Improving Reactivity to I/O Events in Multithreaded Environments Using a Uniform, *Scheduler-Centric* API

Luc Bougé[2], Vincent Danjean[1], and Raymond Namyst[1]

[1] LIP, ENS Lyon, 46 allée d'Italie, F-69364 Lyon Cedex 07, France
[2] PARIS Project, IRISA/ENS Cachan, Campus Beaulieu, F-35042 Rennes, France

Abstract. Reactivity to I/O events is a crucial factor for the performance of modern multithreaded distributed systems. In our *scheduler-centric* approach, an application detects I/O events by requesting a *service* from a *detection server*, through a simple, uniform API. We show that a good choice for this detection server is the thread scheduler. This approach simplifies application programming, significantly improves performance, and provides a much tighter control on reactivity.

1 Introduction

The widespread use of clusters of SMP workstations for parallel computing has lead many research teams to work on the design of portable multithreaded programming environments [1,2,3]. A major challenge in this domain is to reconcile *portability* with *efficiency*: parallel applications have to be portable across a wide variety of underlying hardware, while still being able to exploit much of its performance. Most noticeably, much effort has been focused on performing efficient communications in a portable way [4,5], on top of high-speed networks [6,7]. However, a major property has often been overlooked in the design of such distributed runtimes: the *reactivity* to communication events. We call *reactivity* of an application its ability to handle external, asynchronous events as soon as possible within the course of its regular behavior. The response time to a network event is indeed a critical parameter, because the observed *latency* of messages directly depends on it: if the application is not reactive enough, the observed external latency can be arbitrary larger the nominal, internal latency offered by the underlying communication library. For instance, all communication primitives including a back and forth interaction with a remote agent (e.g., to fetch some data) are extremely sensitive to the reactivity of the partner [8].

Berkeley's Active Messages Library [9] provides a good reactivity to network events. However, the communication system is highly dependent on the hardware, and it has only been implemented on specific message passing machines. Princeton's Virtual Memory Mapped Communication Library [10] can offer a good reactivity. However, once again, these mechanisms are highly hardware-dependent and need specific OS modifications or extensions. Our goal is not to

propose yet another powerful I/O or communication library. Instead, we intend to design a *generic* approach, allowing to use already *existing* I/O libraries in a multithreaded environment, so that we can *ensure a good reactivity*.

An application may use several strategies to detect I/O events. The most common approach is to use *active polling* which consists in checking the occurrence of some I/O events by repeatedly calling an appropriate function of the I/O subsystem. Such an elementary test is usually inexpensive, with an overhead of a few assembly instructions. However, repeating such a test millions of times may exhaust computing resources in a prohibitive way. Alternatively, the application can rely on *passive waiting*, using blocking system calls, signal handlers, etc. In this latter case, I/O events are signaled to the operating system by hardware interrupts generated by the I/O device, which makes the approach much more *reactive*. However, catching such an interrupt is usually rather costly, of the order of tenth of microseconds, disregarding the additional latency of rescheduling the application.

Usually, the choice of the I/O detection strategy is made within the application. This results in mixing *application-specific* algorithmic code with *system-dependent* reactivity management code. Moreover, this approach suffers from several severe drawbacks:

Determining the available methods. The operating system (i.e., the underlying I/O driver) may only offer a restricted set of methods. In some cases, only a single explicit polling primitive may be provided to the user. In other cases, handling interrupts may be the only way to check the completion of I/O operations. In this latter situation, the operating system may even provide no other choice but a single mechanism to handle interrupts. Moreover, complexity and portability requirements may often prevent the use of some mechanisms. For instance, raw asynchronous delivery of signals imposes hard reentrance constraints on the whole application code, if the consistency of *all* data structures accessed within signal handlers has to be guaranteed.

