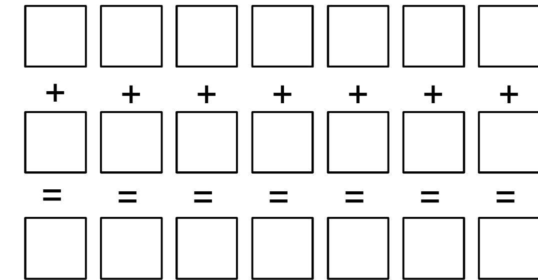
Pack adjacent values into larger containers

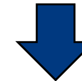
- SIMD: Single Instruction Multiple Data

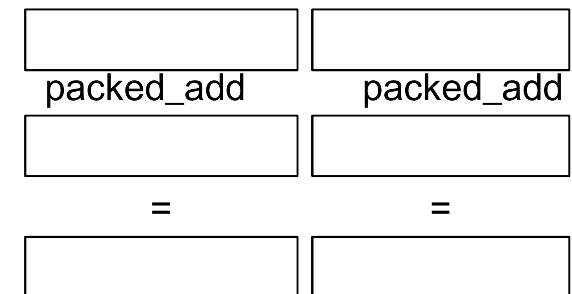
Exploit SIMD instruction set extensions

Benefits

- fewer loop iterations
- fewer instructions
- fewer memory accesses








```
char a[N], b[N], c[N];
for(i=0; i<1024; i++) {
  a[i] = b[i] + c[i]
}
```

```
char a[N], b[N], c[N];
for(i=0; i<1024; i+=4) {
  a[i..i+3] = b[i..i+3] + c[i..i+3]
}
/* handle remaining iterations*/
```

Diversity of SIMD ISA

Vector lengths

- 64 bits: MMX, ARM NEON
- 128 bits: Intel SSE, PowerPC AltiVec, ARM NEON
- 256 bits: upcoming Intel AVX
- 512 bits: Larrabbee (?)

Data types

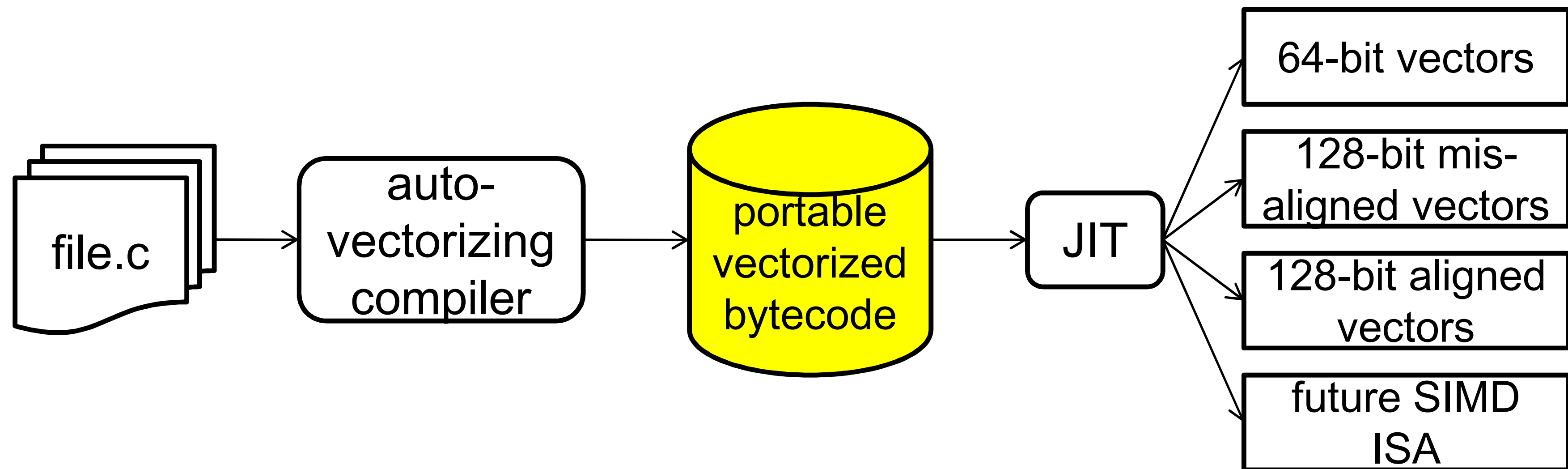
- AltiVec has no 64-bit operations
- MMX has only integer operations

Alignment constraints

- NEON and SSE support misaligned accesses
- AltiVec does not
- AVX instructions have different requirements

Supported idioms

Split-Vectorization Illustrated



Split-Vectorization

Compute vectorization opportunities offline

Let the JIT materialize the actual hardware characteristics

Objectives for JIT vectorizer

1. Robust

   Code must execute, both with and without SIMD support,

   or when using an unmodified, non-vectorizing JIT compiler

2. Risk-free

   Minimum penalty when running vectorized bytecode without SIMD support

3. Efficient

   Maximum speed-up when running vectorized bytecode with SIMD support

Definitions

Conscious JIT

- Understands semantics of annotations
- Generates SIMD code when possible
- Falls back to scalar if not

Agnostic JIT

- Falls back to scalar code
- Limited penalty, thanks to standard compilation techniques
  - inlining
  - copy propagation
  - dead code elimination

Split-Vectorization Details

Vectorize for “large” vectors

- GCC requires vector-size to be known

Only vectorize in the absence of loop-carried dependences

- good enough in practice
- could easily add dependence hints

Generate re-alignment scheme, to cope with

- misaligned accesses
- aligned accesses (when known)
- re-aligned accesses

Encode information as builtins

Provide support for agnostic JIT

Vector Length Builtins

int get_VF(type)

- return the number of elements that fit in vector register

vector init_uniform(val)

- return a vector initialized to `m` copies of val

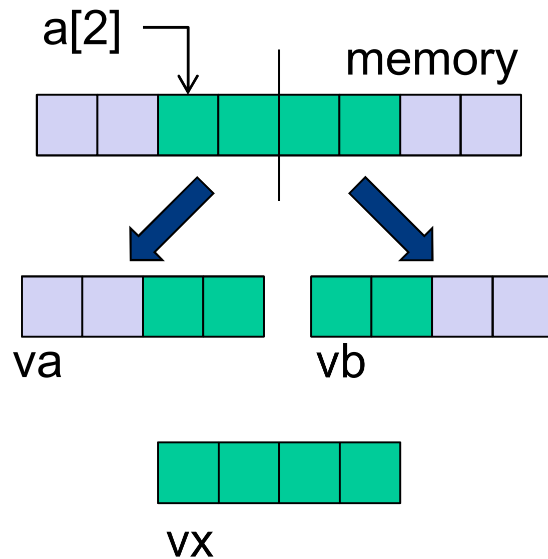
vector init_affine(val, inc)

- return a vector initialized to (`val,val+inc,val+2inc,...,val+(m-1)inc`)

vector init_reduc(val, defaults)

