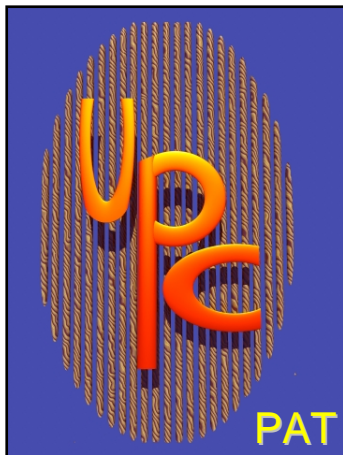




UNIVERSITY OF
FLORIDA

www.hcs.ufl.edu
High-performance Computing & Simulation Research Lab

UPC Performance Tool Interface



Professor Alan D. George, Principal Investigator
Mr. Hung-Hsun Su, Sr. Research Assistant
Mr. Adam Leko, Sr. Research Assistant
Mr. Bryan Golden, Research Assistant
Mr. Hans Sherburne, Research Assistant

HCS Research Laboratory
University of Florida

Motivation for Tool Interface



- UPC performance
 - Can be comparable with MPI code
 - But, generally requires hand tuning
- No performance tool support for UPC programs. Why?
 - New language, but...
 - Complicated compilers
 - Several different implementation strategies
 - Direct compilation (GCC-UPC, Cray)
 - Library approach (Berkeley w/GASNET, MTU UPC, HP)
 - “Wrappers” and binary instrumentation must be handled on a compiler-by-compiler basis
 - One-sided memory operations
 - Relaxed memory model, compiler optimizations/reorganization
 - Direct source instrumentation not accurate enough!



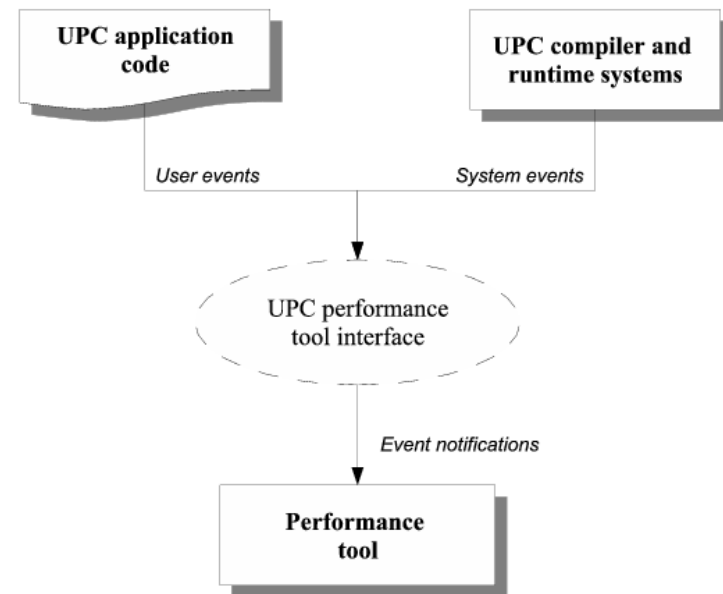
Proposed Interface

- Event-based interface

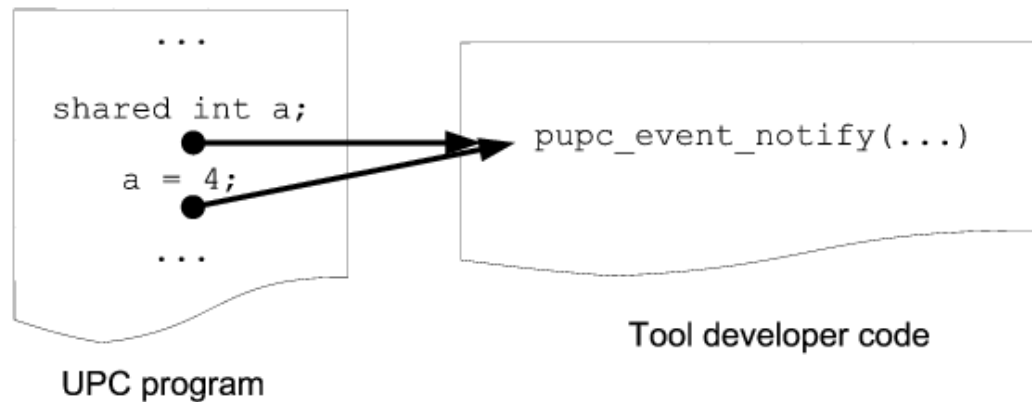
- UPC compiler/runtime communicate with performance tools using standard interface
- Performance tool is notified when certain actions happen at runtime

- Notification structure

- Function “callback” to tool developer code
- Use a single function name
 - Pass in event ID and source code location
 - Use `varargs` for rest of arguments (like `printf`)
- Notifications can come from compiler/runtime (system events) or from code (user events)
- Callback has to be threadsafe, re-entrant



(Very) Simple Example



```
enum pupc_event_type {pupc_event_type_start, pupc_event_type_end,  
                      pupc_event_type_atomic};  
  
void pupc_event_notify(  
    unsigned int event_id,  
    enum pupc_event_type event_type,  
    const char* source_file,  
    unsigned int source_line,  
    unsigned int source_col,  
    ...);
```

Event ID Conventions

- Assumption: 32-bit, unsigned integer
- System-level events
 - Events that arise from actions defined in spec
 - Convention shown right
 - Range: 0x00000000 to 0x5FFFFFFF
- Implementation-specific events
 - Defined by implementation
 - Useful for software cache miss events, etc
 - Range: 0x60000000 to 0xBFFFFFFF
- User-level events
 - Users obtain identifiers from tool at runtime with `pupc_create_event(const char* name)`
 - Used for marking phases or basic blocks of a program's computation
 - Range: 0xC0000000 to 0xFFFFFFFF

