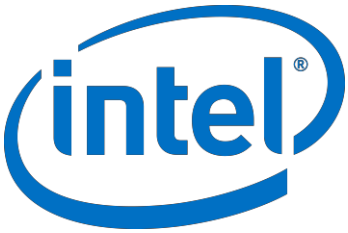


Intel® Architecture Code Analyzer

User's Guide

Copyright © 2009-2017 Intel Corporation

All Rights Reserved

Document Number: 321356-001US

Revision: 2.3

World Wide Web: http://www.intel.com

Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order. Copies

of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

This document contains information on products in the design phase of development.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

BlueMoon, BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Cilk, Core Inside, E-GOLD, Flexpipe, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Insider, the Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow., the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel vPro, Intel XScale, InTru, the InTru logo, the InTru Inside logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Moblin, Pentium, Pentium Inside, Puma, skoool, the skoool logo, SMARTi, Sound Mark, Stay With It, The Creators Project, The Journey Inside, Thunderbolt, Ultrabook, vPro Inside, VTune, Xeon, Xeon Inside, X-GOLD, XMM, X-PMU and XPOSYS are trademarks of Intel Corporation in the U.S. and/or other countries.

* Other names and brands may be claimed as the property of others.

Copyright (C) 2009-2017, Intel Corporation. All rights reserved.

Contents

1 Introduction

Intel® Architecture Code Analyzer helps you statically analyze the throughput of instruction sequences (kernels) on Intel® microarchitectures.

For a given binary, Intel Architecture Code Analyzer:

- Identifies the binding of the kernel instructions to the processor ports under ideal front-end, out-of-order engine and memory hierarchy conditions.
- Performs static analysis of the kernel throughput and reports its cycle count.
- Identifies the critical path.

1.1 Intel® Architecture Code Analyzer Accuracy

Intel Architecture Code Analyzer enables you to do a first order **estimate** of the relative performance of sections of code on different microarchitectures. It **does not** provide absolute performance numbers.

The performance data reported by the tool may significantly deviate from actual performance observed on an Intel® processor. You can achieve the most accurate throughput measurements by executing the analyzed code on the processor itself. The Intel® Architecture Code Analyzer complements such measured data with information on port binding, bottlenecks, and critical paths.

1.2 Processor Support

Intel Architecture Code Analyzer supports analysis for 1st to 6th generation Intel® Core™ processors, which correspond to Intel® microarchitectures codenamed Sandy Bridge (2nd gen), Ivy Bridge (3rd gen), Haswell (4th gen), Broadwell (5th gen) and Skylake (6th gen), including Skylake Server.

1.3 Platform Support

Intel Architecture Code Analyzer is a command-line utility that can analyze a binary file that contains code with special markers that delimit the analyzed code. The tool analyses Intel® 64 code including Intel® Advanced Vector Extensions (Intel® AVX), AVX2 and AVX3 instructions.

Intel Architecture Code Analyzer is available on Windows*, Linux*, Mac OS X* operating systems (for 64-bit editions).

NOTE:	Intel® Architecture Code Analyzer has been validated on 64-bit SUSE* 11, Mac OS X* 10.12.1 and Microsoft* Windows 8.1 64-bit. It should work on other versions of Linux*, Mac OS X* and Microsoft* Windows operating systems.

2 Analysis

2.1 Throughput Analysis

Throughput Analysis is used to analyze the throughput and bottlenecks of a loop body; it treats the contents of the analyzed block as an infinite loop, including considering inter-iteration dependencies between instructions within the analyzed block. The Throughput Analysis report provides the following information:

- Throughput of the whole analyzed block, counted in cycles. The block throughput is calculated as the maximum between:
  - Throughput of the processor's ports
  - Maximum front-end throughput (4 micro-ops per cycle)
  - Divider unit throughput
- Bottleneck source that limited the throughput: front-end, port number, divider unit, or long dependency chains.
- Total number of cycles each processor port was bound by micro-ops.