- return a vector initialized to (`val`, `default`, `default`,...,`default`)

Alignment Builtins



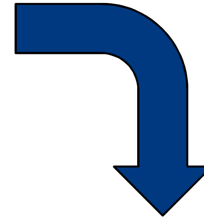

```
float sum=0;
for (i=0; i<n; i++) {
  sum += a[i+2];
}
```

bytecode

```
int vf = get_VF(fp);
float sum;
vfloat vsum = init_uniform(fp, 0);
rt = get_rt(&a[2],8,32);
vfloat va = align_load(&a[0]);
for (i=0; i<n; i+=vf) {
  vb = align_load(&a[i+4]);
  vx = realign_load(va, vb, rt, &a[i+2], 8, 32);
  vsum = vadd(vx, vsum);
  va = vb;
}
sum = reduc_plus(vsum);
```

Builtins Materialization: SSE

```
int vf = get_VF(fp);
float sum;
vfloat vsum = init_uniform(fp, 0);
rt = get_rt(&a[2],8,32);
vfloat va = align_load(&a[0]);
for (i=0; i<n; i+=vf) {
  vb = align_load(&a[i+4]);
  vx = realign_load(va, vb, rt, &a[i+2], 8, 32);
  vsum = vadd(vx, vsum);
  va = vb;
}
sum = reduc_plus(vsum);
```



```
int vf = 4;
float sum;
vfloat vsum = {0, 0, 0, 0};
rt = get_rt(&a[2],8,32);
vfloat va = align_load(&a[0]);
for (i=0; i<n; i+=vf) {
  vb = align_load(&a[i+4]);
  vx = movdqu(va, vb, rt, &a[i+2], 8, 32);
  vsum = vadd(vx, vsum);
  va = vb;
}
sum = reduc_plus(vsum);
```

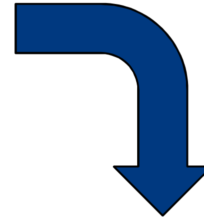
VF=4

vector size = 16

misaligned accesses

Builtins Materialization: NEON

```
int vf = get_VF(fp);
float sum;
vfloat vsum = init_uniform(fp, 0);
rt = get_rt(&a[2],8,32);
vfloat va = align_load(&a[0]);
for (i=0; i<n; i+=vf) {
  vb = align_load(&a[i+4]);
  vx = realign_load(va, vb, rt, &a[i+2], 8, 32);
  vsum = vadd(vx, vsum);
  va = vb;
}
sum = reduc_plus(vsum);
```



```
int vf = 2;
float sum;
vfloat vsum = {0, 0};
rt = get_rt(&a[2],8,32);
vfloat va = align_load(&a[0]);
for (i=0; i<n; i+=vf) {
  vb = align_load(&a[i+4]);
  vx = vld1_f32(va, vb, rt, &a[i+2], 8, 32);
  vsum = vadd(vx, vsum);
  va = vb;
}
sum = reduc_plus(vsum);
```

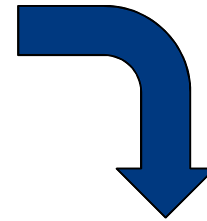
VF=2

vector size = 8

aligned accesses guaranteed

Builtins Materialization: AltiVec

```
int vf = get_VF(fp);
float sum;
vfloat vsum = init_uniform(fp, 0);
rt = get_rt(&a[2],8,32);
vfloat va = align_load(&a[0]);
for (i=0; i<n; i+=vf) {
  vb = align_load(&a[i+4]);
  vx = realign_load(va, vb, rt, &a[i+2], 8, 32);
  vsum = vadd(vx, vsum);
  va = vb;
}
sum = reduc_plus(vsum);
```



```
int vf = 4;
float sum;
vfloat vsum = {0, 0, 0, 0};
rt = get_permute_vec(&a[2],8,32);
vfloat va = lvx(&a[0]);
for (i=0; i<n; i+=vf) {
  vb = lvx(&a[i+4]);
  vx = vperm(va, vb, rt, &a[i+2], 8, 32);
  vsum = vadd(vx, vsum);
  va = vb;
}
sum = reduc_plus(vsum);
```

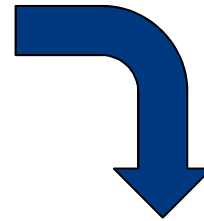
VF=4

vector size = 8

realignement

Builtins Materialization: Scalarization

```
int vf = get_VF(fp);
float sum;
vfloat vsum = init_uniform(fp, 0);
rt = get_rt(&a[2],8,32);
vfloat va = align_load(&a[0]);
for (i=0; i<n; i+=vf) {
  vb = align_load(&a[i+4]);
  vx = realign_load(va, vb, rt, &a[i+2], 8, 32);
  vsum = vadd(vx, vsum);
  va = vb;
}
sum = reduc_plus(vsum);
```



```
int vf = 1;
float sum;
float vsum = 0;
rt = get_permute_vec(&a[2],8,32);
vfloat va = align_load(&a[0]);
for (i=0; i<n; i+=vf) {
  vb = lxv(&a[i+4]);
  vx = load(va, vb, rt, &a[i+2], 8, 32);
  vsum = vsum + vx;
  va = vb;
}
sum = reduc_plus(vsum);
```

VF=1

vector size 4

misaligned accesses

x86 Code Generation Example

```
float a[N];
float b[N];
float c[N];
for (i =0; i<n; ++i)
{
  a[i]=b[i]+c[i];
}
```

offline vectorization

```
.locals (VectorSF* 'a')
ldloc 'a'
ldloc 'b'
ldloc 'c'
call VectorSF::Add
...
ldloc 'a'
ldc.i4 'vf'
stloc 'a'
...
```

```
.method VectorSF::Add
…
```

