User Level DB: a Debugging API for User-Level Thread Libraries

Kevin Pouget* , Marc Pérache† , Patrick Carribault† and Hervé Jourdren†
*kevin.pouget@gmail.com; †{marc.perache, patrick.carribault, herve.jourdren}@cea.fr
CEA, DAM, DIF
F-91297 Arpajon France

***Abstract*—With the advent of the multicore era, parallel programming is becoming ubiquitous. Multithreading is a common approach to benefit from these architectures. Hybrid M:N libraries like MultiProcessor Communication (MPC) or MARCEL reach high performance expressing fine-grain parallelism by mapping M user-level threads onto N kernel-level threads. However, such implementations skew the debuggers' ability to distinguish one thread from another, because only kernel threads can be handled. SUN MICROSYSTEMS' THREAD_DB API is an interface between the debugger and the thread library allowing the debugger to inquire for thread semantics details. In this paper we introduce the USER LEVEL DB (ULDB) library, an implementation of the THREAD_DB interface abstracting the common features of user-level thread libraries. ULDB gathers the generic algorithms required to debug threads and provide the thread library with a small and focused interface. We describe the usage of our library with widely-used debuggers (GDB, DBX) and the integration into a user-level thread library (GNUPTH) and two high-performance hybrid libraries (MPC, MARCEL).**

***Keywords*-Multithreading, Debugging, MPC,GDB.**

I. INTRODUCTION

For a few years, the surge of clock frequency of processors has reached a limit. Power consumption and heating constraints prevent most of the processors from being clocked faster than 4 GHz (the IBM POWER6 is the only one cadenced faster than 5 GHz [1]). That is why industry started to increase the number of cores on a die. A die generally contains between 2 and 16 cores, but they can sometimes be up to 64 or 80 [2]. In the domain of high performance computing (HPC), multithreading is a programming model used to exploit such architectures. This can be done through explicit multithreading (*e.g.,* POSIX threads) or language-generated multithreading (*e.g.,* OPENMP [3]). Hybrid thread-libraries (mixing user-level and kernel-level threads, see Section II) fit the requirements of HPC by exploiting efficiently parallel architectures and avoiding unnecessary and costly system calls.

Debugging of multithreaded processes is much complicated than single thread processes. First of all, the debugger needs the support of the thread library. This can be done through the THREAD_DB API [4] introduced by SUN MICROSYSTEMS. This API is implemented by the thread library and exposed to the debugger. It allows the debugger to handle thread *semantics*. Linux NPTL [5] and Solaris Threads provide a more or less complete implementation of THREAD_DB, which lets them be controlled by common debuggers like GDB [6], DBX [7] or TOTALVIEW [8].

The use of user-level or hybrid thread libraries provides good performances but most of these thread libraries do not implement the THREAD_DB API. Without this accurate support, debuggers can only handle LWPs (Light Weight Process) distinction, rather than end-user level threads. LWPs are related to the underlying OS. Their events can be monitored through the `ptrace` system call [9]. Also this function offers memories and registers access primitives. However, user threads only exist within their library. The kernel is not aware that user threads are handled on a LWP and cannot have information about them as soon as they are unscheduled.

Example 1 shows the list of threads natively known by the debugger for an application with five user threads mapped onto two LWPs (a), and the *same* execution with the additional knowledge provided by our USER LEVEL DB (ULDB) module (b). One can see that the threads 1, 2 and 5, blocked in the user-level thread scheduler, are natively unknown. Only the user threads currently scheduled on a LWP are visible.

