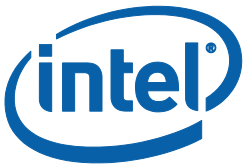

Intel® Cilk™ Plus Application Binary Interface Specification

Part 2 of the Intel® Cilk™ Plus Language Specification version 1.1

Document Number: 324512-002US

Revision: 1.1

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

More information about Intel® Cilk™ Plus can be found at the following web site:

http://cilk.com

Feedback on the specification is encouraged and welcome, please send to:

cilkfeedback@intel.com

Cilk™ Plus Trademark License

Intel hereby grants to you a limited, nonexclusive, non-transferable, royalty-free, revocable (as described in this paragraph) worldwide license to use the trademark "Cilk™ Plus" (the "Licensed Trademark") solely in connection with Compliant Products and no other goods or services. For the purpose of this license grant, Compliant Products mean your software tools, applications and technology which implement and comply with the "Cilk™ Plus Language Specification" or "Cilk™ Plus Application Binary Interface Specification" as set forth in this document. You must use the Licensed Trademark only with an applicable noun that identifies your Compliant Product, such as (but not limited to) the following examples: "the Cilk™ Plus compiler", "the Cilk™ Plus-compliant software", "XYZ Cilk™ Plus compiler" or "XYZ Cilk™ Plus software." Use of the Licensed Trademark is subject to the disclaimer set forth in this document. Should Licensee desire to use the INTEL® mark, the INTEL® logo (or any other Intel marks) in connection with its advertising or promotional materials respecting Compliant Products, any such use must be approved in writing by Intel prior to any such use.

Intel may immediately terminate the license grant described above if you are in breach of any conditions and such breach is not cured within thirty (30) days of written notice from Intel. Any claim or dispute arising under or relating to this license grant shall be governed by the internal substantive laws of the State of Delaware, without regard to principles of conflict of laws.

Non-Trademark Use of Cilk™ Plus

Provided that it is accurate statement, you may fairly state that your software tools, application or technology complies with the "Cilk™ Plus Language Specification" or "Cilk™ Plus Application Binary Interface Specification", such as (but not limited to) the following examples: "XYZ compiler is compliant with the Cilk™ Plus Language Specification", and "XYZ software is compliant with the Cilk™ Plus Application Binary Interface Specification".

Disclaimer and Other Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL(R) 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 (A) SALE AND/OR USE OF INTEL PRODUCTS, (B) THE SPECIFICATION AND (C) THE LICENSED TRADEMARK, INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, VALIDITY OF RIGHTS OR INFRINGEMENT OF ANY PATENT, COPYRIGHT, TRADEMARK OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

NEITHER PARTY SHALL BE LIABLE TO THE OTHER PARTY FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, EVEN IF SUCH PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

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 by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

1 Changelog

15-Oct-2010	BTannenbaum	Initial version based on John Carr's document. Updated for final 12.0 changes to __cilkrts_stack_frame
18-Oct-2010	BTannenbaum	Changed name to "Cilk Plus ABI 0.9", added note about coming ABI changes, incorporated comments from Arch and Angelina

19-Oct-2010	BTannenbaum	• Added explicit note about file names
		• Removed confusion about "frame" in description of handling of CILK_FRAME_STOLEN in __cilkrts_leave_frame()
		• changed definition of CILK_FRAME_MBZ.
		• Noted that __cilkrts_stack_frame.size is unused (and not initialized by Intel compiler)
		• Noted that __cilkrts_worker.saved_protected_tail is unused and initialized to NULL by the runtime
		• Added section on runtime initialization and rundown
		• Incorporate Robert's comments
		  ○ Use keywords with leading _Cilk
		  ○ Use "spawning function" instead of "Cilk function"

20-Oct-2010	BTannenbaum	• Fixed typo in definition of CILK_FRAME_MBZ
21-Oct-2010	PHalpern	Corrections and clarifications of flag meanings. Added to description of cilk_for. Other minor fixes.
21-Oct-2010	BTannenbaum	Responded to Pablo's comments and questions
25-Oct-2010	BTannenbaum	Added standard cover page, legal information, reference to Intel® Cilk™ Plus Language Specification, a few formatting fixes
26-Oct-2010	BTannenbaum	Added common description from John
1-Aug-2011	BTannenbaum	Update for Composer 12.1 release
3-Aug-2011	BTannenbaum	Incorporated comments, added doc for versioned enter frame functions
8-Dec-2011	BTannenbaum	Added Cilk™ Plus Trademark License section

2 Description

This document is part of the Intel® Cilk™ Plus Language Specification version 1.1. The language specification comprises a set of technical specifications describing the language and the run-time support for the language. Together, these documents provide the detail needed to implement a compliant compiler. At this time the language specification contains these parts:

- Part 1. The Intel® Cilk™ Plus Language Specification, document number 324396-002US.
- Part 2. The Intel® Cilk™ Plus Application Binary Interface, document number 324512-002US.

This document describes the Intel® Cilk™ Plus Application Binary Interface, the interface between compiler-generated code and the Intel® Cilk™ Plus runtime. The purpose of this document is to allow a compiler writer to generate code to use the runtime. This interface is version-specific. Previous versions of Cilk have used a different interface and future versions may change the interface. This version matches the version shipped with Compiler Pro 12.1, also known as Composer 2011 and Composer XE 2011.

On Windows, the Cilk Plus runtime is shipped as cilkrts20.dll. Applications link against cilkrts.lib. On Linux, the Cilk Plus runtime is shipped as libcilkrts.so.5. Applications link against libcilkrts.so. The Cilk ABI consists of two data structures and several functions. The structure definitions are shared by the compiler and runtime and so have a defined layout as part of the ABI. All other structure types are opaque to user code. See also header <internal/abi.h>.

It is possible, if somewhat tedious and error-prone, for humans to code to the same interface. C++ exceptions cannot be implemented properly without compiler support. See header <internal/fake.h> for some helpful macros used with a slightly older version of the runtime.

3 Definitions and background

- **Spawning function**. A function that spawns is called a spawning function. The simplest approach is to consider every function that contains a _Cilk_spawn to be a spawning function.

  A function with a _Cilk_for statement is not necessarily a spawning function. Parallel for is implemented as a library call that invokes a nested function.

- **C function**. The term "C function" is used to distinguish ordinary functions from spawning functions and includes C++ functions.
- **Spawn helper**. A function that encapsulates the call that is spawned. It includes any constructors and destructors necessary for the call, and is a spawning function. That is, it has a __cilkrts_stack_frame.
- **Nontrivial sync**. A nontrivial sync is a sync statement in a function that is unsynched, i.e. a sync statement that needs to call into the runtime. A function becomes unsynched when it is stolen at a _Cilk_spawn. See the discussion of the CILK_FRAME_ UNSYNCHED flag.
- **User thread**. The thread that runs `main()` or any other thread explicitly not created by the Cilk Plus runtime is a user thread.

