Lithe API Reference

Release 1.0

Kevin Klues, Andrew Waterman, Krste Asanović

February 07, 2013

CONTENTS

CHAPTER

ONE

WELCOME TO LITHE'S API REFERENCE

For the software industry to take advantage of multicore processors, we must allow programmers to arbitrarily compose parallel libraries without sacrificing performance. We believe that high-level task or thread abstractions built around a common global scheduler cannot provide this composition effectively. Instead, the operating system should expose unvirtualized processing resources that can be shared cooperatively between parallel libraries within an application.

Lithe is a system that standardizes and facilitates the exchange of these unvirtualized processing resources between libraries.

CHAPTER

TWO

API REFERENCE

2.1 Lithe Runtime

```
int lithe_sched_enter(lithe_sched_t *child);
int lithe_sched_exit();
lithe_sched_t *lithe_sched_current();
int lithe_hart_request(int k);
void lithe_hart_grant(lithe_sched_t *child, void (*unlock_func) (void *), void *lock);
void lithe_hart_yield();
void lithe_context_init(lithe_context_t *context, void (*func) (void *), void *arg);
void lithe_context_reinit(lithe_context_t *context, void (*func) (void *), void *arg);
void lithe_context_cleanup(lithe_context_t *context);
lithe_context_t *lithe_context_self();
int lithe_context_run(lithe_context_t *context);
int lithe_context_block(void (*func) (lithe_context_t *, void *), void *arg);
int lithe_context_unblock(lithe_context_t *context);
void lithe_context_yield();
void lithe_context_set_cls(lithe_context_t *context, void *cls);
void *lithe_context_get_cls(lithe_context_t *context);
```

int **lithe_sched_enter** (lithe_sched_t **child*)

Passes control to a new child scheduler. The 'funcs' field of the scheduler must already be set up properly or the call will fail. It hands the current hart over to this scheduler and then returns. To exit the child, a subsequent call to lithe_sched_exit() is needed. Only at this point will control be passed back to the parent scheduler. Returns -1 if there is an error and sets errno appropriately.

int **lithe_sched_exit** ()

Exits the current scheduler, returning control to its parent. Must be paired with a previous call to lithe_sched_enter(). Returns -1 if there is an error and sets errno appropriately.

lithe_sched_t ***lithe_sched_current** ()

Return a pointer to the current scheduler. I.e. the pointer passed in when the scheduler was entered.

int **lithe_hart_request** (int *k*)

Request a specified number of harts from the parent. Note that the parent is free to make a request using the calling hart to their own parent if necessary. These harts will trickle in over time as they are granted to the requesting scheduler. Returns 0 on success and -1 on error.

void **lithe_hart_grant** (lithe_sched_t **child*, void (*unlock_func) (void **)*, void **lock*)

Grant the current hart to another scheduler. Triggered by a previous call to lithe_hart_request() by a child scheduler. This function never returns. The 'unlock_func()' and its corresponding 'lock' parameter are passed in by a parent scheduler so that the lithe runtime can synchronize references to the child scheduler in the rare case that the child scheduler may currently be in the process of exiting. For example, the unlock function passed in should be the same one used to add/remove the child scheduler from a list in the parent scheduler.

void **lithe_hart_yield**()
Yield current hart to parent scheduler. This function should never return.

void **lithe_context_init**(lithe_context_t *context, void (*func) (void *), void *arg)
Initialize the proper lithe internal state for an existing context. The context parameter MUST already contain a valid stack pointer and stack size. This function MUST be paired with a call to lithe_context_cleanup() in order to properly cleanup the lithe internal state once the context is no longer usable. To reinitialize a context with a new start function without calling lithe_context_cleanup() first, use lithe_context_reinit() instead. Once the context is restarted it will run from the entry point specified.

void **lithe_context_reinit**(lithe_context_t *context, void (*func) (void *), void *arg)
Used to reinitialize the lithe internal state for a context already initialized via lithe_context_cleanup(). Normally each call to lithe_context_init() must be paired with a call to lithe_context_cleanup() before the context can be reused for anything. The lithe_context_reinit() function allows you to reinitialize this context with a new start function any number of times before pairing it with lithe_context_cleanup(). Once the context is restarted it will run from the entry point specified.

