A Thread-Aware Debugger with an Open Interface

Daniel Schulz
Qcentic GmbH
Max-Planck-Str. 39A
50858 Köln
Germany

Frank Mueller
Humboldt University Berlin
Institut f. Informatik
10099 Berlin (Germany)
mueller@informatik.hu-berlin.de

ABSTRACT

While threads have become an accepted and standardized model for expressing concurrency and exploiting parallelism for the shared-memory model, debugging threads is still poorly supported. This paper identifies challenges in debugging threads and offers solutions to them. The contributions of this paper are threefold. First, an open interface for debugging as an extension to thread implementations is proposed. Second, extensions for thread-aware debugging are identified and implemented within the Gnu Debugger to provide additional features beyond the scope of existing debuggers. Third, an active debugging framework is proposed that includes a language-independent protocol to communicate between debugger and application via relational queries ensuring that the enhancements of the debugger are independent of actual thread implementations. Partial or complete implementations of the interface for debugging can be added to thread implementations to work in unison with the enhanced debugger without any modifications to the debugger itself. Sample implementations of the interface for debugging have shown its adequacy for user-level threads, kernel threads and mixed thread implementations while providing extended debugging functionality at improved efficiency and portability at the same time.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.5 [**Software Engineering**]: Testing and Debugging; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Design, Standardization, Languages

Keywords

Debugging, Concurrency, Threads, Active Debugging, Open Interface

1. INTRODUCTION

Threads have become an accepted abstraction of concurrency using the shared-memory programming paradigm and provide the means to exploit parallelism in a shared-memory multi-processor environment. Today, many thread implementations adhere to the POSIX Threads (Pthreads) standard [21], which defines a common application interface (API) to exhibit the functionality of threads. The Pthreads standard describes the semantics in terms of the observable behavior for this API but excludes constraints on implementation choices. Hence, Pthreads implementations range from user-level libraries [14, 17] via mixed-mode threads [16, 20, 8] to kernel-level implementations [2, 22, 1, 11].

Software development of threaded programs should be facilitated by adhering to the Pthreads API at the level of program design and implementation. The testing and debugging stage, however, lacks support for threaded applications. The motivation for special testing and debugging tools is given by a number of properties that distinguish multi-threaded programs from single-threaded ones:

1. The control flow of threads may interleave or even execute in parallel.
2. Threads may suspend and resume execution voluntarily, due to preemption or as a result of events (signals).
3. Synchronization between threads defines a partial order of program execution.

The debugging process, which takes at least 50% of the development effort together with testing, is affected for threaded programs in several ways [7]. The following issues illuminate common problems.

- Conventional breakpoint debugging does not suffice to capture a single flow of control for a program. The programmer is accustomed to follow the control flow of one thread. When two consecutive breakpoints within a thread are hit, other threads may have been executing between these breakpoints. Furthermore, a breakpoint in a subroutine called by different threads may be hit in sequence for different threads at a time.
- The state of threads and synchronization objects is not visible during debugging due to a lack of debugger information. However, state information would be vital

to allow inferences about the execution stage of the program and its progress relative to the partial order of synchronization.

- Thread scheduling cannot be controlled by the debugger. It may, however, be desirable to forcibly suspend or resume the execution of selected threads to identify problems in the application by reducing interference between or ensuring reaction to other threads, respectively.

Thus, a thread-aware debugger should provide the following features that address these issues to facilitate the debugging of threads:

Thread-specific breakpoints stop the application only when a certain thread reaches the breakpoint.

Status inquiries about threads and synchronization objects show the progress of execution and the current state of the objects.

Scheduling control provides the means to forcibly suspend and resume threads.

Scheduling breakpoints halt the application upon a context switch and serve as a means to track the interleaving of execution between threads.

The work described in this paper also aims at providing a flexible platform for both the debugger and a variety of thread implementations to support thread-aware debugging. Instead of customizing the debugger for each thread implementation, a common framework for controlling threads is provided, which communicates with the threads of the application. The thread implementation, on the other hand, provides a standard interface for debugging to serve requests by the debugger. This approach has several advantages:

Portability is ensured through an open interface for debugging threads on one side and functional extensions to the debugger on the other side. The former requires that thread implementations support this interface by providing at least part of the functionality but does not assume any particular API for thread implementations, *e.g.*, POSIX compliance is optional. The latter is independent of the actual thread implementation and remains unchanged, regardless of the extend of the support by the open interface or the source language of the application.

Extensibility is guaranteed by the communication interface between the debugger and the threaded application. This interface only defines a query language but not the actual messages themselves to allow the addition of new functionality and new messages in the future without changes in the communication interface on either side.

Flexibility is provided for partial support instead of the full functionality of the interface for debugging threads. The debugger remains functional but provides less information and less control over threads when only a part of the interface for debugging threads exists. This allows partial implementations of the debug interface where certain information is not available or not accessible, *e.g.*, when kernel threads prohibit access/control of threads.

Optional invocation allows the application to run without the thread debugging support while the same executable may be used for debugging when needed. The thread debug support can be dynamically loaded as an add-on library only upon activation of the debugger.

The technical issues of these features are presented in detail in the description of the design and implementation of the thread-aware debugger and the interface for debugging threads.

This paper is structured as follows. Section 2 gives an overview of the design. Section 3 describes the open interface for debugging threads whose implementation will depend on the thread implementation. Section 4 introduces the thread debug interface common to all implementations. Section 5 presents the communication structure between debugger and application. Section 6 summarizes the extensions to the debugger. Section 7 describes the implementation. Section 8 lists the extended commands for thread-aware debugging. Section 9 discusses related work. Finally, Section 10 presents the conclusions.

2. DESIGN OVERVIEW

The components of the framework for thread-aware debugging comprise two executable components and two interfaces. The executable components are the application on one side and the debugger on the other side. Since the application is assumed to be multi-threaded, it also utilizes a thread implementation. The debugger includes enhancements for thread-aware debugging and for communication with the application. The interfaces consist of a *thread debug interface (TDI)* and the *thread extensions for debugging (TED)*. The TDI includes a query language interpreter and provides the communication interface between debugger and application. The TED comprises the open interface for debugging threads as a thin layer over the actual thread implementation.

