Switching Supports for Stateful Object Remoting on Network Processors

Chung-Kai Chen, Yu-Hao Chang, Yu-Tin Chen, Chih-Chieh Yang and Jenq-Kuen Lee
Department of Computer Science, National Tsing-Hua University, Taiwan
Email:{ckchen, yhchang, ytchen, ccyang}@pllab.cs.nthu.edu.tw
jklee@cs.nthu.edu.tw

***Abstract*— Distributed object-oriented environments have become important platforms for parallel and distributed service frameworks. Among distributed object-oriented software, .NET Remoting provides a language layer of abstractions for performing parallel and distributed computing in .NET environments. In this paper, we present our methodologies in supporting .NET Remoting over meta-clustered environments. We take the advantage of the programmability of network processors to develop the content-based switch for distributing workloads generated from remote invocations in .NET. Our scheduling mechanisms include stateful supports for .NET Remoting services. In addition, we also propose scheduling policy to incorporate work-flow models as the models are now incorporated in many of tools of grid architectures. The result of our experiment shows that the improvement of EFT is from 5% to 21% when compared to ETT and is from 8% to 34% when compared to RR while the stateful task ratio is 50%. Our schemes are effective in supporting the switching of .NET Remoting computations over meta-cluster environments.**

I. Introduction

Distributed object-oriented environments have become important platforms for parallel and distributed service frameworks. Among distributed object-oriented software, .NET Remoting provides a framework that allows objects to interact with each other across the boundaries. In the .NET Remoting Framework, channels are used to transport messages to and from remote objects, and the .NET Remoting infrastructure provides two types of channels that can be used to provide a transport mechanism for the distributed applications - the TCP channel and HTTP channel. TCP channel is a socket-based transport that utilizes the TCP protocol for transporting the serialized message stream across the .NET Remoting boundaries while HTTP channel utilizes the HTTP protocol for transporting the serialized message stream across the Internet and through firewalls. As other networking applications, such as HTTP, server cluster is deployed for serving tremendous request. To leverage the request load among servers and optimize the cluster utilization, it is necessary to apply a load balancing mechanism in server clusters.

With the initial deployments of component services on grid environments, there are more exciting and challenging issues ahead in the runtime aspects of optimizations for component architectures on grids. Meanwhile, the arrival of network processors provides aggregate computation and I/O bandwidth. It looks promising to explore possible runtime optimization and paradigms for addressing the issues with the deployments of network processors. Among IXP platforms for network processing, IXP 1200 provides 6 micro-engines for packet switching, IXP 2400 provides 8 micro-engines for packet optimizations, and IXP 2800 provides 16 micro-engines to deliver performance of OC 192 (10 Gb/S). Interesting application aspects of runtime component switching are given below. For example, there are currently four software frameworks for grid services known as Java RMI, CCA service [6], .NET remoting, and OGSA remoting. For software compatibility and the re-use of resources of different frameworks, it should be interesting to explore runtime transcoding among different services. Second, it comes the idea of runtime adaptations of software components and re-configurations of systems to respond to enviroment changes and service traffics [2], [3]. As the network processor is a good candidate for gateways in terms of switching speeds and bandwidth, it is interesting to see how the software can work for network processors to address these issues. Note that network processors are with heterogeneous computing engines and memory hierarchies. Issues remain open on how to utilize network processors effectively for the switching of high-level applications. Finally, we need the load-balancing dispatcher for component services for serving tremendous request. In addition, a combination of three scenarios above might be possible. In our research work, we are studying issues with such scanarios by exploring the capabilities of network processors [4]. As a first step toward this idea, we try out with the issue of load-balancing dispatchers for .NET remoting services with network processors in this work.

In this paper, we address the issues in supporting

.NET Remoting over meta-cluster environments. We take the advantage of the programmability of network processors to develop the content-based switch. Stateful supports for .NET Remoting services are also incorporated. Our work has .NET Remoting applications classified into two separate channels in one application, one is for stateful, and another is for stateless. We then try to dispatch jobs for stateless applications, and also for the scheduling of stateful invocations. In addition, we also incorporate work-flow models for tasks to be scheduled into our frameworks. This is due to many of the tools of grid architectures now are with work-flow model supports [6]. In the first step of our scheduling policy, we perform scheduling policy for statful jobs in the work-flow models. With the initial placements of processor allocations, we then perform the scheduling policy for stateless applications in the second phase. Timeout constraints for stateful tasks are incorporated so that it might roll back processor assignments for stateful tasks during the second phase. This mechanism gives load-balancing for stateless tasks while also performs load-balancings of stateful tasks when the timing constraints are met. Our work, to our best knowledge, is the first work to address issues in supporting .Net remoting services for both stateful and stateless methods with network processor supports.