4 Versioning

Version 1.1 introduces versioning to the Intel Cilk Plus ABI.

ABI document version	__CILKRTS_ABI_VERSION	Bind frame function	Enter frame functions
Version 0.9	0	__cilkrts_bind_frame()	__cilkrts_enter_frame __cilkrts_enter_frame_fast
Version 1.1	1	__cilkrts_bind_frame_1()	__cilkrts_enter_frame_1 __cilkrts_enter_frame_fast_1

Versioning is accomplished in three ways:

1. The top 8 bits of the flags field have been reserved for the version number. The compiler must set the ABI version into the version number of the portion the flags when the

__cilkrts_stack_frame is initialized. This informs the runtime what fields are valid in the __cilkrts_stack_frame.

2. The __cilkrts_bind_frame call is versioned. The compiler must call the version that matches the ABI it implements. If a program attempts to load an older version of the Intel Cilk Plus runtime, a loader error will occur.
3. The __cilkrts_enter_frame call is versioned. The compiler must call the version that matches the ABI it implements and the __cilkrts_stack_frame it allocates. Compilers are encouraged to inline these functions.

5 General concepts and code generation

Only spawning functions are visible to the Cilk runtime. Non-spawning functions called by spawning functions are treated as part of the calling spawning function.

All spawning functions require separate stack and frame pointers. Incoming arguments and local variables must be accessed using the frame pointer. Only outgoing arguments should be on the stack. The stack pointer may change unpredictably after spawn. Specifically, when a function is stolen the continuation runs on a new stack. The correct stack pointer, the same as in the serial code, will be restored after sync. The runtime tracks stack pointer changes within a function for whatever stack they are on.

A spawn statement is extracted into a separate function called a I. The spawn helper function is a closure which:

- Initializes the __cilkrts_stack_frame. Note that it can assume that the thread has been bound to the Cilk Plus runtime, so it can use `__cilkrts_enter_frame_fast()` instead of `__cilkrts_enter_frame()`
- Computes the function arguments before the detach
- Detaches
- Calls the function
- Copies the return value if the spawned function isn't a void function
- Calls the destructors for any computed temporaries
- Pops the frame and calls `__cilkrts_leave_frame()` to exit

For example, the following spawn statement:

```
...
x = _Cilk_spawn f(y);
...
```

becomes:

```
void spawn_f(int *x, int y)
{
```

```
        __cilkrts_stack_frame sf;
        __cilkrts_enter_frame_fast(&sf);
        __cilkrts_detach();
        *x = f(y);
        __cilkrts_pop_frame(&sf);
        if (sf->flags)
            __cilkrts_leave_frame(&sf);
    }

    ...
    if (!setjmp(frame.ctx))
        spawn_f(&x, y);
    ...
```

The `__cilkrts_detach()` runtime call is described later. A spawn helper function is a spawning function. A spawn helper function must not be inlined.

The `setjmp()` at point of spawn saves the continuation in case the parent is stolen. If `setjmp()` returns nonzero (always 1) the parent has been stolen; the continuation after the spawn statement has been executed by a different worker which used `longjmp()` to pick up the execution after the `setjmp()` branch test.

6 Runtime initialization and shutdown

The runtime can be manually initialized by calling `__cilkrts_init()` and shutdown by calling `__cilkrts_end_cilk()`. These functions are defined in cilk_api.h. These calls are optional. Normally, the runtime will be initialized by the first call to `__cilkrts_bind_thread()`.

By default the number of workers is the number of cores on the system. The default can be overridden by setting the environment variable CILK_NWORKERS. An application can explicitly set the number of workers by calling `__cilkrts_set_param("nworkers", "N")`, where the second parameter is the number of workers to use, as a string. This call must be made before the runtime has been started; if the runtime is already running, the call will fail and return an error code. Changing the number of workers requires the application to shut down the runtime and restart it.

Unless explicitly shut down by the application, the runtime does not shut down until the application terminates. When the last user thread calls `__cilkrts_leave_frame()` with a __cilkrts_stack_frame which has CILK_FRAME_LAST set in the flags field, the runtime will suspend all of the worker threads it created. The worker threads will wake up at the next call to `__cilkrts_bind_thread()`.

7 __cilkrts_stack_frame

A spawning function contains a frame descriptor object with type `struct __cilkrts_stack_frame`. The descriptor is referred to as "frame" in code fragments.

```
struct __cilkrts_stack_frame
{
    /**
     * flags is an integer with values defined below.  Client code
     * initializes flags to CILK_FRAME_VERSION before the first Cilk
     * operation.
     *
     * The low 24-bits of the 'flags' field are the flags, proper.  The high
     * 8-bits are the version number.
     *
     * IMPORTANT: bits in this word are set and read by the PARENT ONLY,
     * not by a spawned child.  In particular, the STOLEN and UNSYNCHED
     * bits are set on a steal and are read before a sync.  Since there
     * is no synchronization (locking) on this word, any attempt to set
     * or read these bits asynchronously in a child would result in a race.
     */
    uint32_t flags;

    /** Not currently used.  Not initialized by Intel compiler. */
    int32_t size;

    /**
     * call_parent points to the __cilkrts_stack_frame of the closest
     * ancestor spawning function, including spawn helpers, of this frame.
     * It forms a linked list ending at the first stolen frame.
     */
    __cilkrts_stack_frame *call_parent;

    /**
     * The client copies the worker from TLS here when initializing
     * the structure.  The runtime ensures that the field always points
     * to the __cilrts_worker which currently "owns" the frame.
     */
    __cilkrts_worker *worker;

    /**
     * Unix: Pending exception after sync.  The sync continuation
     * must call __cilkrts_rethrow to handle the pending exception.
     *
     * Windows: the handler that _would_ have been registered if our
     * handler were not there.  We maintain this for unwinding purposes.
     * Win32: the value of this field is only defined in spawn helper
     * functions
     *
     * Win64: except_data must be filled in  for all functions with a
     * __cilkrts_stack_frame
     */
    void *except_data;

    /**
     * Before every spawn and nontrivial sync the client function
```

```
     * saves its continuation here.
     */
    __CILK_JUMP_BUFFER ctx;

#if __CILKRTS_ABI_VERSION >= 1
    /**
     * Architecture-specific floating point state.  mxcsr and fpcsr should be
     * set when CILK_SETJMP is called in client code.  Note that the Win64
     * jmpbuf for the Intel64 architecture already contains this information
     * so there is no need to use these fields on that OS/architecture.
     */
    uint32_t mxcsr;
    uint16_t fpcsr;         /**< @copydoc mxcsr */


    /**
     * reserved is not used at this time.  Client code should initialize it
     * to 0 before the first Cilk operation
     */
    uint16_t reserved;

    /**
     * Pedigree information to support scheduling-independent pseudo-random
     * numbers.  There are two views of this information.  The copy in a
     * spawning function is used to stack the rank and communicate to the
     * runtime on a steal or continuation.  The copy in a spawn helper is
     * immutable once the function is detached and is a node in the pedigree.
     * The union is used to make clear which view we're using.
     *
     * In the detach sequence Client code should:
     *    - copy the worker pedigree into the spawn helper's pedigree
     *    - copy the worker pedigree into the call parent's pedigree
     *    - set the worker's rank to 0
     *    - set the worker's pedigree.next to the spawn helper's pedigree
     */
    union
    {
        __cilkrts_pedigree spawn_helper_pedigree;/* Used in spawn helpers */
        __cilkrts_pedigree parent_pedigree;      /* Used in spawning funcs */
    };
#endif  /* __CILKRTS_ABI_VERSION >= 1 */
};
```