The separation of TDI and TED was a design choice aiming at separating generic parts of the framework, such as the TDI, from non-generic parts that depend on the actual thread implementation, such as the TED. Without the distinction between these interfaces, the TDI of a thread-aware debugger would need to be modified each time when support for a new thread implementation is added. Figure 1(a) depicts this case where the TDI includes interface components $I_1, ..., I_n$ for each thread implementation. These interface components would be required to extract internal information from the thread implementation and transform them into a normalized representation. Even if the threads API was restricted to POSIX threads, as depicted here, the components of data structures (*e.g.* `pthread_t`) would differ from one implementation to the next requiring the interface components as a mediator. This, in turn, forces a rebuild of

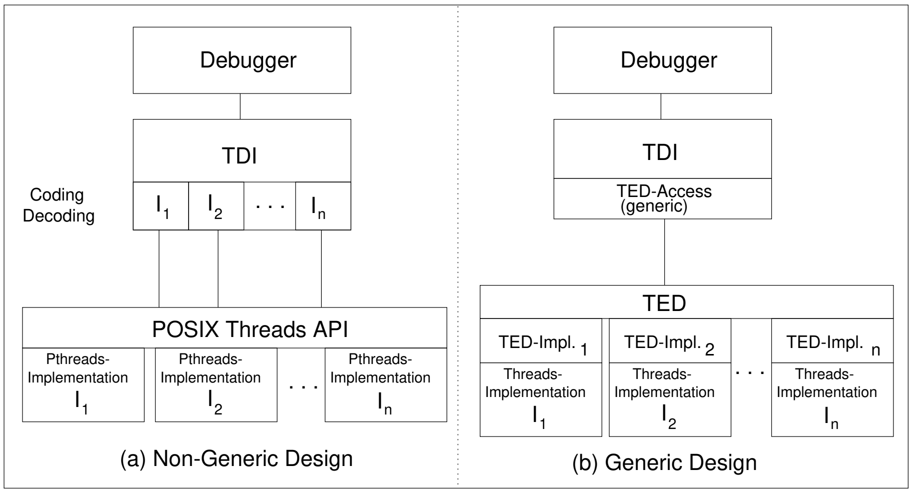


Figure 1: Design Options for Encapsulation

the TDI each time support for a new thread implementation is added.

Figure 1(b) shows better encapsulation chosen for the implementation. The TDI uses a generic interface to the TED component. The TED provides access to the internals of a thread implementation, *i.e.*, the TED has an implementation-dependent part. Since the TED only provides a thin layer, it reduces the amount of implementation-dependent code considerable compared to Figure 1(a) where the TDI is implementation dependent (and the TED is missing). The TED also provides opportunities to integrate non-standard thread implementations since the abstraction from the thread API occurs early while the non-generic approach requires adherence to a certain thread interface on the TDI level.

The encapsulation by TDI and TED also provides the means of *active debugging*. In active debugging, the application is enhanced by special routines that may provide and collect information about the state or perform manipulations on the executing of an application. This approach facilitates and speeds up the debugging process. *Passive debugging* only probes the application. Instead of extensions for debugging on the application side, the debugger is enhanced to contain knowledge about the thread implementation. Table 1 compares active to passive debugging. Generally, debuggers extract information from an application using a procedural approach through probing data, even if data may later be processed within the debugger under a different paradigm. Active debugging allows preprocessing on the application side to communicate data following an arbitrary paradigm, *e.g.*, using a declarative paradigm, as given in this paper. The encapsulation by TDI and TED hides implementation details of the threads to enhance portability, as discussed before. In addition, the TDI maintains a database of the application's state. Queries to the database are performed in a uniform and extensible query language. Furthermore, requests for the state of distinct objects from the debugger can be clustered and are optimized to remove redundancies. As a result, such a declarative query interface performs better than a procedural interface where each information request would require a separate action by the debugger. Post-mortem debugging, *i.e.*, debugging core files of a prematurely terminated execution of the application, provides support for debugging threads in the passive case. Active debugging does not work with post-mortem debugging since the program is no longer executable. Hence, the TED functionality cannot be utilized.

3. THREAD EXTENSION FOR DEBUGGING (TED)

The objective of the TED layer is to provide uniform access to implementation-dependent thread structures. Basic primitives to manipulate sets within the TDI realize a uniform method to access information. This information can either be extracted directly from the thread implementation (if the API supports direct access) or has to be extracted by extension of the thread implementation for debugging.

For example, a thread within an application has a state similar to a process: it may be running, ready, blocked or terminated.[1] The Threads API, however, may not provide access to the internal state of a thread. A non-standard function

[1]The implementation actually distinguishes the cause of blocking. A thread may be blocked on a *mutex*, a *condition variable*, a *timer* object, due to *suspension* or for an unspecified reason (*other*).

Issue	Active Debugging	Passive Debugging
details of thread implementation	not known to debugger	must be known by debugger
change/add new thread impl.	+ no changes of debugger	− debugger must be enhanced
extract info from application	declarative approach	procedural approach
query overhead	+ lower, no redundancies	− higher, redundant requests
post-mortem thread debugging	− not possible	+ always possible

Table 1: Active vs. Passive Debugging

to access the state of a thread is added in such a case to provide the required access. The state is then translated to a standard encoding defined by TED uniform for all thread implementations.

The TED provides access functions for attributes with a common signature to simplify and unify access to any internal data structure.

$$S_r : T_{D_O} \rightarrow T_{D_A}$$

$$S_w : T_{D_A} \times T_{D_O} \rightarrow T_{D_A}$$

Functions for reading S_r and writing S_w with domains $dom(T_{D_A}) = D_A$ and $dom(T_{D_O}) = D_O$ for types T of the domain of objects D_O and addresses D_A allow arbitrary values to be associated with objects for later inquiries.[2] The objects are active or passive entities of the threads implementation, such as threads, mutex objects and condition variables. Objects of a common entity can be accessed using set operations that are either mapped onto the threads API or onto functions that serve as debugging extensions of the API and access internal structures. For example, the set of all threads within an application may not be accessible through the threads API but there commonly exists an internal data structure with access functions, which can be utilized by TED. Mutex objects, on the other hand, are typically not linked to each other so that the set of mutex objects has to be maintained on the TED level. For this purpose, call-outs of the thread implementation to the TED layer upon object creation provide the means to register these objects in a common set within TED. These call-outs are part of the modifications to the thread implementation to ensure debugging support.