The rest of this paper is organized as follows. Section II presents presents the frameworks for meta-cluster supports for .NET Remoting with the assistance of IXP network processors. Next, Section III presents load-balancing schemes for work-flow models. Experimental results are then presented in Section IV. Finally, Section V concludes this paper.

II. Efficient Switching Support for .NET Remoting

For meta-cluster supports with .NET remoting, the workload dispatcher is generally needed. Loading balancing mechanism is divided into centralized [5] [1] and distributed [7] versions. We focus on the centralized version in our work. The centralized mode installs a gateway in front of the cluster. The gateway parses incoming request and makes appropriate routing decisions according to specific request attribute (such as source IP address and URL) and server workload feedbacks. The bottleneck for the .NET remoting dispatchers often occurs in the gateway because it needs high computation power to process a huge number of remoting requests. In addition, if the application is stateful, the gateway will consume additional cost to keep the coherence of sessions. We demonstrate how to distribute workloads of .NET remoting with the assistance of IXP 1200 network processors.

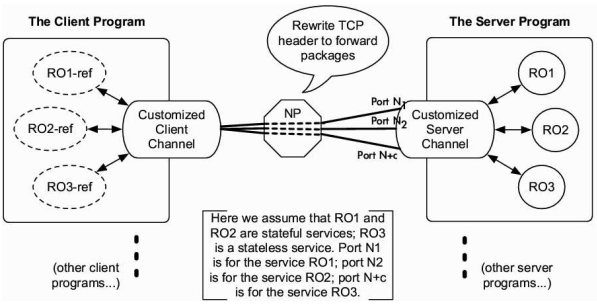


Fig. 1. The system architecture of using network processors as the remoting service gateway.

A. Remoting Switch

We take advantage of the programmability of network processor to develop the content-based switch. Figure 1 shows the system architecture of our design. The network processor `NP` serves as the gateway of remoting services hosted on each backend servers. All TCP channel connections of remoting going to the servers are brokered by the network processor. It uses its special hardware architecture to do fast TCP/IP header rewriting for directing packets back and forth. A TCP connection table is maintained in the memory space of the network processor to keep track of the connection information. It includes the IP and port information of the client and the connected server for each connection.

As a gateway of the backend servers, the job of `NP` is to dispatch remoting invocations concerning the load-balancing issues and the session semantics. For stateless remoting services, `NP` chooses the least load server to dispatch invocations; for stateful remoting services, `NP` has to make sure that invocations belonging to the same session will be dispatched to the same server. In Figure 1, `RO1`, `RO2` and `RO3` are all remoting objects that contain the intended operations for remote invocations. The `RO1-ref`, `RO2-ref` and `RO3-ref` in the client side are the TransparentProxy objects referring to `RO1`, `RO2` and `RO3` respectively. Both the proxy object and the remoting object use a channel object to manage network connections for data transportation. In this system, we design and deploy a pair of extended channel objects to automatically distribute remoting invocations into different TCP connection ports according to their service types. By doing this, `NP` can identify the service types through the examination of the destination port of incoming request packets. On the distribution of services on different ports, we use a map data structure to record the assigned port for each remoting service. All stateless services are bound to the port number large than `c`, where `c` is a selected constant. This map information can be a part

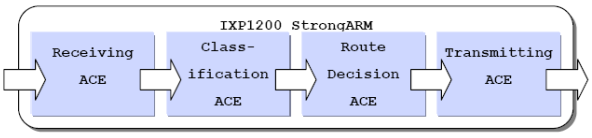


Fig. 2. A packet processing flow organized by ACEs.

of the remoting service deployment configurations and is accessible by the clients and the servers. We describe the distribution mechanism done by the channel objects below.

- **Client Channel Object** When the `SyncProcessMessage` or `AsyncProcessMessage` method of the client channel object is called in order to start a remoting invocation, it analyzes the parameter `IMessage` object to fetch the remoting service name. The mapped port for that remoting service is looked up by the map and is used for sending request packets.
- **Server Channel Object** When the server channel object is first instantiated, it looks up the map for all the currently used ports for remoting services. Then it opens corresponding server sockets on these ports to listen to connections.