The low 24 bits of the flags field are used as flags to signal the state of a frame:

CILK_FRAME_STOLEN	0x01	Set if the frame has ever been stolen or a full frame was created for the stack frame. Set by runtime.
CILK_FRAME_UNSYNCHED	0x02	Set if the frame has been stolen and is has not yet returned from `__cilkrts_sync()`. It is technically a misnomer in that a frame can have this flag set even if all children have returned. Set by runtime.
CILK_FRAME_DETACHED	0x04	Is this frame detached (spawned)? If so the runtime needs to undo-detach in the slow path epilogue. Set by

		generated code, in `__cilkrts_detach()` See section 9.6 for a sample implementation of `__cilkrts_detached()`.
CILK_FRAME_EXCEPTION_PROBED	0x08	Set if the frame has been probed in exception handler first pass (Windows only). Set by runtime.
CILK_FRAME_EXCEPTING	0x10	Is this frame receiving an exception after sync? Set by runtime.
CILK_FRAME_LAST	0x80	Is this the last (oldest) Cilk frame? Set by runtime when the initial __cilkrts_stack_frame is initialized. See section 9.1 for a sample implementation of `__cilkrts_enter_frame()`.
CILK_FRAME_EXITING	0x0100	Is this frame in the epilogue, or more generally after the last sync when it can no longer do any Cilk operations? Set by runtime.
CILK_FRAME_SUSPENDED	0x8000	Is this frame suspended? (used for debugging) Set by runtime.
CILK_FRAME_UNWINDING	0x10000	Set by runtime.
All other bits are reserved for future extensions and must be zero.		

The high 8 bits of the flags field are the version number. ABI version 0 corresponds to Version 0.9 of the Intel Cilk Plus ABI Specification. This document has been extended to correspond to ABI version 1.

The stack frame descriptor has a constructor and destructor. Call `__cilkrts_enter_frame()` before any other use of this structure. Once `__cilkrts_enter_frame()` has been called, call `__cilkrts_pop_frame()` and `__cilkrts_leave_frame()` before returning. Together these are the destructor for the frame descriptor. The function must be synched when calling these functions.

As an optimization, `__cilkrts_leave_frame()` need not be called if the flags field is zero. This is the reason for dividing the destructor into two functions. Frame flags will never be zero when exiting a spawn helper so the test should be omitted in that context. (Either the spawn needs to be undone and the CILK_FRAME_DETACHED bit is set or an exception is propagating and the CILK_FRAME_EXCEPTING bit is set.)

As another optimization, the frame descriptor need not be constructed until the first spawn and may be destructed after the last sync.

WARNING: The Cilk Plus runtime only supports one __cilkrts_stack_frame per spawning function and the call order described above.

8 __cilkrts_worker

The worker structure holds thread local state that needs to be visible to the compiler.

```
struct __cilkrts_worker {
    /**
     * T, H, and E pointers in the THE protocol See "The implementation of
     * the Cilk-5 multithreaded language", PLDI 1998:
     * http://portal.acm.org/citation.cfm?doid=277652.277725
     */
    __cilkrts_stack_frame *volatile *volatile tail;
    __cilkrts_stack_frame *volatile *volatile head;  /**< @copydoc tail */
    __cilkrts_stack_frame *volatile *volatile exc;   /**< @copydoc tail */

    /**
     * Addition to the THE protocol to allow us to protect some set of
     * entries in the tail queue from stealing.  Normally, this is set
     * beyond the end of the task queue, indicating that all entries are
     * available for stealing.  During exception handling, protected_tail
     * may be set to the first entry in the task queue, indicating that
     * stealing is not allowed.
     */
    __cilkrts_stack_frame *volatile *volatile protected_tail;

    /** Limit of the Lazy Task Queue, to detect queue overflow */
    __cilkrts_stack_frame *volatile *ltq_limit;

    /** Worker id */
    int32_t self;

    /** Global state of the runtime system, opaque to the client */
    global_state_t *g;

    /**
     * Additional per-worker state of the runtime system that we want
     * to maintain hidden from the client
     */
    local_state *l;

    /** map from reducer names to reducer values */
    cilkred_map *reducer_map;

    /** A slot that points to the currently executing Cilk frame. */
    __cilkrts_stack_frame *current_stack_frame;

    /** Saved protected tail.  Set to NULL by runtime.  No longer used. */
    __cilkrts_stack_frame *volatile *volatile saved_protected_tail;

    /** System-dependent part of the worker state */
    __cilkrts_worker_sysdep_state *sysdep;

#if __CILKRTS_ABI_VERSION >= 1
    /**
     * Per-worker pedigree information used to support scheduling-independent
     * pseudo-random numbers.
     */
    __cilkrts_pedigree   pedigree;
#endif  /* __CILKRTS_ABI_VERSION >= 1 */
};
```

User code can treat the worker as an opaque structure or may choose to inline some operations.

9 Saving Cilk state

Some runtime calls require a function's state to be saved in the stack_frame. On Windows this is done with `setjmp()`. On Linux (or more generally, in gcc compatible mode on Unix-like operating systems) this is done with `__builtin_setjmp()`. On Linux only, when an uncaught exception is active the CILK_FRAME_EXCEPTION bit must be set in the flags field and the raw exception pointer from the runtime saved in the except_data field. This happens only when sync is called implicitly during stack unwinding.

When saving Cilk state on IA32 and Intel64 architecture CPUs, the compiler must also save the floating point state so it can be restored by the runtime on a steal.

10 Cilk runtime calls

10.1 `void __cilkrts_enter_frame_1(struct __cilkrts_stack_frame *sf);`
`void __cilkrts_enter_frame_fast_1(struct __cilkrts_stack_frame *sf);`

Call one of these to initialize a spawning function's stack_frame object before using it. The fast variant can be called if a parent of the current function has called enter_frame. It skips a test for whether Cilk is initialized on the user thread.