online code generation

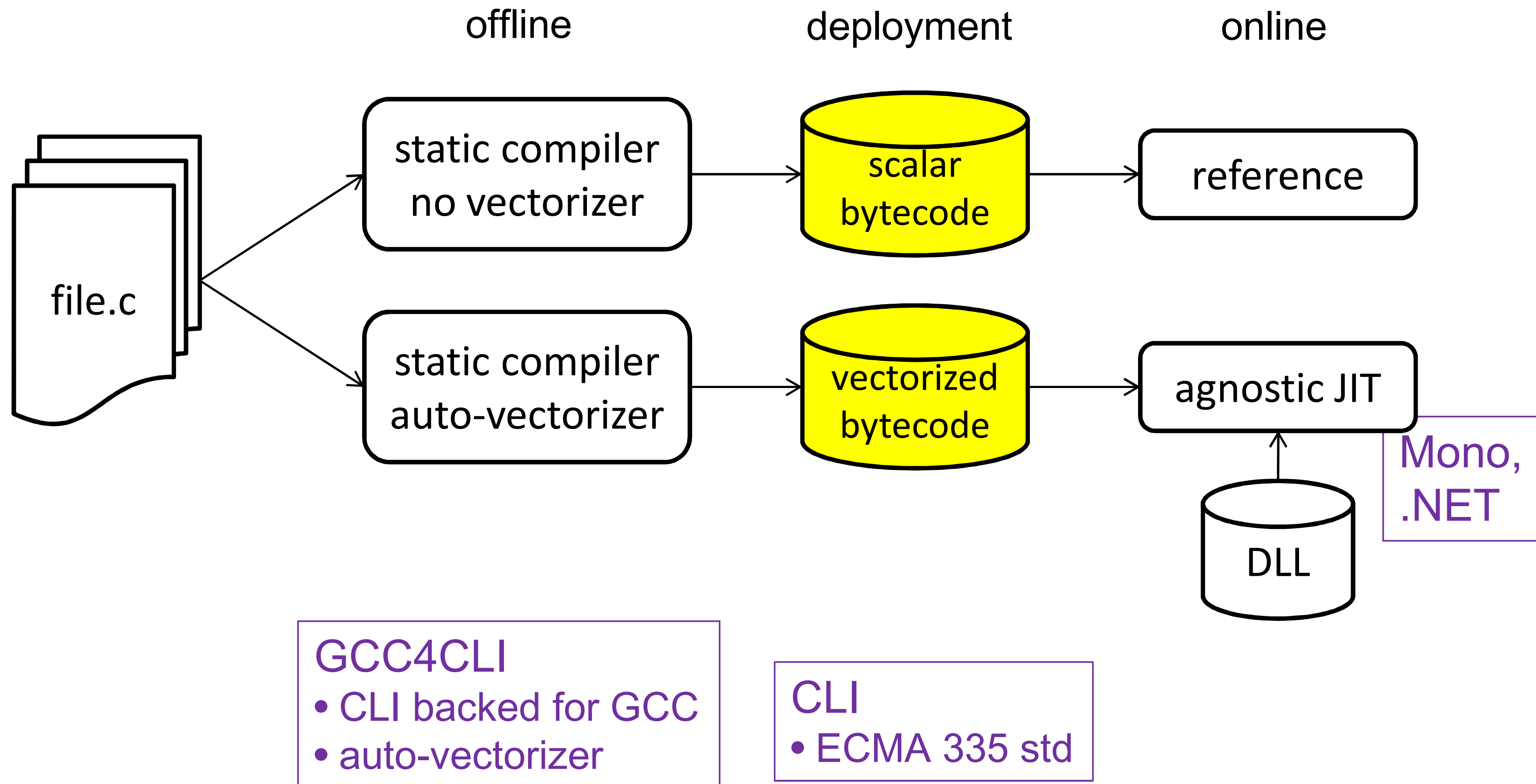
conscious JIT

```
movups (%esi),%xmm0
movups (%ecx),%xmm1
addps %xmm1,% xmm0
movups %xmm0 ,(%eax)
add $16, %ecx
add $16, %esi
...
```

agnostic JIT

```
flds (%ebx)
flds (%ecx)
faddp %st ,%st(1)
flds 0x4 (%ebx)
flds 0x4 (%ecx)
faddp %st ,%st(1)
flds 0x8 (%ebx)
...
fstps 0x8 (%eax)
fstps 0x4 (%eax)
fstps (%eax)
```

Agnostic Vectorization Flows



Risk-free: Agnostic JIT

(Unexpected) speedups

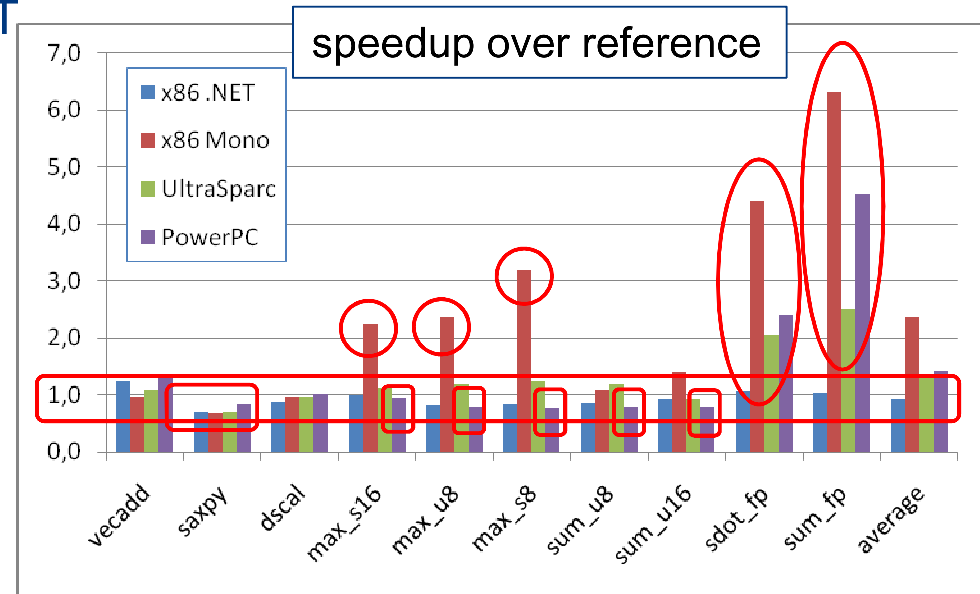
- especially on reductions

Contained slowdowns

- Average: -7%,+138%,+30%,+41%
- Worst case -30% for `saxpy`.NET
- `saxpy` in general, and integer reductions on PowerPC
  - lack of register promotion
  - additional spilling on PowerPC due to unrolling

