RAP - Resource Adaptive Programming with an application to robust and fast file copying

Utkarsh Upadhyay
IIT Kanpur
utkarshu@iitk.ac.in *

Nikolaj Bjørner
Microsoft Research
nbjorner@microsoft.com

Abstract

Network file copy programs are prime examples of system applications that use various resources: CPU, disk and network. The resources used in a local-area network, where network throughput can be higher than disk throughput, are incompatible with resources used when copying files over a high-latency, low-bandwidth remote network. A customary solution to scaling network applications is by using asynchronous programming, but the model is not directly amenable to chatty exchanges. We here provide a programming abstraction based on resource adaptive thread-pools as a layer for network and disk resources. The layer is encapsulated using *workflows* (monads) in the F# programming language; and we report on experiments with *RoboClone*, a parallelized version of the Windows RoboCopy program. In addition to the resource adaptive layer, RoboClone also includes the remote differential compression (RDC) protocol.

1 Introduction

The advent of multi-core processors has renewed attention to using parallelism for utilizing all available cores on computation intensive applications. Yet, systems software is rarely solely CPU bound; resources such as networks and disks are prolific bottlenecks in most system applications. The importance of utilizing parallelism is no less in these contexts, but applications will have to adapt to a heterogeneous set of resources, each with different capabilities. The situation of utilizing these resources is aggravated by additional two factors: operating system layers provide abstraction boundaries, many of which make it difficult for applications to take advantage of the underlying resources optimally, and main-stream programming language constructs are not tuned to automatically and transparently allocate system resources, such as threads, when they are useful.

Our main quest is designing and building abstractions to take optimal advantage of system resources. Ideally, the abstractions should allow applications to be resource agnostic. We will address this partially in the following by examining a *weighted thread-pool* layer and a particular application.

A good example of an application that use various resources are network file copy programs. They access disk to read and write files; the content of files are sent over a network with different latencies and bandwidths, and when the copy programs use compression algorithms, they also rely on the CPU cores as a set of resources. Often, systems software addresses these resources only as an afterthought. As a case in point, a substantial improvement between Microsoft Outlook 11 and Outlook 12 was the use of asynchronous remote procedure calls instead of synchronous remote procedure calls in Outlook 11.

A customary solution to scaling network applications is by using asynchronous programming paradigms [17] [11], [1]. Asynchronous programming facilitates using operating system thread resources in useful ways while network or file system calls are pending. Asynchronous programming alone does not solve the problem of scaling protocols between chatty, but bandwidth sparse, versus non-chatty, but bandwidth intensive phases.

*This work was performed while interning at Microsoft Research

We here provide a programming abstraction based on resource adaptive thread-pools as a layer for network and disk resources. The layer is encapsulated using *workflows* (monads) in the F# [19] programming language; and we report on experiments with *RoboClone*, a parallelized version of the Windows RoboCopy program [6]. The use of monads for encapsulating threading or asynchronous programs is not new, for instance, run-times for asynchronous programming in the context of network applications have been pursued in [13][22], but to our knowledge these approaches do not consider adapting the workload according to physical constraints, rather these approaches use monads for program layering. While RoboClone is a research vehicle for resource adaptive programming, it does offer an API that is compatible with RoboCopy. Network administrators or every day users can therefore substitute RoboCopy by RoboClone in the same environment with minimal change in setup.

1.1 Workflows

Workflows is a programming paradigm that is supported in F#. Similar to traditional monadic programming, known from programming language theory [14], [21], VDM [15], Opal [10] and Haskell [4], they allow encapsulating various side-effects. In particular, it is possible to use workflows to encapsulate asynchronous programming. Asynchronous programming features are well known to address resource optimization in the context of high-latency network systems. Standard asynchronous programming methodologies require carefully organizing all control flow to yield when reaching blocking procedure calls and requires carefully organizing code for callbacks. It is easy to overlook blocking calls during coding and it may be very difficult to re-organize code to introduce context switches afterwards. Workflows have the prospect of guaranteeing that context switches are performed on blocking calls while allowing the impression of a sequential programming style.