An implementation, which may be inlined, is

```
void __cilkrts_enter_frame_1(struct __cilkrts_stack_frame *sf)
{
    struct __cilkrts_worker *w = __cilkrts_get_tls_worker();
    if (w == 0) { /* slow path, rare */
        w = __cilkrts_bind_thread_1();
        sf->flags = CILK_FRAME_LAST | CILK_FRAME_VERSION;
    } else {
        sf->flags = CILK_FRAME_VERSION;
    }
    sf->call_parent = w->current_stack_frame;
    sf->worker = w;
    /* sf->except_data is only valid when CILK_FRAME_EXCEPTING is set */
    w->current_stack_frame = sf;
}
```

`__cilkrts_enter_frame_fast()_1` assumes that `__cilkrts_get_tls_worker()` will never return 0. An implementation, which may be inlined, is

```
void __cilkrts_enter_frame_fast_1(struct __cilkrts_stack_frame *sf)
{
    struct __cilkrts_worker *w = __cilkrts_get_tls_worker();
    sf->flags = CILK_FRAME_VERSION;
    sf->call_parent = w->current_stack_frame;
    sf->worker = w;
    /* sf->except_data is only valid when CILK_FRAME_EXCEPTING is set */
    w->current_stack_frame = sf;
}
```

Implementations of these functions to initialize ABI 0 __cilkrts_stack_frames are also available. The compiler must call the version which matches the __cilkrts_stack_frame it allocates.

10.2 struct __cilkrts_worker *__cilkrts_get_tls_worker(void); struct __cilkrts_worker *__cilkrts_get_tls_worker_fast(void);

These functions return the current thread's worker structure, or NULL if the current thread is not bound to Cilk. The fast variant may malfunction if Cilk is not yet initialized.

10.3 struct __cilkrts_worker *__cilkrts_bind_thread_1(void);

Call this function if `__cilkrts_get_tls_worker()` returns NULL. It notifies the runtime that a new user thread has entered Cilk. The function returns the user thread's new worker.

Set the CILK_FRAME_LAST bit in the flags field of the frame descriptor if __cilkrts_bind_thread_1 was called. This will remind __cilkrts_leave_frame to undo the bind operation.

Note that the runtime also exports __cilkrts_bind_thread() to support code built with compilers that generated ABI 0 code. Compilers that generate ABI 1 code should use __cilkrts_bind_thread_1().

10.4 void __cilkrts_rethrow(struct __cilkrts_stack_frame *sf);

Except on Windows, call this function after a sync if the CILK_FRAME_EXCEPTION flag is set in the frame descriptor. It will reinstate a suspended exception.

10.5 void __cilkrts_sync(struct __cilkrts_stack_frame *sf);

This function implements nontrivial sync. Call this function at a sync statement and before function exit if and only if the function is not synched, i.e. the flags field of the frame descriptor has the CILK_FRAME_UNSYNCHED bit set.

Prior to calling this interface, save the function's current state in the stack_frame. The `setjmp()` to save the state will return 1 after the sync completes. `__cilkrts_sync()` returns if the sync is successful (i.e., we can continue with the user code). On the other hand, `__cilkrts_sync()` does not return if
the sync is not successful (i.e., a spawned function has not yet returned). Eventually, after an unsuccessful sync, the last child will return and a different worker will resume via a `longjmp()`, picking up the execution from after the `setjmp()` branch test.

```
if (frame.flags & CILK_FRAME_UNSYNCHED)
{
    if (!__builtin_setjmp(frame.ctx))
        __cilkrts_sync(&frame);
    /* Function is now synched.  An asynchronous exception
       may be pending. */
}
```

10.6 void __cilkrts_detach(struct __cilkrts_stack_frame *sf);

This function implements the spawn operation by pushing its parent onto the tail end of the spawn deque. Pass the spawn helper function's frame descriptor as the argument. It is implemented as below and can be inlined.

```
void __cilkrts_detach(struct __cilkrts_stack_frame *self)
{
    struct __cilkrts_worker *w = self->worker;
    struct __cilkrts_stack_frame *parent = self->call_parent;
    struct __cilkrts_stack_frame *volatile *tail = w->tail;

    self->spawn_helper_pedigree.rank = w->pedigree.rank;
    self->spawn_helper_pedigree.next = w->pedigree.next;

    self->call_parent->parent_pedigree.rank = w->pedigree.rank;
    self->call_parent->parent_pedigree.next = w->pedigree.next;

    w->pedigree.rank = 0;
    w->pedigree.next = &sf->spawn_helper_pedigree;

    /*assert (tail < w->ltq_limit);*/
    *tail++ = parent;
    /* The stores are separated by a store fence (noop on x86)
       or the second store is a release (st8.rel on Itanium) */
    w->tail = tail;
    self->flags |= CILK_FRAME_DETACHED;
}
```

10.7 void __cilkrts_cilk_for_32(void (*body)(void *, uint32_t, uint32_t),

```
                                  void *context,
                                  uint32_t count,
                                  int grain);
    void __cilkrts_cilk_for_64(void (*body)(void *, uint64_t, uint64_t),
                                  void *context,
                                  uint64_t count,
                                  int grain);
```

These functions implement `_Cilk_for`.

The first two arguments are a closure that executes the loop body. The argument `count` is passed as the first argument to every call to `body`.

The third argument is the number of loop iterations to execute.

The last argument is the grain size, specified by the `cilk grainsize` pragma. 0 indicates that no pragma was specified, so the runtime should pick a grain size according to its own heuristic. Negative values for grain size are reserved.

The loop body should count up from its second argument (inclusive) to its third argument (exclusive). The loop body function is always called with the third argument strictly greater than the second.

The internal indices of _Cilk_for (i.e, the values passed to the second and third arguments of the body function) run up from 0 to `count-1` (inclusive).The body function is responsible for mapping the internal zero-based unit-stride index to the user-visible index. For example, if the user's code is "`cilk_for( int i=a; i<b; i+=c )...`" and the generated loop body function is "`void foo(void*, uint32_t l, uint32_t u )`", then the loop body function should execute the user's loop body with i=a+k*c for k∈{l,l+1,l+2,...,u-1}, with k in ascending serial order.

10.8 Void __cilkrts_pop_frame(struct __cilkrts_stack_frame *sf);

Pops a frame off of the chain of __cilkrts_stack_frame's rooted in __cilkrts_worker.current_stack_frame. It is implemented as below and can be inlined:

```
void __cilkrts_pop_frame(struct __cilkrts_stack_frame *sf)
{
    struct __cilkrts_worker *w = sf->worker;
    w->current_stack_frame = sf->call_parent;
    sf->call_parent = 0;
}
```

10.9 void __cilkrts_leave_frame(struct __cilkrts_stack_frame *sf);

Handles all processing for leaving a spawning function. `__cilkrts_pop_frame()` should be called before `__cilkrts_leave_frame()` to remove the frame from the list rooted in current_stack_frame in the __cilkrts_worker.