Example 1 Hybrid thread debugging

```
(gdb) info threads
Thread  2 (LWP 2) in bar ()
Thread  1 (LWP 1) in foo ()
```

(a) Natively

```
(gdb) info threads
Thread  5 (LWP 2) in mpc_setjmp ()
Thread  4 (LWP 2) in bar ()
Thread  3 (LWP 1) in foo ()
Thread  2 (LWP 1) in mpc_setjmp ()
Thread  1 (LWP 1) in mpc_setjmp ()
```

(b) With the ULDB module

In this paper, we describe our ULDB library, an implementation of the THREAD_DB API. ULDB is designed to be used by any thread library to deal with the debugger without requiring a huge development effort. In Section II we introduce the different models of thread libraries. Then in Section III we describe the THREAD_DB architecture. in Section IV we discuss our ULDB library and in Section V its API. In Section VI, we highlight some examples of utilization of our library. We conclude this paper with the further perspectives of work.

II. Related Work

The thread libraries lie between the user application and the operating system. They offer parallelism and synchronization primitives to design multithreaded programs. Three classes of libraries can be distinguished (see Figure 1):

Kernel-level libraries: Thread creations, scheduling and synchronizations are managed by the system kernel. Such threads are named *virtual processors* or LWPs on Unix systems. They efficiently exploit parallel architectures, but each thread operation is costly because of the system calls. (Futexes [10] provide an alternative by staying in userland for the acquisition/release of free mutexes.) As one user thread is mapped to one kernel thread, these libraries are said to be "1:1" (Figure 1.a).

User-level libraries: These only use user-space functions to handle threads. The stacks of the threads are built with context switches or through an alternative signal stack (*e.g.,* `makecontext` or `sigalt stack`, respectively; see [11] for further details) and the execution flows from one thread to another *via* `swapcontexts` or `longjmps`. The latency of these operations is short, and most of them are widely portable. However, a blocking system-call will freeze all the threads (the kernel in unaware that the program is multithreaded). These libraries are "N:1" (Figure 1.b shows a "3:1" mapping).

Hybrid-level libraries: They gather the principles of both previous models: multiprocessors are exploited with the creation of several *kernel threads* (*i.e.,* LWPs), whereas operations done frequently are handled in user space. Stalls due to system calls can be avoided by creating a kernel thread which will run the remaining threads on the behalf of the blocked one. Obviously, these advantages come at the price of a high complexity in the library implementation. These libraries are "N:M" (Figure 1.c shows a "6:2" mapping).

III. Thread_DB Architecture

The Thread_DB interface describes many aspects of thread debugging. The current version of this library addresses the questions of thread distinction and backtrace display (features supported by GDB). These features are enabled by the thread library related part of Thread_DB:

Iteration: The thread list must be traversed according to the request criteria and a callback function applied on each thread (`td_ta_thr_iter`);

Mapping: The debugger needs to know which thread is running on a given LWP (`td_thr_map_lwp2thr`);

Event Notification: Some events of the thread life (*e.g.,* birth, death) must be notified to the debugger, through the execution of breakpointed instructions (`td_{ta|thr}_event_*`);

Information: Details about a thread current characteristics like state, LWP, ...(`td_thr_get_info`);

Register accessors: The general registers of a thread must be read, either on its LWP if the thread is active, or somewhere else, at the discretion of the implementation, if the thread is `asleep`. These registers are used to build the thread backtrace (`td_thr _{get|set}regs`).

As the debugger and the thread library (the debuggee) are two distinct processes, they do not share an address space. This implies that they cannot communicate directly. Thus, the implementation of the Thread_DB API is divided into two modules: the first one, doing the actual implementation of the interface, lives in the debugger address-space, whereas the second lives with the debuggee. The exchanges between the two sides are only one way, debugger to debuggee, and are done through the Proc-service interface of the debugger (memcpy-like functions). Figure 2 draws a diagram of this architecture.