1.2 RoboCopy

RoboCopy [6] is a widely used program for robustly copying files over faulty networks. RoboCopy manages download resumption and makes it easy to administrate and retrieve copy job summaries. RoboCopy uses the SMB [18] protocol for copying files. The SMB protocol is exposed transparently over the APIs used for accessing files locally or remotely. For instance, `FileCopyEx`, `Backup Read/Write`, `Set/Get FileTime/FileAttributes` use either local file system calls, or perform networked calls over SMB when the files that are accessed reside on remote machines. RoboCopy works well in the field, but the version released with Windows Server 2008 is subject to some bottlenecks that cause administrators of large copy jobs to look for alternatives:

1. Most of the Win32 APIs used by RoboCopy translate to synchronous remote procedure calls. In scenarios with high latency, such synchronous calls form a bottleneck, which is noticeable since RoboCopy runs on a single thread.

2. Even if a file changes slightly from one copy to another, FileCopyEx, or Backup Read/Write will not be able to take advantage of the pre-existing data on the destination[1]

We would like to stress that the first limitation with RoboCopy is not specific to it. Bottlenecks due to blocking calls appear in several applications that use network protocols; and engineering these applications to being resource adaptive can require a substantial effort.

[1]This limitation is somewhat specific to RoboCopy. Microsoft offers products, such as DFS-R, that minimize the amount of data copied over when files are changed only slightly.

1.3 Parallelism and Concurrency

Note that we used the terminology parallelism and not concurrency: we use parallelism when multiple threads *cooperate* in accomplishing a task, we use concurrency when threads *compete* to accomplish each their task. Concurrency control is important, also in the context of parallelism, but it is not the prevalent problem. The main objective with parallelism is maximizing the use of available resources.

2 F# workflows

F# contains facilities for building custom workflows. We will build a *continuation passing style* (CPS) workflow. It can be viewed as a special case of the `Async` monad in F# [19]. The CPS workflow encapsulates a basic type argument `'a` into a continuation. That is, it wraps `'a` into the type `('a -> unit) -> unit`. F# requires to supply the basic primitives `Bind`, `Return`, `Let` and `Delay` in order to construct a workflow. These primitives are inserted by the compiler when using the custom workflow syntax. Listing 2.1 shows the described CPS workflow.

Listing 2.1 A CPS workflow

```
module CPS =
  type 'a prim = P of (('a -> unit) -> unit)

  let Primitive f = P f

  let RunWait (P pfn) =
    let doneE =
      new EventWaitHandle(false, EventResetMode.ManualReset)
    pfn (doneE.Set >> ignore);
    ignore (doneE.WaitOne())

  let RunCont (P pfn) cont = pfn cont

  type CPSBuilder() =
     member p.Bind((P pfn), f) =
      P (fun cont -> pfn (fun a -> let (P g) = f a in g cont))
     member p.Return(x) = P (fun cont -> cont x)
     member p.Let(x, f) = f x
     member p.Delay(f) =  f ()

  let cps = new CPSBuilder()
```

It contains two methods for invoking the workflow with a continuation. The function `RunWait` invokes the workflow with a continuation that sets an event. It waits for the continuation to be called. This allows using the workflow in a multi-threaded context. The function `RunCont` just applies the workflow to a continuation that is passed in.

