Intrinsics for Low Overhead Tool Annotations

Document Number: 326357-001US

Revision: 1.0

World Wide Web: http://software.intel.com/en-us/articles/intel-cilk-plus-specification/

Legal Information

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

.

Contents

1. Overview

Program analysis tools are valuable. Some of these tools require reporting of events during program execution. For example, race analyzers for OpenMP and Cilk require knowledge of synchronization events. However, using ordinary code to report these events is surprisingly expensive, even if the reporting code is conditionally executed only if a global flag is set. Not only is there the cost of the conditional branch, but also a cost to look up the flag in position-independent code.

This document specifies two compiler intrinsics that eliminate the overhead in most real use cases, and certainly reduces it. The intrinsics enable richly annotating libraries for tools without incurring significant run-time costs when the tools are not in use. Each annotation is more than a mere mark in the instruction stream. It can accept an expression argument like a call to a routine does.

The intrinsics are something that programmers can write in their source code and program analysis tools consume. In some cases, such as for OpenMP or Cilk, a compiler can automatically add specific annotations based on language semantics. The compiler is the middleman between production and consumption. The intrinsics are designed so that the rest of the compilation tool chain, such as linkers and loaders, do not need to understand the data to correctly pass it along.

The primary audience for this document is authors of compilers and program analysis tools. It describes how to produce and consume the annotations generated by the intrinsics. A secondary audience is authors of software. It shows how the intrinsics can be applied to communicate events from source code to a program analysis tool. It is up to the authors of the software and program analysis tool to agree on what kind of events should be communicated.

The rest of this document is structured as follows. Section 2 presents the intrinsics as seen by the programmer. Section 3 describes the code and tables a compiler produces from the intrinsics, or implicitly from language semantics. Section 4 describes a real-life example used in a production system.

2. Programmer Interface

This section describes the intrinsic in the form of the initial *producer*, the programmer. There are two forms of the intrinsic, with the following signatures:

```
extern "C" void
__notify_intrinsic( const char *annotation, const volatile void *tag);
extern "C" void
__notify_zc_intrinsic(const char *annotation, const volatile void *tag);
```

The string *annotation* must be a compile-time constant. It specifies the type of the annotation.

The pointer *tag* is computed at run time. It specifies the data associated with the annotation.

Each intrinsic implies a compiler fence: the compiler must not move any memory operation across it. The reason for this restriction is that annotation might denote an event that must be precisely placed with respect to memory operations.

The difference between the two intrinsics is that `__notify_intrinsic` must leave a *probe-ready* instruction sequence in the instruction stream where the instrinsic occurs. The next section explains what this sequence is. The `__notify_zc_intrinsic` does not leave such a sequence, and hence is closer to "zero cost".

For sake of a concrete example, consider a tool that needs to know when control flow enters and exits a critical region. Suppose a critical region is coded ad-hoc by the user. The user can write it like this to inform the tool:

```
    if( x->synch_variable.compare_and_swap(1,0)==0 ) {
        __notify_intrinsic("enter_critical_region", &x->synch_variable);
        x->protected_value<<=1;
        __notify_intrinsic("exit_critical_region", &x->synch_variable);
        x->synch_variable = 0;
    }
```

The example shows why memory barrier semantics are necessary. Without the implied barrier, the compiler might move `x->protected_value<<=1` above or below the notify calls, which would put it outside the reported critical section, thus confusing the tool.

The memory barrier implied by the intrinsic is only a compiler barrier, *not* a hardware memory barrier, which could have significant cost. If the tool needs a hardware barrier for correct operation, it should insert one in the instruction stream.

3. Tool Interface

This section describes the intrinsic in the form produced by the compiler and the consumed by an analysis tool that executes the program under special control. For concreteness, the examples are given for IA-32 instructions. Extension to other instruction sets is straightforward.

