Inserting Data Prefetches into Loops in Dynamically Translated Code in IA-32EL

Agenda

- IA-32EL **Brief** Overview
- Prefetching in Loops
- IA-32EL Prefetching in Loops Implementation
- Performance #'s
- Related and Future Work

What is IA-32 EL ?

- A dynamic binary translator
- Enables the execution of IA-32 applications on Intel® Itanium® Processor (IPF) platforms
  - Only user-level, 32-bit protected flat applications
- Runs above the IPF Operating System
  - Entirely in user mode
  - Invoked by the OS (entirely transparent – to user, OS must be 32bit aware)
  - 99% of the translator is OS-independent, with a small OS-specific glue-layer
- SW only solution
  - NO IA-32 instructions executed on dedicated IA-32 IPF hardware

What is IA-32 EL ? (2)

- Three phase translation – Interpreter, Cold, Hot
  - Interpretation phase
    - The first n times a basic-block is executed – it is executed by the interpreter
      - Saves cost of translation of blocks that are executed <= n times
  - Cold phase
    - minimal optimizations
    - pre-prepared per IA-32 instruction IPF machine code
    - used/taken count instrumentation
  - Hot phase
    - cold block whose used-count reaches a threshold
    - trace-hyperblock formation
      - sequence of cold blocks that flow into each other with high prob
        - May form Loop
    - per IA-32 instruction Intermdiate Language (IL)
    - optimizations on IL (**including prefetch**), scheduling IL
    - IPF machine code

Prefetching

- Early transfer of data from memory into the data cache – so at access time it is in the cache
  - => know far enough ahead the data access address.
    - Loops may provide opportunity

Software Prefetching in Loops

- issuing a prefetch instruction in iteration i, in order that data accessed in iteration i+k will be cache resident in iteration i+k
- ld8 [r35]

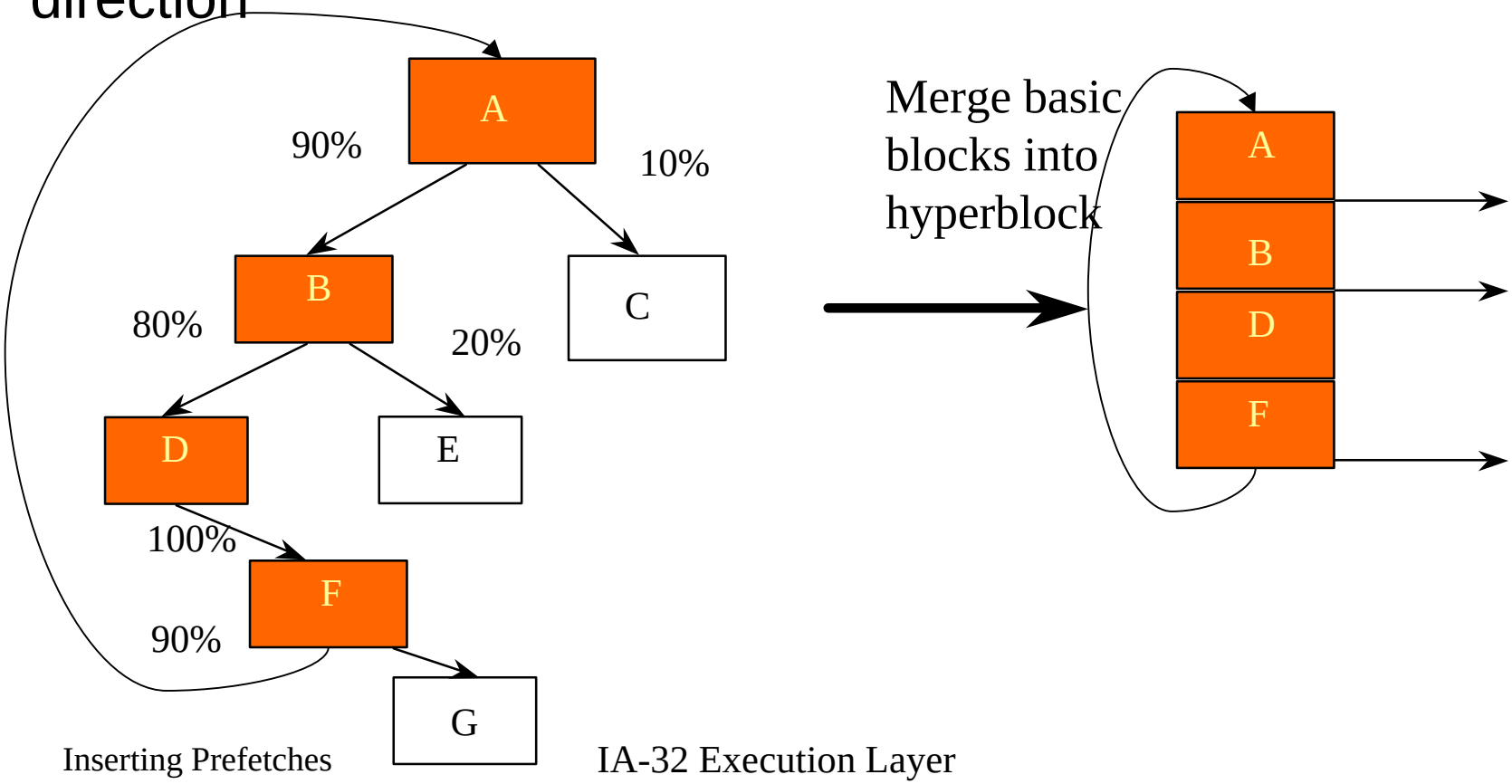
  lfetch [r35+distance(access[i],access[i+k])]
- Determine k
- Determine distance