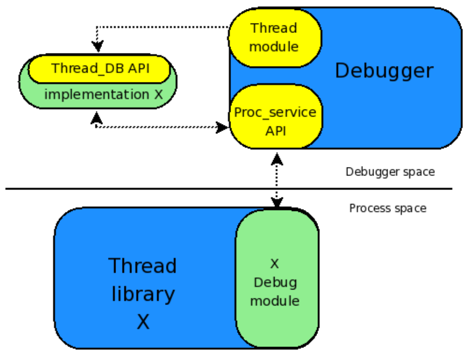

Figure 2. The Thread_DB architecture

In this architecture, each thread library needs to provide a library which implements Thread_DB (libthread_db). This implementation is strictly bound to its thread library. For instance, the location and structure of the thread list is nowhere standardized.

IV. User Level DB Library

We propose ULDB library that offers an implementation of the Thread_DB API, which lightens thread-library programmers from the implementation of the generic algorithms of the API (See Figure 3). ULDB API is small and focused (only 20 functions). It defines only few rules to respect. This debug library provides the thread library with standardized debug features. ULDB is easily portable over different operating systems: one optional system call — to read the current LWP and architectures: the register set has to be ported.

A special care has been taken to manage user- and hybrid-thread libraries. Their non-bijective mappings (respectively N:1 and N:M) lead the debugger to misunderstand the

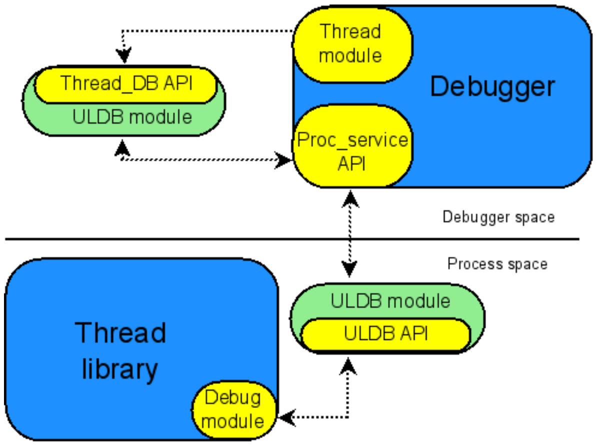

Figure 3. The ULDB architecture

actual user-thread configuration. In fact, only LWPs, created by the system kernel, can be followed in a generic way. Debuggers are natively able to distinguish one kernel thread from another, build their backtrace, manage their signals, *etc*. Our library provides debuggers with the ability to do these operations at the user-thread scope.

The process side of the library has to be linked up with the program to debug, or at least with the thread library. The thread manager provides ULDB with application-specific details during the initialization, and then with thread-specific data, at each thread creation. In both cases, only a few information are required: the register offset and a lock object for the initialization, and the address of the context buffer for each thread. Further optional information can be provided to precise the thread characteristics: its type (created by user or owned by the thread manager), the boundaries of its stack, its start function, …

A. Thread states and event notification

In addition to these static information, ULDB must be provided with up-to-date details about the thread life. The most important ones are its state and the LWP it is executed on. The former is used to determine whether the registers of a thread need to be read on memory (if it is `asleep`), or directly on its LWP. These values are essential for a correct behavior and the closer update before/after the switch of state, the better. Wrong informations about the thread's state lead to incorrect reads into memory. For example, to backtrace one blocked thread B, the debugger may read registers from thread A actually executing on the LWP if the state of the thread B is not `asleep` whereas the data required are located into memory.
Thread birth and death must also be notified to the debugger. According to the THREAD_DB specifications, much more events shall be reported, but debuggers (*i.e.,* GDB, DBX) only manage these two ones. The report of events is done with the help of special functions, breakpointed by the debugger. The handling of these functions is done internally. Notification of events are conditioned by two bit-masks, the first one is valid for the whole application, an the second one is specific to a thread. Each event is activated or disabled at the willing of the debugger. The management of these bit-masks and the reporting of events is also done internally, the thread manager only has to inform ULDB that birth or death occurred.