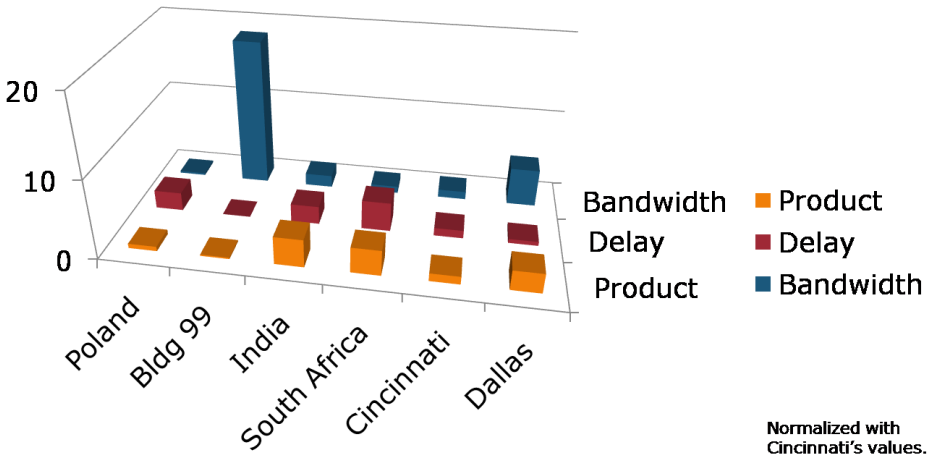

Figure 2: Delay-bandwidth numbers sampled from Buildling 99, Microsoft, Redmond

3 The delay-bandwidth product

The capacity of a network is characterized by the so-called delay-bandwidth product. This product, which is the time for a round-trip multiplied by the number of bytes sent per time unit, indicates how a sender or receiver should scale their send and receive buffers. This situation is illustrated in Figure 1, where data is transmitted from a sender on the left to a receiver on the right. The sender (and receiver) should maintain a buffer of size at least $delay \times bandwidth$ in order to allow the transmission control protocol (TCP) to utilize the available bandwidth maximally while also re-sending data until acknowledgment is received.

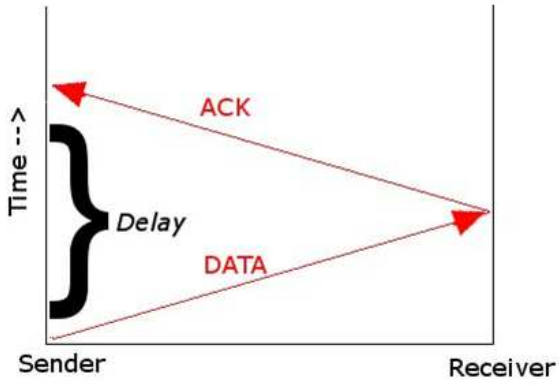

Figure 1: Network round trip delays

Figure 2 shows a number of delays and bandwidths and their products for samples taken from Buildling 99, Microsoft, Redmond. Observe how the delay-bandwidth product compares between Dallas and South Africa: the higher bandwidth to Dallas compensates for the lower delay, resulting in a similar product.

3.1 Modeling the effect of system delays

An application utilizes the available network resources optimally if it is able to fill up the send and receive buffers on both parties. Modern implementations of the TCP/IP stack will automatically tune these buffers to fit the effective delay-bandwidth product, but leaving it up to applications to take advantage of this. The operating system utility `FileCopyEx` does scale the number of bytes copied during round-trips over SMB, but this only has an effect for large files. Whenever copying over files and directories whose size is smaller than the network buffers, this advantage goes away. To summarize, we have the two cases:

- Chatty communication: small files but round-trip intensive. The application will be responsible for filling up the send/receive buffers.
- Transfers of large files: Existing file copy functions, `FileCopyEx`, scales the send/receive buffer internally according to network resources.

Based on these observations we assign a *weight* to each file copy activity. The weight is set to the number of bytes in the file. Assuming the underlying protocol can fill up the send-buffer with enough bytes copied in parallel, the job of the thread-pool is to spawn precisely a number of threads that is sufficient for filling up the send-buffer. This gives as estimate, the number of threads that can be spawned for concurrent copy tasks, as:

$$\begin{aligned} network\_buffer &= delay \times bandwidth \times congestion && (1)\\ file\_data &= min(network\_buffer, max(packet\_size, file\_size)) && (2)\\ \#threads &\leq \frac{network\_buffer}{file\_data} && (3) \end{aligned}$$

But this estimate is not accurate. It does not take into account the time it takes to retrieve data from disk. If we assume that each thread incurs an additional disk seek based on the data they access, the seek time may dominate the equation if we allocate too many threads.

Specifically, we can also provide a rough bound on the number of threads by the ratio of the delay with the disk seek time.

$$\#threads \leq \frac{delay}{seek\_time} \quad (4)$$

We will be using the two bounds from (3) and (4) in the following when controlling the number of spawned threads. However, these inequations can be improved further by considering these possibilities together as follows.

