OptBPEL: A Tool for Performance Optimization of BPEL Process

Sheng Chen, Liang Bao, and Ping Chen

Software Engineering Institute, Xidian University.
Xi'an, 710071, China
chensheng_cs@yahoo.com, baoliang@mail.xidian.edu.cn,
chenping@sei.xidian.edu.cn

Abstract. The Business Process Execution Language (BPEL) is now a de facto standard for specifying and executing business process for web service composition and orchestration. As more and more web services are composed using BPEL, tuning these compositions and gain better performance becomes increasingly important. This paper presents our approach for optimizing the BPEL process and introduces *OptBPEL*, a tool for performance optimization of BPEL process. The approach starts from the optimization of synchronization structure concerning *link* in BPEL. After that, some concurrency analysis techniques are applied to obtain further performance improvement. Finally, we give some experiments and prove the efficiency of these optimization algorithms used in *OptBPEL*.

Keywords: Performance Optimization, *OptBPEL*, Synchronization Analysis, Concurrency Analysis, Optimization Algorithms.

1 Introduction

Service-Oriented Architecture (SOA) is now a prevalent architectural style for creating an enterprise IT architecture that exploits the principles of service-orientation computing to achieve a tighter relationship between the business and the information systems that support the business [1].

With the growing adoption of service oriented computing, Web services composition is an emerging paradigm for enabling application integration within and across organizational boundaries. Business Process Execution Language (BPEL) [2] is now a promising and de facto language describing the Web services composition in form of business process.

BPEL supports concurrency and synchronization, hence BPEL processes may suffer from deadlocks and time-dependent data races [3] due to the erroneous use of *flow* and *link* like any other multi-threaded programs. For these reasons, while orchestrating processes, business modelers may hesitate to use concurrent paradigm, they prefer to invoke services sequentially even they could be executed concurrently. In this paper, we propose OptBPEL, a tuning tool for performance optimization of BPEL process. It first reads a BPEL process and performs some

© Springer-Verlag Berlin Heidelberg 2008

synchronization analysis on the *link* structure in process, generates a refined BPEL process. After that, the refined process is transformed by applying some concurrency analysis techniques and gains further optimization promotion.

2 Tool Description

Fig. 1 depicts the role of OptBPEL in the design and execution of BPEL process. The BPEL process may be manually written or generated by a BPEL design tool, e.g. ActiveBPEL Designer [4]. OptBPEL takes the BPEL code as input and performs some synchronization related optimization (in synchronization optimizer), generates the refined BPEL process. After that, concurrency related optimization (in concurrency optimizer) is applied to this refined BPEL process and gives the final optimized BPEL process. In section 3 and 4, we describe these two types of optimization that currently supported by OptBPEL.

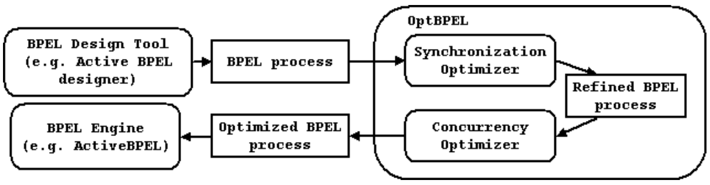
OptBPEL
BPEL Design Tool (e.g. Active BPEL designer)
BPEL process
Synchronization Optimizer
Refined BPEL process
BPEL Engine (e.g. ActiveBPEL)
Optimized BPEL process
Concurrency Optimizer

Fig. 1. The architecture of OptBPEL

3 Synchronization Optimization

In this section, we focus on the optimization opportunities with respect to the *link* construct in BPEL. These opportunities can categorized into three types: namely refining synchronization granularity, removing unnecessary *link* and augmenting concurrency.

When optimizing a process, it is critical to guarantee that the original process and its optimized version have equivalent semantics. Since this issue may take pages to describe formally, in this paper, we concentrate on the data race and deadlock aspect. If optimization operations that are related to *link* only don't induce data race and deadlock, then we can safely say that the optimized process is equivalent to the original one.

3.1 Transform BPEL Process to BSG

The approach we exploited focuses on activity segment [5] rather than activity itself, in order to reduce complexity and thus enhance efficiency. BPEL uses *flow* and *link* to express concurrency and synchronization respectively. While the *flow* could be modeled as *fork* and *join*, the *link* could be modeled as *wait* and *notify*, as the semantics of them stated in BPEL specification [2]. To concentrate on how we tune the process, we intentionally omit the details as how to handle the *transition condition* of the *link* and the *join condition* of activity.