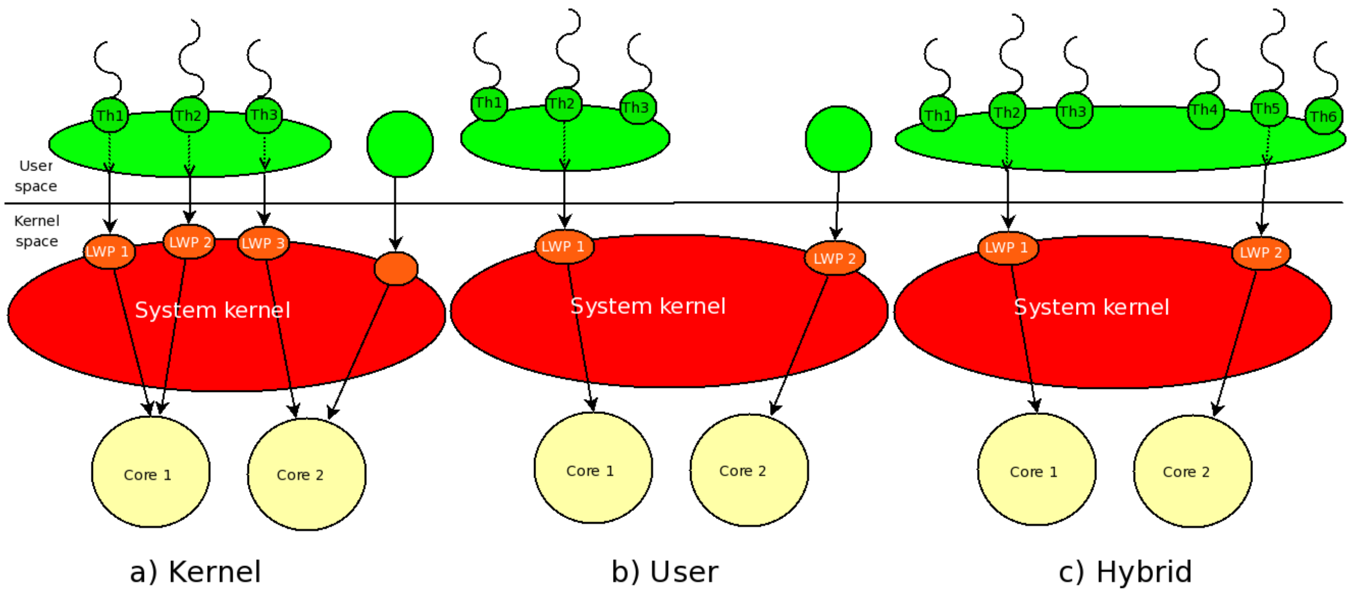

Figure 1. Levels of scheduling

B. Performance matters

As one of the purposes of user-level thread libraries is to build high performance programs, we paid attention to lower as much as possible our footprint on the execution. Therefore, we defined two sets of functions:

- one for a fast integration of the module, where the thread identifier is defined by the library, usually the Thread ID (TID). No modification of the thread library structures is required. Internally, we browse a linked list to retrieve our handles. We may improve this search thanks to a hash table;
- the second set offers a better scalability: the handle is directly managed by the thread library, which may store it in its thread data. As most of our functions are short and inlined, the overhead is very limited. This solution is the best way to use our library.

V. User Level DB API

The ULDB API is used by the thread library to deal with the debugger (see Figure 3). It describes how to initialize ULDB and the thread-related parameters. It is divided into a static set and a dynamic one. A `DEBUG` mode is also available to help the thread library's developer to set up correctly ULDB with the notification of incoherent or unordered use of the functions.

A. Library initialization

The thread manager provides here informations valid for the whole run. The offsets of the registers inside the context buffer must first be detailed. The context of a thread will be read directly into memory by the debugger when the thread is not be active (`asleep` state). Here are some example of Linux/x86 offsets for `jmb_buf` and `mcontext_t` contexts:

Register	eip	esp	ebp	ebx	edi
mcontext_t	14	7	6	8	4
jmp_buf	5	4	3	0	2

Then, the thread manager must provide an initialized lock object (*e.g.,* `mutex, spinlock`) and the corresponding `lock`, `release` and `free` functions. This lock will be used to ensure that the critical region of the ULDB library (*i.e.,* the thread list management) is serialized. The lock will be freed, if necessary, at the end of the debugging session (triggered by the debugger side of the module). If the thread library runs at user level and is not preemptive (*e.g.,* the GnuPth), no lock is required. This part of the API in presented in Source 1.