TED also supports relations between objects that are created or revoked when certain events occur. Upon occurrence of such an event, a call-back from the thread implementation updates a relation. Let D_T, D_M, D_{CV} be the domains of threads, mutexes and condition variables, respectively. Then, the following relations may hold (see Figure 2):

1. **OwnedBy**{ThreadID:D_T, MutexID:D_M}
2. **BlockedOn** {`ThreadID:`D_T`,` `MutexID:`D_M}
3. **WaitFor** {`ThreadID:`D_T`,` `CondVarID:`D_{CV}}
4. **SignaledBy**{`CondVarID:`D_{CV}`,` `MutexID:`D_M}

Relations 1 to 3 have a cardinality of $1 : N$, *i.e.*, a thread may own multiple mutexes, multiple threads may be blocked on the same mutex or may wait for the same condition variable. Relation 4 has a cardinality of $1 : 1$ for Pthreads since only one mutex may be associated with a condition variable that threads are blocked on at a time. Other cardinalities can be supported as well. *E.g.*, MIT Threads has a $M : N$ model that allows threads to be blocked on the same condition variable even if they used different mutexes before suspending. Notice that $M : N$ cardinalities would require

[2]In the implementation, T_{D_O} is substituted by T_{D_A} since objects can be uniquely identified by their addresses. This simplifies the mapping even further.

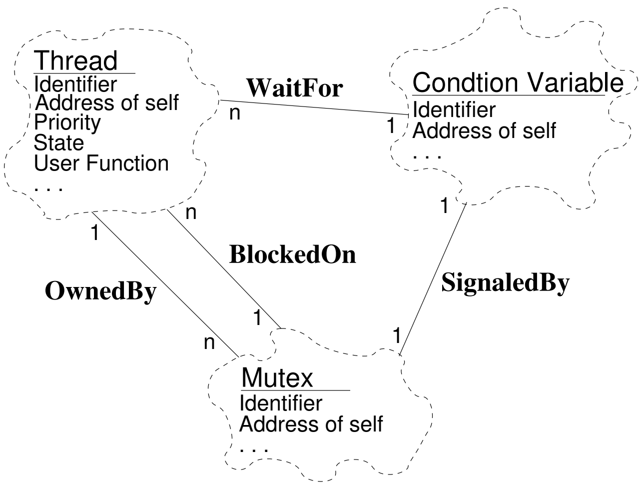


Figure 2: Booch Class Diagram of Object Classes

object classes since only scalar attributes are currently support by the TED domain T_A, *i.e.*, extensions to the TED would be required.

The open interface for debugging threads encompasses access functions for sets to iterate over its members and access functions for attributes of a member. Table 2 depicts the iterators and attribute functions and their operational description. Each interface function is registered with the TDI, which uses it to build a database of the application's state as explained in the next section. The domain of values returned by the iterators and functions is D_A and $\mathbb{N}\cup\{\text{NULL}\}$, respectively[3]. If a thread implementation supports only a subset of this functionality, it simply does not register the function. Invalid requests return NULL. Persistent objects are discussed in the next section.

4. THREAD DEBUG INTERFACE (TDI)

The objective of the TDI component is an abstraction from the thread implementation on one side and the debugger on the other side. The TDI keeps a database of the application's state. This approach supports the paradigm of active debugging. The database maintained by the TDI is only updated when the application changes its state wrt. multi-threading objects. The TED may register a set of operations that will inform the TDI of updates during the application's execution. Notice that this approach is unique to active debugging since passive debugging does not allow application-side execution of auxiliary operations. The TDI exports the following functions for the registration purpose:

```
int RegisterObject (RelT Rel, ObjRefT ObjRef);
int DeregisterObject (RelT Rel, ObjRefT ObjRef);
int IsRegistered (RelT Rel, ObjRefT ObjRef);
```

The signature of these operations includes the relation and an object of the same type (thread, mutex or condition variable). If the thread application supports the registra-

[3]with the following exceptions: `id` and `rstate` have a domain of $\mathbb{Z} \cup \{\text{NULL}\}$, `state` has a domain of {*undef=0, running, ready, blocked_m, blocked_c, blocked_t, blocked_s, blocked_o, exiting* } where the different blocking states refer to the cause of blocking: mutex, condition variable, timer, suspension (forced) and other.

Iterator	Description
GetFirstThread, GetNextThread	get thread from set
GetFirstMutex, GetNextMutex	get mutex from set
GetFirstCond, GetNextCond	get condition variable from set

Attribute	Description
for threads:	
id	process-persistent thread-ID
addr	address of the thread structure
prio	priority
state	execution state
rstate	implementation-dependent state
entry	address of thread's function
earg	address of function's argument
newpc	next program counter
sp	stack pointer
mbo	blocked on this mutex
cvwf	blocked on this condition variable
pid	ID of process executing the thread
for mutexes:	
id	process-persistent ID
addr	address of mutex structure
owner	mutex owner (thread ID)
for condition variables:	
id	process-persistent ID
addr	address of cond var structure
cmutex	associated mutex

Table 2: Open Interface for Debugging Threads (Iterators and Attributes)

tion process, it will invoke the corresponding functions, *e.g.*, when locking, unlocking and destroying a mutex. The TED functionality described in the open interface for debugging threads is also registered with the TDI. This allows the TDI to generically invoke TED operations to resolve database queries. The registration occurs *via* the following interface:

```
int SetIterFunc (RelT Rel,ObjRefT (*GetFirst)(),
    ObjRefT (*GetNext)(ObjRefT Last));
int SetAttrFunc (RelT Rel, AttrT Attr,
    AttrDomainT (*GetFunc) (ObjRefT Obj),
    AttrDomainT (*SetFunc) (ObjRefT Obj,
        AttrDomainT value));
```

The exchange between TED and TDI about the range of debugging support is further generalized by letting the TDI inform the TED upon activation that a number of functions are expected to be registered. This registration request of the TDI includes the list of attribute functions, iterators and registration procedures for all objects. The TED may then register a subset or all of these functions, depending on the range of support. This initial exchange only assumes a known layout of the data types to be registered and serves the portability of the involved software components.