Agnostic targets

- Mono on Core2 Duo, UltraSparc, PowerMac,
- .NET on Core2 Duo

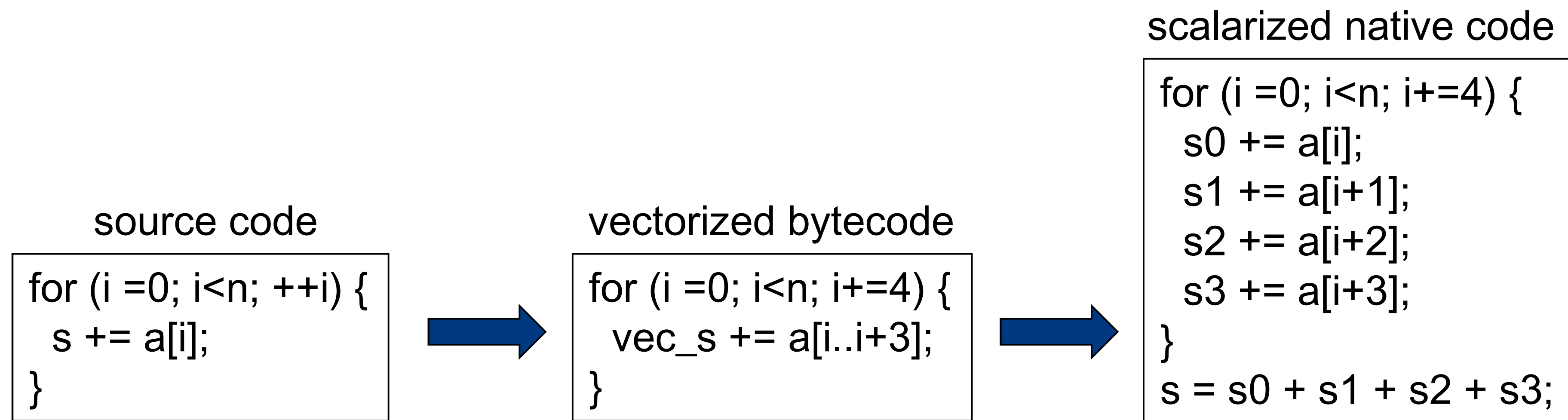


Scalarization Side-Effect

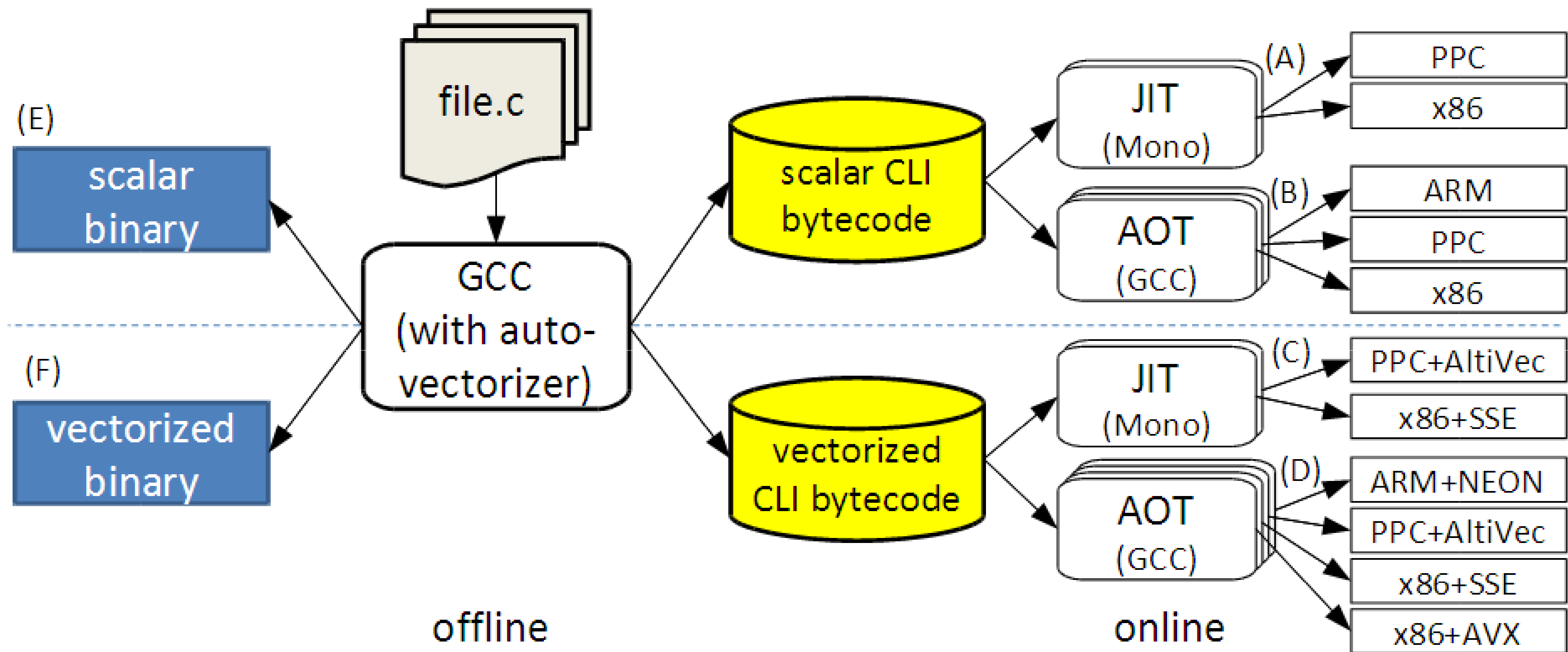
Agnostic JIT inlines the builtins

Practical effect: unrolling + modulo variable expansion (MVE)

- reduces cost of loop control
- removes circular data dependences in case of reductions
- possible spilling



Conscious Vectorization Flows



Vectorization Impact

Run with Mono 2.7-dev

Impact = (A/C) / (E/F)

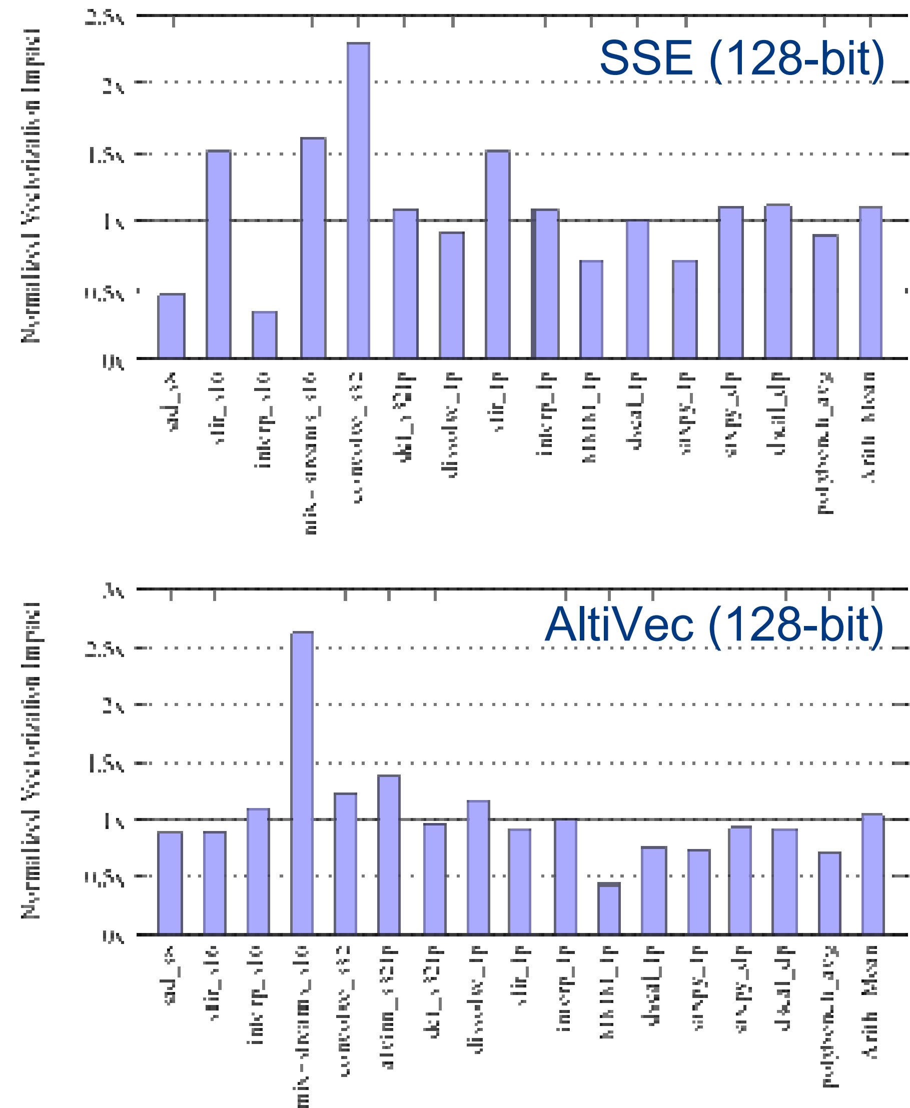
- evaluate split approach
- not vectorization itself