Source 1 Initialization of the ULDB module

```
/* Enable or disable the ULDB library */
tdb_err_e  uldb_enable_lib_thread_db (void)
tdb_err_e  uldb_disable_lib_thread_db (void)

/* Provide the offsets of x86 registers */
tdb_err_e
  uldb_set_eip_offset (size_t offset)
tdb_err_e
  uldb_set_esp_offset (size_t offset)
tdb_err_e
  uldb_set_ebp_offset (size_t offset)
tdb_err_e
  uldb_set_ebx_offset (size_t offset)
tdb_err_e
  uldb_set_edi_offset (size_t offset)

/* Provide a lock object and its accessors */
tdb_err_e
  uldb_set_lock (void *lock,
                 int (*acquire) (void *),
                 int (*release) (void *),
                 int (*lock_free) (void *))
```

B. Thread attributes

They describe the characteristics of a thread. Some of them are static, like the context address, the type or the boundaries of the stack; whereas others must be kept up-to-date. For example, the LWP to which a thread is bound may vary over the time if the library allows work stealing or thread migrations. As well, the thread state (*e.g.,* `active, asleep, runnable`) is frequently changed by the scheduler. The states `TD_THR_ACTIVE` and `!TD_THR_ACTIVE` must be emphasized, for they are used to settle where the context is read: on the LWP for the former (the thread is actually executed on the LWP), in the memory of the latter (the thread is blocked or wait for execution). Some attributes like the start function, the type of thread or the stack boundaries are not mandatory for the library; they only provide further details to be displayed by the debugger. Source 2 presents this second part of the API.

C. Performance facilities

The functions suffixed by `*_tid` exempt the user from the ULDB thread-handles management, and only ask for a TID. The function `uldb_get_thread` is used internally to map from this TID to the handle. The structure used to store the thread handles is actually a linked list. Therefore, it may become expensive to traverse the list often when numerous threads are used. Libraries looking for high performance shall avoid the utilization of the `*_tid` functions and manage internally the mapping TID to the handle.

D. System and architecture

The system architecture is important to be known to deal with thread contexts. As well, a system-dependent call is used to determine the LWP a thread is running on (in `uldb_get _lid`). These information must be provided to ULDB *via* preprocessor macros.

Source 2 Initialization and update of the thread attributes

```
/* add or remove a thread to the internal list */
uldb_add_thread (const void *tid, tdb_thread_debug_t **thread)
uldb_remove_thread (tdb_thread_debug_t *thread)

/* report the events monitored by the debugger*/
uldb_report_creation_event (tdb_thread_debug_t *thread)
uldb_report_death_event  (tdb_thread_debug_t *thread)

/* Provide the address of the context of a sleeping thread */
uldb_set_thread_context (tdb_thread_debug_t *thread, void *context)
/* Provide further details about a thread (optional) */
uldb_set_thread_startfunc (tdb_thread_debug_t *thread, char *tls)
uldb_set_thread_stkbase (tdb_thread_debug_t *thread, void *stkbase)
uldb_set_thread_stksize (tdb_thread_debug_t *thread, int stksize)
uldb_set_thread_type (tdb_thread_debug_t *thread, td_thr_type_e type)

/* Provide the current LWPid / state (active or not) of a thread */
uldb_update_thread_lid (tdb_thread_debug_t *thread, lwpid_t lid)
uldb_update_thread_state (tdb_thread_debug_t *thread,  td_thr_state_e state)
```

VI. Experimentations

Several experimentations have been carried out to validate the design of the library and its API. In this section we describe how we managed to use our library with debuggers like GDB and DBX, with some modifications of the source code of the former. Then we tackle the work required to allow threads from user and hybrid-thread libraries (GnuPth, MPC and Marcel) to be debugged.