The TDI handles the communication with the debugger in such a way that the debugger receives a consistent view of the multi-threaded objects in the application, which is discussed in section 7. This abstraction provides the means to support *persistent* identifiers, as seen in Table 2. A persistent identifier is a unique identifier assigned to an object for its life time. This provision circumvents problems rooted within thread implementations that recycle object identifiers. *E.g.*, the Pthreads API only defines a common interface including the signature of operations and their types. A thread object has a certain type but the meaning of the value is transparent, *i.e.*, a value may refer to a thread object A as long as it exists. Once A terminates, the value may be recycled to refer to thread B. In a passive debugging approach, threads A and B cannot be distinguished explicitly, it would be the user's responsibility to detect A's termination and infer that another thread with the same identifier truly refers to B. By active debugging, the TDI receives notice of the creation and termination of a thread through the registration procedure. This allows the TDI to assign its own values to identify objects and use these values to communicate with the debugger. The user is only exposed to the debugger, which provides the persistent identifiers and allows a distinction between threads A and B for active debugging.

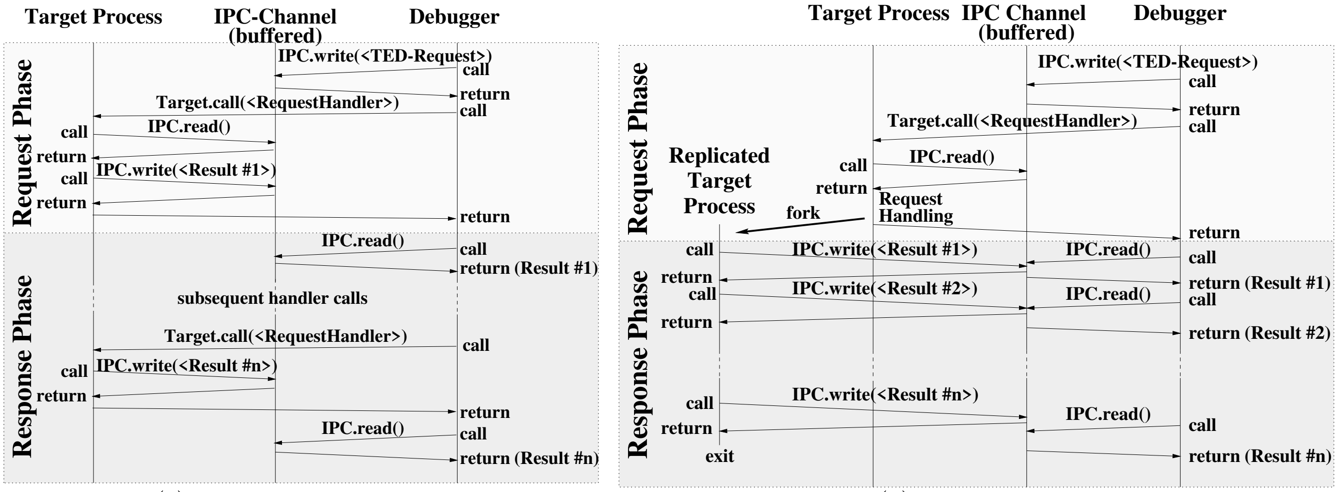
The actual queries for the database are issued in a uniform and extensible query language. Queries are defined according to a specification of a relational algebra. Each query is preceded by a mode that either refers to a TED query or a user-defined extension with its own operational framework. Queries can be selections and projections limiting the set of objects in question and defining the requested attributes, respectively. Each query includes a set of selections of

- a relation with values or
- projections of a relation with assignments.

The queries are resolved by a list of values corresponding to the projections in the query or by an error message. The results are a set of answers reduced to ensure that no duplicates are contained within the set. Furthermore, requests for the state of distinct objects can be clustered in one query and are optimized to remove redundancies. As a result, this declarative query interface performs better than a procedural interface where each request would require a separate function call by the debugger. An example is given in Section 7.

5. COMMUNICATION STRUCTURE

Breakpoint debugging is generally supported by a service of the operating system. This service, *e.g.*, the system call `ptrace` under UNIX, provides access to the trace of a process as depicted in Figure 3. The debugging process can *peek* or *poke* one word of the application process at a time. It may also *continue* in the execution of the application. The performance of the debugger is often constrained by the granularity of its data accesses, which will be quantified in Section 7. When the approach of active debugging is utilized, large amounts of data may be exchanged between the debugger and the application rendering the `ptrace` approach less efficient. The responses of the TDI for a query issued by the debugger may contain large amounts of data depending on the number of active multi-threaded objects in the application. Although queries are often much shorter, a symmetric approach was chosen. The queries issued by the debugger as well as the responses from the TDI are transmitted using inter-process communication (IPC).