Factors impacting performance

- lack of global register allocation
- unoptimized x87 code

Exceptions

- *MMM_fp*: constant not folded across nested loop
- *mix-streams*: versioning for alignment



INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE
INRIA
centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Split- vs. natively vectorized

Run with GCC CLI front-end

- more mature compilation technology

Normalized execution time close to 1x

- no inherent slowdown due to split-vectorization

Scalarized double kernels on AltiVec

- performance ok

Most differences come from small code generation details

AVX similar

This Talk

Recent Work

Ongoing work

- performance of interpreters
- performance guarantees

Thoughts for future work

Performance of Interpreters

Interpreters used to be the way to execute bytecode

Declined as JIT matured

Renewed interest

- portability
- profiling capabilities while hotspots are detected
- good for code that executes few times
- some vendor’s policies

Improving Interpreters

Bottleneck is dispatch

For each bytecode, the interpreter must

- read bytecode
- dispatch
- access arguments
- perform the function
- store result

About 20 instructions per bytecode

```
while (bytecode = read()) {
  switch(OPCODE(bytecode)) {
    case SUB:
        x = pop();
        y = pop();
        push(x-y);
      break;
```

```
.L10:
        movzwl  (%esi), %edx
        movl    %esi, %eax
        subl    -536(%ebp), %eax
        movzwl  %dx, %edi
        sarl    %eax
        movl    %edi, %ecx
        movl    %eax, -532(%ebp)
.L841:
        cmpw    $665, %dx
        ja      .L841
        jmp     *.L512(,%ecx,4)
    …
.L124:  /* case SUB: */
        movl    %eax, -428(%ebp)
        leal    -16(%ebx), %eax
        movl    %eax, -532(%ebp)
        movl    -16(%ebx), %eax
        subl    %eax, -32(%ebx)
        leal    2(%esi), %eax
        jmp     .L513
    …
.L513:
        movl    %eax, %esi
        movl    -532(%ebp), %ebx
        jmp     .L10
```

Superinstructions

Reduce the number of dispatches

Group instructions before execution

- for example, `ldc.i4` is often followed by an `add`
- create specialized new bytecode
- “pay” single dispatch loop for 2 bytecodes

Static or dynamic

Note: increases number of indirect jump targets

```
.L10:
        movzwl  (%esi), %edx
        movl    %esi, %eax
        subl    -536(%ebp), %eax
        movzwl  %dx, %edi
        sarl    %eax
        movl    %edi, %ecx
        movl    %eax, -532(%ebp)
.L841:
        cmpw    $665, %dx
        ja      .L841
        jmp     *.L512(,%ecx,4)
   ...
.L124:  /* case SUB: */
        movl    %eax, -428(%ebp)
        leal    -16(%ebx), %eax
        movl    %eax, -532(%ebp)
        movl    -16(%ebx), %eax
        subl    %eax, -32(%ebx)
        leal    2(%esi), %eax
        jmp     .L513
   ...
.L513:
        movl    %eax, %esi
        movl    -532(%ebp), %ebx
        jmp     .L10
```

Vector Superinstructions

Make vector builtins superinstructions

- “conscious” interpreter
- one superinstruction factorizes up to 100 instructions
- removes call, return, and parameter passing

```
switch(OPCODE(bytecode)) {
  case ADD_VEC:
      vx = pop(); vy = pop();
      for(i=0; i<16; i++)
        vz[i] = vx[i] + vy[i];
      push(vz);
    break;
}
```

