MIAMI: Machine Independent Application Models for performance Insight

Computing Recipes for Performance Tuning

Gabriel Marin
Background

• There is a need for deeper performance analysis
  – Gaining insight into performance bottlenecks
• MIAMI: performance modeling based on static and dynamic analysis of optimized x86-64 binaries
  – Language independence, code coverage, capture optimization effects
• Application centric, single node performance models
  – Identify performance limiters at loop level
    • Insufficient ILP, uneven resource utilization, contention on machine resources, memory latency or bandwidth
    • Insight into what code transformations are needed
  – Estimate potential for performance improvement
  – Understand when not to fix an apparent problem
MIAMI Diagram

- **x86 object code**
  - **CFGs**, edge counts
  - MIAMI code IR, instr/µop/registers
  - PIN

- **Dependence graph at loop level**
  - Loop nesting structure
  - Performance predictions, performance limiters, potential for performance improvement
    - map metrics to source code and data structures

- **Dependence graph customized for machine**
  - instruction latencies, idiom replacement

- **Set assoc. cache miss predictions**
  - data reuse insight

- **Machine model (MDL)**

- **Memory reuse distance analysis**

- **Performance predictions, performance limiters, potential for performance improvement**

- **XML performance database**

- **hpcviewer**

- **modulo scheduler**
  - binutils

- **PIN**
Dynamic Analysis

• Light weight tool on top of PIN
  – Discover CFGs incrementally at run-time
  – Selectively insert counters on edges
    • Understand routine entry points, function calls that do not return or return multiple times
  – Save CFGs and selected edge counts
  – 2x – 3x slowdown with PIN

• There are other alternatives
  – Sampling on the branch target buffer
    • Trade overhead for complexity and accuracy
  – Somewhat independent of the rest of the analysis, can be a replacement
Static Analysis

• Input:
  – CFGs with partial edge counts

• Methodology:
  – Recover execution counts for all blocks and edges
  – Understand routine entry points, function calls that do not return or return multiple times
  – Compute loop nesting structures
  – Infer executed paths and their execution frequencies
  – Compute instruction schedule for executed paths
Static Analysis

• Rebuild CFGs and recover execution counts for all blocks and edges
Static Analysis

- Rebuild CFGs and recover execution counts for all blocks and edges
- Compute loop nesting structures
Static Analysis

- Rebuild CFGs and recover execution counts for all blocks and edges
- Compute loop nesting structures
- Infer executed paths and their execution frequencies
  - at loop level from the inside out
  - each block is considered at most at one loop level
Instruction Scheduling

• Compute instruction schedule one path at a time
  – Emulates ideal branch predictor
• Decode native instructions into generic instructions
  – Generic instructions resemble RISC instructions or x86 micro-ops
• Build dependence graph for path
• Machine description language → architecture model
  – Tailor dependence graph for machine
  – Instantiate scheduler with architecture description
• Compute modulo instruction scheduling
  – Emulates out-of-order execution
Instruction Decoding

- Built on top of XED
- Map instructions onto a 5-D space
  - Instruction type (~ 45 bins)
  - Exec unit style: vector, scalar
  - Operands type: fp, int
  - Bit width: 16, 32, 64, 80, ...
  - Vector width: 64, 128, 256, ...
- Together with the CFG defines the MIAMI IR of the application
Instruction Decoding

• Only Load, Store and Loadstore micro-ops operate on memory
• For an x86 instruction, each memory operand results into a new Load or Store micro-op, in addition to the micro-op for the main operation
  – Exception: moves that simply copy a value to or from memory
    • they are decoded to a single Store or Load
• Stack push/pop (implicit) operations result in multiple micro-ops (stack pointer increment + mem uop)
• REP instructions have a branch uop appended
• Care must be taken into assigning original x86 operands to the new micro-ops
  – Instruction dependencies and dataflow analysis are computed on IR
Instruction Decoding