The activity segments and their relation are captured by abstract graph model, which we call BPEL Segment Graph (BSG), is a directed graph $G =< S, E >$. Where the non-empty set of segments S contains all the segments in G, the edges set E expresses the relation among them. We classify the edges into two types: the first is sequential edge represented by SEQ used to express sequential relation; the second, and more important type is the synchronization edge represented by SYN used to express synchronization dependencies.

The BSG of a BPEL process can be established statically by the virtue of BPEL synchronization semantics and that BPEL can not create thread dynamically [2]. Koenraad [6] provided great details about how to construct series-parallel tack graph for parallel program, the construction of BSG for a BPEL process is kind of similar to that.

3.2 Optimization Algorithm

A deadlock can not arise during the optimizing process due to the fact that any type of the three optimization operations that impairs the synchronization of the process may in turn incur data races. If any two segments in the BSG will not conflict in variable access, then the process is free from data race.

Two legs of the race detection between two segments are their concurrency and variable sharing. Whether two segments are concurrent or not can be obtained by deciding they are reachability in BSG. Given two segments, if there exist a path from one segment to the other, then they are ordered. Otherwise, they are concurrent. When two concurrent segments access the same variable, and, at least one access is write, then the race occurs. Since BPEL can not create variable dynamically, the read/write variables set of a segment can calculated statically [2].

The "Link Optimizing" algorithm, which takes BSG as its parameter and optimizes it, sketches the optimizing process. The algorithm terminates when no optimizing can be done any more. During each iteration of **while**, we inspect each SYN edge in *bsg* for optimizing opportunities. In the part marked (1), we survey the edge if it can be removed, if we get it, then no further optimizing is need. Otherwise, in part (2), we try to refine it. We keep on refining a SYN edge until it can be removed or it can not be refined any more. In part (3), we deal with the augmenting of concurrency.

4 Concurrency Optimization

4.1 Process Modeling and PDGs

In order to apply static analysis to a BPEL process, we must first convert the BPEL process into its equivalent representation of TCFG. As described in [7], the transformation is straightforward. Note we distinguish the *interaction nodes* (representation of *reply*, *receive* and *invoke* activities) and *calculation nodes* (representation

```
Input: BSG bsg
Output: Optimized bsg
while 1 do
    foreach SYN edge se in bsg do
        let t and h be the tail and head node of se respectively;
        if exist a path from t to h besides se then                    // (1)
            remove se from bsg;
        else if no node in bsg conflict with h then                    // (2)
            $heads = (s) \mid (h, s) \in E \ \ in \ \ bsg$;
            add a SYN edge from t to each node in heads, remove se; continue;
        end
        seq_nodes=the set of nodes that are reachable from h by sequential edges
        merely and are not reachable from t, excluding h;
        if no node in bsg conflicts with none of seq_nodes then        // (3)
            delete the sequential edges entering and leaving h;
    end
    if no optimizing operation is done during this iteration then break;
end
```

Algorithm 1. Link Optimizing

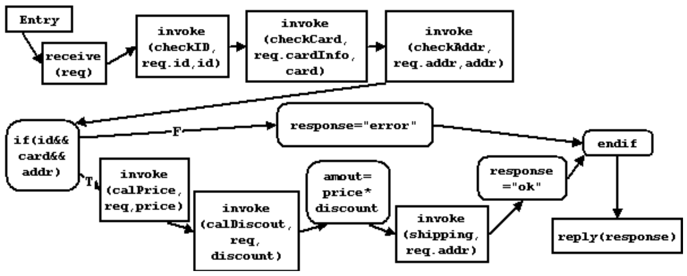
Entry
receive (req)
invoke (checkID, req.id,id)
invoke (checkCard, req.cardInfo, card)
invoke (checkAddr, req.addr,addr)
if(id&& card&& addr)
F
T
response="error"
endif
invoke (calPrice, req,price)
invoke (calDiscout, req, discount)
amout= price* discount
invoke (shipping, req.addr)
response ="ok"
reply(response)

Fig. 2. The TCFG representation of BPEL process

of other activities). This distinction is meaningful because the experiment result in [8] shows that the time cost of execution of *interaction nodes* is at least 5–10 times higher than that of *calculation nodes.* Fig. 2 shows an example of TCFG, the *interaction nodes* are represented by rectangular boxes and the *calculation nodes* by rounded boxes.

To obtain a program dependence graph (PDG) representation of this process, we need to insert control and data dependency that model the partial ordering on activities in the BPEL process that must be followed to preserve the semantic of the original process, [8] gives a detailed description.