Selecting the right one. When several methods are available at the OS level, selecting the most appropriate one depends on many factors. A key factor is the level at which the thread scheduler is implemented. Actually, there are many flavors of thread schedulers (user-level, kernel-level, hybrid) and each of them features its own characteristics as far as its interaction with the operating system is concerned. For instance, in the context of a pure user-level thread scheduler, operations such as blocking system calls, are usually prohibited, except if some sophisticated OS extensions (such as *Scheduler Activations* [11,12,13]) are available. Even hybrid schedulers, which essentially implement a user-level scheduler on top of a kernel-level one, suffer from this limitation.

Tuning for performance. Most I/O subsystems (i.e., device drivers) natively provide a low-overhead polling mechanism. However, efficiently using such a mechanism is a difficult challenge in a multithreaded context [14,15]. As for monothreaded applications, the polling frequency has a crucial impact on the overall application performance. If the I/O subsystem is not polled

frequently enough, then the application reactivity may become severely altered. In contrast, a too aggressive polling policy leads to many unproductive polling operations, which wastes computing resources. Even if the optimal frequency can be predicted in advance, it may be difficult to instrument the application to effectively enforce it. Actually, those threads waiting for the completion of some I/O event, loop over a sequence of instructions: each iteration consists in a polling operation, followed by a `thread_yield` instruction in case the operation failed.

The contribution of this paper is to introduce a new approach to the problem of reacting to I/O events in multithreaded environments. We define it as *scheduler-centric*. In our view, the environment should provide the application with a *uniform* paradigm for reactivity management. The actual selection of the strategy, active polling and/or passive waiting, is then left to the scheduler. This allows to centralize all the reactivity-management mechanisms within the scheduler, thereby relieving the programmer from this difficult task. Moreover, this enables the scheduler to adjust its scheduling strategy with the reactivity level required by the applications, independently of the system load. Finally, it allows to aggregate multiple requests issued by concurrent applications to the same NIC, resulting in more efficient interactions. We demonstrate the feasibility of this new approach in the context of a the user-level thread scheduler of the PM^2 multithreaded, distributed environment. Significant performance gains are observed.

2 Our Proposition: a *Scheduler-Centric* Approach

We propose to *centralize* the management of I/O events at a single point within the scheduler, providing the application with a *uniform* mechanism to wait for the completion I/O events. Instead of making I/O completion detection an explicit part of the algorithmic code of the application, we view such an action as a *event detection service* requested by the application to an *external server*, namely the scheduler. The client thread is removed from the running list while waiting for the completion of the service. It is the task of the scheduler to determine the very best way of serving the request: polling, interrupt handling, etc., or any kind of dynamic, adaptive mix of them, and to return the control to the requesting thread.

2.1 Serving the I/O Event Detection Requests

We propose to let the thread scheduler serve the I/O event detection requests for several reasons. First, it is system-aware, in contrast with an application whose code has to be portable across various systems. Thus, the scheduler has full freedom to safely use all mechanisms provided by the OS, including the most sophisticated ones. For instance, a pure user-level thread scheduler "knows" that

it is dangerous to invoke a system call that may potentially block the process, except when there is only one single active thread in the system. Furthermore, if some form of asynchronous mechanism is available, then the thread scheduler can provide signal-safe synchronization primitives to the threads waiting for I/O events, while providing regular and fast ones to other threads.

Second, the scheduler is probably the best place where *efficient* polling can be done. In particular, the specific frequency of polling for each requesting thread can be precisely mastered by the scheduler, as it can hold all relevant information, and an optimal decision can possibly be made at each context switch. Also, the scheduler can maintain for each request type a history of previous requests, so as to select the most efficient mechanism: a possible strategy should be to first actively poll and then switch to passively wait for a NIC interrupt after some time. Also, the scheduler can use the otherwise idle time to perform intensive polling if this has been given a high priority. Finally, the scheduler enjoys full freedom regarding the next thread to schedule: it can thus schedule a thread as soon as the corresponding I/O event has been detected.