void **lithe_context_cleanup**(lithe_context_t *context)
Cleanup an existing context. This context should NOT currently be running on any hart, though this is not enforced by the lithe runtime.

lithe_context_t ***lithe_context_self**()
Returns a pointer to the currently executing context.

int **lithe_context_run**(lithe_context_t *context)
Run the specified context. MUST only be run from hart context. Upon completion, the context is yielded, and must be either retasked for other use via lithe_context_reinit() or cleaned up via a call to lithe_context_cleanup(). This function never returns on success and returns -1 on error and sets errno appropriately.

int **lithe_context_block**(void (*func) (lithe_context_t *, void *), void *arg)
Invoke the specified function with the current context and block that context. This function is useful for blocking a context and managing when to unblock it by some component other than the scheduler associated with this context. The scheduler will recieve a callback notifying it that this context has blocked and should not be run. The scheduler will receive another callback later to notify it when this context has unblocked and can once again be resumed. Returns 0 on success (after the context has been resumed) and -1 on error and sets errno appropriately.

int **lithe_context_unblock**(lithe_context_t *context)
Notifies the current scheduler that the specified context is now resumable. This chould only be called on contexts previously blocked via a call to lithe_context_block(). Returns 0 on success and -1 on error and sets errno appropriately.

void **lithe_context_yield**()
Cooperatively yield the current context to the current scheduler. The scheduler receives a callback notifiying it that the context has yielded and can decide from there when to resume it.

void **lithe_context_set_cls**(lithe_context_t *context, void *cls)
Set the context local storage of the current context.

void ***lithe_context_get_cls**(lithe_context_t *context)
Get the context local storage of the current context.

2.2 Lithe Contexts

2.2.1 Lithe Context Stacks

```
typedef struct {
  void *bottom;
  ssize_t size;
} lithe_context_stack_t;
```

lithe_context_stack_t
A generic type representing the stack used by a lithe context.

void ***lithe_context_stack_t.bottom**
Pointer to the bottom of the stack for a lithe context.

ssize_t **lithe_context_stack_t.size**
Size of the stack pointed to by bottom.

2.2.2 Lithe Contexts

```
struct lithe_context {
  uthread_t uth;
  struct lithe_sched *sched;
  void (*start_func) (void *);
  void *arg;
  lithe_context_stack_t stack;
  void *cls;
  size_t state;
};
typedef struct lithe_context;
```

struct **lithe_context**
lithe_context_t
Basic lithe context structure. All derived scheduler contexts MUST have this as their first field so that they can be cast properly within the lithe scheduler.

uthread_t **lithe_context_t.uth**
Userlevel thread context.

struct lithe_sched ***lithe_context_t.sched**
The scheduler managing this context.

void **(*lithe_context_t.start_func)** (void *)
Start function for the context.

void ***lithe_context_t.arg**
Argument for the start function.

lithe_context_stack_t **lithe_context_t.stack**
The context_stack associated with this context.

void ***lithe_context_t.cls**
Context local storage.

size_t **lithe_context_t.state**
State used internally by the lithe runtime to manage contexts.

2.3 Lithe Schedulers

2.3.1 Callback API

```
struct lithe_sched_funcs {
  int (*hart_request) (lithe_sched_t *__this, lithe_sched_t *child, int k);
  void (*hart_enter) (lithe_sched_t *__this);
  void (*hart_return) (lithe_sched_t *__this, lithe_sched_t *child);
  void (*child_enter) (lithe_sched_t *__this, lithe_sched_t *child);
  void (*child_exit) (lithe_sched_t *__this, lithe_sched_t *child);
  void (*context_block) (lithe_sched_t *__this, lithe_context_t *context);
  void (*context_unblock) (lithe_sched_t *__this, lithe_context_t *context);
  void (*context_yield) (lithe_sched_t *__this, lithe_context_t *context);
  void (*context_exit) (lithe_sched_t *__this, lithe_context_t *context);
};
typedef struct lithe_sched_funcs lithe_sched_funcs_t;
```

struct **lithe_sched_funcs**
lithe_sched_funcs_t
Lithe scheduler callbacks/entrypoints.