A. Cooperation with GDB and DBX

GDB is a widely used debugger, part of the Gnu project. However, its Linux versions are explicitly designed to work with the kernel-level thread library GlibC/NPTL [5] of the Thread_DB interface. That is, it is not possible in the current GDB (6.8) to choose at runtime the path of the `libthread_db` to use; and only kernel threads can be handled.

To cope with these difficulties, we prefixed the shared library loading by the check of an environment variable (`GDB_LIBTHREAD_DB`), which shall point to the Thread_DB implementation. The issue of kernel-thread debugging is not so obvious. As we mentioned before, when the scheduler lies in the kernel space, a "1:1" mapping is assumed. Thus GDB considers that the registers of all the threads can be directly read on their LWP. This assumption is clearly false in user and hybrid libraries. To solve this problem, we analyzed the Solaris implementation of the GDB thread module. The thread library used on Solaris systems was hybrid a few years ago and GDB was able to debug it. We extracted the functions dealing with register handling (`to_fetch_registers` and `to_store_registers`) and adapted them to the Linux version.

The Linux version of DBX (7.7) suffers from a bug which limits debugging of our hybrid libraries. For user-level libraries, ULDB could be used transparently. The only exception is that the state `TD_THR_SLEEP` crashes the debugger and should not be used. The environment variable (`_DBX_LIBTHREAD_DB_OVERRIDE`) sets the path of the `libthread_db` to load.

B. Integration into GnuPth

The GnuPth thread library [12] runs at user level, without preemption. The integration of ULDB has been carried out in less than a day, without further help than the on-line documentation. The resulting patch has around 100 lines and modifies five files.

The interface accesses are gathered in the `pth_debug` module of the library. One function enables the library, by providing the offset of the registers — the `System V/setcontext` context controller are used — and no lock are required — thanks to the non-preemptive user scheduler. Then a function adds each thread in the module and specify its context location, start function, type and lid; and another remove them at their death. The only further calls to our module deal with state switching (`TD_THR_ACTIVE` vs. `TD_THR_SLEEP`) when a thread is scheduled or fells `asleep`.

C. Integration into MPC

MPC [13], [14] is an environment aimed at providing programmers with efficient runtime system for their existing MPI, Posix Thread or MPI+Thread applications. It features an hybrid-level thread scheduler, optimized to deal with tasks communications and synchronizations.

The modifications are globally the same as in GNUPTH, except that MPC uses two context controllers, according to the underlying system. This specificity is easy to take into account with ULDB: only the register offsets and the address of the context must be changed, as described in Source 3.

Source 3 Choice of the context controller

```
#if SCTK_MCTX_MTH(mcsc)
/* swapcontext/makecontext */
  uldb_set_thread_context \
  (thread, ttid->ctx.uc.uc_mcontext.gregs) ;
#else
/* setjmp/longjmp */
  uldb_set_thread_context \
  (thread, ttid->ctx.jb);
#endif
```

MPC also allows to create system threads used internally. These threads (*i.e.,* `idle tasks`, `timing tasks`, ...), created by MPC for its own use, do not follow the classic thread-creation path. The flag `TD_THD_SYSTEM` is triggered to distinguish them from final-user threads, as quoted in Example 2.

The effort to use ULDB with MPC is limited and requires only a 600 patch.

Example 2 User and System threads

```
(gdb) info tdb-threads
#1) system thread #0x20d76a, lwp 25937,
    (run)    startfunc: idle_task
[...]
#3) user   thread #0x386020, lwp 25937,
    (active) startfunc: erato
```

D. Integration into MARCEL

MARCEL [15] is the thread library of the *Parallel Multithreaded Machine* project. It offers high-performance hybrid-thread facilities.