We know that the seek time on the disk depends on the number of threads trying to simultaneously read from the disk and the sizes of the files the threads are reading. The average disk seek time is 12ms for a 5,400 rpm disk while the network latency in a closely knit LAN might be as low as 1ms. Also, a number of active threads which might result in frequent thrashing of the disk, thereby increasing the effective seek time. Figure 3 illustrates this situation. The throughput is measured on writing files of size 1, 2 and 4 MB, using between 1 and 31 threads. The throughput clearly decreases as the number of threads competing to write to the same disk increase.

Hence, the ideal number of threads would be that when the amount of data being sent over the network does not have to wait at all in the Sender's TCP buffer. In other words, let $s(n)$ be the seek time used by n threads to retrieve $file\_data_i$ from files $1, \ldots, n$, then $|s(n) - delay|$ should be kept at a minimum too. Combining this with the condition that the amount of data being sent should at any time be close to the $delay \times bandwidth$ product, we have the following condition to give us the ideal number of threads:

$$\min_n \left(|\sum_{i=1}^{n} file\_data_i - delay \times bandwidth| + \lambda |s(n) - delay| \right) \quad (5)$$

where:

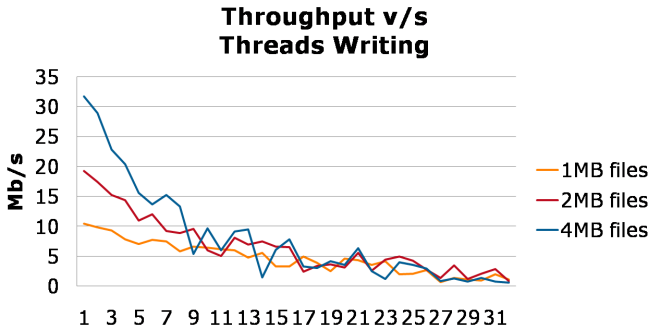

Figure 3: Throughput by number of threads writing to disk

n	is the number of threads
$\{file\_data\}$	is the ordered set of all file data (defined above)
$s(n)$	can be set to $seek\_time \times n + \sum_{i=1}^{n} \frac{file\_data_i}{disk\_transfer\_rate}$
λ	is a Lagrangian multiplier

However, the expression suffers from various implementation related drawbacks. Foremost, the determination of exactly how the $seek\_time$ depends on its parameters that can only be roughly estimated. Hence, arriving at a value of the multiplier λ is practically difficult.

3.2 Measuring system delays

We here summarize how our prototype implementation measures system delays.

3.2.1 Measuring bandwidth

We measure bandwidth simply by maintaining a wall-clock of the time copying started. We also maintain the total number of bytes copied. This number is updated every time a file copy completes, and the bandwidth is set to the total number of bytes divided by value of the wall-clock.

3.2.2 Measuring delay

Delay is estimated by sampling some of the file system calls made by RoboClone. While SMB2 may use a different number of round-trips for some of the calls, the average number of round-trips will be constant. We have found it useful to use the following calls for estimating the round-trip time:

- `FindFirstFile`, used to start enumerating the contents of a directory. This calls requires at least 2 round-trips over SMB.
- `FindNextFile` continues directory enumeration. It requires either no or at least one round-trip. It could be the case that the call requires no round-trip. This is the case when the result is already cached on the caller. The latency will then be below a millisecond. On the other hand, when the result is not cached, the round-trip latency will be noticeable.
- `GetFileInformationByHandle` retrieves file information given a file handle. It requires at least one round-trip.

3.2.3 Measuring disk seek time

Magnetic disk technology has not changed much in the last few decades. Standard manufacturing characteristics of disks remain yielding disks operating at 5400-10K RPM (revolutions per minute). This translates to a maximal seek time in the area of 5-10ms. Our approach is therefore currently to fix the seek time on all systems to 10ms. The physics of solid state disks are of course completely different, and they are not covered by this rough model.