int **(*lithe_sched_funcs_t.hart_request)** (lithe_sched_t *__this*, lithe_sched_t **child*, int *k*)
Function ultimately responsible for granting hart requests from a child scheduler. This function is automatically called when a child invokes lithe_hart_request() from within it's current scheduler. Returns 0 on success, -1 on failure.

void **(*lithe_sched_funcs_t.hart_enter)** (lithe_sched_t *__this*)
Entry point for hart granted to this scheduler by a call to lithe_hart_request().

void **(*lithe_sched_funcs_t.hart_return)** (lithe_sched_t *__this*, lithe_sched_t **child*)
Entry point for harts given back to this scheduler by a call to lithe_hart_yield().

void **(*lithe_sched_funcs_t.child_enter)** (lithe_sched_t *__this*, lithe_sched_t **child*)
Callback to inform that a child scheduler has entered on one of the current scheduler's harts.

void **(*lithe_sched_funcs_t.child_exit)** (lithe_sched_t *__this*, lithe_sched_t **child*)
Callback to inform that a child scheduler has exited on one of the current scheduler's harts.

void **(*lithe_sched_funcs_t.context_block)** (lithe_sched_t *__this*, lithe_context_t **context*)
Callback letting this scheduler know that the provided context has been blocked by some external component. It will inform the scheduler when it unblocks it via a call to lithe_context_unblock() which ultimately triggers the context_unblock() callback.

void **(*lithe_sched_funcs_t.context_unblock)** (lithe_sched_t *__this*, lithe_context_t **context*)
Callback letting this scheduler know that the provided context has been unblocked by some external component and is now resumable.

void **(*lithe_sched_funcs_t.context_yield)** (lithe_sched_t *__this*, lithe_context_t **context*)
Callback notifying a scheduler that a context has cooperatively yielded itself back to the scheduler.

void **(*lithe_sched_funcs_t.context_exit)** (lithe_sched_t *__this*, lithe_context_t **context*)
Callback notifying a scheduler that a context has run past the end of it's start function and completed it's work. At this point it should either be reinitialized via a call to lithe_context_reinit() or cleaned up via lithe_context_cleanup().

2.3.2 Lithe Schedulers

```
struct lithe_sched {
  const lithe_sched_funcs_t *funcs;
  int harts;
  lithe_sched_t *parent;
  lithe_context_t *parent_context;
  lithe_context_t start_context;
};
typedef struct lithe_sched lithe_sched_t;
```

struct **lithe_sched**
lithe_sched_t
Basic lithe scheduler structure. All derived schedulers MUST have this as their first field so that they can be cast properly within lithe.

const lithe_sched_funcs_t ***lithe_sched_t.funcs**
Scheduler functions. Must be set by the implementor of the second level scheduler before calling lithe_sched_entry().

int **lithe_sched_t.harts**
Number of harts currently owned by this scheduler.

lithe_sched_t ***lithe_sched_t.parent**
Scheduler's parent scheduler.

lithe_context_t ***lithe_sched_t.parent_context**
Parent context from which this scheduler was started.

lithe_context_t **lithe_sched_t.start_context**
The start context for this scheduler when it is first entered.

CHAPTER

THREE

ABOUT

3.1 People

Current Contributors:

- Kevin Klues <klueska@cs.berkeley.edu>
- Andrew Waterman <waterman@cs.berkeley.edu>
- Krste Asanović <krste@cs.berkeley.edu>

Past Contributors:

- Heidi Pan
- Benjamin Hindman <benh@cs.berkeley.edu>
- Rimas Avizienis <rimas@cs.berkeley.edu>

3.2 Publications

Composing Parallel Software Efficiently with Lithe. (PDF)
Heidi Pan, Benjamin Hindman, and Krste Asanović
ACM Conference on Programming Language Design and Implementation (PLDI'10), Toronto, Ontario, Canada. June 2010.

Lithe: Enabling Efficient Composition of Parallel Libraries. (PDF)
Heidi Pan, Benjamin Hindman, and Krste Asanović
USENIX Workshop on Hot Topics in Parallelism (HotPar'09), Berkeley, CA. March 2009.

INDEX

L