- One x86 (CISC) instruction can translate to a sequence of generic instructions

iclass LEAVE  category MISC  ISA-extension BASE  ISA-set I186
instruction-length 1  operand-width 64  effective-operand-width 64
effective-address-width 64
Operands
#   TYPE            DETAILS        VIS  RW       OC2 BITS BYTES NELEM
#   ====           =======        ===  ==       === ==== ===== =====
0   MEM0           (see below) SUPPRESSED  R     V  64  8  1
1  BASE0             BASE0=RBP SUPPRESSED  R ASZ  64  8  1
2   REG1              REG1=RBP SUPPRESSED RW V  64  8  1
3   REG2              REG2=RSP SUPPRESSED RW V  64  8  1

0) IB:    Move
Width:   64
Veclen:  1
ExUnit:  SCALAR
ExType:  int
Primary: yes
SrcOps: 1  (REGISTER/2)
DstOps: 1  (REGISTER/3)
ImmValues: 0

1) IB:    Load
Width:   64
Veclen:  1
ExUnit:  SCALAR
ExType:  int
Primary: no
SrcOps: 1  (MEMORY/0)
DstOps: 1  (REGISTER/2)
ImmValues: 0

2) IB:    Add
Width:   64
Veclen:  1
ExUnit:  SCALAR
ExType:  int
Primary: no
SrcOps: 2  (REGISTER/3) (IMMED/0)
DstOps: 1  (REGISTER/3)
ImmValues: 1  (s/8/8)
Matrix Multiply Example

register int i, j, k, r;
for (r=0 ; r<reps ; ++r) {
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            for (k = 0; k < n; k++) {
                c[i][j] += a[i][k]*b[k][j];
            }
        }
    }
}

Assembly code for inner most loop:
- compiler unrolled the loop 16 times

movaps xmm1,XMMWORD PTR [rcx+r9*8+0x609120]
movaps xmm2,XMMWORD PTR [rcx+r9*8+0x609130]
movaps xmm3,XMMWORD PTR [rcx+r9*8+0x609140]
movaps xmm4,XMMWORD PTR [rcx+r9*8+0x609150]
movaps xmm5,XMMWORD PTR [rcx+r9*8+0x609160]
movaps xmm6,XMMWORD PTR [rcx+r9*8+0x609170]
movaps xmm7,XMMWORD PTR [rcx+r9*8+0x609180]
movaps xmm8,XMMWORD PTR [rcx+r9*8+0x609190]
mulpd xmm1,xmm0
mulpd xmm2,xmm0
mulpd xmm3,xmm0
mulpd xmm4,xmm0
mulpd xmm5,xmm0
mulpd xmm6,xmm0
mulpd xmm7,xmm0
mulpd xmm8,xmm0
adddp xmm1,XMMWORD PTR [rcx+r9*8+0x60d920]
adddp xmm2,XMMWORD PTR [rcx+r9*8+0x60d930]
adddp xmm3,XMMWORD PTR [rcx+r9*8+0x60d940]
adddp xmm4,XMMWORD PTR [rcx+r9*8+0x60d950]
adddp xmm5,XMMWORD PTR [rcx+r9*8+0x60d960]
adddp xmm6,XMMWORD PTR [rcx+r9*8+0x60d970]
adddp xmm7,XMMWORD PTR [rcx+r9*8+0x60d980]
adddp xmm8,XMMWORD PTR [rcx+r9*8+0x60d990]
movaps XMMWORD PTR [rcx+r9*8+0x60d920],xmm1
movaps XMMWORD PTR [rcx+r9*8+0x60d930],xmm2
movaps XMMWORD PTR [rcx+r9*8+0x60d940],xmm3
movaps XMMWORD PTR [rcx+r9*8+0x60d950],xmm4
movaps XMMWORD PTR [rcx+r9*8+0x60d960],xmm5
movaps XMMWORD PTR [rcx+r9*8+0x60d970],xmm6
movaps XMMWORD PTR [rcx+r9*8+0x60d980],xmm7
movaps XMMWORD PTR [rcx+r9*8+0x60d990],xmm8
add r9,0x10
cmp r9,0x30
jb 0x400aa0 <main+528>
Dependency Graph