The detailed section of the throughput analysis report contains one line for each instruction in the analyzed block. Each line contains:

- Number of the instruction micro-ops.
- Average number of cycles per iteration that the instruction was bound to each processor port. For most instructions this simply means the number of cycles the instruction was bound to each port. However, if a particular micro-op may execute on more than one port, the average number of cycles per iteration may be a partial cycle for each port because that micro-op may bind to a different port on each iteration.
- An indication whether the instruction is on the critical path of the analyzed code.
- Instruction disassembly in Intel® Software Developer's Manual (MASM) style

Some ports have both a regular pipe and a secondary pipe. These ports are separated by a hyphen, and look like two separate ports in the detailed report. Specifically:

- Port 0 has the Divider pipe split from it. In the first cycle they are both busy, then port 0 is available for the next micro-op and the Divider pipe is kept busy for the duration of the divide operation.
- Load ports 2 and 3 have an Address Generation Unit (AGU) split from them. For 256-bit load operations that keep the port busy for two cycles on SnadyBridge and IvyBridge micro-architectures, the AGU gets freed after the first cycle and can process a store address generation if such micro-op isavailable for execution.

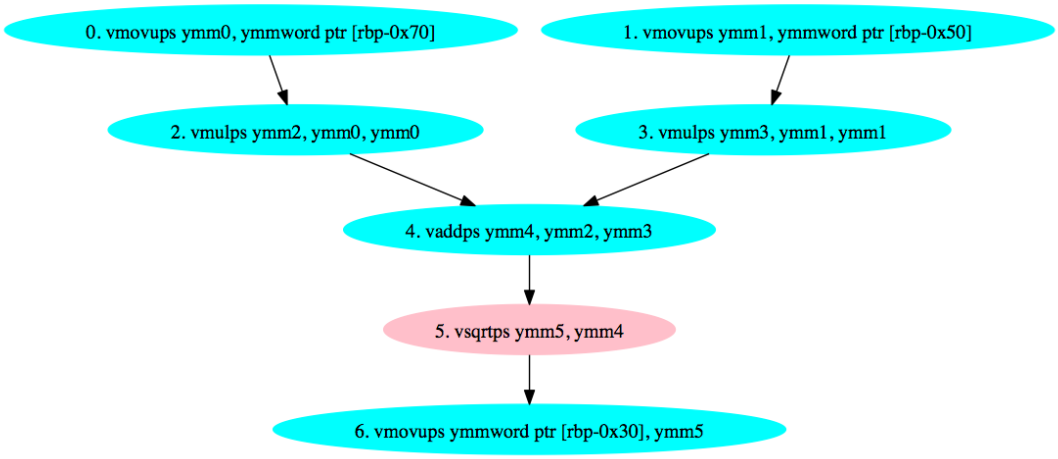
Following is an example Throughput Analysis report:

```
Throughput Analysis Report
--------------------------
Block Throughput: 28.00 Cycles       Throughput Bottleneck: Divider

Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |
---------------------------------------------------------------------------
| Cycles | 4.0    28.0 | 1.0  | 1.5    2.0  | 1.5    2.0  | 2.0  | 1.0  |
---------------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256/AVX512 instruction, dozens of cycles penalty is expected
X - instruction not supported, was not accounted in Analysis

| Num Of |                    Ports pressure in cycles                     |    |
|  Uops  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |    |
---------------------------------------------------------------------------
|   1    |             |      | 1.0    2.0  |             |      |      |    | vmovups ymm0, [rbp-0x70]
|   1    |             |      |             | 1.0    2.0  |      |      |    | vmovups ymm1, [rbp-0x50]
|   1    | 1.0         |      |             |             |      |      |    | vmulps ymm2, ymm0, ymm0
|   1    | 1.0         |      |             |             |      |      |    | vmulps ymm3, ymm1, ymm1
|   1    |             | 1.0  |             |             |      |      |    | vaddps ymm4, ymm2, ymm3
|   3    | 2.0    28.0 |      |             |             |      | 1.0  | CP | vsqrtps ymm5, ymm4
|   2^   |             |      | 0.5         | 0.5         | 2.0  |      |    | vmovups [rbp-0x30], ymm5
```

2.2 Graphs

Use the –graph option to set Intel® Architecture Code Analyzer to output the data dependency graph.

TIP: Graph files produced by Intel® Architecture Code Analyzer can be opened with graphviz.