Software Prefetching in Loops (2)

- k (#iterations ahead)
  - Is a function of the #cycles in an iteration of the loop, and the cache-miss penalty
    - cache_miss_penalty/iteration_cycles
      - Assume miss goes to memory
- distance (from current access to prefetch access)
  - Need to determine how the access changes over the iterations
    - When register(s) used in the address computation are changing by a constant in each iteration
      - Strided access => distance = k*stride

Loops in IA-32EL

- Hyperblock recognition phase
  - Sequence of cold (basic) blocks whose used/taken counters indicate a high probability of flow in one direction



Analysis for Inserting Prefetches

- Determine strides – use only info from analysis of IA-32 instructions in the loop
  - For IL generation, IA-32 instructions in the loop are decoded
  - Record information about all IA-32 registers used in address calculations, in order to determine how those registers change in the loop-hyper-block.
- At end of IL generation use the information to insert prefetch ILs
  - No IA-32 prefetch instructions from source binary are translated

Analysis for Inserting Prefetches (2)

- IA-32 integer registers are classified as one of the following for each data access:
  1) Invariant
  2) Changed by add/sub of explicit constant, invariant register, or invariant memory
  3) Loaded from apparent invariant memory – both location and content
  4) Changed in a manner that can not be determined

Analysis for Inserting Prefetches (3)

- IA-32 memory access of form

  [base_reg, index_reg*scale] offset

- Candidate for constant stride prefetching
  - base_reg is invariant, index_reg is changing by constant (explicit, register or memory) – or vice-versa

Prefetch Example(1)

- Address is being changed by explicit constant

```
fld qword ptr [eax] // eax is += 4

addp4 r35 = r.eax, r0 // zxt32
ldfd r.fp.stack0 = [r35]

add r32 = r35,(k*4)
// k is the computed #iterations
// ahead to prefetch for the fld
lfetch.nt1 [r32]
```

Prefetch Examples (2)

- Address is being changed by register

```
fld qword ptr [eax] // eax is += edx

addp4 r35 = r.eax, r0
ldfd r.fp.stack0 = [r35]

shladd r32 = r.edx,z,r35
// z is the log2 of the
// computed #iterations ahead to
// prefetch for the fld
lfetch.nt1 [r32]
```

Prefetch Examples (3)

- Address is being changed by invariant memory

```
fld qword ptr [eax]
:::::::
add eax, [esp+0xc] // [esp+0xc] is most likely
                   // loop invaraint

addp4 r35 = r.eax, r0
ldfd r.fp.stack0 = [r35]
:::
add r34 = r.esp + 0xc
ld4 r33 = [r34]
add r.eax = r.eax, r33

shladd r32 = r33,z,r35
// z is the log2 of the computed
// #iterations ahead to prefetch for the fld
lfetch.nt1 [r32]
```

Prefetch Examples (4)

- Unoptimized code

```
Mov   edx, DWORD PTR [ebp-32]
Add   edx, edi
Mov   eax, DWORD PTR [ebp-32]
Add   eax, DWORD PTR [ebp-96]
Fld   st(0)
Mov   esi,  DWORD PTR [ebp+020h]
Fmul QWORD PTR [esi+eax*8-8]
Mov   eax, DWORD PTR [ebp+034h]
Fadd QWORD PTR [eax+edx*8-8]
Fstp QWORD PTR [eax+edx*8-8]
Inc   DWORD PTR[ebp-32]
Dec   ecx
Jns   $-35
```