B. Programming Network Processors

In the following, we introduce the framework use to programming the network processors. Intel provides a set of tool, IXA (Intel Exchange Architecture)SDK, to develop IXP1200 application which contains toolchains for both StrongARM and Microengine. To encapsulate and modularize individual tasks in the packet processing flow, IXA SDK define a type of software block, Active Component Element(ACE), constructed by C/C++ language. Each ACE performs its own task and forwards the packet(or drops) to the next ACE in the chain. We bind the ACEs together using targets to define the packet processing flow of the application as Figure 2 shows. An ACE is a normal C program encapsulated by IXA SDK. The ACE structure was defined as Figure 3 shows. Function ix_init() and ix_fini() were left for ACE programmer to override. After an ACE starts and performs internal initialization, the main program calls the programmer's ix_init function to complete the initialization. The main program then enters an event loop. The Intel_event_loop function will dispatch the exception and timer event to their corresponding handler function which was also written by programmer. Once the event loop terminates, the code calls the programmer's ix_fini function to release resources. Finally, the main program performs internal cleanup and exits.

```
Main() {
    Intel_init();
    //core component of an ACE
    ix_init();
    //Perform internal initialization
    Intel_event_loop();
    //Call user's initialization
    ix_fini();
    //perform internal event loop
    Intel_fini();
    //Call user's termination function
    exit();
    //Terminate the Linux process
  }

Intel_event_loop(){
   do forever{
     E = getnextevent();
     if(E is termination event)
      Return to caller;
     else if (E is exception event)
      Call exception handler function;
     else if (E is timer event)
      Call timer handler function;
    }
  }
```

Fig. 3. Conceptual structure of the main program in an ACE

The whole system implementation is divided into two parts, one is the control system executed in StrongARM core and the other one is the data path system executed in microengines. The control system is implemented in ANSI C code; its feature includes downloading the microcode to microengines, maintaining the related tables in SRAM and SDRAM, and determining the routing path for new remoting request. Figure 4 gives a code segment for rewriting the packet header. The functions have prefix started with "ix_" are SDK library provided by Intel. Variables "iphdr" and "tchhdr" are pointers to ip header and tcp header of a packet, respectively.

The data path system is implemented in microcode, a kind of assembly codes designed for microengines of IXP 1200. The functionality of a data path system includes parsing and rewriting the packet header and delivering the exception packet to StrongARM core. The communication between StrongARM core and microengines is achieved by a resource manager and scratch memory. Figure 5 gives a code segment for extracting ip and tcp header. The syntax, ".local", is a directive to declare a register for later usage. Macro "xbuf_extract" extracts a numeric byte field from the transfer register buffer, "$$ip_header", to a general-purpose register.

III. Load Balancing Mechanisms

We now present a scheduling method which incorporates work-flow models for task scheduling. A

```
ix_tcphdr_src_port_write(tcphdr, current->ip_dport);
ix_tcphdr_dest_port_write(tcphdr, current->ip_sport);
ix_tcphdr_seq_write(tcphdr, seq));
ix_tcphdr_ack_write(tcphdr,ack));
ix_tcphdr_flags_write(tcphdr, ACK_SYN_MASK));
ix_checksum_calc_segment_checksum(iphdr, (void*)tcphdr, &chksum,1);
```

Fig. 4. C programs for rewriting packet headers.

```
.local nsrc ndst nsport ndport chksum_delta seq
    xbuf_extract(nsrc, $$ip_header, 0, NSIP)
    xbuf_extract(ndst, $$ip_header, 0, NDIP)
    xbuf_extract(nsport, $$ip_header, 0, NSPORT)
    xbuf_extract(ndport, $$ip_header, 0, NDPORT)
    xbuf_extract(chksum_delta, $$ip_header, 0, DT)
    xbuf_extract(seq, $$ip_header, 0, SEQ)
.endlocal
```

Fig. 5. Microcodes for parsing TCP packet headers.