2.3 Trace

New in version 2.3

To generate a trace use '–trace <path>' option to generate an '.iacatrace' file in <path>.
Run 'pt.py <path-to-.iacatrace-file>' to generate a readable output.

NOTE: Iaca trace requires Python 2.7.5 or Python 3.6.1 to run.

Traces include in-depth information about different operation stages inside the processor. A trace can be used to identify bottlenecks and pressure points.

```
it|in|Dissasembly                                          :0123456789012345678901234567890123456789
0 |0 |vmovups xmm2, xmmword ptr [rdx+r10*4]                :          |         |         |         |
0 |0 |    LOAD/1 (1 uops)                                 :s---deeeeew    R    |         |         |
0 |1 |vmovaps xmm0, xmmword ptr [r8+r10*4]                 :          |         |         |         |
0 |1 |    LOAD/1 (1 uops)                                 :s---deeeeew    R    |         |         |
0 |2 |vdivps xmm2, xmm2, xmm0                              :          |         |         |         |
0 |2 |    OP/2 (1 uops)                                   :A---------|deeeeeeee|ew    R  |         |
0 |3 |vmovaps xmmword ptr [rcx+r10*4], xmm2                :          |         |         |         |
0 |3 |    STORE ADDRESS/3 (1 uops)                        :s---cdeeeew         |      R  |         |
0 |3 |    STORE DATA/4 (1 uops)                           : A--------|---------|--dw    R|         |
0 |4 |vmovups xmm5, xmmword ptr [rdx+r10*4+0x10]           :          |         |         |         |
0 |4 |    LOAD/1 (1 uops)                                 : s---cdeee|ew       |        R|         |
0 |5 |vmovaps xmm3, xmmword ptr [r8+r10*4+0x10]            :          |         |         |         |
0 |5 |    LOAD/1 (1 uops)                                 : s---deeee|w        |        R|         |
0 |6 |vdivps xmm5, xmm5, xmm3                              :          |         |         |         |
0 |6 |    OP/2 (1 uops)                                   : A--------|--c-cdeee|eeeeeew  | R       |
0 |7 |vmovaps xmmword ptr [rcx+r10*4+0x10], xmm5           :          |         |         |         |
0 |7 |    STORE DATA/4 (1 uops)                           :  A-------|---------|-------dw|   R     |
0 |7 |    STORE ADDRESS/3 (1 uops)                        :  s---deee|w        |         |   R     |
0 |8 |add r10, 0x8                                        :          |         |         |         |
0 |8 |    OP/2 (1 uops)                                   :  sdw     |         |         |   R     |
0 |9 |cmp r10, rax                                        :          |         |         |         |
0 |9 |    OP/2 (1 uops)                                   :   s-dw   |         |         |   R     |
```

Above is an example of a trace output.

The kernel instructions are modeled, in order, from top to bottom while the processor's cycles run from left to right. The 'it' column shows the iteration count of the entire kernel, the 'in' column shows the instruction count within the kernel and the 'Disassembly' column shows the instruction's disassembly, along with the micro-architectural instruction fragment information. By default the first 150 cycles of the modeled execution are displayed; use the `--start-cycle` and `--start-iteration` options of the `pt.py` script to choose a different window to display.

Each instruction is represented by at most 4 instruction-fragments (OP, STORE DATA, STORE ADDRESS,LOAD). The trace displays the micro-architectural stage of each fragment inside the processor at any given cycle from allocation to retire. If two stages happen at the same cycle the most important one of is shown. Specifically when Alloc & sready stages happen in the same stage the sready stage is shown.

The stages are:
[A] – Allocated
[s] – Sources ready
[d] – Dispatched for execution
[e] – Execute
[w] – Writeback

[R] – Retired
[-] – Idle
[+] – Idle (on critical path)

2.4 Analysis Report Notes

2.4.1 Unbound Instructions

Some instructions do not require a processor functional unit to complete their execution. For example, a `xor eax, eax` instruction does not require an execution port because the register is directly set to 0. As a result, their micro-ops are not bound to any port. Instructions that are not bound to a port are marked with a '*' character next to their number of micro-ops.