- If the frame is detached and the parent has been stolen, the frame will be suspended. `__cilkrts_leave_frame()` will not return.
- If the frame is detached and the parent has not been stolen, the detach will be undone (so the parent can no longer be stolen) and `__cilkrts_leave_frame()` will return normally.
- If CILK_FRAME_LAST is set, control will be marshaled onto the user thread which made the initial call into the Cilk runtime. The thread will be unbound from the Cilk runtime. If this is the last user thread bound to the Cilk runtime, all worker threads created by the runtime will be suspended. Execution will continue on the user thread.
- If CILK_FRAME_UNSYNCHED is set, any pending reducers or exceptions are merged.

Calling `__cilkrts_leave_frame()` can be skipped if __cilkrts_stack_frame.flags is 0.

```
10.10 void __cilkrts_hyper_create(__cilkrts_hyperobject_base *key);
      void __cilkrts_hyper_destroy(__cilkrts_hyperobject_base *key));
      void* __cilkrts_hyper_lookup(__cilkrts_hyperobject_base *key);
```

These functions are called by the reducer library to implement reducers. These are normal function calls, from the standpoint of calling conventions. However, the compiler writer should be aware that `__cilkrts_hyper_lookup()` will return the same value each time it is called with the same key until the next spawn, sync, or call to `__cilkrts_hyper_destroy`() for that key. This fact allows the compiler to lift the lookup call out of serial loops, etc., in order to avoid excessive lookup overhead. Also, it is not possible for two different keys to return the same value from lookup. Thus, if the compiler can determine that two key pointers are distinct, then it can also assume that the results of calling lookup on the key pointers are also distinct.

11 Exceptions

When an exception occurs, the compiler must ensure that `__cilkrts_pop_frame()` and `__cilkrts_leave_frame()` are called as part of the unwind operation.

The Cilk Plus runtime handles only C++ exceptions.