3.3 Real TCP scaling

Applications that interact with a disk and network will be subjected to how these layers perform their own tuning. There are several proposed and adapted extensions and algorithms for TCP:

- TCP implementations typically use Nagle's algorithm [16], which provides congestion control for TCP connections. If existing requests have not yet been acknowledged, the algorithm buffers up small outgoing packets on the sender's side. Applications can set the `TCP_NODELAY` bit on packets to force transmission.
- TCP window scaling [12] algorithm allows senders and receivers automatically scaling their send/receive buffer based on observed network delay and bandwidth. The Windows TCP stack did not implement such tuning until Vista. In contrast, in Vista, *the TCP/IP stack contains an auto-tuning mechanism that dynamically changes the advertised receive window based on the characteristics of the path over which the connection is made. Auto-tuning helps improve TCP throughput over high bandwidth-latency paths by not artificially limiting the throughput [9].* In pre-Vista operating systems, network administrators have to set a registry key called `TCPWindowSize` to manually configure the receive and send buffers [7].
- Slow start, congestion avoidance, fast retransmit, and fast recovery algorithms [3]; and initial window size choices [2] exemplify some of the algorithms implemented by TCP stacks to utilize network resources optimally in a setting where network resources change dynamically.

3.3.1 Is the quest attainable?

Since automatic window tuning was only made available in Vista, it was not possible to provide truly automatic adaptive programming layers on Windows platforms. Other obstacles remain in different low-level layers. For example, data sent over RPC are fragmented into buffers of a machine- and implementation- dependent maximal size. While the RPC standard allows implementations to negotiate a fragment size based on mutual capabilities [20], the maximal size is typically chosen relative to how much kernel memory is statically allocated to RPC buffers. In Windows 2000 and later operating systems, one can observe by sniffing the network traffic that RPC buffers are at most 5840 bytes [8]. As the `TCP_NODELAY` bit gets set on every such fragment, it inherently limits what applications can leave to synchronous RPC traffic to tune. Microsoft's implementation of RPC (known as MS-RPC) also offers asynchronous pipes. The asynchronous pipes are not limited by fixed buffer size constraints.

4 A Weighted Thread-pool

Section 2 described a continuation based workflow, and Section 3 described the delay-bandwidth product and how it affects how many threads should be spawned in parallel in order to take advantage of the available network resources. We here put these together in a *weighted thread-pool*, which dynamically adjusts the number of allocated threads according to which network and disk resources are used.

4.1 Parameters

We use a class of parameters to encapsulate the delay-bandwidth product as it gets converted into an upper bound on the admissible weights.

Listing 4.1 Delay-bandwidth parameters

```
type Parameters(b, d, seek, congestion, threads) =
 class
  let mutable delay      = d
  let mutable bandwidth  = b
  member p.update d bw =
     delay <- d; bandwidth <- bw
  member p.max_weight  with get () =
     uint64(bandwidth * delay * congestion / 1000.0)
  member p.max_threads with get () =
     min threads (max 1. (delay/seek) |> int)
 end
```

The parameters maintained are

- `delay` - the round-trip delay in milliseconds.
- `bandwidth` - an estimate of the network bandwidth in bytes per second.
- `seek` - estimated disk seek time.
- `congestion` - a congestion factor. The congestion factor indicates to which extent we let RoboClone over-commit the send-buffer with data. Over-committing the send-buffer causes potentially threads to block during network communication, but also allows us to scale the network communication upwards should the available network bandwidth fluctuate.
- `threads` - the maximal number of threads we are willing to spawn within an application. It can be set based on processor capabilities.

From these parameters, we can compute the maximal number of bytes RoboClone is allowed to push over the network at any given time. It is calculated as the delay-bandwidth product multiplied by the congestion ratio. Since the delay was given in milliseconds and the bandwidth used bytes per second, the calculation includes a division by 1000. Similarly, the maximal number of threads that can be spawned is bounded by the ration of the delay and seek time.