Third, the scheduler appears thereby as the single *entry point* for event detection requests. This provides an interesting opportunity to *aggregate* the event detection requests issued by various threads. For instance, if several threads are waiting for a message on the same network interface, then there is no need in having all of them polling the interface: the scheduler can aggregate all the requests and poll the interface on their behalf; once an event has been detected, then it can lookup its internal requests tables to determine which thread is to be served. Observe that this aggregation ability is fully compatible with the other aspects listed above: one can well use a mixing of active polling and passive waiting in detecting common events for multiple I/O requests! Thus, our proposal generalizes the `MPI_testany()` functionality of MPI, to any kind of event detection request, using any kind of communication interface.

2.2 A Uniform API to Request Event Detection

We have designed the programming interface so as to insulate the application from the idiosyncrasies of the specific events under detection. The general idea is that the client application, most often a communication library, should register the specific callback functions to be used by the scheduler in serving its requests.

The application has first to register which kind of events it is intended to detect, and how, into the scheduler. This is done by filling the fields of a structure `params` with a number of parameters: callback functions to poll for the events and to group requests together, objective frequency for polling, etc. The `thread_IO_register` primitive returns a *handle* to be used for any subsequent request. Only requests issued with the same handle may be aggregated together.

```
thread_IO_t  thread_IO_register  (thread_IO_registration_t params);
```

Client threads are provided with a *single* primitive to wait for the occurrence of an I/O event. The `thread_IO_wait` primitive is a blocking one (for the caller

thread). If needed, asynchronous I/O event detection can be achieved in multithreaded environment by creating a new thread to handle the communication. Argument `arg` will be transmitted to the previously registered, callback functions, so that these functions can get specific data about the particular request. The scheduler itself does not know anything about these functions. This primitive returns from the scheduler as soon as possible after an event is ready to be handled.

```
void  thread_IO_wait  (thread_IO_t IO_handle, void *arg);
```

For example, registering a *polling routine* and issuing an asynchronous receive for MPI would look like:

```
thread_IO_registration_t MPI_params;
thread_IO_t MPI_handle;
...
MPI_params.blocking_system_call = NULL;
MPI_params.group = &MPI_group();
MPI_params.poll = &MPI_poll();
MPI_params.frequency = 1;

MPI_handle=thread_IO_register(&MPI_params);
```

```
MPI_Request request;
MPI_IO_info_t MPI_IO_info;
...
MPI_Irecv(buf, size, ..., &request);

MPI_IO_info.request = request;
thread_IO_wait(MPI_handle,
       (void *) &MPI_IO_info);
```

3 Implementation Details

We implemented our generic mechanism within the "chameleon" thread scheduler of the PM^2 multithreaded environment [3] which can be customized to use any of the following scheduling flavors: *user-level* or *hybrid*. Our mechanism is virtually able to deal with a very large number of *scheduling flavors/device driver capabilities* combinations. We focus below on the most common situations.

3.1 Active Polling

A number of callback functions are needed for the scheduler to handle polling efficiently. They are passed to the scheduler through the `params` structure. If the I/O device interface allows it, then the function assigned to the `group` field should be able to aggregate all the requests for this device. Otherwise, a `NULL` pointer should be specified for this field. This function is called each time a new request is added or removed with respect to the given handle. The `poll` field holds the function which effectively does the polling job. This function should return `-1` if no pending event exists, or the index of a ready request if there is any. Furthermore, a few other parameters have to be specified in the `params` structure including a `frequency` integer, which stores the number of time slices between each polling action. Thereby, various I/O devices can be polled with different frequencies, even though they are all accessed through the same interface. Figure 1 displays a skeleton of a callback `poll` function for the MPI communication interface, which actually generalizes the `MPI_Testany` primitive of MPI.

```
MPI_Request MPI_requests[MAX_MPI_REQUEST];
int MPI_count;

typedef struct { MPI_Request request; } MPI_IO_info_t;

void MPI_group(void) {
  MPI_IO_info_t *MPI_info;
  MPI_count=0;
  thread_IO_for_each_request(MPI_info) {  /* Macro iterating on
                                             pending requests */
    MPI_requests[MPI_count++] = MPI_info->request;
  }
}

int MPI_poll(void) {
  int index, flag;
  MPI_Testany(MPI_count, MPI_requests, &index, &flag, ...);
  if (!flag) return -1;
  return index;
}
```