Vector Superinstructions Results

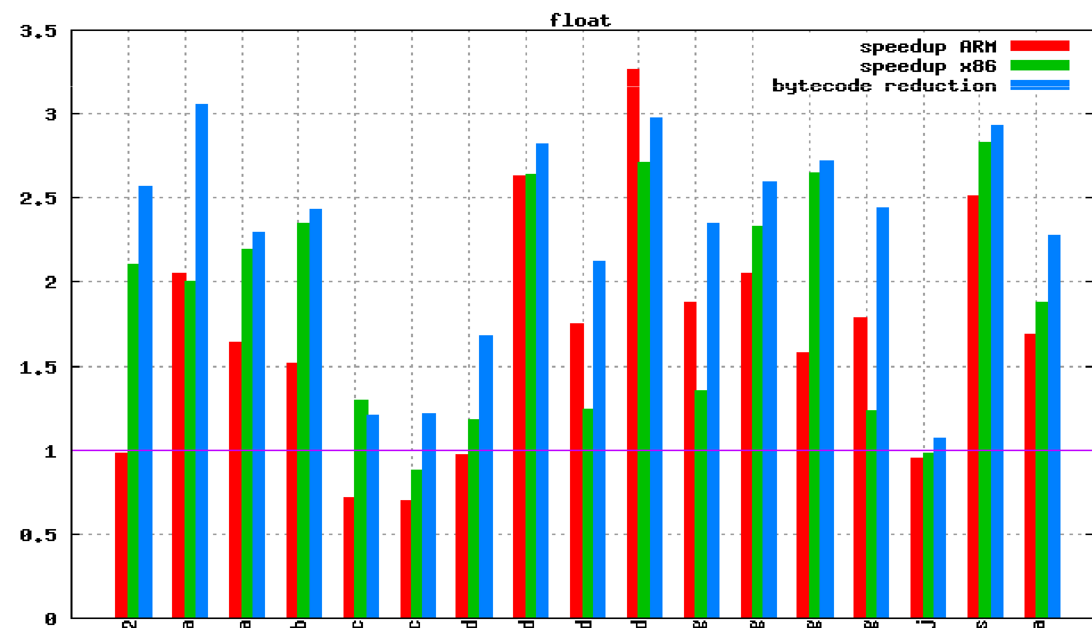
Experiment with ARM and x86

Good speedups

Depend on type of data manipulated

- speedup ~VF/2

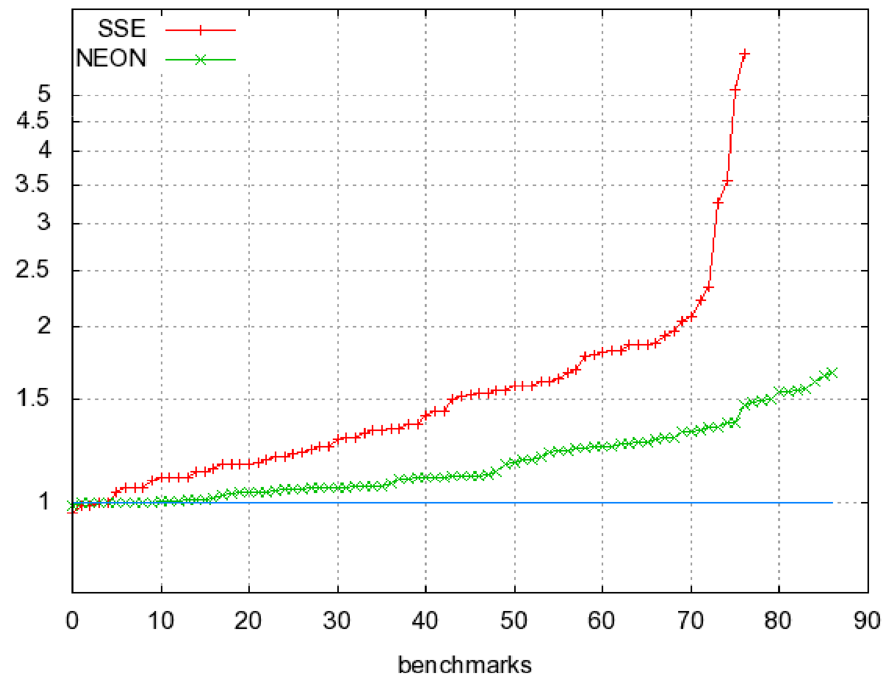
Mostly bytecode reduction



Vector Superinstructions Bonus

Can map vector superinstructions to actual SIMD instructions

- Intel SSE
- ARM NEON




```
switch(OPCODE(bytecode)) {
  case ADD_VEC:
      vx = pop(); vy = pop();
      vz = __builtin_ia32_paddb128(vx,vy)
      push(vz);
    break;
}
```

This Talk

Recent Work

Ongoing work

- performance of interpreters
- performance guarantees

Thoughts for future work

Performance Guarantees

Many computing systems have real-time constraints

- result must be provided in given amount of time
- worst-case execution time (WCET) must be estimated

JIT compilers add new layer of indeterminism

- perceived as inappropriate

Is it hopeless?

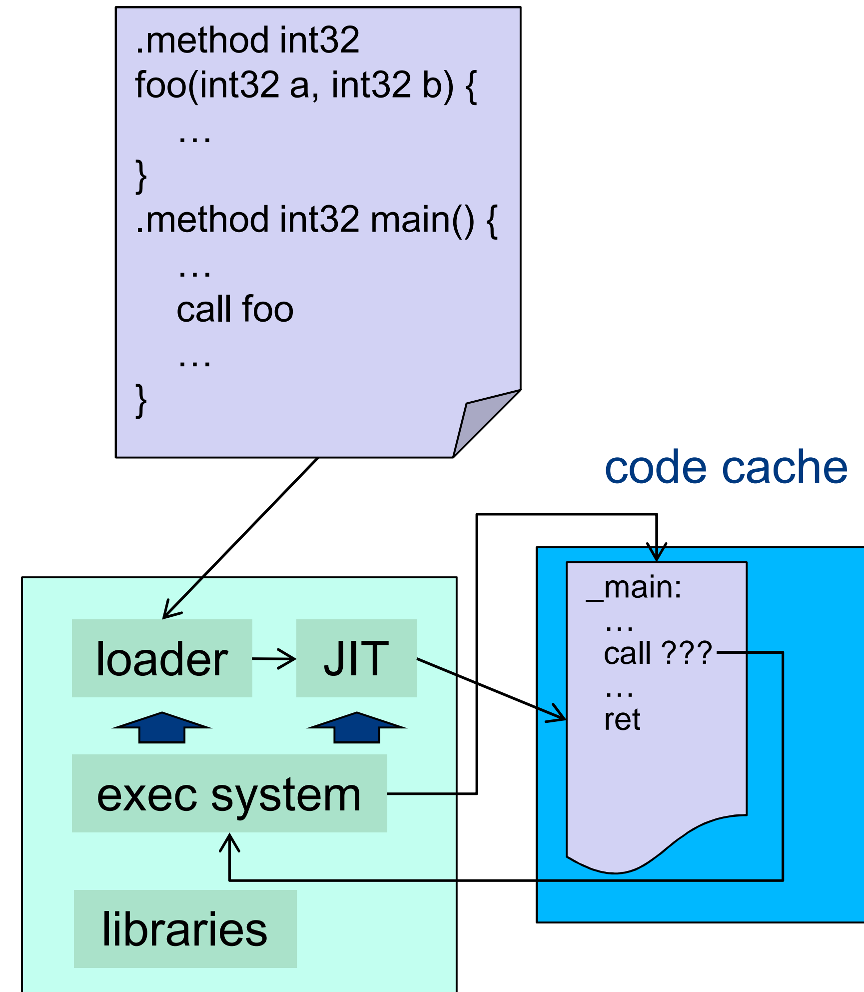
Very First Step: Behavior of Code Cache

Focus on code cache

- software managed cache
- number of (re-)compilations

Simplifying Assumptions

- all code is compiled, no interpreter
- functions compiled at *call* or *return* only, when absent from code cache
- only one level of optimization
- …



WCET and Abstract Interpretation Basics

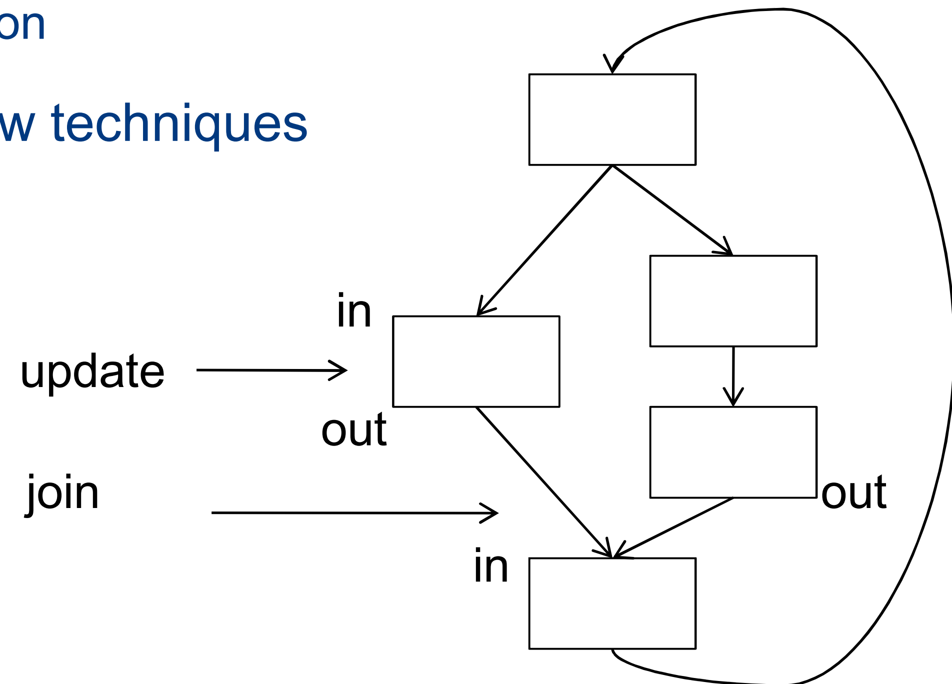
Goal: determine program properties statically