4.2 ThreadPool utilities

The weighted thread-pool maintains two counters: *num_threads*, which counts the number of threads currently spawned, and *total_weight*, which carries the current sum of weights that are assigned. We also summarize utilities that update these counters and check for whether a new weight (and possibly also a new thread) can be added. For example `can_fork` checks whether a thread may be spawned to run a task with a given weight. It calls `check_weight1`, which checks if the given weight plus the current total weight does not exceed the maximal weight. It also checks if the current number of running threads, plus one, does not exceed the maximal number of threads. The `fork` utility encapsulates spawning threads. It increments the current weight, and current number of threads (`can_fork` holds). It then creates a background thread. Background threads do not prevent an application from shutting down.

The queue class is defined in Appendix A.

Listing 4.2 Weighted thread-pool utilities

```
type WeightedThreadPool(param : Parameters) =
  class
    let token = ref 0
    let lock fn = lock token fn
    let task_queue = new queue<(uint64 * (unit -> unit))>()
    let mutable num_threads = 0
    let mutable total_weight = 0UL
    let inc_thread () = num_threads <- num_threads + 1
    let dec_thread () = num_threads <- num_threads - 1
    let check_weight1 w = w + total_weight <= param.max_weight
                          || total_weight = 0
    let check_weight (weight, task) = check_weight1 weight
    let can_fork w = check_weight1 w &&
                     num_threads < param.max_threads
    let inc_weight w = total_weight <- total_weight + w
    let dec_weight w = total_weight <- total_weight - w
    let fork weight fn =
        inc_weight weight;
        inc_thread();
        let thread = new Thread(ThreadStart(fn))
        thread.IsBackground <- true;
        thread.Start()
```

4.3 Thread-pool core

The core of the thread-pool is given in listing 4.3, and we will here describe the main methods. The main method for activating a given `task` with a `weight` is to call the `run` method.

- `run` calls `run_locked`, which is executed within a lock. We here assume for simplicity that the weight passed to `run` is below the maximal allowed weight.
- `run_locked` checks if the thread-pool can fork a thread given the `weight`. If this is the case, then a thread gets forked with the `task`, otherwise the task and weight are en-queued into a queue.
- `run_task` gets invoked within a thread. It executes the task, then calls `run_next_locked` within a lock.
- `run_next_locked`, decrements the weight associated with a task that was just executed. It then attempts to deque one task to run on the current thread, and if successful, it also tries to fork additional threads to run tasks using `fork_more_tasks`.
- `fork_more_tasks` attempts to deque tasks from the task queue as long as there are a sufficient number of spare threads and a sufficient amount of weight credits available.

We therefore see that the weighted thread-pool allows to dynamically adjust the number of threads that are spawned based on an estimate of the workload and an estimate of which resources are consumed. The thread-pool abstracts from the fact that it is used for network and disk resources.

Listing 4.3 Weighted thread-pool core

```
let rec run_locked weight task =
    if can_fork weight then
      fork (weight, fun () -> run_task weight task)
    else
      task_queue.Enqueue ((weight, task))

and run_task weight task =
    task ();
    match lock (fun () -> run_next_locked weight) with
    | None -> dec_thread ()
    | Some (weight, task) -> run_task weight task

and fork_more_tasks () =
    if can_fork 0 then
      match task_queue.TakeIf check_weight with
      | None -> ()
      | Some (weight, task) ->
        fork (weight, fun () -> run_task weight task);
        fork_more_tasks ()

and run_next_locked weight =
    dec_weight weight;
    match task_queue.TakeIf check_weight with
    | None ->  None
    | Some (weight', task) ->
      inc_weight weight';
      fork_more_tasks ();
      Some (weight', task)

member this.run weight task =
    lock (fun () -> run_locked weight task)
```

4.4 Parameter updates

The parameters described in listing 4.1 maintain a current estimate of the delay and bandwidth. These estimates may need to be adjusted during a copy operation, so while the runtime that spawns new threads query the parameters for the maximally allowed weight and maximally allowed number of threads, we also need to provide feedback into the parameters to adjust the the delay and bandwidth parameters as they are observed or change. Listing 4.4 contains a method in the `WeightedThreadPool` class that takes the same lock that gets held when the parameters are read, and updates the parameters with updated delays and bandwidth estimates.

