Decentralizing Composite Web Services

Mangala Gowri Nanda
IBM India Research Laboratory
mgowri@in.ibm.com

Satish Chandra
IBM India Research Laboratory
satishchandra@in.ibm.com

Vivek Sarkar
IBM T. J. watson Research Center
vsarkar@us.ibm.com

Abstract

Web services make information and software available programmatically via the Internet and may be used as building blocks *for applications. A composite web service is one that is built using multiple component web services. Composite web services are emerging as a programming model of distributed computation applicable to business process. Once its specification has been developed, the composite service may be* orchestrated *either using a* centralized *engine or in a* decentralized *fashion. Decentralized orchestration brings performance benefits, and improves scalability and concurrency. Decentralized orchestration is similar to execution on multi-processors. However standard techniques for automatic parallelization cannot be applied directly to decentralizing composite services. We propose a novel algorithm for automatic parallelization and code partitioning that is applicable to decentralized orchestration of web services. We show how to generate a decentralized web service specification given a specification for centralized orchestration.*

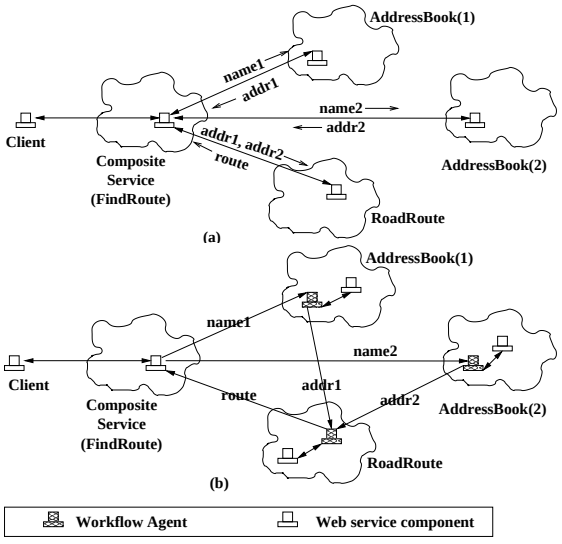

Figure 1. Centralized and Decentralized Architecture

1 Introduction

Web services encapsulate information, software or other resources, and make them available over the network via standard interfaces and protocols [7]. Complex web services may be created by aggregating the functionality provided by simpler ones. This is referred to as *service composition*. Typically a *composite* service is defined by using a specification language such as *Business Process Execution Language* (BPEL)[3] and its execution is driven by a centralized engine that interprets the specification.

As an example, Figure 1(a) shows the centralized orchestration of a $FindRoute$ composite service built on two address book services, $AddrBook(1)$ and $AddrBook(2)$, and a $RoadRoute$ service which computes driving directions. The $FindRoute$ service sends a name, $name1$, to $AddrBook(1)$, and another name, $name2$, to $AddrBook(2)$. The two addresses returned are then sent to the $RoadRoute$ service which returns the driving directions from one address to the other. Thus, the $FindRoute$ service acts as a centralized coordinator for all interactions among the component services.

In *decentralized orchestration*, messages can be sent directly from a component where the data is produced to a component where the data is consumed, without using the composite service as a centralized coordinator. For example, the addresses generated at $AddrBook(1)$

and $AddrBook(2)$ can be sent directly to $RoadRoute$, as shown in Figure 1(b). The message send operations are assumed to be *asynchronous* (non-blocking),where as invoke and receive operations are assumed to be *synchronous* (blocking). Doing so will increase parallelism (since $AddrBook(1)$ and $AddrBook(2)$ can be executed concurrently) and also reduce the message overhead (since fewer messages are sent). Decentralized orchestration requires an *engine* or *agent* at each component site that gives the component the capability of performing the following steps:

- Receive data/control messages from previously-invoked component(s)
- Execute application logic
- Create a new invocation of a web service component
- Evaluate exit conditions and send data / control messages to newly-invoked component(s)

The example in Figure 1 was chosen for the sake of simplicity. In practice, there are two domains where decentralization of composite web services is becoming increasingly important — *business processes* and *grid applications* [5]. Business processes consists of activities that may be executed by humans or by agents (computer processes). On the other hand, grid applications are compositions of subprograms that are invoked using web service interfaces with little or no human interactions. In this case, both increasing concurrency and reducing message overheads can result in very tangible performance benefits for composite web services. Our primary interest in decentralization is targeted to grid applications and other applications where web services are composed with little or no human interactions.

In this paper we

- give a code partitioning algorithm that permits decentralized orchestration of composite web services, and
- for the BPEL language, we show how to generate the decentralized specification from a centralized specification.

2 The Language and its Graphical Representation

In this section, we outline the subset of BPEL assumed in this paper as the programming language model for composition of web service components.

BPEL processes use *containers* to exchange data between tasks. The type of each container is specified as a message type in the Web Services Definition Language (WSDL) [7]. A message type declaration includes a set of