Work in an abstract domain

- define partial order, lattice
- define *join* and *update* function

Compute invariant using data flow techniques

$$\alpha : \wp(D) \longrightarrow \hat{D}$$



Abstract Interpretation Applied to Code Cache

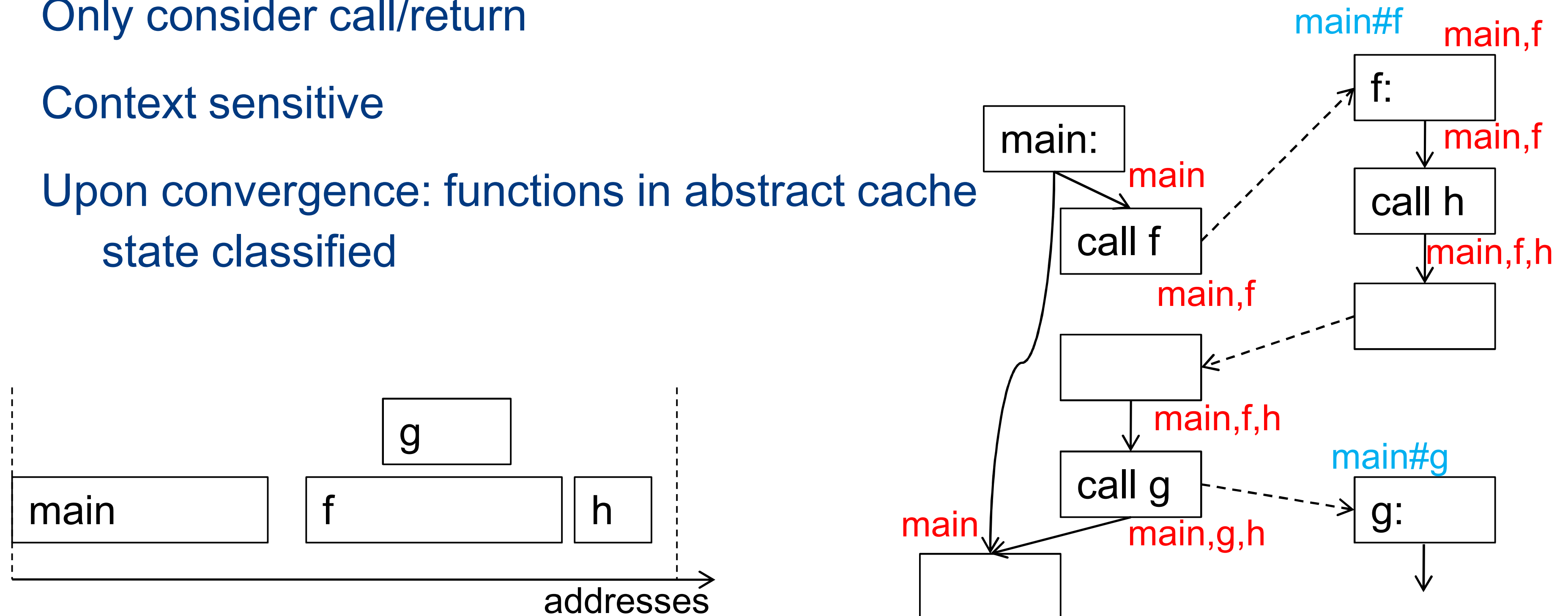
Concrete domain D: actual content of code cache

Abstract domain $\hat{D}$: domain of function identifiers

Only consider call/return

Context sensitive

Upon convergence: functions in abstract cache state classified



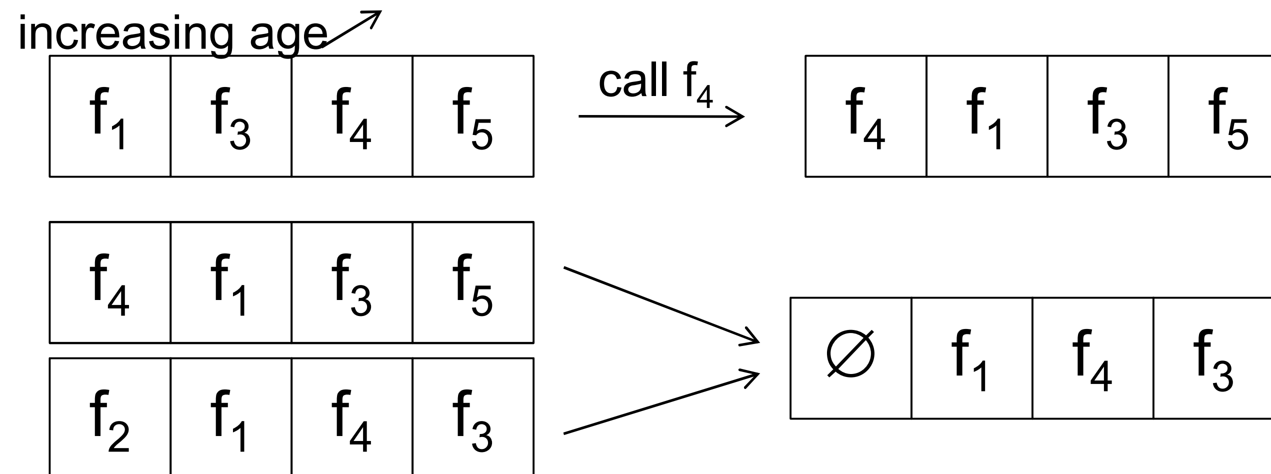
Cache Structure 1: Fixed size blocks, LRU

Cache decomposed in fixed-size blocks

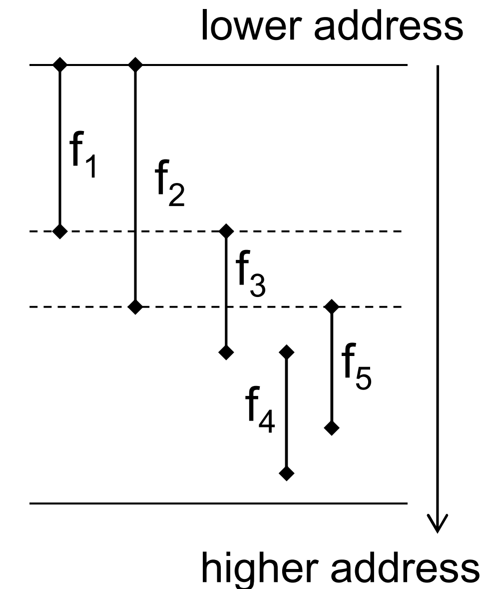
- size of largest function

No external fragmentation

LRU known to be amenable to accurate analysis



Cache Structure 2: Fixed Layout

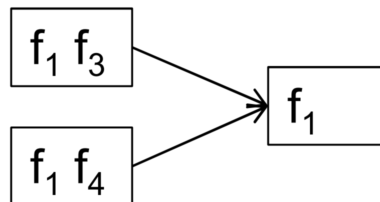


Pre-compute layout

Try to minimize caller-callee interactions

Layout Computation

- sequential (reference)
- WCET-directed



```
f1() {
  f2();
  for (i=0;i<10;i++) {
    if (cond) f3();
    else f4();
  }
  f5();
}
```