4.2 Node Partitioning and Merging

In this section, we describe a simple algorithm called *merge-reorder* partitioning algorithm. The aim of this algorithm is to determine the best partition at which

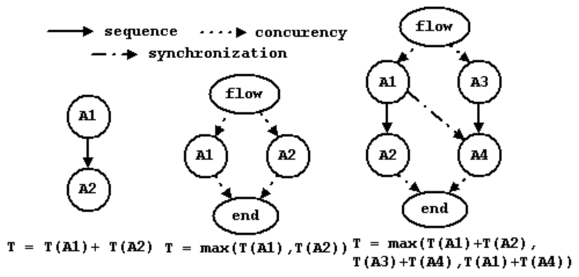

Fig. 3. Time cost model

each *calculation node* must be executed in some *interaction nodes* in order to minimize the total execution time of the BPEL process.

Before giving the description of the algorithm, we first define the time cost model of the execution. Fig. 3 shows the time cost under different (basic) situation (Note T(A_i) represents the time cost of the execution of activity A_i).

***Merge-reorder* algorithm**. An informal description of the merging and reordering algorithm is as follows:

1. Locate a control node, in the PDG whose child nodes are all leaf nodes. For all nodes that have the same control dependence condition on, repeat steps 2 through 6. Continue till all control nodes have been processed.
2. **Merging:** identify the set of dependence edges E, that pertain to a dependence between siblings with the control dependence condition chosen in step 1, such that at least one of the sibling is a *calculation node*. Pick an edge in E and merge the source and destination nodes of the edge. The resultant dependence of the merged task is the union of the component nodes.
3. When a *calculation node* gets merged with an *interaction node*, the combined node is an *interaction node*. When a *calculation node* gets merged with another *calculation node*, the combined node is also marked as a *calculation node*.
4. **Reordering:** for all configurations generated in step 4, using the time cost model to choose the merging configuration that likely to yield the minimum time cost value. For all partitions that only have one single *calculation node*, merge them into other different partitions averagely.
5. Exhaustively consider all merging configurations of siblings that can be generated by merging some subset of the dependence edges in E. Since the size of E for a single region is usually small, this exhaustive search is usually feasible in practice.
6. Once a region (subgraph) has been merged, we treat the whole subgraph as a single node for the purpose of merging at the next higher level. The dependence of the merge is a union of all dependence.

This algorithm is revised from the one in [8], the main difference between them is that in [8], author finds the partitions to maximize throughput, whereas the objective function we used here is to minimize completion time, another difference is that the our algorithm must create partitions such that each partition has exactly one *interaction node* and zero or more *calculation nodes*.

5 Performance Evaluation

Our experimental setup for testing optimized orchestration is as follows. We use a cluster of Intel Pentium based Windows machines (2.8G, 512MB RAM) connected by a 100 Mb/s LAN. Each test case we use runs on ActiveBPEL+Axis2 and RCbpel+RCWS respectively. RCbpel is a portable engine which is BPEL specification compliant developed by our research center implemented in C++, and RCWS is a web service container implemented by means of hybrid programming in C++ and Python. The reason why we use two settings is to demonstrate that performance enhance brought by OptBPEL is generic and not BPEL engine dependent, rather to compare execution duration and resource consumption between them.

Since BPEL is a relatively new language, there are currently no standardized BPEL benchmarks that we could use in our performance evaluation. However, we have tested many BPEL process including these taken from real-world, such as banking system. Fig. 4(a) presents the result of the optimization of travel reserve process (note the letter R represents the RCbpel engine and A represents the ActiveBPEL engine in these figures), Fig. 4(b) shows the optimization result of online book purchase process. The gain is minor when average service response time is relatively short, this is because the gain is offset by lengthy and frequent network connecting to some extent. However, when average service response time is long, as is the most case of real services, and the network connecting impact is negligible, we see drastic performance enhancement.

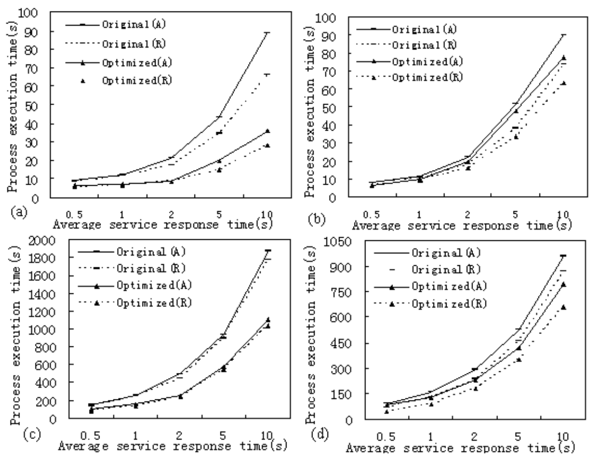

Fig. 4. Experimental results

Fig. 4(c),(d) show the optimization result by OptBPEL of two complex BPEL processes, the left one is a banking process and the other is a real-life train ticket purchase process, both them consist of hundreds of *invoke* and *receive* activities and twisted control flow. We once more see a significant performance improvement after the optimization by OptBPEL. The performance gain, which is average service response time correlated, ranges from 15% to 57%. When the process is complicated, there is always, as can be seen in banking process and ticket purchase process, bundles of performance optimization opportunities. OptBPEL taps and exploits all these opportunities, and thus speeding up the process dramatically.