(a) mutual exclusive execution

(b) parallel execution

Figure 4: Communication between Debugger and Application

Another problem is posed by the fact that the application process is stopped while the debugger is active (and vice versa). The debugger can only make progress when its queries are handled by the TDI, which is part of the application. This problem is solved by letting the debugger issue a call to a handler function within the application. A `ptrace` call for continuation activates the server side of the TDI, which receives the request, resolves the query and initiates the response. The response may contain a large amount of data that cannot be transfered in one buffer since the buffer length is generally constrained by the IPC mechanism. The debugger could issue repeated `ptrace` calls to receive one packet at a time but this would result in a large number of context switches of the debugger and the application process as a side effect of using `ptrace` (see Figure 4(a)). Instead, the application process forks a child upon long responses

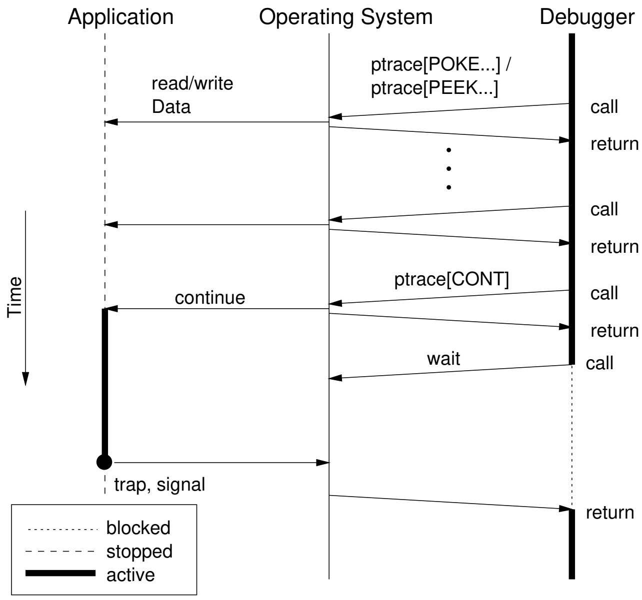


Figure 3: Breakpoint Debugging with Ptrace

(see Figure 4(b)). The child receives the responsibility to fill the IPC buffers before terminating while the debugger can receive the IPC packets in parallel.[4]

6. DEBUGGER EXTENSIONS

The debugger was extended in two respects. First, the IPC interface to the TDI was added. Second, new user commands to control the debugging process were included and their resolution was handed off to the TDI. The IPC extensions are bundled in one module, the TDI client, and can be bound with the debugger during the build process. The TDI client handles the client side of the IPC communication. The debugger may invoke send and receive functions of the TDI client to send a query and receive the response, in both cases as a string. After sending a query the application is continued (`ptrace` call) within the `TDI-Server`, the main function on the application side of the TDI. The TDI server evaluates the query, hands it off to the parser of the query language, which may update the state of the database using the TED interface. Once the response has been formatted, it is returned using IPC and the debugger can act upon the result. The second extension of the debugger defines a number of new user commands and their actions. This additional functionality is detailed in Section 8.

7. IMPLEMENTATION

The implementation comprises changes to the debugger and the threads implementation. The Gnu debugger GDB 4.18 was chosen for this purpose since the sources are available, it is widely used and actively maintained [19]. The chosen thread implementations range from kernel threads (LinuxThreads) [11] over mixed threads (Solaris) [16] to user-level threads (FSU and MIT Pthreads) [14, 17].

One of the challenges of active debugging is posed by the interaction between the activation of debugging operations

[4]Even on a uniprocessor, the child process and the debugger may run concurrently and do not require context switches for each packet anymore.

within the application and the regular execution of the application itself. The TDI server may be a separate thread for kernel threads while the server may simply be invoked in the context of the active thread for user-level threads. But this approach may result in scheduling actions due to

1. a skew of the consumed execution time of the TDI,
2. event notification or
3. calls to library functions that use synchronization.

When round-robin scheduling is active, additional execution time consumed by the TDI server may cause a context switch of the current thread. If the switch occurs before the TDI server finishes, the results obtained for debugging may be inconsistent. One part of the results may originate before the context switch and another part after the switch subject to a modified thread state since application threads had been active meanwhile. This problem will not only occur upon timer expiration but may also be caused by other signals. Context switches may also be caused by synchronization, in particular when the TDI server calls a library function whose entry is protected by a mutex. The mutex may already be locked by a thread in the application resulting in a context switch from the TDI server to the application thread. Even worse, a deadlock may occur if the application thread is the same thread that executes the TDI server.

First, the problem of calling library functions that contain synchronization was addressed by providing TDI-specific replacements for heap allocation and string manipulation. Other functions used by the TDI server do not contain potentially blocking library calls.

Second, the problem of signal handling during TDI activation shall be discussed. One solution would be to mask signals in the application for a limited time. But masking could only be accomplished by the application itself, which causes a race: A signal may arrive while the TDI tries to mask signals so that the TDI would loose control and other threads may be scheduled. The race can be avoided if the debugger forced the masking of signals for the application but most operating systems only provide such an interface for the *current* process and not for *another* process. Instead of an operating system interface, the thread implementation was enhanced to provide a flag that, when set, collects signals for later handling as depicted in Figure 5. The debugger uses the `ptrace` call to set the flag in the application (1). Incoming signals are collected but their handling is postponed during TDI activation. Once the TDI activities are complete, the debugger reads the collected signals (2), clears the flag and collected signals and sends each signal to the application (3).[5]

The implementation of the TDI server contains a communication subsystem, a query parser and a query evaluator. The communication structure was implemented *via* shared memory IPC between processes. The performance was evaluated by comparing the IPC variant using a page size of 8kB with a `ptrace` implementation using a 32 bit word size, both under Linux 2.0.36 on a 150MHz Pentium with FSU Pthreads. Figure 6 shows that the response time for `ptrace` is five times higher than the performance for IPC. The results underline the advantages of the IPC approach for the TDI communication.

The query parser was generated from lexical and syntactical specifications by the generators Flex and Bison, respectively. The parser reports errors for illegal queries or transforms legal ones into a tuple representation, which is then fed to the query evaluator. The evaluator may optimize the query, invoke TED functions to resolve the query and compile a response. Examples for a query may be as follows:

```
thread:id,entry,state:state == 1 || mbo == 0 (1)
thread:id,prio=10,state: (prio+10<20) && cvwf ! =0x10 (2)
```

Query (1) requests the identifier, state and function of all threads that are running or not blocked on a mutex. Query (2) requests the same information (except for the function) for threads whose priority plus 10 is less than 20 and who are not blocked on a condition variable (second conjunct).