2.4.2 Combining 256-bit Intel® AVX and Legacy Intel® SSE

Transitioning between 256-bit Intel® AVX instructions and legacy Intel Streaming SIMD Extensions (Intel® SSE) instructions will cause performance penalties. Intel® Architecture Code Analyzer detects these transitions between 256-bit Intel® AVX and legacy Intel® SSE within the analyzed block, and **ignores** the associated performance penalty in the total throughput and total latency summary report. Instead, the summary report includes two additional lines at the top indicating that such sequence(s) exist in the analyzed block, and marks the first transition instruction with a '@' character in the Num of Uops columns.

For more information on transitions between Intel® AVX and Intel® SSE, see Avoiding AVX-SSE Transition Penalties.

2.4.3 Unsupported Instructions

Intel® Architecture Code Analyzer does not support a small subset of the Intel® Architecture Instruction Set. When it reaches an unsupported instruction in the analyzed block it ignores the instruction. It does not take the instruction into account in the port binding analysis or in the throughput calculations.

In such cases, the summary report includes two additional lines at the top indicating that such instruction(s) exist in your code, and marks the instruction with a 'X' character in all columns.

2.4.4 Bubbles in the execution of the front end

The Intel® Architecture Code Analyzer models some of the internal resources of the microarchitecture front end. It may report "front end bubbles" if some or any of these resources become a bottleneck.

2.4.5 VDIV / VSQRT Latency

For some values of their operands (e.g. zero or one) VDIV and VSQRT instructions can produce results earlier than their specified latency. The Intel® Architecture Code Analyzer does not model this behavior. As a result it could be more "pessimistic" for kernels that use these instructions.

3 Using Intel® Architecture Code Analyzer

This section explains how to build your binary so that the Intel® Architecture Code Analyzer can analyze it, and it lists the tool command-line options.

3.1 Building Your Binary

The file **iacaMarks.h** contains macros to denote the start (IACA_START) and end (IACA_END) of the code section for the Intel® Architecture Code Analyzer to evaluate. The Intel Architecture Code Analyzer is a static tool. It treats the analyzed code section as a single consecutive block of instructions. It does not follow branch instructions, not even unconditional branches.

When analyzing a loop construct, place the macros at the following locations:

```
while ( condition )
{
    IACA_START
    <loop body>
}
IACA_END
```

This placement skips the loop initialization and includes the loop-end branch instruction.

These macros modify the **rbx** register in IA-64 code. As a result, the compiler saves this register just before the macro and restores it immediately after the macro.

Once you insert the macros into your code, build your code into an executable file or an object file.

For Microsoft* Visual C++ compiler, 64-bit version, use **IACA_VC64_START** and **IACA_VC64_END**, instead.

NOTE:	Input files generated with the Intel compiler option –Qipo are not supported.

3.2 Command Line Options

The following command runs the Intel® Architecture Code Analyzer:

```
iaca <options> <input file name>
```

`<input file name>` represents the name of the input file.

Available `<options>`:

`-64`	64-bit input file – For backward compatibility.
`-arch <type>`	Architecture type. These are the available types: `NHM`, `WSM`, `SNB`, `IVB`, `HSW`, `BDW`.
`-o <file>`	Specifies an output file. The default is `stdout`. To ensure your output appears correctly, specify an output file. The stdout output line width is limited to 80 characters, but output files have no line width limit.
`-graph <file>`	Specifies an output file for the analysis graph, which can be viewed with graphviz.
`-ignore <boolean>`	Ignores added `pop ebx / push ebx` due to Intel Architecture Code Analyzer Markers. `true` ignores, `false` does not (default: true).
`-reduceout`	Output is reduced.
`-report`	Generate error report.
`-trace <file>`	Generate a trace file to be consumed by IACA's pt.py tool.
`-v`	Version.

3.3 Analysis Errors

Should the analysis fail, the following error messages may appear:

Error message	Possible Cause
`COULD NOT OPEN FILE - <file name>`	The supplied path for the input or output file was incorrect, the input file is not readable or failed to create the output file.
`ILLEGAL INSTRUCTION - <offset>`	Code contains an illegal instruction in the specified byte offset.
`INCORRECT XED2 VERSION`	Mixed files between multiple Intel® Architecture Code Analyzer releases.
`COULD NOT FIND START_MARKER` `COULD NOT FIND END_MARKER`	Code did not contain the proper marker(s). See section 3.1 for more details.