In the executable file, each annotation is quadruple (*ip, probespace, annotation, expr*) in a table called a *ZCA Table* where:

- *ip* describes a location corresponding to where the intrinsic was called. The intrinsic is implied to have been called *before* the actual instruction at *ip*.
- *probespace* is the size in bytes of the *probe-ready* instruction sequence starting at *ip*. The rules for such a sequence are explained shortly.
- *annotation* is the annotation parameter to the intrinsic.
- *expr* describes how to compute the *tag* parameter to the intrinsic.

If probespace is greater than 0, the half-open interval [*ip,ip+probespace*) of bytes is guaranteed to be a *probe-ready* instruction sequence, which means that:

a. The sequence is single-entry single-exit. It does not contain branches out of it. No instruction except the first is the target of a branch.

b. The sequence can be copied bitwise elsewhere still run correctly. All instructions in it are position independent.

c. After the sequence is copied, a jmp can be overwritten onto the sequence. The table below shows minimal sequence length and example jmp.

Architecture	**Minimum *probespace***	**Instruction**	**Encoding**
IA-32	5 bytes	jmp *.L*x*	E9 *xx xx xx xx*
Intel® 64	6 bytes	jmp *.L*x*(%rip)	FF 25 *xx* xx *xx xx*

d. Two sequences must not partially overlap. In other words, if two sequences have a byte in common, they must be identical sequences.

e. Identical sequences must be consecutive rows in the ZCA Table. They will have identical *ip* and *probespace* values, but possibly differing *annotation* and *expr* values. Because there is room for only one probe, the tool should insert a single probe that treats said rows as a consecutive sequence of calls to `__notify_intrinsic`.

f. Annotations created with `__notify_zc_intrinsic` will have a probespace value of 0.

The *expr* is a DWARF encoding of how to compute the tag argument from the current machine state. Whether the decoding is done interpretively, JITing, etc. is up to the tool implementer.

The instruction stream for the example from Section 2 would look something like the following on 64-bit Linux. The insertion point addresses are underlined.

```
0000000000400ab0 <Update>:
  400ab0:       ba 01 00 00 00          mov    $1,%edx
  400ab5:       33 c0                   xor    %eax,%eax
  400ab7:       f0 0f b1 57 08          lock cmpxchg %edx,8(%rdi)
  400abc:       85 c0                   test   %eax,%eax
  400abe:       75 0f                   jne    400acf <Update+0x1f>
  400ac0:       8b 47 04                mov    4(%rdi),%eax
  400ac3:       03 c0                   add    %eax,%eax
```

```
400ac5:        89 47 04                mov     %eax,4(%rdi)
400ac8:        c7 47 08 00 00 00 00    movl    $0x0,8(%rdi)
400acf:        c3                      retq
```

The table would have two rows that represent the following information:

ip	***probespace***	***annotation***	***expr***
400ac0	8	"enter_critical_region"	8+%rdi
400ac8	7	"exit_critical_region"	8+%rdi

Note that the first row has a *probespace* value that covers 3 instructions, in order to meet the minimum *probespace* requirement. In this example, the first probe-ready sequence barely avoids overlapping the second probe-ready sequence. If overlap had been an issue, or the instructions that were illegal for a probe-ready sequence, then the compiler would have to insert a nop to create a valid probe-ready sequence.

3.1 Faithful Control Flow

The compiler must ensure that insertion points retain the original control flow written by the programmer. Conceptually, each invocation of `__notify_intrinsic` or `__notify_zc_intrinsic` generates `notify` corresponding hypothetical `notify` or `notify_zc` operation in the instruction stream. Two rules require attention:

a. A sequence of one or more consecutive `notify` operations in a basic block must be immediately followed by a probe-ready instruction sequence in the same basic block. Enforcing this requirement may require inserting a nop[1], indeed always when a `notify` operation appears at the end of a basic block.

b. If consecutive `notify` or `notify_zc` operations occur in a basic block, their order must be maintained in the ZCA entries.