- For the innermost loop
Machine Description Language

• Construct a model of the target architecture
  – Enumerate machine resources
  – Describe instruction execution templates & resource usage
  – Scheduling constraints between resources
  – Idiom replacement
    • Account for differences in ISAs, micro-architecture features
  – Memory hierarchy characteristics
  – Various machine features
**AMD 10h Architecture**

\[
\text{CpuUnits} = U\_\text{ALU} \times 3, U\_\text{AGU} \times 3, U\_\text{Mul}, U\_\text{ABM},\\
U\_\text{IDiv}, U\_\text{LS} \times 2,\\
U\_\text{FpAdd}, U\_\text{FpMul}, U\_\text{FpStore},\\
O\_\text{Port} \times 3;
\]
/* f2iConvert32 */
Instruction Convert{32}:int template = U_FpAdd+U_FpStore+U_ALU, NOTHING*7;
Instruction Convert{32}:int,vec{128} template = U_FpStore, NOTHING*3;

/* f2iConvert64 */
Instruction Convert{64}:int template = U_FpAdd+U_FpStore+U_ALU, NOTHING*7;

/* i2fConvert32 */
Instruction Convert{32}:fp template = U_FpAdd+U_FpStore, NOTHING*8 |
        U_FpMul+U_FpStore, NOTHING*8;
Instruction Convert{32}:fp,vec{128} template = U_FpStore, NOTHING*3;

/* i2fConvert64 */
Instruction Convert{64}:fp template = U_FpAdd+U_FpStore, NOTHING*8 |
        U_FpMul+U_FpStore, NOTHING*8;
Instruction Convert{64}:fp,vec{128} template = U_FpStore, NOTHING*3;

/* i2fConvert80 - old x87 instruction, only scalar */
Instruction Convert{80}:fp template = U_FpStore, NOTHING*3;

/* Prefetch does not create a dependence, so latency is irrelevant. Just takes
issue bandwidth to execute it. */
Instruction Prefetch template = U_AGU + U_LS;
Instruction Prefetch:vec{512} template = U_AGU + U_LS;
**AMD 10h Architecture**

“The L1 data cache can support two 128-bit loads or two 64-bit store writes per cycle or a mix of those. The LSU consists of two queues—LS1 and LS2. LS1 can issue two L1 cache operations (loads or store tag checks) per cycle. It can issue load operations out-of-order, subject to certain dependency restrictions. LS2 effectively holds requests that missed in the L1 cache after they probe out of LS1. Store writes are done exclusively from LS2. 128-bit stores are specially handled in that they take two LS2 entries, and the store writes are performed as two 64-bit writes.”

/* AMD 10h has only 64 bit stores. 128bit stores are split into two 64bit stores. */
Replace \(\text{Store:} \text{int,vec}\{128\} \) \(rX \rightarrow [rA]\) with
- \(\text{Store:} \text{int,vec}\{64\} \) \(rX \rightarrow [rA] +\)
- \(\text{Store:} \text{int,vec}\{64\} \) \(rX \rightarrow [rA] \) "Store 64b int";

Replace \(\text{Store:} \text{fp,vec}\{128\} \) \(rX \rightarrow [rA]\) with
- \(\text{Store:} \text{fp,vec}\{64\} \) \(rX \rightarrow [rA] +\)
- \(\text{Store:} \text{fp,vec}\{64\} \) \(rX \rightarrow [rA] \) "Store 64b fp";
Dependency Graph