Analysis for Inserting Prefetches (4)

- IA-32 memory access of form

  [base_reg, index_reg*scale] offset

- Candidate for speculative stride prefetching
  - base_reg is loaded and that load is an explicit constant stride
    - e.g. array of pointers
      - Speculatively load pointer of iteration i+k
      - Prefetch the location that pointer i+k points to

Prefetch Examples (5)

- Speculative Prefetch

```
mov eax, dword ptr[ebp]
fld qword ptr [eax]
add ebp, 4

add r32 = r.ebp,(k*4)
// k is the computed #iterations  ahead
// to prefetch for the fld
ld4.s r32 = [r32]
lfetch.nt1  [r32]
```

Analysis for Inserting Prefetches (5)

- IA-32 memory access of form

  [base_reg, index_reg*scale] offset

- Candidate for dynamic stride detection prefetch
  - If (stride on iteration[I-1] == stride on iteration[I])
    - Prefetch using stride
  - FP access which is not constant or speculative, and in heavy FP loop

Prefetch Examples (6)

- Dynamic Prefetch

```
fld qword ptr [eax]

// compute current stride
sub r32 = r.eax, r.prev_addr
// compare current stride to previous stride
cmp.eq.unc p44, p.true = r32, r.prev_stride
// z is the log2 of the computed
// #iterations ahead to prefetch for the fld
shladd r33 = r.prev_stride, z, r.eax
// record current address
or r.prev_addr = 0x0, r.eax
// record current stride
or r.prev_stride = 0x0, r32
(p44) lfetch.nt1 [r33]
```

Analysis for Inserting Prefetches (6)

- Candidates grouped – can be serviced by same prefetch
  - when the address range of the group does not exceed 0.5 of the L3 cache line size
    - [eax]
    - [eax]+8

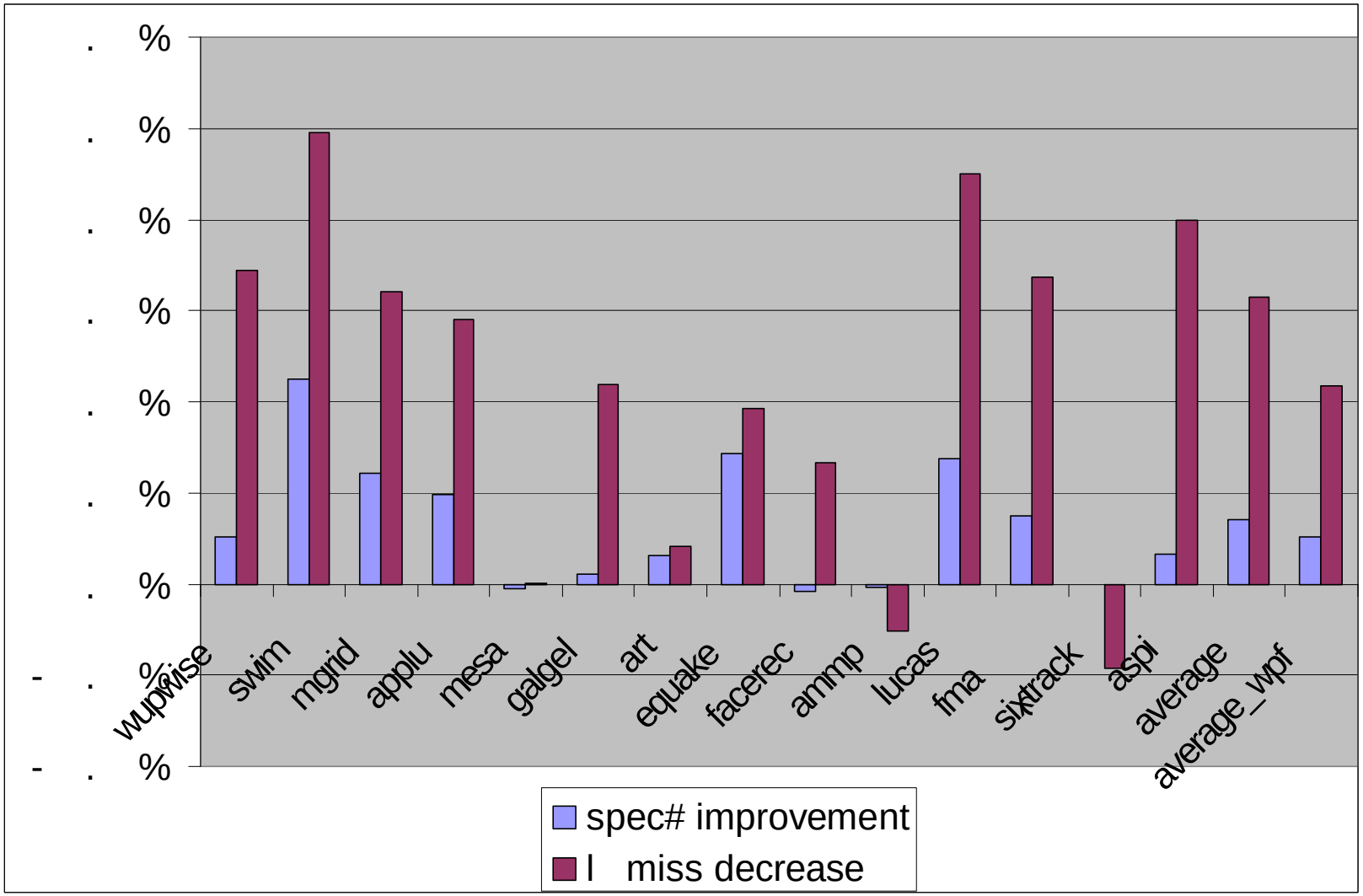
Analysis for Inserting Prefetches (7)