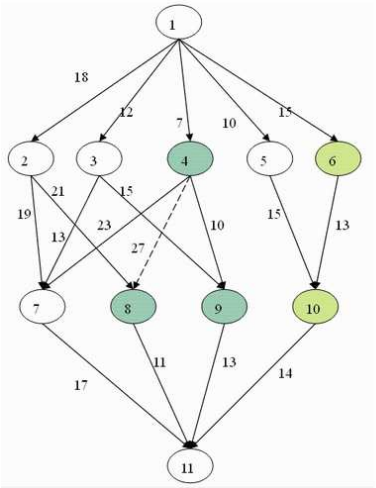
work-flow of tasks is represented as directed acyclic graph (DAG) [8], [9], [10]. An example of such a graph is shown in Figure 6. Nodes represent application tasks and edges represent data communication. The computation costs and communication costs are stored in a $n \times 1$ and $n \times n$ matrix, respectively. In the example graph, tasks $n_4, n_6, n_8, n_9, n_{10}$ are stateful tasks associating with two different services. The graph also comes with information to mark the stateful tasks when the timeout constraint for expiration is raised. In this case, the successors in the stateful tasks can be redirected to other servers for load balancing. This timeout information is presented as the dotted line of the edge. In our example graph, the edge between tasks n_4 and n_8 is with timeout edge. We assume every server can execute maximum k tasks in parallel. Tasks will be queued until the running tasks are less than k in a server and the computation cost will be n times of the original execution time of a task when there are n tasks executed on a server.

We have defined several attributes for task scheduling. The rank of the tasks represent the priorities of the scheduling order. The $rank(n_i)$ is the approximation of the length of the longest path from the task n_i to the exit task. The rank of task n_i is defined by

$$rank(n_i) = w_i + \max_{n_j \in succ(n_i)} (c_{i,j} + rank(n_j)), \quad (1)$$

where w_i is the computation cost of task n_i, $succ(n_i)$ is the set of the immediate successors of task n_i, $c_{i,j}$ is the communication cost of edge(i, j). According to the rank, we schedule tasks by decreasing order of a rank.

Our scheduling algorithm presents a two-phase scheduling policy. In the first phase, we perform a pre-scheduling for all stateful tasks, and then we perform



Computation Costs

Task	W_i
1	8
2	15
3	7
4	5
5	11
6	17
7	18
8	13
9	10
10	5
11	8

Fig. 6. a task graph with 11 tasks.

scheduling for stateless tasks in the second phase. In our first phase, we first mark all the stateful tasks by traversing the graph. If the edge before the task is marked as timeout, all the tasks following the edge will be recognized as a new stateful group. After separating the stateful tasks into different groups, we can then schedule each group one by one. We use the following equation to estimate the load of the stateful tasks which have been scheduled to the server.

$$Load(s_i) = \sum_{\forall g_j \ has \ been \ scheduled \ to \ s_i} \{ \sum_{\forall n_k \in g_j} R_k \}, \quad (2)$$

where s_i is the i-th server, R_k is the remaining computation time of task n_k, and g_j is the j-th group

of the stateful task groups. We also have

$$AddLoad(s_i, g_t) = Load(s_i) + \{ \sum_{\forall n_k \in g_t} R_k \}. \quad (3)$$

In order to balance the group load of the stateful tasks, we use the AddLoad function to calculate the total computation cost of each group when adding a new scheduled group. We then dispatch them to servers by picking up the minimum one. The scheduling algorithm is illustrated in the routine $Phase1\_Stateful\_Scheduler()$ of Figure 7.

After all the stateful tasks have been scheduled, we subsequently schedule the stateless tasks by the order generated by rank. The phase2_stateless_scheduler routine in Figure 7 presents the algorithm for the second phase of the scheduling. When a stateful task leaves the queue and prepare to be executed, we check the timeout value of the stateful group which was separated by the given timeout mark. To see the timeout will happen or not, if not, we will redirect the rest stateful tasks to the original server to keep the correctness of the stateful service. In this case, we also indicate the roll-back of the scheduling results for stateful tasks, and re-run the stateful scheduler in the phase one for the remaining stateful tasks. For a stateless task, we use the following function to estimate the finish time of the stateless task executing on the servers.

$$EFT(n_i, s_j) = Exec(w_i, avail[s_j], k) + \max_{n_m \in pred(n_i)} (AFT(n_m) + c_{m,i}), \quad (4)$$

where $pred(n_i)$ is the set of immediate predecessor tasks of task n_i, and $avail[s_j]$ is the earliest time at which server s_j is ready for task execution. AFT(n_m) is the actual finish time of the task n_m. $Exec(w_i, avail[s_j], k)$ is the execution cost of task n_i with computation cost w_i executed on the server s_j which can parallel execute at most k tasks from time $avail[s_j]$. And we choose the server with the minimum EFT to schedule. The last paragraph of the second routine in Figure 7 illustrates this idea.