The following source demonstrates the importance of rule a.

```
__notify_intrinsic ("spin_wait",&x->synch_variable);
while( x->synch_variable.compare_and_swap(0,1)==0 )
    continue;
```

The corresponding machine code might be:

```
        lea       8(%rdi), %rdx
        movl      $1, %ecx
        notify "spin wait",%rdx
.L2:
        xorl      %eax, %eax
        lock      cmpxchg %ecx, 8(%rdi)
```

[1] For examples of nop instructions of various lengths, see section "Using Nops" of the *Intel(R) 64 and IA-32 Architectures Optimization Reference Manual* for recommended 1-9 byte nop instructions.

```
        testl     %eax, %eax
        jne       .L2
```

The operation in bold is at the end of the first basic block of the code. Simply deleting the operation and recording the *ip* address of the next real instruction (`xorl`) would erroneously move the `notify` into the second basic block. Inserting a nop at the end of the first basic block before deleting the `notify` operation removes the hazard. Since the example is for the Intel® 64 architecture, the nop should be 7 bytes, so that it is a probe-ready instruction sequence.

3.2 Table Encoding

This section deals with physical encoding of the ZCA tables.

The ZCA data is stored in a named section. On Windows* the section is named ".itt_not". On Linux* the section is named ".itt_notify_tab".

The linker must concatenate ZCA subtables from separate object files. Thus a function in a linkonce COMDAT sections should have its subtable in an associated COMDAT section. The approach is similar to how exception unwinding tables are handled.

Some linkers been observed to insert arbitrary padding between subtables in COMDAT sections. To make subtables easy to find, each subtable is prefixed with a header consisting of:

- *magic number*: Constant that identifies start of a header described in Section 3.2.1.
- *version number*: Version for format of this information.
- *number of triples:* The number of rows in the subsequent table.

A tool can scan for *magic number* to skip past padding.

3.2.1 ZCA table header

Header for a group of annotations. Multiple tables may be present in a ZCA section. Tools are expected to read all tables in the section to find all of the annotations.

Offsets for the strings and expression tables are added to the address of the `zca_header_t` to generate an absolute address.

```
struct zca_header_t
{
    static const std::size_t magic_sz = 16;

    char      magic[magic_sz];// Magic value - ".itt_notify_tab"
    uint8_t   version_major;  // Major version number
    uint8_t   version_minor;  // Minor version number
    uint16_t  entry_count;    // Count of entries that follow
    uint32_t  strings;        // Offset in bytes to string table
    uint32_t  strings_len;    // Size of string table (bytes)
    uint32_t  exprs;          // Offset in bytes to expression table
    uint32_t  exprs_len;      // Size of expression table (bytes)
};

#define ZCA_MAJOR_VERSION 1
#define ZCA_MINOR_VERSION 1
```

```
#define ZCA_HEADER_MAGIC ".itt_notify_tab"
```

3.2.2 ZCA table row

Note that table rows are packed on 4 byte boundaries.

Offsets for annotation strings are added to the start of the string table pointed to by the previous `zca_header_t`.

Offsets for DWARF expressions are added to the start of the expression table pointed to by the previous `zca_header_t`.

```
struct zca_entry_t
{
    uint64_t    ip;          // Instruction pointer of entry
    uint32_t    probespace; // Bytes of instruction for probe
    uint32_t    annotation; // Offset in bytes into strings table
    uint32_t    expr;       // Offset in bytes into expression table
};
```

4. Annotations for Intel® Cilk™ Plus ABI

The annotations are a general mechanism for marking a code stream, and thus can be used for a variety of tools. This section describes a specific example: a set of annotations for Intel® Cilk™ Plus to be used by race detectors. The annotations described provide a way for an executable compiled from Intel® Cilk™ Plus to inform a tool about synchronization events in a program that need to be noted, and about apparent races that should be ignored.