Experiments

Typical WCET benchmarks

State-of-the-art IPET (Implicit Path Enumeration Technique)

- build task control flow graph
- upper bound of execution time of basic block t_i
- count variable f_i
- maximize $\sum_i f_i \times t_i$
- ILP solver

Results

WCET decreases as cache size increases

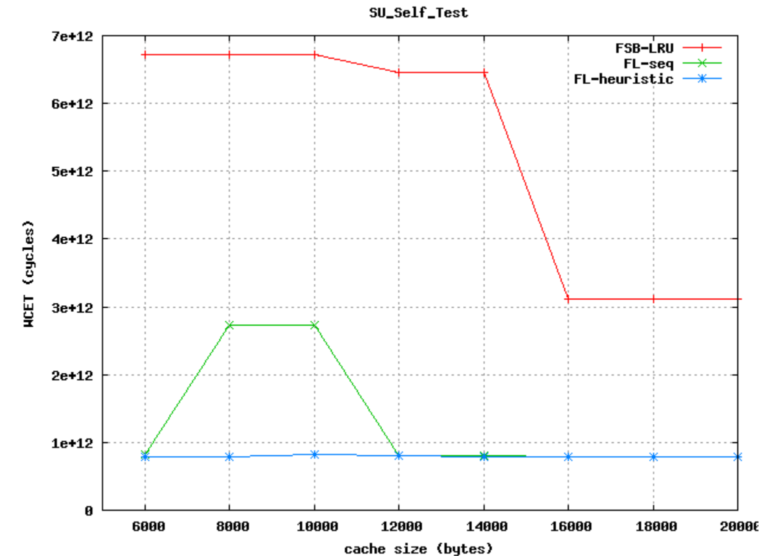
- but irregularities

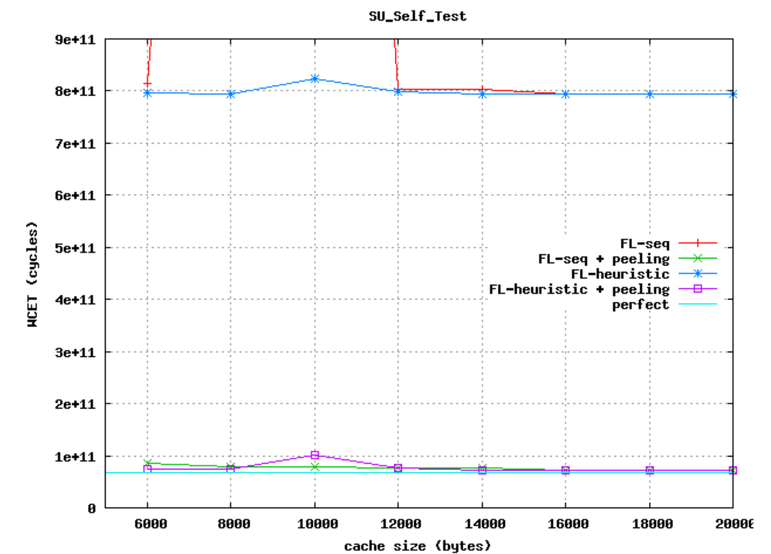
Fixed size blocks + LRU worse than fixed layout

- fragmentation
- replacement strategy

Especially bad at small cache sizes

- diversity of functions sizes





Peeling: Computation Time vs. Tightness

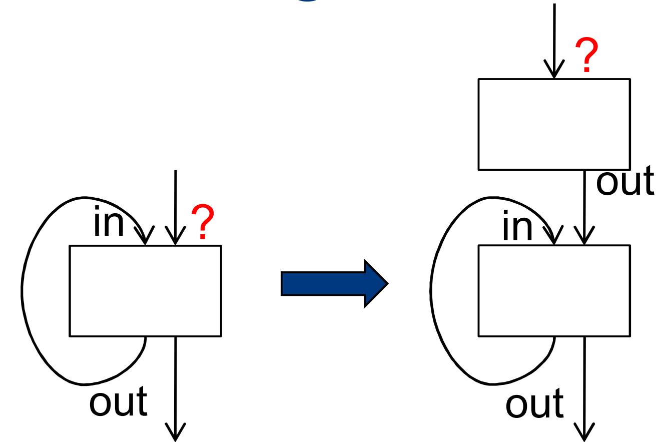
Virtual loop peeling

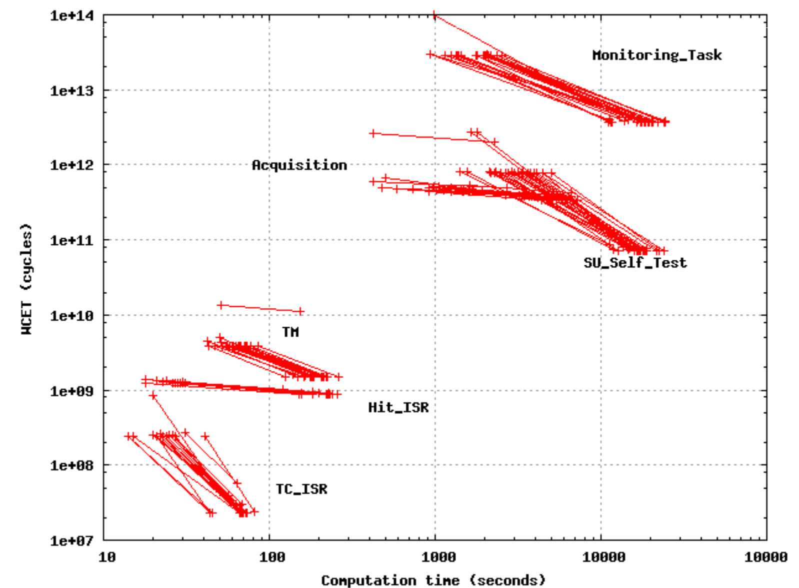
- only for analysis purposes

Improves analysis accuracy in loop

- removes uncertainty

Increases computation time





Conclusion

Recent disruption in the way architectures evolve

- ubiquitous manycores in near future
- no more magic increase of performance
- heterogeneous

Applications are (partly) sequential

- Amdahl's law

Application's lifetime if longer that hardware's

Need new ways to address performance