12 <internal/abi.h>

This is a copy of <internal/abi.h> as of 3-Aug-2011.

```
/*
 *  abi.h
 *
 *  Copyright (C) 2009-2011 Intel Corporation.  All Rights Reserved.
 *
 *  The source code contained or described herein and all documents related
 *  to the source code ("Material") are owned by Intel Corporation or its
 *  suppliers or licensors.  Title to the Material remains with Intel
 *  Corporation or its suppliers and licensors.  The Material is protected
 *  by worldwide copyright laws and treaty provisions.  No part of the
 *  Material may be used, copied, reproduced, modified, published, uploaded,
 *  posted, transmitted, distributed, or disclosed in any way without
 *  Intel's prior express written permission.
 *
 *  No license under any patent, copyright, trade secret or other
 *  intellectual property right is granted to or conferred upon you by
 *  disclosure or delivery of the Materials, either expressly, by
 *  implication, inducement, estoppel or otherwise.  Any license under such
 *  intellectual property rights must be express and approved by Intel in
 *  writing.
 *
 *********************************************************************************/

/**
 * @file abi.h
 *
 * @brief Defines the application binary interface between the compiler and
 * the Intel Cilk Plus runtime.
 */

#ifndef CILK_INTERNAL_ABI_H
#define CILK_INTERNAL_ABI_H

#include <cilk/common.h>

/**
 * Jump buffers are OS and architecture dependent
 */
#if ! defined(_MSC_VER)
/* Non-Windows - only need 5 registers for the jump buffer for both IA32 and Intel64 */
typedef void *__CILK_JUMP_BUFFER[5];

/** OS-specific implementation of setjmp */
#   define CILK_SETJMP(X) __builtin_setjmp(X)
/** OS-specific implementation of longjmp */
#   define CILK_LONGJMP(X) __builtin_longjmp(X,1)
#else
/* Windows - things are a little more complicated */
#   if defined(_M_X64)
/* Intel64 - Use an OS-defined jump buffer */
#       include <setjmp.h>
typedef jmp_buf __CILK_JUMP_BUFFER;

#       define CILK_SETJMP(X) setjmp(X)
#       define CILK_LONGJMP(X) longjmp(X, 1)
#   elif defined(_M_IX86)
/**
 * Windows x86 - Use a simplified version of the Windows jump buffer for x86
 * setjmp is provided by __cilkrts_setjmp which passes jump buffer in EAX and
 * destination in EDX longjmp is provided by an internal routine which uses
 * this structure
 */
typedef struct
{
    unsigned long Ebp;
```

```
    unsigned long Ebx;
    unsigned long Edi;
    unsigned long Esi;
    unsigned long Esp;
    unsigned long Eip;
    unsigned long Registration;
    unsigned long TryLevel;
} __CILK_JUMP_BUFFER;

#    else
#    error Unexpected architecture - Need to define __CILK_JUMP_BUFFER
#    endif  /* _M_X64 */

#endif  /* defined(_MSC_VER) */

/* struct tags */
typedef struct __cilkrts_worker       __cilkrts_worker;
typedef struct __cilkrts_worker*      __cilkrts_worker_ptr;
typedef struct __cilkrts_stack_frame __cilkrts_stack_frame;

// Forwarded declarations
typedef struct global_state_t        global_state_t;
typedef struct local_state           local_state;
typedef struct cilkred_map           cilkred_map;
typedef struct __cilkrts_worker_sysdep_state
                                     __cilkrts_worker_sysdep_state;

/**
 * Version number assigned to frames.  When considering this value in code, use
 * CILK_FRAME_VERSION which has this value appropriately shifted.
 */
#define __CILKRTS_ABI_VERSION 1

/** Pedigree information kept in the worker and stack frame */
typedef struct __cilkrts_pedigree
{
    /** Rank at start of spawn helper. Saved rank for spawning functions */
    uint64_t rank;

    /** Link to next in chain */
    struct __cilkrts_pedigree *next;
} __cilkrts_pedigree;

/**
 * The worker struct contains per-worker information that needs to be
 * visible to the compiler, or rooted here.
 *
 * For 32-bit Windows we need to be aligning the structures on 4-byte
 * boundaries to match where ICL is allocating the birthrank and rank
 * in the __cilkrts_stack_frame.  It's 4-byte aligned instead of 8-byte
 * aligned.  This is OK because the compiler is dealing with the 64-bit
 * quantities as two 32-bit values.  So change the packing to be on
 * 4-byte boundaries.
 */
#if defined(_MSC_VER) && defined(_M_IX86)
#pragma pack(push, 4)
#endif

struct __cilkrts_worker {
    /**
     * T, H, and E pointers in the THE protocol See "The implementation of
     * the Cilk-5 multithreaded language", PLDI 1998:
     * http://portal.acm.org/citation.cfm?doid=277652.277725
     */
    __cilkrts_stack_frame *volatile *volatile tail;
    __cilkrts_stack_frame *volatile *volatile head;  /**< @copydoc tail */
    __cilkrts_stack_frame *volatile *volatile exc;   /**< @copydoc tail */

    /**
     * Addition to the THE protocol to allow us to protect some set of
     * entries in the tail queue from stealing.  Normally, this is set
```

```
     * beyond the end of the task queue, indicating that all entries are
     * available for stealing.  During exception handling, protected_tail
     * may be set to the first entry in the task queue, indicating that
     * stealing is not allowed.
     */
    __cilkrts_stack_frame *volatile *volatile protected_tail;

    /** Limit of the Lazy Task Queue, to detect queue overflow */
    __cilkrts_stack_frame *volatile *ltq_limit;

    /** Worker id */
    int32_t self;

    /** Global state of the runtime system, opaque to the client */
    global_state_t *g;

    /**
     * Additional per-worker state of the runtime system that we want
     * to maintain hidden from the client
     */
    local_state *l;

    /** map from reducer names to reducer values */
    cilkred_map *reducer_map;

    /** A slot that points to the currently executing Cilk frame. */
    __cilkrts_stack_frame *current_stack_frame;

    /** Saved protected tail.  Set to NULL by runtime.  No longer used. */
    __cilkrts_stack_frame *volatile *volatile saved_protected_tail;

    /** System-dependent part of the worker state */
    __cilkrts_worker_sysdep_state *sysdep;

#if __CILKRTS_ABI_VERSION >= 1
    /**
     * Per-worker pedigree information used to support scheduling-independent
     * pseudo-random numbers.
     */
    __cilkrts_pedigree   pedigree;
#endif  /* __CILKRTS_ABI_VERSION >= 1 */
};


/**
 * Every spawning function has a frame descriptor.  A spawning function
 * is a function that spawns or detaches.  Only spawning functions
 * are visible to the Cilk runtime.
 */
struct __cilkrts_stack_frame
{
    /**
     * flags is an integer with values defined below.  Client code
     * initializes flags to CILK_FRAME_VERSION before the first Cilk
     * operation.
     *
     * The low 24-bits of the 'flags' field are the flags, proper.  The high
     * 8-bits are the version number.
     *
     * IMPORTANT: bits in this word are set and read by the PARENT ONLY,
     * not by a spawned child.  In particular, the STOLEN and UNSYNCHED
     * bits are set on a steal and are read before a sync.  Since there
     * is no synchronization (locking) on this word, any attempt to set
     * or read these bits asynchronously in a child would result in a race.
     */
    uint32_t flags;

    /** Not currently used.  Not initialized by Intel compiler. */
    int32_t size;

    /**
```

```
     * call_parent points to the __cilkrts_stack_frame of the closest
     * ancestor spawning function, including spawn helpers, of this frame.
     * It forms a linked list ending at the first stolen frame.
     */
    __cilkrts_stack_frame *call_parent;

    /**
     * The client copies the worker from TLS here when initializing
     * the structure.  The runtime ensures that the field always points
     * to the __cilrts_worker which currently "owns" the frame.
     */
    __cilkrts_worker *worker;

    /**
     * Unix: Pending exception after sync.  The sync continuation
     * must call __cilkrts_rethrow to handle the pending exception.
     *
     * Windows: the handler that _would_ have been registered if our
     * handler were not there.  We maintain this for unwinding purposes.
     * Win32: the value of this field is only defined in spawn helper
     * functions
     *
     * Win64: except_data must be filled in  for all functions with a
     * __cilkrts_stack_frame
     */
    void *except_data;

    /**
     * Before every spawn and nontrivial sync the client function
     * saves its continuation here.
     */
    __CILK_JUMP_BUFFER ctx;

#if __CILKRTS_ABI_VERSION >= 1
    /**
     * Architecture-specific floating point state.  mxcsr and fpcsr should be
     * set when CILK_SETJMP is called in client code.  Note that the Win64
     * jmpbuf for the Intel64 architecture already contains this information
     * so there is no need to use these fields on that OS/architecture.
     */
    uint32_t mxcsr;
    uint16_t fpcsr;         /**< @copydoc mxcsr */


    /**
     * reserved is not used at this time.  Client code should initialize it
     * to 0 before the first Cilk operation
     */
    uint16_t reserved;

    /**
     * Pedigree information to support scheduling-independent pseudo-random
     * numbers.  There are two views of this information.  The copy in a
     * spawning function is used to stack the rank and communicate to the