This section refers to many internals Intel® Cilk™ Plus. These internals are described in an open specification of the Cilk ABI: "The Intel® Cilk™ Plus Application Binary Interface Specification" (http://software.intel.com/sites/products/cilk-plus/cilk_plus_abi.pdf).

Some annotations are inserted into the code by the compiler when compiling function that spawns another function. Some annotations are inserted by hand into the Intel® Cilk™ Plus runtime library. Others are available for users to insert into their code.

4.1 Compiler generated annotations

The code generated by the compiler accesses two structures, `__cilkrts_worker` and `__cilkrts_stack_frame`. Bracketing all modifications of these with begin/end notifications allows tools to ignore those modifications, so there will not be any need to know the size or layout of these structures.

4.1.1 cilk_enter_begin

Parameter: The address of the __cilkrts_stack_frame.

This annotation notifies a tool that a spawning function has been entered. It is inserted by the compiler in any frame which has a `_Cilk_spawn`. The `cilk_enter_begin` notification must precede the call to `__cilkrts_get_tls_worker()`. If full initialization of the __cilkrts_stack_frame is delayed until the first spawn, the `cilk_enter_begin` notification should be delayed until the full initialization.

Note that a different annotation is used in spawn helper functions.

Cilk screen performs the following actions on `cilk_enter_begin`:

1. Push the Cilk frame onto its internal stack.
2. Clean memory allocated in the frame

4.1.2 cilk_enter_helper_begin

Parameter: The address of the __cilkrts_stack_frame.

Notifies a tool that a Cilk spawn helper function has been entered. The `cilk_enter_helper_begin` notification must preceed any initialization of the __cilkrts_stack_frame structure.

Note that a different annotation is used in non-spawn helper functions.

Cilk screen performs the following actions on `cilk_enter_helper_begin`:

1. Push the Cilk frame onto its internal stack.
2. Clean memory allocated in the frame

4.1.3 cilk_enter_end

Parameter: The stable stack pointer

Notifies a tool that the initialization of the __cilkrts_stack_frame for this frame has been completed, along with any associated modifications to the __cilkrts_worker. Inserted for any frame which has either a `cilk_enter_begin` or `cilk_enter_helper_begin` notification. This notification must be matched by a `cilk_enter_begin` or `cilk_enter_helper_begin` notification. If no Cilk functions are called, then a `cilk_enter_end` notification should not be given.

4.1.4 cilk_spawn_prepare

Parameter: The address of the __cilkrts_stack_frame.

Notifies a tools that a spawning function is about to call a spawn helper. Inserted by the compiler before any accesses to the `__cilkrts_stack_frame` or `__cilkrts_worker` associated with the _Cilk_spawn call.

4.1.5 cilk_spawn_or_continue

Parameter: The 0 if this is a spawn, non-zero if this is a continuation.

Notifies a tools that a spawning function is about to call a spawn helper. Inserted by the compiler after any accesses to the `__cilkrts_stack_frame` or `__cilkrts_worker` associated with the _Cilk_spawn call, and before the call to the spawn helper.

4.1.6 cilk_detach_begin

Parameter: The address of the `__cilkrts_stack_frame` of the *parent*

Notifies a tool that the frame is detaching. That is, that it will be available for stealing. Generated in the detach sequence in a spawn helper function.

Cilk screen uses the `cilk_detach_begin` notification to identify a parallel region of code.

4.1.7 cilk_detach_end

Parameter: 0

Notifies a tool that the detach is complete and the parent can be stolen.

4.1.8 cilk_sync_begin

Parameter: The address of the __cilkrts_stack_frame.

Notifies a tool that a `_Cilk_sync` is beginning.

The `cilk_sync_begin` notification may be matched by either a `cilk_sync_end`, if this is the last child to reach the sync, or by a `cilk_resume`, if there are other children outstanding and the runtime steals other work that can be executed.

4.1.9 cilk_sync_end

Parameter: The address of the __cilkrts_stack_frame.

Notifies a tool that all spawned calls must have completed before passing this point. The `cilk_sync_end` notification must be given even if no steals have occurred.