Listing 4.4 Updating weights

```
   member this.update_weight bw delay =
     lock ( fun () -> param.update delay bw)
```

4.5 Task parallelism

Listing 4.5 contains the two methods for spawning thread-pool tasks. The method `do_spawn` taks a list of tasks, which are closures of type `unit -> unit CPS.prim`, paired with weights. Each task is run potentially in parallel.

Listing 4.5 Spawning parallel threads with continuations

```
member this.do_spawn tasks =
  CPS.Primitive (fun cont ->
    if tasks = [] then
      cont()
    else
      let count = ref (List.length tasks)
      let cont1 () =
         if 0 = Interlocked.Decrement count then cont()
      List.iter (fun (task, w) ->
         this.run w (fun () -> CPS.RunCont (task()) cont1)) tasks
  )

member this.run_wait asy = CPS.RunWait asy
```

5 Applying the thread-pool

We are finally in a position where we can illustrate the use of the weighted thread-pool and encapsulation in F# workflows.

Listing 5.1 illustrates how the weighted thread-pool can be used in the context of a file copy program. At the heart of a file copy program as a recursive walk over the directory structure of the source. Files and directories that are found at the source are copied over to the destination. The structure suggests that file copying can be easily parallelized: files can be copied independently, and as soon as a parent directory has been copied, then all sub-directories and files residing under the parent can be copied.

In the context of RoboClone, we found one case that required concurrency control. After each file copy operation, the attributes, the security settings and the time stamp of the parent directory needs to be updated. This task can cause a sharing violation if multiple threads attempt to write to the parent directory at the same time. Hence, a lock had to be placed on the parent directory prior to copying these settings.

6 Experience

We conducted several experiments with various incarnations of the RoboClone prototype. We used a pair of machines, one in Redmond, Washington, USA, and the other in Warshaw, Poland. The observed delay varied between 200ms and 250ms. Our main working set consisted of a few thousand files files taken from a standard Windows Server 2008 installation. It comprised a mixed set of directories and files of various sizes (from 20KB and up to 10MB). Preliminary results were encouraging. The prototype implementation offered speedups over RoboCopy ranging from 2x (for a mix of larger files) to 7x (for a mix of smaller files).

Listing 5.1 Parallel walk

```
type files = | File of string | Dir of string * (files list)

let acc_file acc f =
    match f with | File n -> acc n | _ -> ()
let file_walk acc name dir_list =
    List.iter (acc_file acc) dir_list

let acc_dir  acc f =
    match f with | Dir (n,fs) -> acc n fs | _ -> ()
let dir_walk acc name dir_list =
    List.iter (acc_dir acc) dir_list

let copy name = ...

let walk_main name dir_list =
 let p = Parameters(...) in
 let pool = new WeightedThreadPool(p) in
 let rec walk name dir_list =
   CPS.cps {
     let tasks  = ref []
     do  copy name
     do  dir_walk  (acc_d tasks) name dir_list
     do  file_walk (acc_f tasks) name dir_list
     do! pool.do_spawn !tasks
     return ()
     }
 and walk_file file =
     (fun () -> CPS.cps {return copy file}, sz file)
 and acc_f list file =
     list := (walk_file file)::!list
 and walk_dir dir dir_list =
     (fun () -> walk dir dir_list, sz dir)
 and acc_d list dir dir_list =
     list := (walk_dir dir dir_list)::!list
 pool.run_wait (walk name dir_list)
```

7 RDC - Remote Differential Compression

In addition to experimenting with a parallelism, we also added the remote differential compression protocol (RDC) to RoboClone. The RDC protocol allows compressing files over low-bandwidth networks. It does so by using a previous incarnation of a similar file to negotiate which file data needs to be retrieved from a remote server.

RDC - Remote Differential Compression is a protocol used for compressing files over the network by negotiating which file data to send by using checksums from a file. The MS-RDC DLL is distributed with Windows Vista [5]. One of the features of MS-RDC is the ability to identify similar files with the purpose of aiding the remote differential compression.