6 Related Works

There has been considerable research effort paid to BPEL. WofBPEL [9] translates BPEL processes to Petri nets and imposes existing Petri nets analysis techniques to perform static analysis on processes, [10] modifies the CWB to support BPE-calculus by means of PAC to ensure that each *link* has one source and target activity exactly, and to guarantee that the process is free of deadlocks. Mads [11] describes some region-based memory techniques for programs that perform dynamic memory allocation and de-allocation, which is similar with our *merge-reorder* algorithm.

A mathematical performance model of BPEL process is addressed in [12], thus we may capture a deeper understanding of the performance of a process; on the other hand, it does not mention how to optimize a process.

Much work has been done on automatic parallelization of sequential programs based on PDGs, e.g. [13]. In contrast, this paper focuses on the use of PDGs in partitioning of composite web service applications for reducing execution time of BPEL process. Although the IBM Symphony project [8] employs a similar way of partitioning composite web services, its final goal is to implement the decentralized orchestration, which is a totally different problem.

7 Conclusions and Future Work

This paper presents our approach for tuning and optimizing the BPEL process and introduces OptBPEL, a tool for performance optimization of BPEL process. The approach starts from the architecture of OptBPEL, which contains two optimizers, namely synchronization optimizer and concurrency optimizer respectively. Then two important analysis methods, synchronization analysis and concurrency analysis, are introduced. We argue the efficiency of these algorithms used in OptBPEL and give some experiments to prove it.

Our further work will focus on the following three issues: 1) in some specific situations(e.g. grid computing), the resources, particularly computing resources, may be restricted. While optimizing under these circumstances, we will take the constraints into account. 2) the approach imposed in synchronization optimizer will emphasize on activity level rather than activity segment level since the current algorithm misses some optimizing opportunities; and 3) the elaborate time cost model of BPEL execution is needed and more efficient *merge-reorder* algorithm deserves a deeper observation.

References

1. Papazoglou, M.P.: Service-oriented computing: Concepts, characteristics and directions. In: 4th International Conference on Web Information Systems Engineering (WISE), pp. 3–12. IEEE Press, New York (2003)
2. Jordan, D.: Web services business process execution language version 2.0. OASIS Specification (2007)
3. Savage, S., Burrows, M., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multi-threaded programs. ACM Transactions on Computer Systems 15, 391–411 (1997)
4. Active-Endpoints: Active Endpoints Corp. (2007), `http://www.active-endpoints.com/active-bpel-designer.htm`
5. Christiaens, M., Bosschere, K.: Trade: a topological approach to on-the-fly race detection in java programs. In: Java Virtual Machine Research and Technology Symposium (JVM), Usenix Association (2001)
6. Audenaert, K., Levrouw, L.: Space efficient data race detection for parallel programs with series-parallel task graphs. In: 3rd Euromicro Workshop on Parallel and Distributed Processing, pp. 508–515. IEEE Press, New York (1995)
7. Yuan, Y., Li, Z.J., Sun, W.: A graph-search based approach to bpel4ws test generation. In: International Conference on Software Engineering Advance (ICSEA), pp. 16–22. IEEE Computer Society Press, Los Alamitos (2006)
8. Nanda, M., Chandra, S., Sarkar, V.: Decentralizeing execution of compostite web services. In: 19th Object-Oriented Programming, System, Languages, and Applications (OOPSLA), pp. 170–187. ACM Press, New York (2004)
9. Ouyang, C., Wil, M.P., van der Aalst, Breutel, S.: Wofbpel: A tool for automated analysis of bpel processes. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 484–489. Springer, Heidelberg (2005)
10. Koshkina, M., Breugel, F.: Modelling and verifying web service orchestration by means of the concurrency workbench. TAV-WEB Proceedings/ACM SIGSOFT 29–5 (2004)
11. Tofte, M., Talpin, J.-P.: Region-based memory management. Information and Computation 132, 109–197 (1997)
12. Rud, D., Schmietendorf, A., Dumke, R.: Performance modeling of ws-bpel-based web service compositions. In: IEEE Services Computing Workshops (SCW), pp. 140–147. IEEE Computer Society Press, Los Alamitos (2006)
13. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9 (1987)