- Select candidate groups to prefetch for
- Determine how many iterations prefetch can cover
  - Explicit constant stride – total access length over n iterations is <= 0.25 L3 cache line size
  - Prefetch is predicated accordingly – or omitted in unrolled iterations
- Determine how many iterations ahead to prefetch
  - Iteration_cycles is guestimate derived from the estimates of the cycles of the cold blocks that compose the hyperblock

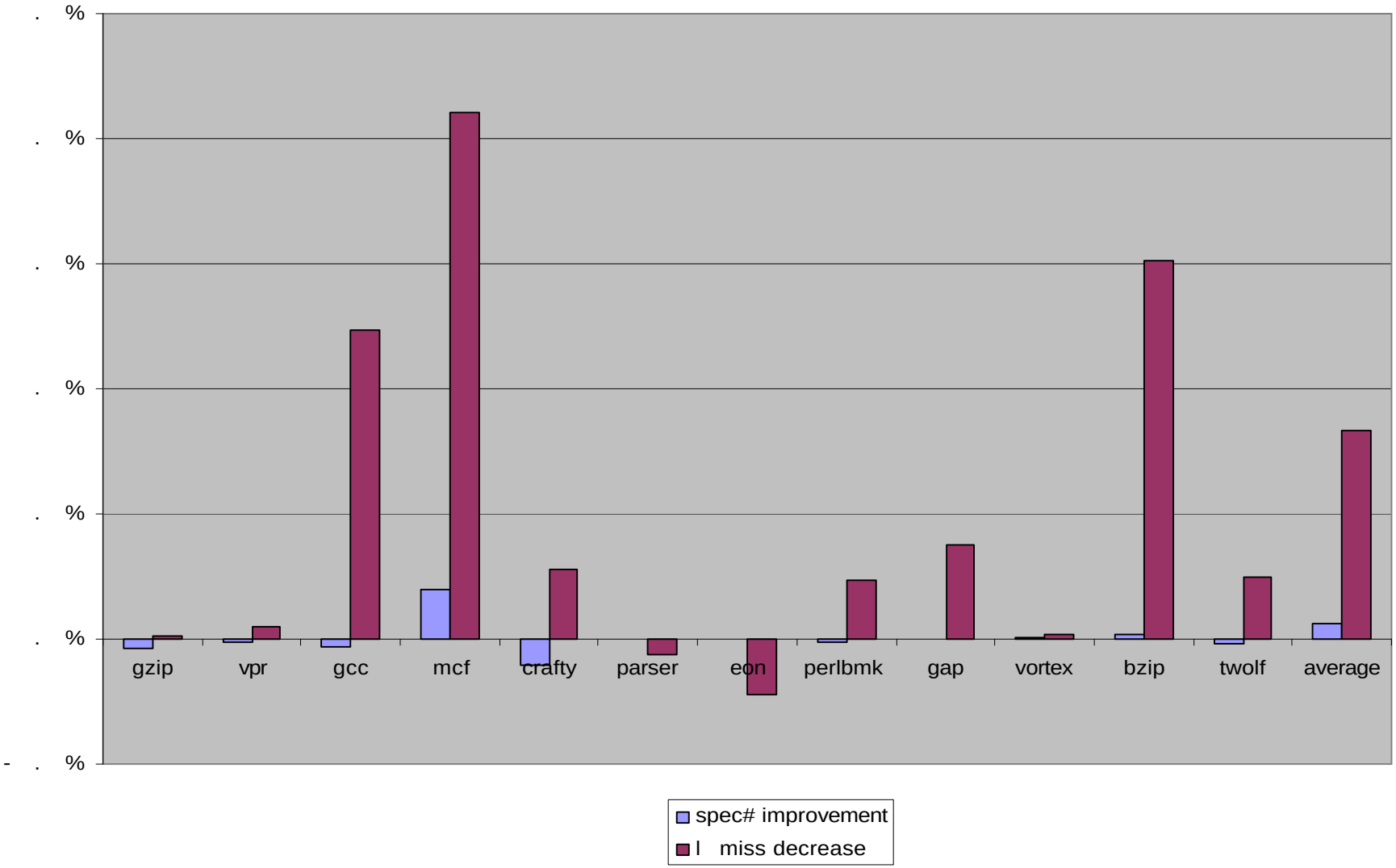
CPU2K Performance Results

- ~15% improvement on FP for IA-32 Binaries with NO prefetch instructions
- ~10% improvement on FP for IA-32 Binaries with prefetch instructions (average_wpf in graph)
  - Naïve translation of IA-32 prefetch instructions
    - VS