Fig. 1. Polling callback functions in the case of a MPI communication operation.

3.2 Passive Waiting

The end of a DMA transfer generates an interrupt. Most network interface cards are able to generate an interrupt for the processor when a event occurs, too. Because the processor handles interrupts in a special mode with kernel-level access, the application can not be directly notified by the hardware (network card, etc.) and some form of OS support is needed. Even when communication systems provide direct network card access at the user level (as specified in the VIA [16] standard for example), the card needs OS support to interrupt and notify a user process. Indeed, hardware interruption cannot be handled at user-level without losing all system protection and security. The simplest way to wait for an interrupt from the user space is thus to use blocking system calls. That is, the application issues a call to the OS, that suspends it until some interrupt occurs. When such blocking calls are provided by the I/O interface, it is straightforward to make them usable by the scheduler. The `blocking_system_call` field of the `params` structure should reference an intermediate application function, which effectively calls the blocking routine.

Note that I/O events may also be propagated to user space using Unix-like *signals*, as it is proposed by the POSIX *Asynchronous I/O* interface. When such a strategy is possible, our mechanism handles I/O signals by simply using the aforementioned polling routines to detect which thread is concerned when such a signal is caught. Threads waiting for I/O events are blocked using special *signal-safe* internal locks, without impacting the regular synchronization operations performed by the other parts of the application.

3.3 Scheduler Strategies

A main advantage of our approach consists in selecting the appropriate method to detect I/O events *independently* of the application code. Currently, this selection is done according two parameters: the flavor of the thread scheduler, and the range of methods registered by the application.

When the thread scheduler is entirely implemented at the user level, the active polling method is usually selected, unless some specific OS extensions (such as *Scheduler Activations* [11]) allow the user-level threads to perform blocking calls. Indeed, this latter method is then preferred because threads are guaranteed to be woken up very shortly after the detection of the interrupts. The same remark applies to the detection method based on signals, which is also preferred to active polling.

Two-level hybrid thread schedulers, which essentially run a user-level scheduler on top of a fixed pool of kernel threads, also prevent the direct use of blocking calls by application threads. Instead, we use a technique which uses specific kernel threads that are dedicated to I/O operations. When an application user thread is about to perform an I/O operation, our mechanism finds a new kernel thread on top of which the user thread executes the call. The remaining application threads will be left undisturbed, even if this thread gets blocked. Note that these specific kernel threads are idle most of the time, waiting for an I/O event, so little overhead will be incurred. Also, observe that the ability to aggregate event detection requests together has a very favorable impact: it decreases the number of kernel-level threads, and therefore alleviates the work of the OS.

Observe finally that all three methods (active polling, blocking calls and signals handling) are compatible with a kernel-level thread scheduler.

4 Experimental Evaluation

Most of the ideas of this paper have been implemented in our multithreaded distributed programming environment called PM^2 [3] (full distribution available at URL `http://www.pm2.org/`). First, we augmented our thread scheduler with our mechanism. It allows the applications to register any kind of event detected by system calls or active polling. (Support for asynchronous signals notification has not been implemented yet.) Then, we modified our communication library so that it uses the new features of the scheduler. At this time, MPI, TCP, UDP and BIP network protocols can be used with this new interface. Various platforms are supported, including Linux i386, Solaris SPARC, Solaris i386, Alpha, etc.