4 Examples

This section provides examples of how to analyze and optimize code using Intel® Architecture Code Analyzer.

4.1 Throughput Analysis – 4x4 Matrix Multiply

This example performs a multiply of two 4x4 matrices using Intel® AVX. The initial code and throughput analysis report for SandyBridge micro-architecture are shown below.

4.1.1 Initial Code Version

```
Throughput Analysis Report
--------------------------
Block Throughput: 12.00 Cycles       Throughput Bottleneck: Port5

Port Binding In Cycles Per Iteration:
-------------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |
-------------------------------------------------------------------------
| Cycles | 8.0    0.0  | 6.0  | 4.0    4.0  | 4.0    4.0  | 4.0  | 12.0 |
-------------------------------------------------------------------------

| Num Of |              Ports pressure in cycles               |    |
|  Uops  |  0  - DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |    |
---------------------------------------------------------------------
|   2^   |           |      | 1.0    1.0  |             |      | 1.0  | CP | vbroadcastf128 ymm9, xmmword ptr [rcx]
|   2^   |           |      |             | 1.0    1.0  |      | 1.0  | CP | vbroadcastf128 ymm10, xmmword ptr [rcx+0x10]
|   2^   |           |      | 1.0    1.0  |             |      | 1.0  | CP | vbroadcastf128 ymm11, xmmword ptr [rcx+0x20]
|   2^   |           |      |             | 1.0    1.0  |      | 1.0  | CP | vbroadcastf128 ymm12, xmmword ptr [rcx+0x30]
|   1    |           |      | 1.0    2.0  |             |      |      |    | vmovaps ymm0, ymmword ptr [rax]
|   1    |           |      |             |             |      | 1.0  | CP | vpermilps ymm1, ymm0, 0x0
|   1    |           |      |             |             |      | 1.0  | CP | vpermilps ymm2, ymm0, 0x55
|   1    |           |      |             |             |      | 1.0  | CP | vpermilps ymm3, ymm0, 0xcc
|   1    |           |      |             |             |      | 1.0  | CP | vpermilps ymm4, ymm0, 0xff
|   1    |           |      |             | 1.0    2.0  |      |      |    | vmovaps ymm0, ymmword ptr [rax+0x20]
|   1    |           |      |             |             |      | 1.0  | CP | vpermilps ymm5, ymm0, 0x0
|   1    |           |      |             |             |      | 1.0  | CP | vpermilps ymm6, ymm0, 0x55
|   1    |           |      |             |             |      | 1.0  | CP | vpermilps ymm7, ymm0, 0xcc
|   1    |           |      |             |             |      | 1.0  | CP | vpermilps ymm8, ymm0, 0xff
|   1    | 1.0       |      |             |             |      |      |    | vmulps ymm1, ymm1, ymm9
|   1    | 1.0       |      |             |             |      |      |    | vmulps ymm2, ymm2, ymm10
|   1    | 1.0       |      |             |             |      |      |    | vmulps ymm3, ymm3, ymm11
|   1    | 1.0       |      |             |             |      |      |    | vmulps ymm4, ymm4, ymm12
|   1    |           | 1.0  |             |             |      |      |    | vaddps ymm1, ymm1, ymm2
|   1    |           | 1.0  |             |             |      |      |    | vaddps ymm3, ymm3, ymm4
|   1    |           | 1.0  |             |             |      |      |    | vaddps ymm1, ymm1, ymm3
|   1    | 1.0       |      |             |             |      |      |    | vmulps ymm5, ymm5, ymm9
|   1    | 1.0       |      |             |             |      |      |    | vmulps ymm6, ymm6, ymm10
|   1    | 1.0       |      |             |             |      |      |    | vmulps ymm7, ymm7, ymm11
|   1    | 1.0       |      |             |             |      |      |    | vmulps ymm8, ymm8, ymm12
|   1    |           | 1.0  |             |             |      |      |    | vaddps ymm5, ymm5, ymm6
|   1    |           | 1.0  |             |             |      |      |    | vaddps ymm7, ymm7, ymm8
|   1    |           | 1.0  |             |             |      |      |    | vaddps ymm5, ymm5, ymm7
|   2^   |           |      | 1.0         |             | 2.0  |      |    | vmovaps ymmword ptr [rdx], ymm1
|   2^   |           |      |             | 1.0         | 2.0  |      |    | vmovaps ymmword ptr [rdx+0x20], ymm5
```

4.1.2 Optimization

The Throughput Analysis Report shows that the total throughput (Block Throughput) is 12 cycles, and port 5 was most pressured (Throughput Bottleneck), with 12 micro-ops allocated to it.

