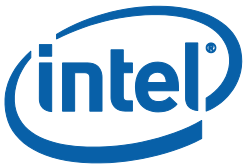

Intel® Cilk™ Plus Application Binary Interface Specification

Part 2 of the Intel® Cilk™ Plus Language Specification version 0.9

Document Number: 324512-001US

Revision: 0.9

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

Disclaimer and Legal Information

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

2 Description

This document is part of the Intel® Cilk™ Plus Language Specification version 0.9. 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 324396-001US.
- Part 2. The Intel® Cilk™ Plus Application Binary Interface, document number 324512-001US.

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 12.0, also known as Composer 2011 and Composer XE 2011.

Note: There are already changes to the ABI being discussed for a future version of the compiler. Specifically, we're considering changes to the __cilkrts_stack_frame for counting spawns and future extensions. Other changes may be considered.

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. When/if the ABI changes incompatibly, the versioned names will be changed. 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 not synched, 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 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 whatever stack they are on.

A spawn statement is extracted into a separate function called a spawn helper function. The spawn helper 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 this isn't a void function
- Calls the destructors for any computed temporaries
- Pops the frame and calls `__cilkrts_leave_frame()` to exit

```
...
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.

5 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_windows.h and cilk_api_linux.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()`.

6 __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 a bitfield with values defined below.  Client code
       initializes flags to 0 before the first Cilk operation. */
    unsigned int flags;
    /* Not currently used.  Not initialized by Intel compiler.  */
    int 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. */
    struct __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 __cilkrts_worker
       which currently "owns" the frame. */
    struct __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. */
#ifdef _WIN32
    jmp_buf ctx;
#else
    void *ctx[5];
#endif
};
```

Values of the flags bitfield

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 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.

7 __cilkrts_worker

The worker structure holds thread local state.

```
struct __cilkrts_worker
{
    /* T, H, and E pointers in the THE protocol (see the PLDI '98
     paper). */
    struct __cilkrts_stack_frame *volatile *volatile tail;
    struct __cilkrts_stack_frame *volatile *volatile head;
    struct __cilkrts_stack_frame *volatile *volatile exc;
    /* 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. */
    struct __cilkrts_stack_frame *volatile *volatile protected_tail;

    /* limit of the lazy task queue, to detect queue overflow */
    struct __cilkrts_stack_frame *volatile *ltq_limit;

    /* worker id */
    int self;

    /* global state of the runtime system, opaque to the client */
    struct __cilkrts_global_state *g;

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

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

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

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

    /* system-dependent part of the worker state */
    struct __cilkrts_worker_sysdep_state *sysdep;
};
```

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

8 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.

9 Cilk runtime calls

9.1 `void __cilkrts_enter_frame(struct __cilkrts_stack_frame *);`
`void __cilkrts_enter_frame_fast(struct __cilkrts_stack_frame *);`

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(struct __cilkrts_stack_frame *sf)
{
    struct __cilkrts_worker *w = __cilkrts_get_tls_worker();
    if (w == 0) { /* slow path, rare */
        w = __cilkrts_bind_thread();
        sf->flags = CILK_FRAME_LAST;
    } else {
        sf->flags = 0;
    }
    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()` assumes that `__cilkrts_get_tls_worker()` will never return 0. An implementation, which may be inlined, is

```
void __cilkrts_enter_frame_fast(struct __cilkrts_stack_frame *sf)
{
    struct __cilkrts_worker *w = __cilkrts_get_tls_worker();
    sf->flags = 0;
    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;
}
```

9.2 `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.

9.3 `struct __cilkrts_worker *__cilkrts_bind_thread(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 was called. This will remind __cilkrts_leave_frame to undo the bind operation.

9.4 `void __cilkrts_rethrow(struct __cilkrts_stack_frame *);`

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.

9.5 `void __cilkrts_sync(struct __cilkrts_stack_frame *);`

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. */
}
```

9.6 void __cilkrts_detach(struct __cilkrts_stack_frame *);

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;
    /*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;
}
```

9.7 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 will always execute at least one iteration of the loop, i.e. the third argument is 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) runs up from 0 to `count-1` (inclusive). If the user-visible stride is not positive 1, then the body function must multiply its second and third arguments by the actual stride. If the user-visible lower bound of the loop is not integer zero, the body function must offset the loop boundaries by the lower-bound value.

9.8 Void __cilkrts_pop_frame(struct __cilkrts_stack_frame *);

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;
}
```

9.9 void __cilkrts_leave_frame(struct __cilkrts_stack_frame *);

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.

9.10 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.

10 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.

11 <internal/abi.h>

This is a copy of <internal/abi.h> as of 20-Oct-2010.