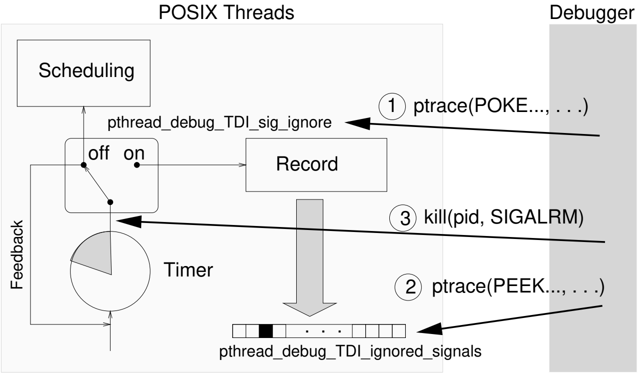


Figure 5: Signal Handling during Active Debugging

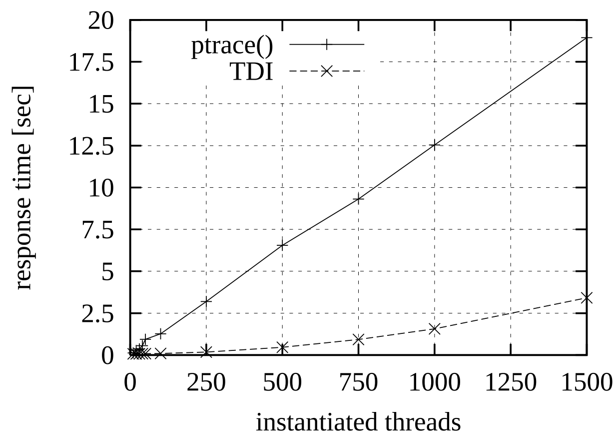


Figure 6: Response Times: IPC vs. Ptrace

[5]An alternative to re-issuing the signals would be to add them to the pending signals of the thread implementation and force a check on pending signals when resuming the application, which would have the advantage that signal contexts were preserved. Future work may include such a provision.

Query (2) also sets the priority of the selected thread to 10. A response to the debugger may be as follows:

```
8 804b238 2 #7 804b238 2 #10 804b238 1
```

The debugger interprets each of the three tuples as (identifier, function address, state) and translates addresses and states into symbolic names for its output.

The TDI server is combined with the application by dynamically linking the application with the TDI server relocatable library. This has the advantage that an application compiled for testing and debugging only has to be linked with the dynamic linker library using `-ldl`. The TDI will not be invoked or even linked when the application is executed outside the debugger. Once the debugger is invoked, it checks if the threads of the application contain a symbol to indicate debugging support for threads. If the the flag is found, it will be set by the debugger and results in dynamic binding and invocation of the TDI server library during the initialization phase. The debugger then sends a `pthread_TDI_register` message to the TDI server. The TDI presents the TED with the set of functions it expects, and the TED responds with the registration of attribute and iteration functions. Afterwards, the TDI may resolve queries by referencing thread objects through TED functions.

Thread-specific breakpoint debugging also requires changes to the debugger. In GDB, the routine *proceed* calling *normal_stop, wait_for_inferior* and *resume* control the trap handling for breakpoints. A command to *resume* execution activates the application. The debugger then *waits* for the inferior process (application) to hit a trap instruction. When the trap is hit, the debugger resumes control and cleans up its traces from the application's code in *normal_stop*. Thread-specific breakpoints modify this sequence by checking upon resumption of the debugger after *wait* if the breakpoint reached corresponded to the requested thread. If the thread identifiers match, *normal_stop* is called. Otherwise, the breakpoint is reset similar to the cleanup performed in *normal_stop* and *resume* is called again. The cleanup is depicted in Figure 7. When the application traps, the inserted trap instruction has to be replaced by the original instruction A of the application to ensure the correct semantics. Now, B has been replaced by a trap to transfer control to the debugger for resuming execution (2). Then, the trap is replaced by B while A is replaced by a trap to make sure that the application halts at the same breakpoint again the next time around. The execution in step (2) may, however, cause a scheduling action if a signal was received. This is prevented by disabling signals during step (2) using the facilities discussed before. Finally, the thread identifier of the active thread has to be determined at a breakpoint. For user-level threads, it suffices to search for the single running thread using a TDI query. For kernel threads, multiple threads may be running (on a multi-processor) but the system call *wait* returns information about the process that caused a trap. For mixed threads, the low-level scheduling entity has to be identified and it has to be ensured that parallel execution of a trap by different threads of one application results in serial notification of the debugger (on the operating system level). For example, Solaris maps POSIX threads onto light-weight processes (LWPs) whose status information would have to be checked upon encountering a trap. This requires that the LWP and its state for a POSIX thread is determined, for example, through the `/proc` file system. Single step commands, such as `ptstep, ptnext` (see next section), use a similar technique to only count steps executed by the current thread. Attaching and detaching threads also has a similar effect but includes thread-specific breakpoints for the program counter of the target thread. A breakpoint on the next context switch is realized by setting conditional breakpoints at the program counter of all threads except for the one that has trapped. Once such a breakpoint is hit, all other conditional breakpoints set before have to be deleted. Forcing a change in the scheduling pattern results in a TDI query that sets thread attributes and invokes a TED function to affect the scheduler of the thread implementation. A forced suspension also requires that the debugger signal the application. This ensures that the scheduler is invoked to dispatch the next thread eligible to run. This can also be achieved by adding a scheduler signal to the set of collected signals during signal masking, as discussed before.

8. THREAD-AWARE DEBUGGING

This sections describes commands that have been added or modified to make GDB thread aware and provide extended debugging support for multi-threaded applications. Notice that GDB already provides limited debugging support for selected thread implementations. The new commands for debugging threads have been chosen to coexist with the existing functionality. For example, **info threads** may already list the threads for Solaris, Mach and LinuxThreads. The new command **info pthreads** lists threads for any application supporting the TDI/TED facilities and includes extensive information about the state of each thread, the object it may be blocked on, priorities etc. Both commands are available at the same time.