Examination of the instructions that bind to port 5 in the instruction analysis report shows that the instructions were **`broadcasts`** and **`vpermilps`**. The broadcasts can only execute on port 5, but replacing them with **128-bit loads** followed by **`vinsertf128`** instructions reduces the pressure on port 5 because **`vinsertf128`** can execute on port 0. These changes reduced the throughput to 10 cycles.

```
Throughput Analysis Report
--------------------------
Block Throughput: 10.00 Cycles       Throughput Bottleneck: Port0, Port5

Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |
---------------------------------------------------------------------------
| Cycles | 10.0   0.0  | 6.0  | 6.0    6.0  | 6.0    6.0  | 4.0  | 10.0 |
---------------------------------------------------------------------------

| Num Of |             Ports pressure in cycles              |    |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |    |
---------------------------------------------------------------------
|   1    |           |     | 1.0   1.0 |           |     |     |    | vmovaps xmm9, xmmword ptr [rcx]
|   1    |           |     |           | 1.0   1.0 |     |     |    | vmovaps xmm10, xmmword ptr [rcx+0x10]
|   1    |           |     | 1.0   1.0 |           |     |     |    | vmovaps xmm11, xmmword ptr [rcx+0x20]
|   1    |           |     |           | 1.0   1.0 |     |     |    | vmovaps xmm12, xmmword ptr [rcx+0x30]
|   2    | 0.1       |     | 1.0   1.0 |           |     | 0.9 | CP | vinsertf128 ymm9, ymm9, xmmword ptr [rcx], 0x1
|   2    | 0.9       |     |           | 1.0   1.0 |     | 0.1 | CP | vinsertf128 ymm10, ymm10, xmmword ptr [rcx+0x10], 0x1
|   2    |           |     | 1.0   1.0 |           |     | 1.0 | CP | vinsertf128 ymm11, ymm11, xmmword ptr [rcx+0x20], 0x1
|   2    | 1.0       |     |           | 1.0   1.0 |     |     | CP | vinsertf128 ymm12, ymm12, xmmword ptr [rcx+0x30], 0x1
|   1    |           |     | 1.0   2.0 |           |     |     |    | vmovaps ymm0, ymmword ptr [rax]
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm1, ymm0, 0x0
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm2, ymm0, 0x55
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm3, ymm0, 0xcc
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm4, ymm0, 0xff
|   1    |           |     |           | 1.0   2.0 |     |     |    | vmovaps ymm0, ymmword ptr [rax+0x20]
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm5, ymm0, 0x0
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm6, ymm0, 0x55
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm7, ymm0, 0xcc
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm8, ymm0, 0xff
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm1, ymm1, ymm9
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm2, ymm2, ymm10
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm3, ymm3, ymm11
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm4, ymm4, ymm12
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm1, ymm1, ymm2
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm3, ymm3, ymm4
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm1, ymm1, ymm3
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm5, ymm5, ymm9
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm6, ymm6, ymm10
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm7, ymm7, ymm11
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm8, ymm8, ymm12
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm5, ymm5, ymm6
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm7, ymm7, ymm8
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm5, ymm5, ymm7
|   2^   |           |     | 1.0       |           | 2.0 |     |    | vmovaps ymmword ptr [rdx], ymm1
|   2^   |           |     |           | 1.0       | 2.0 |     |    | vmovaps ymmword ptr [rdx+0x20], ymm5
```

5 Release Contents

This section lists the files required for running on Linux*, and Mac OS X* operating systems to analyze Intel® 64 code. Each section also explains which environmental variables to modify.

5.1 Linux* OS

Add the `bin/` directory to the `PATH` environment variable.

Add the `lib/` directory to the `LD_LIBRARY_PATH` environment variable.