- Tailored for the AMD 10h architecture
# Instruction Scheduler Metrics

## Scheduler predictions

<table>
<thead>
<tr>
<th>Scopes</th>
<th>ExecTime</th>
<th>InfCpuRes</th>
<th>NoDepend</th>
<th>MaxGainExtraRes</th>
<th>MaxGainExtraIP</th>
<th>CPUBottleNeck</th>
<th>BOT_U_LS[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Experiment Aggregate Metrics</strong></td>
<td>1.3e09 100.0</td>
<td>3.4e08 100.0</td>
<td>1.15e09 100.0</td>
<td>1.08e09 100.0</td>
<td>2.77e08 100.0</td>
<td>1.11e09 100.0</td>
<td>1.11e09 100.0</td>
</tr>
<tr>
<td><strong>main</strong></td>
<td>1.3e09 100.0</td>
<td>3.4e08 100.0</td>
<td>1.15e09 100.0</td>
<td>1.08e09 100.0</td>
<td>2.77e08 100.0</td>
<td>1.11e09 100.0</td>
<td>1.11e09 100.0</td>
</tr>
<tr>
<td>loop at source.c: 36</td>
<td>1.3e09 100.0</td>
<td>3.4e08 100.0</td>
<td>1.15e09 100.0</td>
<td>1.08e09 100.0</td>
<td>2.77e08 100.0</td>
<td>1.11e09 100.0</td>
<td>1.11e09 100.0</td>
</tr>
<tr>
<td>loop at source.c: 36</td>
<td>1.3e09 100.0</td>
<td>3.4e08 100.0</td>
<td>1.15e09 100.0</td>
<td>1.08e09 100.0</td>
<td>2.77e08 100.0</td>
<td>1.11e09 100.0</td>
<td>1.11e09 100.0</td>
</tr>
<tr>
<td>loop at source.c: 36</td>
<td>1.3e09 100.0</td>
<td>3.4e08 100.0</td>
<td>1.15e09 100.0</td>
<td>1.08e09 100.0</td>
<td>2.77e08 100.0</td>
<td>1.11e09 100.0</td>
<td>1.11e09 100.0</td>
</tr>
<tr>
<td>loop at source.c: 36</td>
<td>1.3e09 93.5</td>
<td>2.53e08 73.3</td>
<td>1.11e09 96.0</td>
<td>1.08e09 100.0</td>
<td>2.30e08 83.3</td>
<td>1.11e09 100.0</td>
<td>1.11e09 100.0</td>
</tr>
</tbody>
</table>

**Path 1 (x): 2.304E7**

**Path 2: 4.608E7**

**Path 1: 2.303769E7**

**Path 2 (x): 2304.0**

<table>
<thead>
<tr>
<th>Path 1 (x): 48.0</th>
<th>Path 2: 2256.0</th>
<th>Path 2 (x): 1.0</th>
<th>Path 1: 47.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6e01 0.0%</td>
<td>0.00e00 0.0%</td>
<td>1.60e01 0.0%</td>
<td>1.60e01 0.0%</td>
</tr>
<tr>
<td>4.00e00 0.0%</td>
<td>4.00e00 0.0%</td>
<td>2.00e00 0.0%</td>
<td>0.00e00 0.0%</td>
</tr>
<tr>
<td>8.00e00 0.0%</td>
<td>8.00e00 0.0%</td>
<td>2.00e00 0.0%</td>
<td>0.00e00 0.0%</td>
</tr>
<tr>
<td>7.00e00 0.0%</td>
<td>7.00e00 0.0%</td>
<td>3.00e00 0.0%</td>
<td>0.00e00 0.0%</td>
</tr>
<tr>
<td>5.00e00 0.0%</td>
<td>5.00e00 0.0%</td>
<td>3.00e00 0.0%</td>
<td>0.00e00 0.0%</td>
</tr>
</tbody>
</table>

**Main performance limiting factor is the issue bandwidth on the Load/Store units**

## HPCToolkit measurements

\[ 48 \times 48 \times 8 \times 3 = 55\text{KB} \]
• 128-bit Mult * 8, 128-bit Add * 8
  – 16 cycles => 50% efficiency with no memory delays
• 128-bit Load * 16, 64-bit Store * 16
  – Issue bandwidth limited, needs blocking for register reuse
Performance loss due to insufficient ILP

MaxGainExtraIP – improvement potential from increased ILP

routine rtotal accounts for 36% of improvement potential;
- loop computing dtotal accounts for 22% of improv. potential

false recurrence on dtotal;
- icomp/jcomp indices take distinct values but are loaded from another array
Insight from the Scheduler