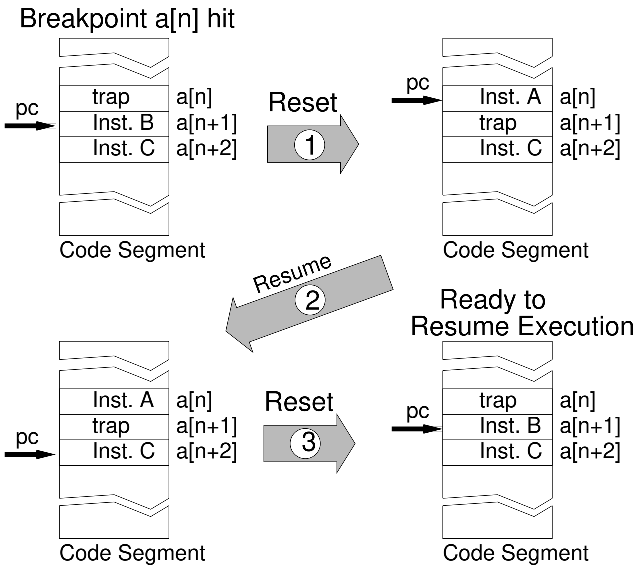


Figure 7: Resetting a Breakpoint

- **info pthreads** lists the set of threads that have not terminated yet including the attributes for threads depicted in Table 2.
- **info pmutex** lists the set of initialized mutexes with the attributes of Table 2.
- **info pcond** lists the set of initialized condition variables with the attributes of Table 2.
- **break <location>pthread <Thread ID>** sets a thread-specific breakpoint for <Thread ID>at <location>
- **ptattach <Thread ID>** stops <Thread ID>the next time it is scheduled at the first possible location and transfers control to the debugger. Once issued, all subsequent breakpoint commands (break, next, step) are thread-specific. This means that these breakpoints only apply to the attached thread while other active threads will not stop at these breakpoints.
- **ptdetach** reverses a ptattach and makes breakpoints applicable to all threads again.
- **ptstack <Thread ID>** prints the call stack of <Thread ID>.
- **continue -cs** continues the execution until a breakpoint is hit or a context switch occurs, which ever comes first. In the latter case, the identifier of the new thread is printed.
- **ptnext/ptnexti/ptstep/ptstepi** [**n**] issues `n` next or step instructions for the current thread and ignores other instructions executed by concurrently running threads.

These facilities go beyond traditional debugging support for threads in the following sense. The cause of blocking of threads and the blocking object can be identified. This may allow the user to identify deadlocks when circular dependencies between synchronization objects and threads are depicted.[6] It provides the user with the call stack of threads, *i.e.*, the user can follow the progress of concurrent executions. Thread-specific breakpoint debugging simplifies the user's task of tracing the execution of selected threads. Interactions with other threads can be detected by notification upon context switches. Finally, scheduling actions forced by the user allow selected activation and suspension to disable or force thread interactions, test their impacts and possibly track down problems between these interactions.

We also assessed the overhead of the active debugging support through TDI and TED. On the application side, overhead may be incurred by the TED. Two Splash-2 benchmarks [24] were measured (processes emulated by FSU Pthreads) on a Pentium II 350 MHz under Linux 2.2.14, as depicted in Figure 6. Fft performed calculations for 2^{20} data points. The numbers for Fft exclude initialization. Barnes used the standard parameters (except for 5 leaves), and the numbers reported represent the computation time, only. During the experiments, the number of processes (threads) was varied between 2 and 128 but this had no effect on the measurements. The overhead represents the portion of the second measurements that were due to active debugging. This overhead depends on the characteristics of the application, *e.g.*, fft uses less synchronization than barnes, which explains the lower overhead of the former. On the debugger side, the overhead of the TDI and of queries are not noticeable to the user, *i.e.*, the response time of TDI queries equals that of any other debugger interaction. However, if a large database is gradually built (thousands of threads, mutexes, etc.) then the response time of queries may be affected since all entries may be probed. We did not experience this problem in practice. Hence, relational queries seem suitable for active debugging.

[6]It may seem that automatic deadlock detection could be easily incorporated. This is true in the sense that the state of threads may not change in the absence of signals. The signal semantics of Pthreads does not provide such a stable state since signals may interrupt a synchronization request and then skip out of the synchronization call from the signal handler. For this reason, automatic deadlock detection has not been implemented.

Program	No Debugging	GDB-TDI	Overhead
fft	14 sec	16 sec	**12.5%**
barnes	33 sec	40 sec	**17.5%**