  - Generated prefetches only
- No significant affect on INT
- L3 Cache miss cost percentage

CPU2K FP – Prefetch Impact



CPU2K INT – Prefetch Impact



Performance #'s

- Overhead of analysis for inserting prefetches is negligible
- Dynamic stride prefetch– significant gains on 183.equake and 189.lucas
  - No losses on others
- Speculative prefetch – no gains or losses – needs further analysis

Related Work

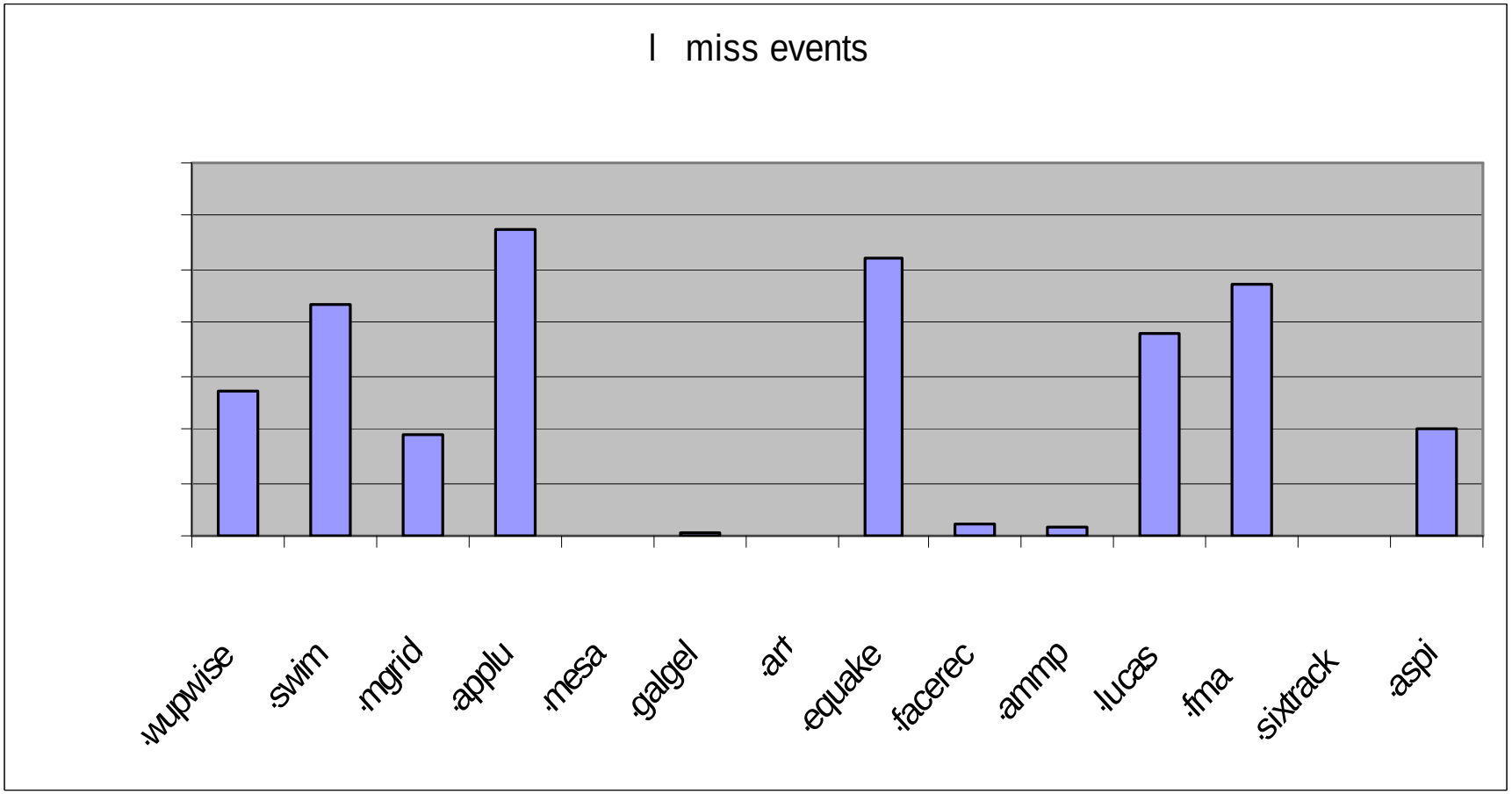
- Based on prefetch literature
  - Dynamic and Speculative stride are new
- Other Approach
  - Instrument IA32-EL hot loops to gather stride info dynamically
  - After specified number of instrumented iterations-
    - Regenerate the hot loop, using the discovered strides for prefetching