0x03 0007 00

Group

Action

Type

Example system event:

Group 3 = library event

Action 7 = `upc_memcpy`

Type 0 = executed

Defined System Events

- Startup and shutdown (group 0)
 - Initialization called by each UPC thread after UPC runtime has been initialized
 - Exit called before all threads stop (two types of events: collective exit & non-collective exit)
- Synchronization (group 1)
 - Fence, notify, wait, barrier start/end
- Work sharing (group 2)
 - Forall start/end
- Library events (group 3)
 - Lock functions & string functions start/end, etc

Defined System Events (2)

- Direct shared variable access (group 4)
 - Two sets of get/put events for relaxed and strict memory accesses
- **Function entry/exit (group 5)**
 - Events occur at the beginning and end of every user function
- Collective events (group 6)
 - Events for all collective routines defined in spec
- All system events have symbolic names defined in pupc.h (along with prototypes for all tool interface functions)
- For complete list and details of each event, see proposal at
<http://www.hcs.ufl.edu/upc/upctoolint/>

System Events Sample Table: Synchronization

Symbolic name	Event identifier	vararg arguments
PUPC_NOTIFY	0x01000000	<code>int</code> named, <code>int</code> expr
PUPC_WAIT	0x01000100	<code>int</code> named, <code>int</code> expr
PUPC_BARRIER	0x01000200	<code>int</code> named, <code>int</code> expr
PUPC_FENCE	0x01000300	(none)

Table 2: Synchronization events

Instrumentation & Measurement Control



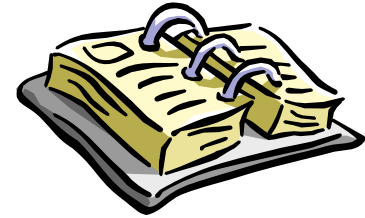
- Need to control overhead of instrumentation for interface
- `--profile` flag
 - Instructs compiler to instrument all events for use with performance tool
 - Compiler should instrument all events, except
 - Shared local accesses
 - Accesses that have been privatized through optimizations
- `--profile-local` flag
 - Instruments everything as in `--profile`, but also includes shared local accesses
- `#pragma pupc [on / off]` directive
 - Controls instrumentation during compile time, only has effect when `--profile` or `--profile-local` have been used
 - Instructs compiler to avoid instrumentation for specific regions of code, if possible
- `pupc_control(int on);` function call
 - Controls measurement during runtime done by performance tool

Tool Access to High-Resolution Timers

- UPC runtime library provides functions to access hardware timers (if available)
- Not a globally synchronized timer, but locally consistent to each thread
- Modelled directly after Berkeley UPC timers
- Get abstract “ticks”, convert to microseconds
- Prototypes:

```
pupc_tick_t pupc_ticks_now();  
uint64_t pupc_ticks_to_us(pupc_tick_t ticks);  
double pupc_tick_granularityus();  
double pupc_tick_overheadus();  
#define PUPC_TICK_MAX ..., #define PUPC_TICK_MIN ...
```

Open Issues



- Column number argument in callback
 - Used to differentiate two statements on a single line
 - But,
 - Available in most systems?
 - Users can split statement, or figure out indirectly what events are from what
 - Suggestion: keep column number, systems that don't support pass in 0
- No direct support for sampling
 - Can use interface directly and keep data structures in memory, then sample those using timer
 - Higher overhead than sampling available structures from runtime directly

Open Issues (2)



- Lines containing multiple events
 - Tools should expect multiple events to come from a single line
 - e.g., `shared int a; shared int b; a = b + b + (++a);`
 - Ideally, should receive an event for each remote get/put
 - Will this limit or change source-to-source translations?
- User function entry/exit points
 - Potential for very high overhead
 - What about mixing in C/MPI code?
 - Can examine stack and relate addresses to source functions, but requires platform-specific code
 - Examining just stack means more instrumentation required for profiling tools

Open Issues (3)



- One-sided RMA events
 - Which thread should get a notification that a remote get/put is starting on the other side, if any?
 - What about relaxed model + nonblocking communication that blocks at fences/barriers?
 - Need a systematic way for creating implementation-specific events
- Non-collective exits
 - Distributed systems (like BUPC) can't guarantee that non-collective exits will be propagated and all threads will receive an exit event
 - Non-collective exits can cause incomplete data due to buffering, unless guards are in place
 - Suggestion: Require users to avoid non-collective exits for profiling runs (reasonable)

Open Issues (4)



- What code is compiled by a UPC compiler?
 - In proposal, tool developer code (`pupc_event_notify` et. al) is compiled by UPC compiler with no `--profile` flag
 - Some compilers might not support `varargs` in UPC code
 - Potential solution: C `pupc_event_notify` function that makes upcalls to UPC code
 - Need way of passing shared void* from C to UPC though!
- Efficient access to MYTHREAD on pthreaded systems
 - How to access MYTHREAD (in TLS) efficiently?
 - How to access at all if C `pupc_event_notify` is used?
 - Suggestion: have `PUPC_INIT` return context pointer that gets passed in on all subsequent profile callbacks

Questions, Comments, Suggestions?

- What can I do to get you into this UPC performance tool interface today?



Changes From Last Proposal (v1.1 to now)

- No source location structs, pass file information directly in callback
- Inclusion of collective event ID category
- New event type argument in callback: start, stop, atomic
 - Halves number of event IDs required
- User-level events
 - Function is now `pupc_user_event()` and also includes event type
 - No more user function events
 - Extra arguments are meant to be used as in `printf`:
 - `pupc_create_event("Step", "%d %f")`
 - `pupc_user_event(i, f);`
- Different flags for instrumentation control
- Timers not "global timer"