8 Conclusions

Our grand goal is to make it easier to write systems software that utilizes resources in ways that adapt to their availability. It is much easier to write such software using standard sequential programming paradigms, but these paradigms do not directly work well when the application needs to be optimized for network and disk utilization. One can of course write software from scratch to use asynchronous I/O, whenever possible, but even in the context of an application that uses asynchronous I/O there will inevitably be blocking system calls. Opening a file, is one such, or more subtly, page faults may appear under global locks. We here proposed an approach based on weighted thread-pools, encapsulated in F# workflows, to optimize an core network application: a file copy program. Our experimental evaluation shows that such an approach offers a realistic run-time layer for our application domain.

References

[1] Acme Labs. thttpd - tiny/turbo/throttling HTTP server. http://www.acme.com/software/thttpd/.

[2] M. Allman, S. Floyd, and S. Partridge. Increasing TCP's Initial Window. http://tools.ietf.org/html/rfc3390.

[3] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control, 1999. http://tools.ietf.org/html/rfc2581.

[4] Koen Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9(3):313–323, 1999.

[5] Microsoft Corporation. Remote Differential Compression. http://msdn.microsoft.com/en-us/library/aa373254$VS.85$.aspx.

[6] Microsoft Corporation. RoboCopy. http://www.microsoft.com/downloads/details.aspx? familyid=9d467a69-57ff-4ae7-96ee-b18c4790cffd&displaylang=en.

[7] Microsoft Corporation. TCP Receive Window Size and Window Scaling. http://msdn.microsoft.com/en-us/library/ms819736.aspx.

[8] Microsoft Corporation. Windows 2000 startup and logon traffic analysis. http://technet.microsoft.com/en-us/library/bb742590.aspx.

[9] Microsoft Corporation. Windows Vista Developer Story. Network Infrastructure: Overview. http://msdn.microsoft.com/en-us/library/bb756985.aspx.

[10] Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. Opal: Design and implementation of an algebraic programming language. In Jürg Gutknecht, editor, *Programming Languages and System Architectures*, volume 782 of *Lecture Notes in Computer Science*, pages 228–244. Springer, 1994.

[11] Duane Wessels. *Squid: The Definite Guide*. O'Reilly and Associates, January 2004.

[12] Van Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance, 1992. http://www.ietf.org/rfc/rfc1323.txt.

[13] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 189–199. ACM, 2007.

[14] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[15] Peter D. Mosses. VDM Semantics of Programming Languages: Combinators and Monads. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 483–503. Springer, 2007.

[16] John Nagle. Congestion Control in IP/TCP Internetworks, 1984. http://rfc.net/rfc896.html.

[17] Douglas C. Schmidt and James C. Hu. Developing flexible and high-performance web servers with frameworks and patterns. *ACM Comput. Surv.*, 32(1es):39, 2000.

[18] SMB2, 2008. url:http://msdn2.microsoft.com/en-us/library/cc246482.aspx.

[19] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2008.

[20] The Open Group. DCE 1.1: Remote Procedure Call. http://www.opengroup.org/.

[21] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[22] Chris Waterson. An ocaml-based network services platform. In *CUFP '07: Proceedings of the 4th ACM SIGPLAN workshop on Commercial users of functional programming*, pages 1–2, New York, NY, USA, 2007. ACM.

A The queue class

Listing A.1 A queue implementation

```
let match_list f list =
  match list with [] -> None | x::xs -> f x xs

type 'a queue() = class
  let mutable xs  : 'a list = []
  let mutable rxs : 'a list = []
  let norm() = if xs = [] then (xs <- List.rev rxs; rxs <- [])
  let matchl f = norm(); match_list f xs
  member q.IsEmpty() = xs = [] && rxs = []
  member q.Enqueue(x) = rxs <- x :: rxs
  member q.ToList() = List.append xs (List.rev rxs)
  member q.Take()  = matchl (fun y ys -> xs <- ys; Some y)
  member q.Size() = norm(); List.length xs
  member q.TopPeek () = matchl (fun y ys -> Some y)
  member q.TakeIf pred =
    norm();
    match xs with
    | y::ys when pred y -> xs <- ys; Some y
    | _ -> None
 end
```