     * runtime on a steal or continuation.  The copy in a spawn helper is
     * immutable once the function is detached and is a node in the pedigree.
     * The union is used to make clear which view we're using.
     *
     * In the detach sequence Client code should:
     *    - copy the worker pedigree into the spawn helper's pedigree
     *    - copy the worker pedigree into the call parent's pedigree
     *    - set the worker's rank to 0
     *    - set the worker's pedigree.next to the spawn helper's pedigree
     */
    union
    {
        __cilkrts_pedigree spawn_helper_pedigree;/* Used in spawn helpers */
        __cilkrts_pedigree parent_pedigree;      /* Used in spawning funcs */
    };
#endif  /* __CILKRTS_ABI_VERSION >= 1 */
};
```

```
/*
 * Restore previous structure packing for 32-bit Windows
 */
#if defined(_MSC_VER) && defined(_M_IX86)
#pragma pack(pop)
#endif

/* Values of the flags bitfield */
/** CILK_FRAME_STOLEN is set if the frame has ever been stolen. */
#define CILK_FRAME_STOLEN    0x01

/**
 * CILK_FRAME_UNSYNCHED is set if the frame has been stolen and
 * is has not yet executed _Cilk_sync.  It is technically a misnomer in that a
 * frame can have this flag set even if all children have returned.
 */
#define CILK_FRAME_UNSYNCHED 0x02

/**
 * Is this frame detached (spawned)?  If so the runtime needs
 * to undo-detach in the slow path epilogue.
 */
#define CILK_FRAME_DETACHED  0x04

/**
 * CILK_FRAME_EXCEPTION_PROBED is set if the frame has been probed in the
 * exception handler first pass
 */
#define CILK_FRAME_EXCEPTION_PROBED 0x08

/** Is this frame receiving an exception after sync? */
#define CILK_FRAME_EXCEPTING 0x10

/** Is this the last (oldest) Cilk frame? */
#define CILK_FRAME_LAST            0x80

/**
 * Is this frame in the epilogue, or more generally after the last
 * sync when it can no longer do any Cilk operations?
 */
#define CILK_FRAME_EXITING   0x0100

/** Is this frame suspended? (used for debugging) */
#define CILK_FRAME_SUSPENDED 0x8000

/** Used by Windows exception handling to indicate that __cilkrts_leave_frame should do nothing
*/
#define CILK_FRAME_UNWINDING 0x10000

/*
 * The low 24-bits of the 'flags' field are the flags, proper.  The high 8-bits
 * are the version number.
 */

/** ABI version left shifted to the high byte */
#define CILK_FRAME_VERSION (__CILKRTS_ABI_VERSION << 24)

/** Mask for the flags field to isolate the version bits */
#define CILK_FRAME_VERSION_MASK  0xFF000000

/** Mask for the flags field to isolate the flag bits */
#define CILK_FRAME_FLAGS_MASK    0x00FFFFFF

/** Convenience macro to provide access the version portion of the flags field */
#define CILK_FRAME_VERSION_VALUE(_flags) (((_flags) & CILK_FRAME_VERSION_MASK) >> 24)

/** Any undefined bits are reserved and must be zero ("MBZ" = "Must Be Zero") */
#define CILK_FRAME_MBZ  (~ (CILK_FRAME_STOLEN | \
                            CILK_FRAME_UNSYNCHED | \
                            CILK_FRAME_DETACHED | \
```

```
                            CILK_FRAME_EXCEPTION_PROBED | \
                            CILK_FRAME_EXCEPTING | \
                            CILK_FRAME_LAST | \
                            CILK_FRAME_EXITING | \
                            CILK_FRAME_SUSPENDED | \
                            CILK_FRAME_UNWINDING | \
                            CILK_FRAME_VERSION_MASK))

__CILKRTS_BEGIN_EXTERN_C

/**
 * Call __cilkrts_enter_frame to initialize an ABI 0 frame descriptor.
 * Initialize the frame descriptor before spawn or detach.  A function that
 * conditionally does Cilk operations need not initialize the frame descriptor
 * in a code path that never uses it.
 *
 * @param sf The __cilkrts_stack_frame that is to be initialized.
 */
CILK_ABI(void) __cilkrts_enter_frame(__cilkrts_stack_frame* sf);

/**
 * Call __cilkrts_enter_frame to initialize an ABI 1 frame descriptor.
 * Initialize the frame descriptor before spawn or detach.  A function that
 * conditionally does Cilk operations need not initialize the frame descriptor
 * in a code path that never uses it.
 *
 * @param sf The __cilkrts_stack_frame that is to be initialized.
 */
CILK_ABI(void) __cilkrts_enter_frame_1(__cilkrts_stack_frame* sf);

/**
 * __cilkrts_enter_frame_fast is the same as __cilkrts_enter_frame, except it
 * assumes that the thread has already been bound to a worker.
 *
 * @param sf The __cilkrts_stack_frame that is to be initialized.
 */
CILK_ABI(void) __cilkrts_enter_frame_fast(__cilkrts_stack_frame *sf);

/**
 * __cilkrts_enter_frame_fast_1 is the same as __cilkrts_enter_frame_1,
 * except it assumes that the thread has already been bound to a worker.
 *
 * @param sf The __cilkrts_stack_frame that is to be initialized.
 */
CILK_ABI(void) __cilkrts_enter_frame_fast_1(__cilkrts_stack_frame *sf);

/**
 * Call leave_frame before leaving a frame, after sync.  This function
 * returns except in a spawn wrapper where the parent has been stolen.
 *
 * @param sf The __cilkrts_stack_frame that is to be left.
 */
CILK_ABI(void) __cilkrts_leave_frame(__cilkrts_stack_frame *sf);

/**
 * Wait for any spawned children of this function to complete before
 * continuing.  This function will only return when the join counter
 * has gone to 0.  Other workers will re-enter the scheduling loop to
 * attempt to steal additional work.
 *
 * @param sf The __cilkrts_stack_frame that is to be synched.
 */
CILK_ABI(void) __cilkrts_sync(__cilkrts_stack_frame *sf);

/**
 * Called when an exception is escaping a spawn * wrapper.
 * The stack frame's except_data field is the C++ runtime
 * exception object.  If NULL (temporary workaround) the
 * currently caught exception should be rethrown.  If this
 * function returns normal exit functions must be called;
 * undo-detach will have been done.
```

```
 *
 * @param sf The __cilkrts_stack_frame for the function that
 * is raising an exception.
 */
CILK_ABI_THROWS(void)
    __cilkrts_return_exception(__cilkrts_stack_frame *sf);

/**
 * Called to re-raise an exception.
 *
 * @param sf The __cilkrts_stack_frame for the function that
 * is raising an exception.
 */
CILK_ABI_THROWS(void) __cilkrts_rethrow(__cilkrts_stack_frame *sf);

/**
 * Called at the beginning of a spawning function to get the worker
 * that this function is running on.  This worker will be used to
 * initialize the __cilkrts_stack_frame.
 *
 * @return The __cilkrts_worker that the function is running on.
 * @return NULL if this thread is not yet bound to a worker.
 */
CILK_ABI(__cilkrts_worker_ptr) __cilkrts_get_tls_worker(void);

/**
 * Similar to __cilkrts_get_tls_worker, but assumes that TLS has been
 * initialized.
 *
 * @return The __cilkrts_worker that the function is running on.
 * @return NULL if this thread is not yet bound to a worker.
 */
CILK_ABI(__cilkrts_worker_ptr) __cilkrts_get_tls_worker_fast(void);

/**
 * Binds a thread to the runtime by associating a __cilkrts_worker with
 * it.  Called if __cilkrts_get_tls_worker returns NULL.  This function will
 * initialize the runtime the first time it is called.
 *
 * This function is versioned by the ABI version number.  The runtime
 * will export all previous versions.  This prevents using an application
 * built with a newer compiler against an old runtime.
 *
 * @return The __cilkrts_worker bound to the thread the function is running
 * on.
 */
CILK_ABI(__cilkrts_worker_ptr) __cilkrts_bind_thread_1(void);

typedef uint32_t cilk32_t;  /**< 32-bit unsigned type for cilk_for loop indicies */

typedef uint64_t cilk64_t;  /**< 64-bit unsigned type for cilk_for loop indicies */

/**
 * Signature for the lambda function generated for the body of a cilk_for loop
 * which uses 32-bit indicies
 */
typedef void (*__cilk_abi_f32_t)(void *data, cilk32_t low, cilk32_t high);

/**
 * Signature for the lambda function generated for the body of a cilk_for lop
 * which uses 64-bit indicies
 */
typedef void (*__cilk_abi_f64_t)(void *data, cilk64_t low, cilk64_t high);

/**
 * @brief cilk_for implementation for 32-bit indexes.
 *
 * @param body The lambda function for the body of the cilk_for.  The lambda
 * function will be called to execute each grain of work.
 * @param data Data passed by the compiler into the lambda function.  Provides
 * access to data outside the cilk_for body.
```

```
 * @param count Number of steps in the loop.
 * @param grain This parameter allows the compiler to pass a value from a
 * \#pragam(grainsize) statement to allow the user to control the grainsize.  If
 * there isn't a \#pragma(grainsize) immediately preceeding cilk_for loop, Pass
 * 0 to specify that the runtime should calculate the grainsize using its own
 * hueristicts.
 */
CILK_ABI_THROWS(void) __cilkrts_cilk_for_32(__cilk_abi_f32_t body,
                                            void *data,
                                            cilk32_t count,
                                            int grain);

/**
 * @brief cilk_for implementation for 64-bit indexes.
 *
 * @copydetails __cilkrts_cilk_for_32
 */
CILK_ABI_THROWS(void) __cilkrts_cilk_for_64(__cilk_abi_f64_t body,
                                            void *data,
                                            cilk64_t count,
                                            int grain);

__CILKRTS_END_EXTERN_C

#endif /* include guard */
```