The integration has been performed in a few weeks, in remote collaboration with contributors of MARCEL. The resulting patch has around 500 lines.

Like for the two other libraries, no important changes have been required for this integration. Most of the thread creations are done in the `marcel_sched_internal_create_[dont]start` functions. Only the kernel threads (created with the `clone` system call) go through a different path. All the context switchs are executed in `marcel_switch _to`, including the preemption event. The only drawback is that the debugger slows down the process execution, what impedes the preemption. Preemption time-slice must be reduced to get back the original behavior.

VII. CONCLUSION AND FUTURE WORK

In this paper, we introduced the USER LEVEL DB library (ULDB), a generic implementation of the THREAD_DB interface. ULDB aims at providing the thread library with a high-level interface, by abstracting away all the generic features. We successfully demonstrated the ability of our library to allow debugging of several user (GNUPTH) and hybrid (MARCEL, MPC) thread libraries. A very low level of expertise was required in each case, and the modifications were minor and lightweight for the thread libraries.
It is also important to highlight that ULDB does not break the ability of graphical debuggers like DDD, Insight, Eclipse-Debug or DDT to use command-line debuggers (GDB, DBX) as back-end. Hybrid and user threads are handled transparently, with the added features of the front-ends.

Currently, ULDB implements all the thread semantic features supported by GDB. The support of thread synchronization debugging needs to be investigated and included in the ULDB API and in the debugger as far as GDB is concerned.

ULDB is distributed as a part of the MPC project available at http://mpc.sourceforge.net.

ACKNOWLEDGMENT

We would like to thank Ivan SOLEIMANPOUR and Leonard LI from SUN MICROSYSTEMS for their help and advices about DBX. We also acknowledge Samuel THIBAULT and Olivier AUMAGE for their help with MARCEL.

REFERENCES

[1] B. Curran, E. Fluhr, J. Paredes, L. Sigal, J. Friedrich, Y.-H. Chan, and C. Hwang, "Power-constrained high-frequency circuits for the IBM POWER6 microprocessor," 2007.

[2] J. E. Savage and M. Zubair, "A unified model for multicore architectures," in *IFMT '08: Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*, 2008.

[3] OpenMP, "OpenMP," http://www.openmp.org.

[4] S. Microsystem, "Thread_DB Man Page (3)," http://docs.sun.com/app/docs/doc/802-1930-03/6i5u95urj?l=en&a=view.

[5] I. Molnar, "The native posix thread library for linux," Tech. Rep., RedHat, Inc, Tech. Rep., 2003.

[6] GNU Project, "The GNU Debugger," http://www.gnu.org/software/gdb/.

[7] Sun Microsystem, "SunStudio DBX," http://developers.sun.com/sunstudio/.

[8] TotalView Tech, "TotalView," http://www.totalviewtech.com/.

[9] J. Keniston, A. Mavinakayanahalli, and P. Panchamukhi, “Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps,” in *In Proc. 2007 Linux Symposium*, 2007.

[10] Ulrich Drepper, “Futexes Are Tricky,” Jan. 2006, http://people.redhat.com/drepper/futex.pdf.

[11] T. S. Stack and R. S. Engelschall, “Portable multithreading,” in *In Proc. USENIX Tech. Conf*, 2000.

[12] GNU Project, “The Gnu Portable Threads,” http://www.gnu.org/software/pth/.

[13] M. Pérache, H. Jourdren, and R. Namyst, “MPC: A Unified Parallel Runtime for Clusters of NUMA Machines,” in *Proceedings of the 14th International Euro-Par Conference*, Aug. 2008.

[14] M. Pérache, P. Carribault, and H. Jourdren, “MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption,” in *Proceedings of the 16th International EuroPVM/MPI Conference*, Sep. 2009.

[15] S. Thibault, “A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines,” in *Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, 2005.