Future Work

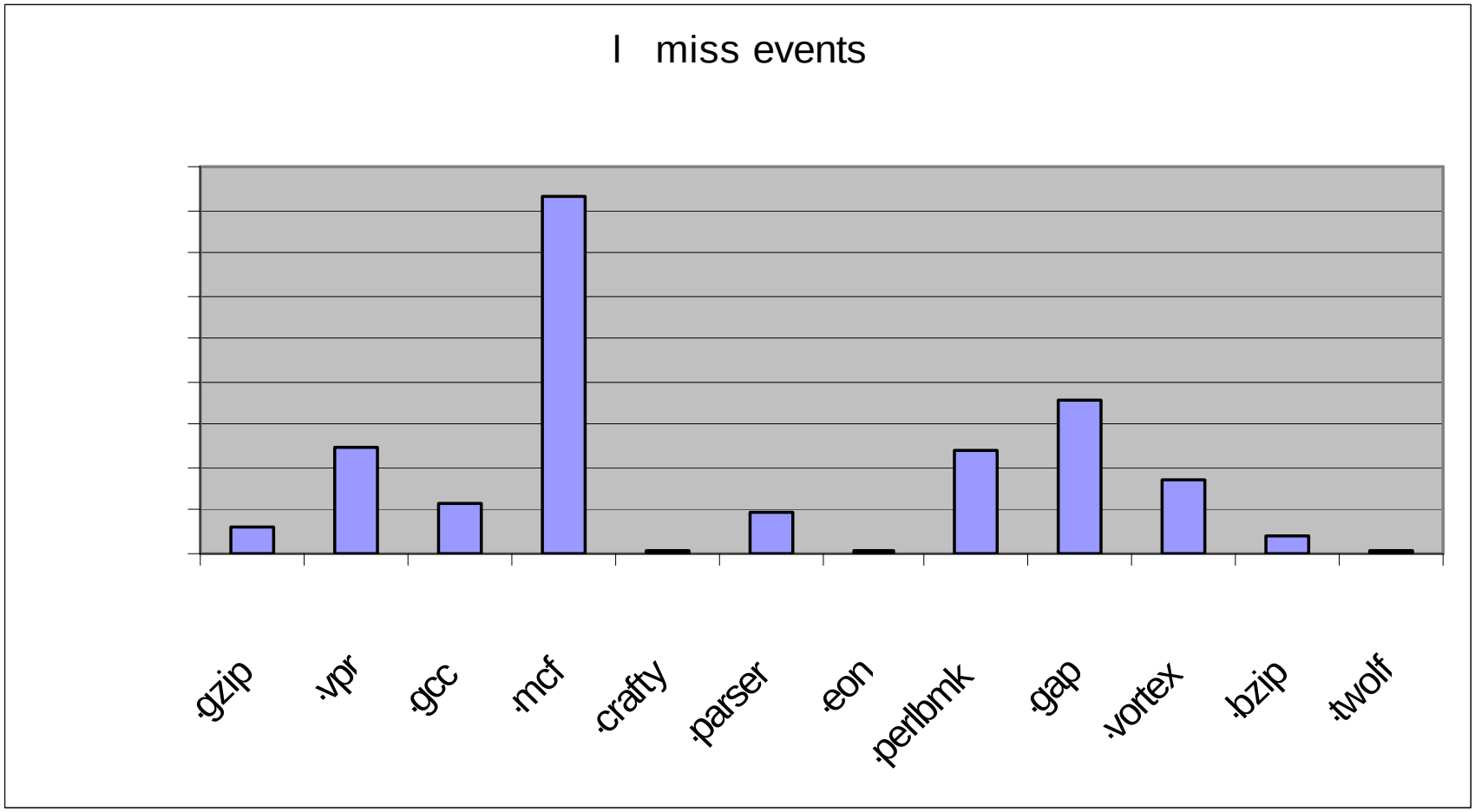
- Sample hardware performance registers to determine strides and misses (DATA_EAR events)
  - Use the collected data
  - Offline binary optimizers do this
- More complex static stride recognition
  - Strides that result from a number of operations within the loop
  - Prefetch for low iteration nested inner loops based on outer loop