The aim of the following tests is to assess the impact of delegating polling to the scheduler, and of aggregating similar requests. They have been run with two nodes (bi-Pentium II, 450 MHz) over a 100 Mb/s Ethernet link. The PM^2 library provides us with both a user-level thread scheduler, and a hybrid two-level thread scheduler on top of Linux, so that it allows using blocking system calls. All durations have been measured with the help of the *Time-Stamp Counter* of

x86 processors, allowing for very precise timing. All results have been obtained as the average over a large number of runs.

4.1 Constant Reactivity wrt. Number of Running Threads

A synthetic program launches a number of threads running some computation, whereas a single server thread waits for incoming messages and echoes them back as soon as it receives them. An external client application issues messages and records the time needed to receive back the echo. We list the time recorded by the client application with respect to the number of computing threads in the server program (Table 1).

	# Computing threads				
Scheduler version	None	1	2	5	10
Naïve polling (ms)	0.13	5.01	10.02	25.01	50.01
Enhanced polling (ms)	0.13	4.84	4.83	4.84	4.84
Blocking system calls (ms)	0.451	0.453	0.452	0.457	0.453

Table 1. Reaction time for a I/O request wrt. the number of computing threads.

With our original user-level thread library, with no scheduler support, the listening server thread tests for a network event each time it is scheduled (*naïve polling*). If no event has occurred, then it immediately yields control back. If n computing threads are running, a network event may be left undetected for up to n quanta of time. The quantum of the library is a classical 10 ms, so $10 \times n/2$ ms are needed to react on average, as shown on the first line of Table 1.

With the modified version of the thread library, the network thread delegates its polling to the user-level scheduler (*enhanced polling*). The scheduler can thus control the delay between each polling action, whatever the number of computing threads currently running. The response time to network requests is more or less constant. On average, it is half the time quantum, that is, 5 ms, as observed on the results.

Using blocking system calls provides better performance: we can observe a constant response time of 450 μs whatever the number of computing threads in the system. However, a two-level thread scheduler is needed to correctly handle such calls.

4.2 Constant Reactivity wrt. Number of Pending Requests

A single computing thread runs a computational task involving a lot of context switches, whereas a number of auxiliary service threads are waiting for messages on a TCP interface. All waiting service threads use a common handle, which uses the `select` primitive to detect events. An external client application generates a random series of messages. We report in Table 2 the time needed to achieve the computational task with respect to the number of auxiliary service threads.

	# waiting service threads							
Scheduler version	1	2	3	4	5	6	7	8
Naïve polling (ms)	80.3	101.3	119.0	137.2	156.6	175.7	195.2	215.7
Enhanced polling (ms)	81.2	84.0	84.0	84.7	86.4	87.9	89.6	91.6

Table 2. Completion time of a computational task wrt. the number of waiting service threads.

This demonstrates that aggregating event detection requests within the scheduler significantly increases performance. Without aggregation, the execution time for the main task dramatically increases with the number of waiting threads. With aggregation, this time remains constant, although not completely, as the time to aggregate the requests depends in this case on the number of requests.

5 Conclusion and Future Work

We have proposed a generic *scheduler-centric* approach to solve the delicate problem of designing a portable interface to detect I/O events in multithreaded applications. Our approach is based on a uniform interface that provides a synchronous event detection routine to the applications. At initialization time, an application registers all the detection methods which are provided by the underlying I/O device (polling, blocking calls, signals). Then, the threads just call a unique *synchronous* function to wait for an I/O event. The choice of the appropriate detection method depends on various complex factors. It is entirely performed by the implementation in a transparent manner with respect to the calling thread.

We showed that the right place to implement such a mechanism is within the thread scheduler, because the behavior of the I/O event notification mechanisms strongly depends on the capabilities of the thread scheduler. Moreover, the scheduler has a complete control on synchronization and context-switch mechanisms, so that it can perform sophisticated operations (regular polling, signal-safe locks, etc.) much more efficiently than the application.