13 <internal/fake.h>

This is a copy of <internal/fake.h> as of 19-Oct-2010.

```
#ifdef _WIN32
/* define macros for synching functions before allowing them to propagate. */
#define CILK_EXCEPT_BEGIN                          \
    if (0 == CILK_SETJMP(sf.except_ctx)) {
#define CILK_EXCEPT_END                                    \
    } else {                                               \
        assert((sf.flags & (CILK_FRAME_UNSYNCHED|CILK_FRAME_EXCEPTING)) ==
CILK_FRAME_EXCEPTING); \
        __cilkrts_rethrow(&sf);                            \
        exit(0);                                           \
    }
#endif

// Define macros for inlining
#ifdef _WIN32
#define INLINE __inline
#else
#define INLINE inline
#endif

#define PRESPAWN(STATE) __builtin_expect(CILK_SETJMP((STATE).ctx) == 0, 1)

/* Helper macro to implement sync. */
#define SYNC(SF) \
  if (__builtin_expect(((SF).flags & CILK_FRAME_UNSYNCHED), 0)) { \
    (SF).worker->pedigree.rank++;                                  \
    if (!CILK_SETJMP((SF).ctx))                                    \
    {                                                \
      __notify_intrinsic((char*)"cilk_leave", &(SF));     \
      __cilkrts_sync(&(SF));                             \
    }                                                \
    else if ((SF).flags & CILK_FRAME_EXCEPTING)         \
      __cilkrts_rethrow(&(SF));                                    \
  } else (void)0

/* Returns nonzero if the frame is not synched. */
INLINE int __cilkrts_unsynched(struct __cilkrts_stack_frame *sf)
{
  return sf->flags & CILK_FRAME_UNSYNCHED;
}

/* Returns nonzero if the frame has been stolen. */
INLINE int __cilkrts_stolen(struct __cilkrts_stack_frame *sf)
{
  return sf->flags & CILK_FRAME_STOLEN;
}

/* Pop the frame off the active stack.  This is separate from
   __cilkrts_leave_frame so it can be inlined. */
/* extern void __cilkrts_pop_frame(struct __cilkrts_stack_frame *) */
INLINE void __cilkrts_pop_frame(struct __cilkrts_stack_frame *sf)
{
    struct __cilkrts_worker *w = sf->worker;
    w->current_stack_frame = sf->call_parent;
    sf->call_parent = 0;
}
```

```
/* Call this in a spawn wrapper once the parent may be safely stolen. */
INLINE void __cilkrts_detach(struct __cilkrts_stack_frame *self)
{
    struct __cilkrts_worker *w = self->worker;
    struct __cilkrts_stack_frame *parent = self->call_parent;
    struct __cilkrts_stack_frame *volatile *tail = w->tail;

    self->spawn_helper_pedigree.rank = w->pedigree.rank;
    self->spawn_helper_pedigree.next = w->pedigree.next;

    self->call_parent->parent_pedigree.rank = w->pedigree.rank;
    self->call_parent->parent_pedigree.next = w->pedigree.next;

    w->pedigree.rank = 0;
    w->pedigree.next = &sf->spawn_helper_pedigree;

    /*assert (tail < w->ltq_limit);*/

    *tail++ = parent;
    /* The stores are separated by a store fence (noop on x86)
       or the second store is a release (st8.rel on Itanium) */
    w->tail = tail;
    __notify_intrinsic((char*)"cilk_detach", self);
    self->flags |= CILK_FRAME_DETACHED;
}

#ifdef _WIN32
/* define boilerplate macros for functions that spawn.  C++ uses an object with
   a destructor, and C uses an explicit __try block. */
# ifdef __cplusplus
class cilk_boilerplate_t : public __cilkrts_stack_frame {
public:
    // Fast enter
    cilk_boilerplate_t() {
        printf("entering frame 0x%p\n", sf_);
        __cilkrts_enter_frame_fast(this);
        /* this + 1 is the start of the actual frame on the stack */
        __notify_intrinsic((char*)"cilk_enter", this + 1);
    }
    // Normal enter
    cilk_boilerplate_t(int) {
        printf("entering frame 0x%p\n", sf_);
        __cilkrts_enter_frame(this);
        /* this + 1 is the start of the actual frame on the stack */
        __notify_intrinsic((char*)"cilk_enter", this + 1);
    }
    ~cilk_boilerplate_t () {
        printf("popping frame 0x%p\n", sf_);
        __cilkrts_pop_frame(sf_);
        __notify_intrinsic((char*)"cilk_leave", this + 1);
        if (__builtin_expect(flags, 0)) {
            printf("leaving frame 0x%p\n", sf_);
            /* this + 1 is the start of the actual frame on the stack */
            __cilkrts_leave_frame(sf_);
        }
    }
private:
    struct __cilkrts_stack_frame *sf_;
};
#   define CILK_BOILERPLATE_BEGIN(sf) cilk_boilerplate_t sf(0); do
#   define CILK_BOILERPLATE_BEGIN_FAST(sf) cilk_boilerplate_t sf; do
#   define CILK_BOILERPLATE_END(sf) while (0)
```

```
# else /* else C on Windows */
#   define CILK_BOILERPLATE_BEGIN(sf)                    \
    struct __cilkrts_stack_frame sf;                     \
    __try { printf("entering frame 0x%p\n", &sf);        \
    __cilkrts_enter_frame(&sf);                          \
    do
#   define CILK_BOILERPLATE_BEGIN_FAST(sf)               \
    struct __cilkrts_stack_frame sf;                     \
    __try { printf("entering frame 0x%p\n", &sf);        \
    __cilkrts_enter_frame_fast(&sf);                     \
    __notify_intrinsic((char*)"cilk_enter", &sf + 1);    \
    do
#   define CILK_BOILERPLATE_END(sf)                      \
    while (0);                                           \
    } __finally {                                        \
        printf("popping frame 0x%p\n", &sf);             \
        __cilkrts_pop_frame(&sf);                        \
        __notify_intrinsic((char*)"cilk_leave", &sf+1); \
        if (sf.flags) __cilkrts_leave_frame(&sf);        \
    } ((void) 0)
# endif /* C on Windows*/
#elif defined __cplusplus /* unix style */
/* TBD -- I think Unix should be like Windows for C++ */
namespace cilk
{
    struct stack_frame : public __cilkrts_stack_frame
    {
        stack_frame()
        {
            __cilkrts_enter_frame(this);
        }
        ~stack_frame()
        {
            /* There used to be a SYNC here, but that is wrong
               when the destructor is not inlined.  SYNC must
               return to the stack pointer of the first spawn.
               Anything under the original stack will be discarded. */
            __cilkrts_pop_frame(this);
            if (__builtin_expect(flags, 0))
                __cilkrts_leave_frame(this);
        }
    };
}

#endif
```