Now we use the algorithm to schedule our sample graph. We assume each server can execute two tasks in parallel, and there are three servers. According to the first phase, we need to schedule the stateful tasks by equation 2. We traverse the graph to find out the stateful tasks and separate them into groups, note that there is a timeout mark between task n_4 and n_8. We therefore can separate them into three groups which are $g_1 = \{n_4, n_9\}$, $g_2 = \{n_6, n_{10}\}$, and $g_3 = \{n_8\}$, and then schedule g_1, g_2, and g_3 to server s_1, s_2, and s_3, respectively by equation 2. Once the stateful tasks have been scheduled, the rank of eack task needs to be calculated to decide the scheduling order.

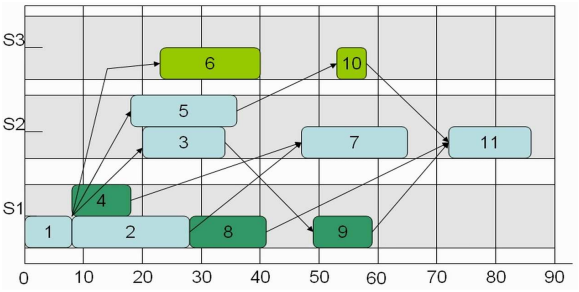


Fig. 8. The result of the sample application using our scheduling algorithm.

By the second phase of EFT algorithm. We need to calculate the dispatching order by using equation 1, the ranks of tasks are $\{n_1 = 93, n_2 = 77, n_3 = 63, n_4 = 71, n_5 = 53, n_6 = 57, n_7 = 43, n_8 = 32, n_9 = 31, n_{10} = 27, n_{11} = 8\}$. Then we sort the rank by decreasing order to get the scheduling order, which is $< n_1, n_2, n_4, n_3, n_6, n_5, n_7, n_8, n_9, n_{10}, n_{11} >$. Once the scheduling order is obtained, the tasks can be scheduled subsequently. By choosing the task with highest rank, task n_1 will be scheduled first. We use the EFT (equation 4) to find out the minimized execution time on servers. Because task n_1 has no predecessor, the result of EFTs is equal to 8: ($\forall s_i$, $EFT(n_1, s_i) = Exec(8, avail[s_i], 2) + 0 = 8$. The avail[$s_i$] is 0, because the task n_1 is the first task of each server.) By the scheduling order, the next task n_2 is also a stateful task whose EFTs need to be calculated: ($EFT(n_2, s_1)$= $Exec(15, avail[s_1], 2)$ + $max\{AFT(n_1)$ + $c_{1,2}\}$ = 15 + (8+0) = 23; $EFT(n_2, s_2)$ = 15 + (8+18) = 41; and $EFT(n_2, s_3)$ = 15 + (8+18) = 41.) From such results, we can schedule the task n_2 to server s_1 which has the minimum EFT value. The task n_4 is the next task to be scheduled by the order. Since the task n_4 is a stateful task, we dispatch task n_4 to the server s_1 which was decided previously. Next, we calculate the EFT value of task n_3:($EFT(n_3, s_1)$= $Exec(7 * 2, avail[s_1], 2)$ + $max\{AFT(n_1)$ + $c_{1,2}\}$ = 14 + (18+0) = 32; $EFT(n_3, s_2)$ = 7 + (8+12) = 27; and $EFT(n_3, s_3)$ = 7 + (8+12) = 27.). As the result, we will dispatch it to server s_2. According to the algorithm, we can dispatch the rest of tasks and get a simulated result of the sample graph shown in Figure 8. The timeout of task n_8 will not happen when the application is executed, it should be redirected to the server s_1 to keep the stateful tasks correctness. The final schedule length of the sample graph is 86.

IV. Experiments

Here we experiment with our workload algorithms EFT by simulations. We have constructed a software

Algorithm: The EFT load-balancing algorithms with work-flow information to handle both stateful and stateless tasks.