We have implemented our scheduler-centric approach within the PM^2 multithreaded environment and we have performed a number of experiments on both synthetic and real applications. In the case of an active polling strategy, for instance, the results show a clear improvement over a classical application-driven approach.

In the near future, we intend to investigate the use of adaptive strategies within the thread scheduler. In particular, we plan to extend the work of Bal *et al.* [14] in the context of hybrid thread schedulers.

References

1. Briat, J., Ginzburg, I., Pasin, M., Plateau, B.: Athapascan runtime: Efficiency for irregular problems. In: Proc. Euro-Par '97 Conf., Passau, Germany, Springer Verlag (1997) 590–599
2. Foster, I., Kesselman, C., Tuecke, S.: The Nexus approach to integrating multithreading and communication. Journal of Parallel and Distributed Computing **37** (1996) 70–82
3. Namyst, R., Méhaut, J.F.: PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In: Parallel Computing (ParCo '95), Elsevier (1995) 279–285
4. Aumage, O., Bougé, L., Méhaut, J.F., Namyst, R.: Madeleine II: A portable and efficient communication library for high-performance cluster computing. Parallel Computing **28** (2002) 607–626
5. Prylli, L., Tourancheau, B.: BIP: a new protocol designed for high performance networking on Myrinet. In: Proc. 1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98). Volume 1388 of Lect. Notes in Comp. Science., Springer-Verlag (1998) 472–485
6. Dolphin Interconnect: SISCI Documentation and Library. (1998) Available from `http://www.dolphinics.no/`.
7. Myricom: Myrinet Open Specifications and Documentation. (1998) Available from `http://www.myri.com/`.
8. Prylli, L., Tourancheau, B., Westrelin, R.: The design for a high performance MPI implementation on the Myrinet network. In: Proc. 6th European PVM/MPI Users' Group (EuroPVM/MPI '99). Volume 1697 of Lect. Notes in Comp. Science., Barcelona, Spain, Springer Verlag (1999) 223–230
9. von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active messages: a mechanism for integrated communication and computation. Proc. 19th Intl. Symp. on Computer Architecture (ISCA '92) (1992) 256–266
10. Dubnicki, C., Iftode, L., Felten, E.W., Li, K.: Software support for virtual memory mapped communication. Proc. 10th Intl. Parallel Processing Symp. (IPPS '96) (1996) 372–381
11. Anderson, T., Bershad, B., Lazowska, E., Levy, H.: Scheduler activations: Efficient kernel support for the user-level managment of parallelism. In: Proc. 13th ACM Symposium on Operating Systems Principles (SOSP '91). (1991) 95–105
12. Danjean, V., Namyst, R., Russell, R.: Integrating kernel activations in a multithreaded runtime system on Linux. In: Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00). Volume 1800 of Lect. Notes in Comp. Science., Cancun, Mexico, Springer-Verlag (2000) 1160–1167
13. Danjean, V., Namyst, R., Russell, R.: Linux kernel activations to support multithreading. In: Proc. 18th IASTED International Conference on Applied Informatics (AI 2000), Innsbruck, Austria, IASTED (2000) 718–723
14. Langendoen, K., Romein, J., Bhoedjang, R., Bal, H.: Integrating polling, interrupts, and thread management. In: Proc. 6th Symp. on the Frontiers of Massively Parallel Computing (Frontiers '96), Annapolis, MD (1996) 13–22
15. Maquelin, O., Gao, G.R., Hum, H.H.J., Theobald, K.B., Tian, X.M.: Polling watchdog: Combining polling and interrupts for efficient message handling. In: Proc. 23rd Intl. Symp. on Computer Architecture (ISCA '96), Philadelphia (1996) 179–188
16. von Eicken, T., Vogels, W.: Evolution of the Virtual Interface Architecture. IEEE Computer **31** (1998) 61–68