Table 3: Performance Overhead of Active Debugging

9. RELATED WORK

McDowell and Helmbold present an overview of the problems and solutions for debugging concurrent programs [13]. Ceswell and Black [5] describe a debugger for Mach threads with thread-specific breakpoints and forced scheduling actions. The approach is limited to kernel threads, uses a non-standard `ptrace` interface and is not as portable as the approach described in this paper. Ponamgi *et al.* [15] continue their work with this debugger by adding event handling to detect deadlocks, livelocks and multiple entry to critical sections. Similar support could be added to our work at the level of the TDI but is subject to the constraints described before, *i.e.*, certain thread standards may not allow deadlock detection due to signal handling. SmartGDB uses the non-generic design of Figure 1(a) where for each thread implementation the debugger has to be modified. Changes in a thread implementation may, in turn, require changes in the debugger. GDB 4.18 [19] requires even more modifications than SmartGDB for each thread implementation. SmartGDB and GDB 4.18 support only a subset of our functionality for debugging threads and still use the slow `ptrace` call for communication. Solaris utilizes the `/proc` file system to efficiently access internal structures of the application, which is an alternative to the communication used by TDI in terms of efficiency but provides neither portability for systems without `/proc` file system nor does it allow a generic encapsulation as seen in Figure 1(b) with its localization of implementation-dependent extensions. Solaris also provides a library for debugging threads with similar functionality as the TED interface but lacks the flexibility of the TDI, which makes our approach portable. Wismüller *et al.* [23] describe a tool set for debugging parallel programs consisting of a debugger (Partop) and a monitoring tool. Partop supports thread-aware debugging and uses the event-action paradigm that executes a certain action when an event occurs, *e.g.*, when a thread is created. A generalization of the event-action paradigm is provided by path expressions and path actions in the context of debugging [3]. In this work, path expressions were proposed at the user level, which few debuggers support today. Our work does, on one hand, uses similar concepts under the paradigm of active debugging. On the other hand, these concepts are utilized for *internal* purposes rather than at the user level, *i.e.*, as a means of communication between debugger components in a portable fashion. The high performance debugging forum (HPDF) specified a command interface for parallel debuggers [9] including thread-aware debugging. Our work does not specify a user interface but rather an interface to a thread library that may be utilized by a debugger. We also implement the thread-aware functionality of the HPDF interface and even go beyond these requirements, *e.g.*, by supplying additional functionality to display synchronization data or execute up to the next breakpoint. Cownie and Gropp [6] propose a debugger interface to display messages within MPI implementations and demonstrate this work in TotalView. Independent from our work, they also conclude that dynamic linking of shared libraries represents the most flexible way to provide debugging facilities for multiple runtime libraries. Panorama [12] is a parallel debugger for MIMD architectures that relies on text-based debuggers to collect information and visualize it in a predefined or a user-defined representation. Panorama's portability is given by its reliance on text-based debuggers at a lower level. Our work differs in that we provide such a text-based debugger extension that may be used by a visualization debugger like Panorama. Kessler [10] introduced fast breakpoints, which can be regarded as a variant on active debugging. Conditional breakpoints are realized by replacing the trap with a call to a debugging-specific handler in the application that checks the condition of the breakpoint and only traps if it evaluates to true, thereby improving performance. We use active debugging to resolve relational queries. KDB [4] supports two-level debugging of user and kernel threads with a unique design. Each kernel thread is controlled separately by a local debugger using the `ptrace` interface and the `/proc` file system. A main debugger interacts with the user and steers all local debuggers. Our work differs in that we require a TED interface for each thread level. Snodgrass [18] utilizes relational queries for monitoring by extending databases to keep histories of traces and providing a temporal operator to query these histories. Our relational queries involve on-line debuggers accessing computing states without histories rather than monitoring data. Overall, none of these tools use active debugging or relational queries for debugging threads nor do they support as much functionality as the work presented in this paper combined with portability at the same time.

10. CONCLUSION

This paper proposes an open interface for debugging as an extension to thread implementations. In addition, extensions for thread-aware debugging are identified and implemented within the Gnu Debugger to provide additional features beyond the scope of existing debuggers. The work is based on the paradigm of active debugging that includes a language-independent protocol to communicate between debugger and application via relational queries to ensure that the enhancements of the debugger are independent of actual thread implementations. Partial or complete implementations of the interface for debugging can be added to thread implementations to work in unison with the enhanced debugger without any modifications to the debugger itself. Sample implementations of the interface for debugging have shown its adequacy for user-level threads, kernel threads and mixed thread implementations while providing extended debugging functionality at improved efficiency and portability at the same time.

Availability

The modified debugger GDB-TDI (sources and binaries) and its documentation are available at http://www.informatik.hu-berlin.de/~mueller/TDI under the Gnu Public License.

11. REFERENCES

[1] R. Alfieri. An efficient kernel-based implementation of posix threads. In *USENIX Conference*, Summer 1994.

[2] F. Armand, F. Herrmann, J. Lipkis, and M. Rozier. Multi-threaded processes in CHORUS/MIX. In *EEUG Conference*, pages 1–13, Spring 1990.

[3] B. Bruegge and P. Hibbard. Generalized path expressions: A high level debugging mechanism. In *Software Engineering Symposium on High-Level Debugging*, pages 34–44, Aug. 1983.

[4] P. Buhr, M. Karsten, and J. Shih. KDB: A multi-threaded debugger for multi-threaded applications. In *Symposium on Parallel and Distributed Tools*, pages 80–87. ACM Press, May 1996.

[5] D. Caswell and D. Black. Implementing a Mach debugger for multithreaded applications. In *Winter USENIX Conference*, pages 25–40, Berkeley, CA, USA, Jan. 1990.

[6] J. Cownie and W. Gropp. A standard interface for debugger access to message queue information in MPI. In *6th European PVM/MPI Users' Group Meeting*, volume 1697 of *LNCS*, pages 51–58. Springer-Verlag, 1999.

[7] C. G. Davis. Testing large, real-time software systems. In *Software Testing, Infotech State of the Art Report*, volume 2, pages 85–105, 1979.

[8] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyond multiprocessing ... multithreading the SunOS kernel. In *USENIX Conference*, pages 11–18, Summer 1992.

[9] H. P. D. Forum. Command interface for parallel debuggers. Draft revision 2.1 for standard, The Parallel Tools Consortium, Sept. 1998. http://www.ptools.org/hpdf/draft.

[10] P. B. Kessler. Fast breakpoints. design and implementation. *ACM SIGPLAN Notices*, 25(6):78–84, June 1990.

[11] X. Leroy. The linuxthreads library. *http://pauillac.inria.fr/~xleroy/linuxthreads*, 1996.

[12] J. May and F. Berman. Retargetability and extensibility in a parallel debugger. *Journal of Parallel and Distributed Computing*, 35(2):142–155, June 1996.

[13] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, Dec. 1989.

[14] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, Jan. 1993.

[15] M. K. Ponamgi, W. Hseush, and G. E. Kaiser. Debugging multithreaded programs with MPD. *IEEE Software*, 6(3):37–43, May 1991.

[16] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *USENIX Conference*, pages 65–80, Winter 1991.

[17] C. Provenzano, G. Hudson, and K. Raeburn. Mit pthreads. *http://www.mit.edu/people/proven/pthreads.html*, 1993.

[18] R. Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[19] R. M. Stallman. GDB manual (the GNU source-level debugger). Technical report, Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, Jan. 1989. Third Edition, GDB version 3.1.

[20] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX Conference*, pages 1–10, Summer 1992.

[21] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].

[22] A. Tevanian, R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young. MACH threads and the UNIX kernel: The battle for control. In *USENIX Conference*, pages 185–197, Summer 1987.

[23] R. Wismüller, M. Oberhuber, J. Krammer, and O. Hansen. Interactive debugging and performance analysis of massively parallel applications. *Parallel Computing*, 22(3):415–442, Apr. 1996.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characteriation and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.