• Understand losses due to insufficient ILP
• Utilization of various machine resources
  – If vector units are available and not used
    • Failed vectorization
    • Lack of ILP or another machine specific reason
• Contention on machine resources
  – Few options from an application perspective, must change instruction mix
  – Contention on load/store unit -> improve register reuse
Understanding Memory Behavior

• Do not focus on predicting memory penalty
  – It is too hard, latency is hidden by overlap with code or with other memory accesses

• Instead, provide better insight to the user on how to improve data reuse
  – Data reuse is not a local phenomenon

• Understand not only where cache misses occur
  – Identify where data has been previously accessed
  – Identify which algorithmic loop is driving the reuse
    • Important for understanding how to shorten the reuse
Understanding data reuse patterns

- Carrier scope of a data reuse
  - algorithmic loop causing data to be reused

\[
\begin{align*}
  &\text{DO } I = 1, N \\
  &\text{DO } J = 1, M \\
  &A(I,J) = A(I,J) + B(I,J) \\
  &\text{ENDDO} \\
  &\text{ENDDO}
\end{align*}
\]

- carrier scope may be also far removed from the location where data is accessed, e.g. time step loop of an iterative algorithm
  - the farther removed the carrier scope, the more difficult to shorten the reuse

Loop independent temporal reuse to \( A(I,J) \). Loop J carries the reuse.
Spatial reuse to array A carried by the I loop
Spatial reuse to array B carried by the I loop
Interpreting Data Reuse Information

S: source scope, D: destination scope, C: carrying scope of a reuse pattern

- Reuse carried within the same iteration of the carrier scope (also same invocation of a routine body)
  - S and D must be the same scope as C (reuse between different statements), or in disjoint loop nests or routines

- If S, D and C are in the same routine
  - fuse S and D
- S and/or D are in routines called from C, e.g. reuse between different sub-steps of a computation
  - strip-mine S and D with the same stripe; promote the loops over stripes outside of C and fuse them
  - the further removed the carrying scope from S and D, the harder it is to shorten the reuse
**Interpreting Data Reuse Information**

*S*: source scope, *D*: destination scope, *C*: carrying scope of a reuse pattern

- Reuse carried across iterations of *C*
  - *S* = *D*, or in the same loop nest
  - *C* iterates over the array’s inner dimension or array indexing independent of *C*
    - apply loop interchange, or
    - apply dimension interchange on the array(s), or
    - apply blocking on a loop inside of *C* and move the loop over blocks outside of *C*
  - *S* and *D* in disjoint loop nests or routines
    - combination of the previous two cases; apply loop fusion + blocking/loop interchange
    - usually, it is harder to optimize
    - Large number of irregular misses and *S* = *D*
      - apply data or computation reordering
About 98% of cache misses and 49% of TLB misses are due to long reuse within the 3rd level loop.

Loop at level 1 carries most of these misses. Moreover, these misses occur on array ‘b’.
- move the i-loop in an inner position, or
- block the j-loop and move the loop over blocks outside of the i-loop.
Bandwidth Constraints

• Miss counts at loop level estimated from reuse distance models

• Minimum bandwidth requirements at loop level
  – \( \text{miss\_count} \times \text{block\_size} / \text{loop\_schedule\_time} \)
  – Assumes ideal prefetching and no memory latency delays
  – Ultimate “loop balance” metric

• One school of thought holds that only bandwidth matters, latency can be hidden
  – Peak machine bandwidth obtained from the machine description file
  – If required loop bandwidth > peak bandwidth
    • do not focus on ILP, vectorization, or register reuse; they increase bandwidth demand
Summary

Putting everything back together

- Analyze full application binaries and create optimization recipes at loop level
  - Compute instruction schedule
    - Understand performance inefficiencies due to lack of ILP, failed vectorization, resource contention
  - Perform memory reuse simulation
  - Compute “loop balance”, compare with peak bdwth
    - Understand if instruction schedule inefficiencies are on the critical path
  - Analyze data reuse patterns to look for improvement opportunities, suggest code transformations
  - Possibly interface with an auto-tuning tool