Backup

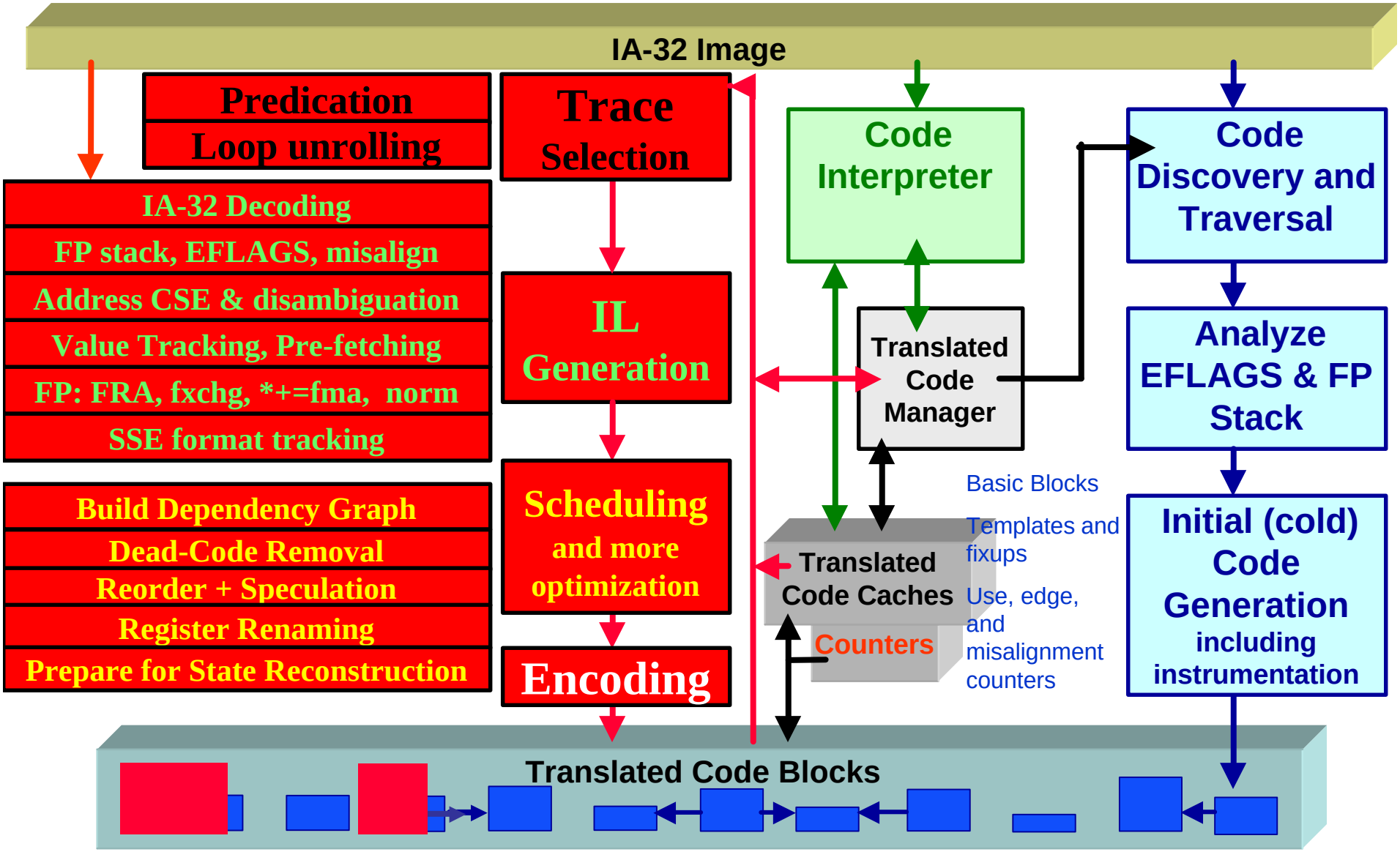
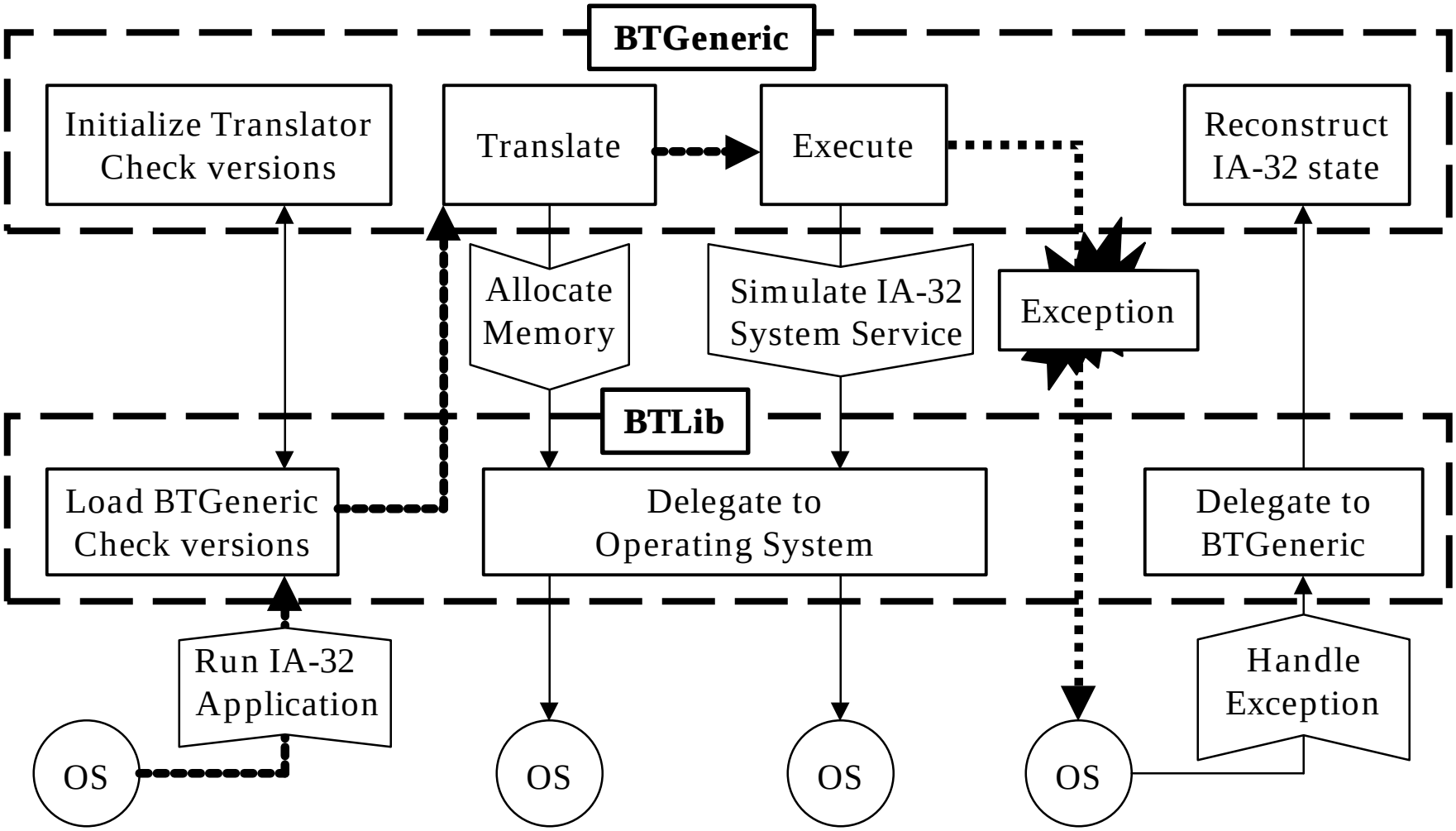
CPU2K FP L3 Miss Events



CPU2K INT L3 Miss Events



Translation Framework



IA-32EL System View