Include `include/iacaMarks.h` in your code.

Filename	Description
`bin/iaca`	Intel Architecture Code Analyzer command-line tool
`bin/iaca.sh`	Intel Architecture Code Analyzer invocation script
`bin/pt.py`	Intel Architecture Code Analyzer Pipetrace
`lib/libiacaLoader.so`	Intel Architecture Code Analyzer shared objects

`lib/libiacaLogicSNB.so`
`lib/libiacaLogicIVB.so`
`lib/libiacaLogicHSW.so`
`lib/libiacaLogicBDW.so`
`lib/libiacaLogicSKL.so`
`lib/libiacaLogicSKX.so`
	Intel Architecture Code Analyzer shared objects for each of the supported architectures

`lib/libiacaArchDataSNB.so`
`lib/libiacaArchDataIVB.so`
`lib/libiacaArchDataHSW.so`
`lib/libiacaArchDataBDW.so`
`lib/libiacaArchDataSKL.so`
`lib/libiacaArchDataSKX.so`
	Instruction databases for each of the supported architectures

`lib/libXED2SNB.so`
`lib/libXED2IVB.so`
`lib/libXED2HSW.so`
`lib/libXED2BDW.so`
`lib/libXED2SKL.so`
`lib/libXED2SKX.so`
	XED2 shared objects for each of the supported architectures

`include/iacaMarks.h`	Header file for the start/end markers

5.2 Mac OS X*

Add the `bin/` directory to the `PATH` environment variable.

Add the `lib/` directory to the `DYLD_LIBRARY_PATH` environment variable.

Include `include/iacaMarks.h` in your code.

Filename	Description
`bin/iaca`	Intel Architecture Code Analyzer command-line tool
`bin/iaca.sh`	Intel Architecture Code Analyzer invocation script
`bin/pt.py`	Intel Architecture Code Analyzer Pipetrace
`lib/libiacaLoader.so`	Intel Architecture Code Analyzer shared objects

`lib/libiacaLogicSNB.so`
`lib/libiacaLogicIVB.so`
`lib/libiacaLogicHSW.so`
`lib/libiacaLogicBDW.so`
`lib/libiacaLogicSKL.so`
`lib/libiacaLogicSKX.so`
	Intel Architecture Code Analyzer shared objects for each of the supported architectures

`lib/libiacaArchDataSNB.so`
`lib/libiacaArchDataIVB.so`
`lib/libiacaArchDataHSW.so`
`lib/libiacaArchDataBDW.so`
`lib/libiacaArchDataSKL.so`
`lib/libiacaArchDataSKX.so`
	Instruction databases for each of the supported architectures

`lib/libXED2SNB.so`
`lib/libXED2IVB.so`
`lib/libXED2HSW.so`
`lib/libXED2BDW.so`
`lib/libXED2SKL.so`
`lib/libXED2SKX.so`
	XED2 shared objects for each of the supported architectures

`include/iacaMarks.h`	Header file for the start/end markers

5.3 Windows* OS

Include `iacaMarks.h` in your code.

Filename	Description
`iaca.exe`	Intel Architecture Code Analyzer command-line tool
`iacaLoader.dll`	Intel Architecture Code Analyzer shared objects
`pt.py`	Intel Architecture Code Analyzer Pipetrace

`iacaLogicSNB.dll`
`iacaLogicIVB.dll`
`iacaLogicHSW.dll`
`iacaLogicBDW.dll`
`iacaLogicSKL.dll`
`iacaLogicSKX.dll`
	Intel Architecture Code Analyzer shared objects for each of the supported architectures

`iacaArchDataSNB.dll`
`iacaArchDataIVB.dll`
`iacaArchDataHSW.dll`
`iacaArchDataBDW.dll`
`iacaArchDataSKL.dll`
`iacaArchDataSKX.dll`
	Instruction databases for each of the supported architectures

`iacaXED2SNB.dll`
`iacaXED2IVB.dll`
`iacaXED2HSW.dll`
`iacaXED2BDW.dll`
`iacaXED2SKL.dll`
`iacaXED2SKX.dll`
	XED2 shared objects for each of the supported architectures

`iacaMarks.h`	Header file for the start/end markers