Input: A task graph **G** with the computation cost, communication costs, and the stateful groups.

```
Phase1_Stateful_Scheduler(){
    while there is a unscheduled group g_i do
        for each server s_j do
            Compute the AddLoad(s_j, g_i).
        Assign the tasks of g_i to the server s_k that minimizes AddLoad(s_k,g_i).
    end while
}
Phase2_Stateless_Scheduler(){
    while there is a un-scheduled task in the graph do
        Find the highest ranked task among un-scheduled tasks, say n_i, for scheduling
        if task n_i is stateful {
            if (the timeout constraint for expiration is raised for task n_i)
                    and ((the current time) - (the time for last done task of this group))< TIMEOUT {
                Assign the tasks of the group to the server which the task of this group has been scheduled.
                Revise this scheduling information to call Phase1_statefull_Scheduler() to re-do remaining stateful tasks.
            }
            else do
                Assign the stateful request to the server assigned at phase one
                  and update the session table.
            end if
        }
        else {/* Schedule stateless tasks */
            for each server s_j do
                Compute EFT(n_i, s_j).
            Assign request n_i to the server s_k that minimizes EFT of request n_i.
        }
        Update the connection table
    end while
}
```

Fig. 7. The EFT load-Balancing algorithm for the application with a work-flow graph.

simulator that emulates the network processor dispatching behavior for scheduling random tasks. The work-flow graphs of tasks are generated by a general graph generator with several parameters:

- **Number of nodes** v: The number of nodes (tasks) in the graph.
- **Shape of graph** s: We use this parameter to control the shape of graphs. The levels of generated graphs form a normal distribution with the mean value equal to $\sqrt{v}/s$. The nodes of each level also form a normal distribution with the mean value equal to $\sqrt{v} * s$.
- **Out degree** O: Out edges of each node. We use this parameter to control the dependence degrees between two tasks.
- **Communication to computation ratio** CCR: It is the ratio of the communication cost to computation cost. We can generate computation-intensive application graphs by assigning low values to CCR.
- **Number of stateful task groups**: It denotes the number of stateful service groups. We can also control the height of each stateful task group by supplied parameters.

Parameter	Value
V	25, 50, 100, 200, 400
S	1
O	2, 3, 4
CCR	0.3
Stateful groups	2, 4, 8
Stateful task ratio	0.25, 0.5

TABLE I

PARAMETER SETS USED IN FIGURE 9 AND FIGURE 10

In order to demonstrate the benefits of our EFT algorithm on dealing with stateful tasks, we use the parameters as listed in Table I. Under the parameter settings in Table I, we show the performance results of two different stateful task ratio 25% and 50% in Figure 9 and Figure 10, respectively. We use 500 graph instances for evaluating each parameter setttings. The x-axis gives different distribution of task nodes. It includes the amount of tasks and the amount of stateful groups as specified in Table I.

We take Round-robin (RR) and Estimated Task Time (ETT) algorithm which was proposed for application without work-flow to compared with our EFT alogirhtm.

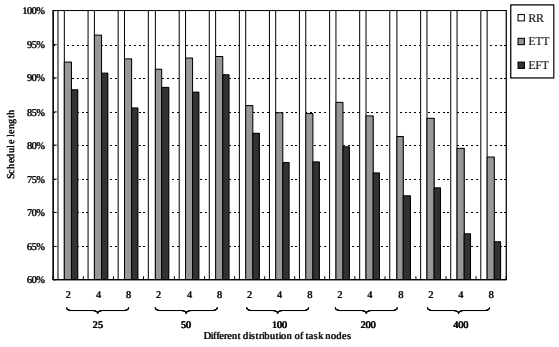


Fig. 9. Performance of EFT scheme with work-flow information (25% stateful tasks in each graph).

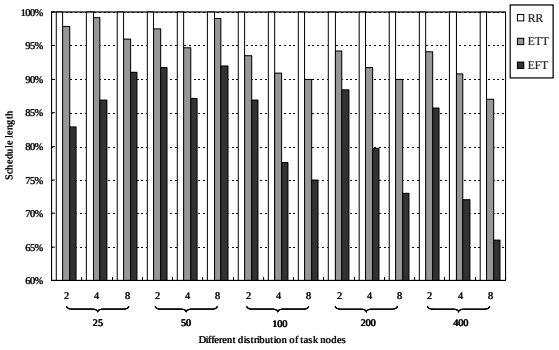


Fig. 10. Performance of EFT schemes with work-flow information (50% stateful tasks in each graph).