```
#ifndef CILK_INTERNAL_ABI_H
#define CILK_INTERNAL_ABI_H

#ifdef _WIN32
#include <setjmp.h>
#endif

#ifdef __cplusplus
#define C "C"
#else
#define C
#endif

#ifndef CILK_ABI
#if defined _WIN32
#define CILK_ABI(WHAT) \
extern C __declspec(dllimport) void __cilkrts_ ## WHAT (struct __cilkrts_stack_frame
*)
#define CILK_ABI0(TYPE,WHAT) \
extern C __declspec(dllimport) TYPE __cilkrts_ ## WHAT (void)
#define CILK_ABI4(WHAT,T1,T2,T3,T4) \
extern C __declspec(dllimport) void __cilkrts_ ## WHAT (T1,T2,T3,T4)
#else
#define CILK_ABI(WHAT) \
extern C void __attribute__((nonnull,visibility("default"))) __cilkrts_ ## WHAT
(struct __cilkrts_stack_frame *)
#define CILK_ABI0(TYPE,WHAT) \
extern C TYPE __attribute__((visibility("default"))) __cilkrts_ ## WHAT (void)
#define CILK_ABI4(WHAT,T1,T2,T3,T4) \
extern C __attribute__((visibility("default"))) void __cilkrts_ ## WHAT (T1,T2,T3,T4)
#endif
#endif

/* struct tags */
struct global_state_t;
struct __cilkrts_local_state;
struct __cilkrts_worker;
struct __cilkrts_stack_frame;

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
     */
    struct __cilkrts_stack_frame *volatile *volatile tail;
    struct __cilkrts_stack_frame *volatile *volatile head;
    struct __cilkrts_stack_frame *volatile *volatile exc;
    /* 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. */
    struct __cilkrts_stack_frame *volatile *volatile protected_tail;

    /* limit of the lazy task queue, to detect queue overflow */
    struct __cilkrts_stack_frame *volatile *ltq_limit;
```

```
    /* worker id */
    int self;

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

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

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

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

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

    /* system-dependent part of the worker state */
    struct __cilkrts_worker_sysdep_state *sysdep;
};


/* 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 a bitfield with values defined below.  Client code
       initializes flags to 0 before the first Cilk operation. */
    unsigned int flags;
    /* Not currently used.  Not initialized by Intel compiler. */
    int 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. */
    struct __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. */
    struct __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. */
#ifdef _WIN32
    jmp_buf ctx;
#define CILK_SETJMP(X) setjmp(X)
#define CILK_LONGJMP(X) longjmp(X, 1)
#else
    void *ctx[5];
#define CILK_SETJMP(X) __builtin_setjmp(X)
#define CILK_LONGJMP(X) __builtin_longjmp(X,1)
```

```
#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

#define CILK_FRAME_UNWINDING 0x10000

/* 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))

/* Call enter_frame to initialize a 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.  */
CILK_ABI(enter_frame);
CILK_ABI(enter_frame_fast);

/* Call leave_frame before leaving a frame, after sync.  This function
   returns except in a spawn wrapper where the parent has been stolen. */
CILK_ABI(leave_frame);
CILK_ABI(sync);

/* Call this 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. */
```

```
CILK_ABI(return_exception);

CILK_ABI(rethrow);

#ifdef _WIN32
typedef unsigned int cilk32_t;
typedef unsigned __int64 cilk64_t;
#else
#include <stdint.h>
typedef uint32_t cilk32_t;
typedef uint64_t cilk64_t;
#endif

typedef void (*__cilk_abi_f32_t)(void *, cilk32_t, cilk32_t);
typedef void (*__cilk_abi_f64_t)(void *, cilk64_t, cilk64_t);

CILK_ABI4(cilk_for_32,__cilk_abi_f32_t,void *,cilk32_t,int);
CILK_ABI4(cilk_for_64,__cilk_abi_f64_t,void *,cilk64_t,int);

// If rts.h is already included, don't make these definitions.  They'll
// only conflict with the definitions in rts.h

#ifndef __CILK_RTS_H__
#if defined _WIN64
extern C __declspec(dllimport) struct __cilkrts_worker *__cilkrts_bind_thread(void);
extern C __declspec(dllimport) struct __cilkrts_worker
*__cilkrts_get_tls_worker(void);
extern C __declspec(dllimport) struct __cilkrts_worker
*__cilkrts_get_tls_worker_fast(void);
#elif defined _WIN32
extern C __declspec(dllimport) struct __cilkrts_worker *__cilkrts_bind_thread(void);
extern C __declspec(dllimport) struct __cilkrts_worker
*__cilkrts_get_tls_worker(void);
extern C __declspec(dllimport) struct __cilkrts_worker
*__cilkrts_get_tls_worker_fast(void);
#else
extern C struct __cilkrts_worker *__cilkrts_bind_thread(void);
extern C struct __cilkrts_worker *__cilkrts_get_tls_worker(void);
extern C struct __cilkrts_worker *__cilkrts_get_tls_worker_fast(void);
#endif
#endif // __CILK_RTS_H__

#undef C

#endif /* include guard */
```

12 <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)) { \
    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;
    /*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
```