4.1.10 cilk_leave_begin

Parameter: The stable stack pointer – same value as passed to `cilk_enter_end`.

This annotation replaces the existing `cilk_leave` notification.

Notifies a tool that a spawning function or spawn helper is about to exit. Inserted by the compiler at the start of the epilogue in any frame which has a `_Cilk_spawn` and in spawn helpers. This annotation must occur before any modifications to the `__cilkrts_stack_frame` and `__cilkrts_worker` performed before exiting the function.

4.1.11 cilk_leave_end

Parameter: 0

Notifies a tool that any modifications to the `__cilkrts_stack_frame` and `__cilkrts_worker` in the epilogue are complete. No Cilk operations may be done after this point.

4.2 Cilk screen annotations in the Cilk runtime

4.2.1 cilkscreen_metacall

Parameter: Address of a metacall_data_t, defined below.

Allows communication with the tools. metacall_data_t is defined in `internal/metacall.h` as follows:

```
typedef struct
{
    uint32_t tool;  // Specifies tool metacall is for
```

```
                    // (eg. system=0, cilkscreen=1, cilkview=2).
                    // All tools should understand system codes.
                    // Tools should ignore all other codes, except
                    // their own.

    uint32_t code;  // Tool-specific code specifies what to do and
                    // how to interpret data

    void        *data;
} metacall_data_t;
```

The following tool codes are currently defined:

METACALL_TOOL_SYSTEM	0	Common calls implemented by the Cilk Plus runtime. Defined in `internal/metacall.h`
METACALL_TOOL_CILKSCREEN	1	Metacalls that are private to Cilk screen. Defined in `cilk/cilkscreen.h`
METACALL_TOOL_CILKVIEW	2	Metacalls that are private to Cilk view. Defined in `cilk/cilkview.h`

All tools are expected to accept METACALL_TOOL_SYSTEM calls. Tools should ignore private calls for another tool.

There may only be ONE cilkscreen_metacall annotation in a process, and that is in the Cilk runtime. Users outside the Cilk runtime should call `__cilkrts_metacall` to have the Cilk runtime make a metacall on their behalf.

4.2.2 cilk_resume

Parameter: address of `__cilk_stack_frame`

Notifies Inspector that the Cilk runtime is about to resume the specified frame in a spawning function.

4.2.1 cilk_leave_stolen

Parameter: 0

Notifies a tool that the parent has been stolen and `__cilkrts_leave_frame` will not return.

4.2.2 cilk_sync_abandon

Parameter: 0

Notifies a tool that a `_Cilk_sync` will not return.

4.3 Cilk screen annotations defined in `cilkscreen.h`

4.3.1 cilkscreen_disable_instrumentation

Parameter: 0

Disables all Cilk screen instrumentation. Normally this will be generated by the Cilk runtime when a region of Cilk code is left.

4.3.2 cilkscreen_enable_instrumentation

Parameter: 0

Enables instrumentation of code by Cilk screen. Normally this will be generated by the Cilk runtime when a region of Cilk code is entered.

4.3.3 cilkscreen_disable_checking

Parameter: 0

Temporarily prevents Cilk screen from monitoring memory accesses. Does not prevent instrumentation of code segments.

4.3.4 cilkscreen_enable_checking

Parameter: 0

Resumes Cilk screen's monitoring of memory accesses.

4.3.5 cilkscreen_clean

Parameter: Address of `void *data[2]` containing start and end address of newly allocated memory. The end address should be 1 past the end of the allocated block.

Notifies Cilk screen that a range of memory is to be considered "clean". That is, newly allocated and can be used without causing races.

4.3.6 cilkscreen_acquire_lock

Parameter: Address or handle of lock.

Notifies Cilk screen that a lock has been acquired.

4.3.7 cilkscreen_release_lock

Parameter: Address or handle of lock.

Notifies Cilk screen that a lock has been released.