In Figure 9 and Figure 10, the results with ETT [12] and EFT are normalized over the results of Round-robin (RR). We can see that the EFT algorithm has significant performance improvement over ETT and RR. The improvement goes higher with bigger task graph and higher stateful task ratio. While the stateful task ratio is 25%, the improvement of EFT is from 2.76% to 12.63% when compared to ETT and is from 9.31% to 34% when compared to RR; While the stateful task ratio is 50%, the improvement of EFT is from 5% to 21% when compared to ETT and is from 8% to 34% when compared to RR. This phenomenon can be explained by the pre-known knowledge of work-flow graphs and the specific handling of stateful tasks in EFT. In phase 1 of the EFT algorithm, it will first consider the scheduling of stateful task groups. It pre-assigns the stateful groups into back-end servers according to the group computation load. In phase 2, we also provide a mechanism for stateful task groups to timeout and rescheduling. This produces a more fine-grained load-balancing scheduling.

V. Conclusion

In this paper, we presented our methodologies in supporting .NET Remoting over meta-clustered environments. Both stateful and stateless supports for .NET Remoting services are incorporated. The result of our experiment shows that the improvement of EFT is from 5% to 21% when compared to ETT and is from 8% to 34% when compared to RR while the stateful task ratio is 50%. Our work gave a comprehensive study for efficient support of .NET remoting in the presence of advanced network architectures such as IXP network processors. Our proposed scheduling methods include schemes with or without work-flow information of tasks. Further efforts to integrate our scheduling policy with CCA grid environments will be important directions for future research explorations.

References

[1] George Apostolopoulos, David Aubespin, Vinod Peris, Prashant Pradhan, and Debanjan Saha, Design, Implementation and Performance of a Content-Based Switch, in *Proceedings of IEEE Infocom 2000*, Mar. 2000.

[2] Chung-Kai Chen, Cheng-Wei Chen, Jenq Kuen Lee. Specification and Architecture Supports for Component Adaptations on Distributed Environments, *Proceedings of the IPDPS Conference*, Santa Fe, April 2004.

[3] Kattamuri Ekanadham, Joefon Jann, Pratap Pattnaik, Ramanjaneya Sarma Burugula, Donna Dillenberger. Anatomy of Autonomic Server Components *IBM Research Report*, November 2002.

[4] National Science Council(NSC) *Research Excellence Project* http://www.ccrc.nthu.edu.tw/PPAEUII/.

[5] Robert Haas, Lukas Kencl, Andreas Kind, Bernard Metzler, Roman Pletka, Marcel Waldvogel, Laurent Frelechoux, and Patrick Droz, IBM Research Clark Jeffries, IBM Corporation, Creating Advanced Functions on Network Processors: Experience and Perspectives, *IEEE Network*, July/August 2003.

[6] Sriram Krishnan, and Dennis Gannon. XCAT3: A Framework for CCA Components as OGSA Services. In *Proceedings of International Workshop on High-Level Parallel Programming Models and Supportive Environments*, April 2004.

[7] G. Teodoro, T. Tavares, B. Coutinho, W. Meira Jr., and D. Guedes, Load Balancing on Stateful Clustered Web Servers, in *15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03)*, November, 2003.

[8] M. Y. Wu, S. Hariri, and H. Topcuouglu, Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. IEEE Trans on Parallel and Distributed Systems *IEEE Trans on Parallel and Distributed Systems*,Vol. 13, 260-274, 2002.

[9] Y. Kwok and I. Ahmad,Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors *IEEE Trans. Parallel and Distributed System*, Vol.7, no.5, pp. 506-521, May 1996.

[10] M. Wu, W. Shu and J. Gu, Local Search for DAG Scheduling and Task Assignment, *Proc. 1997 Int'l Conf. Parallel Processing*, pp. 174-180, 1997.

[11] H. El-Rewini, H.H. Ali, and T. Lewis, Task Scheduling in Multiprocessor Systems, *Computer*, pp. 27-37, Dec. 1995.

[12] Yu-Tin Chen, Wang-Jer Wu, Chung-Kai Chen, Jenq Kuen Lee. Building Java RMI for Meta-Cluster Servers with Network Processor, Compiler Techniques for High-Performance Computing(CTHPC), 2004.

[13] Intel IXP1200 Network Processor Hardware Reference Manual.