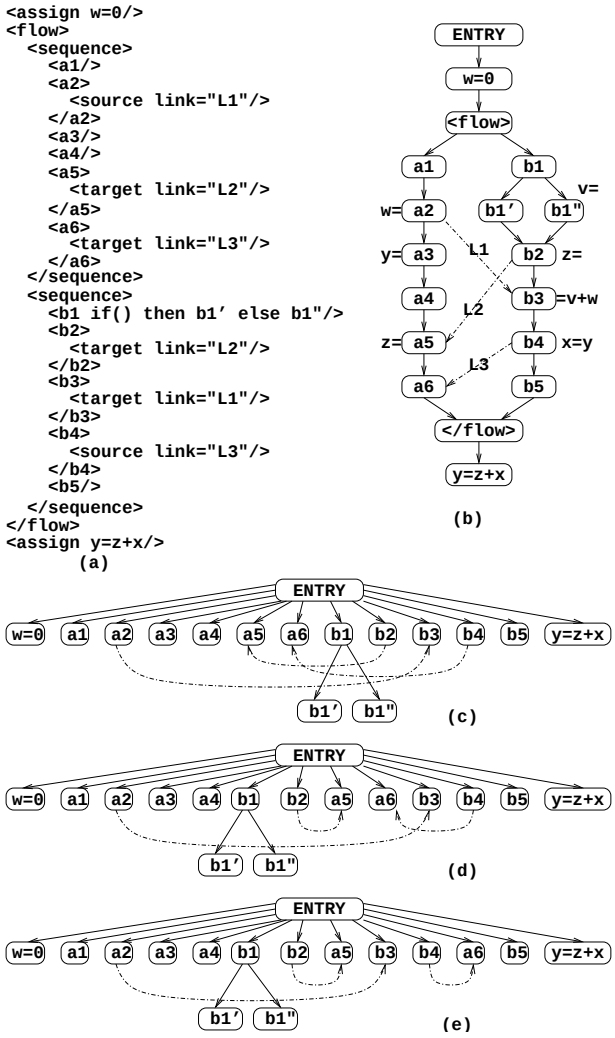

Figure 2. Program Representation

names *parts*, which are analogous to fields in structure declarations. A `process` consists of a set of *activities*:

- The `sequence`, `switch`, and `while` constructs specify sequential, conditional, and iterative control flow among activities respectively.
- The *pick* construct specifies nondeterministic choice based on external events.
- The `flow` construct specifies concurrency among activities, analogous to the cobegin-coend and parallel sections constructs.
- The `link` construct specifies additional synchronization between two concurrent activities in a flow construct, analogous to wait/notify synchronizations. The synchronization ensures that the *source* activity of a link completes before the *target* activity of the link.

- An `invoke` activity *invokes* an `operation` on a `partner` by sending data specified by an `inputContainer` and receiving a response in an `outputContainer`. An `invoke` may be asynchronous, requiring no response and for convenience we term such `invoke` activities `send`.
- A `receive` activity accepts data from a `partner` into a specified `container`.
- A `reply` activity sends data to a `partner` from a specified `container`.
- Standard assignment and arithmetic expressions are also represented as activities.

The quasi-BPEL specification for the `FindRoute` service is shown in Figure 3[1].

Input Program Representation The input BPEL specification can be represented by a program representation that represents explicit parallel control flow, such as the *Threaded Control Flow Graph* (TCFG) [8]. Each node in the TCFG represents an activity in the BPEL `process` and the edges represent flow of control. Control flow generated by `sequence`, `switch` and `while` are similar to any imperative language. Control flow generated by the `flow` construct is similar to the `cobegin-coend` construct of parallel programs. The TCFG extends the standard CFG representation of sequential programs by including two special nodes `<flow>` and `</flow>` to represent the start and end of parallel sections. The parallel sections are represented by whole CFGs, which are embedded in the surrounding TCFG. The TCFG for the BPEL program in Figure 3 is shown in Figure 4.

Intermediate Program Representation — The Program Dependence Graph Our approach for decentralization of composite web services is based on the use of the program dependence graph (PDG) [11], or more generally, the parallel program graph (PPG) [14, 17]. The PDG consists of a Control Dependence Graph (CDG) and a Data Dependence Graph (DDG) superimposed on the same set of nodes which denote statements and predicate expressions of the TCFG,

Since BPEL is a structured program representation, the CDG can be computed as a tree in a syntax-directed manner. All nodes within a `while` loop are control dependent on the loop header. The semantics of a BPEL `switch` is identical to the standard `if-then-else` construct. Nodes in the `then` block are control dependent on the `if(expr)` node and the control dependence edges are labeled *T*, and nodes in the `else` block are control dependent on the `if(expr)` node but the control dependence edges are labeled *F*. A `sequence` of nodes is control dependent on the enclosing control node which may be a `while`, `if` or `Entry`. For parallel sections generated by a `flow` construct we determine control dependence as follows: let N_c be the node on which the `flow` node is control dependent, then every node in each parallel section of the `flow`, is control dependent on N_c as shown in Figure 2.

Data dependences are computed based on read/write accesses to data in `containers`. Standard techniques [9] for analyzing parallel programs may be used to compute the data dependence edges. Since most accesses to containers are by direct (unaliased) names, computing data dependences for BPEL programs is an easier task than computing dependences for array and pointer data accesses with potential aliasing. In addition, the `target` activity of a `link` is considered to have a special synchronization data dependence on a `source` activity of the same `link`.

In the PDG, if all nodes that are control dependent on a node N_c are arranged in a lexical postdominance order from left to right, then all data dependence edges (other than loop-carried dependences) will also be from left to right in the PDG (by definition of data dependence). This condition, however, may be violated if there are explicit "link" induced data dependences as shown in Figure 2(c). In Figure 2(c) the dependence edge from b_2 to a_5 is in the reverse direction. To resolve this, we re-arrange the nodes in the PDG to ensure that all data dependences are from left to right. For the example we need to move b_2 to the left of a_5. However since there is an implicit sequencing from b_1 to b_2, we also need to move b_1 as shown in Figure 2(d). Then we check if all new dependences are in the correct direction. Figure 2(e) shows the final step in the reordering of the edge $b_4 \rightarrow a_6$. This reordering is always possible so long as there is no circular set of dependences. For example, if there was a link from b_5 to a_1 then reordering the nodes would reverse the direction of the edge $a_2 \rightarrow b_3$ indicating that there is an implicit deadlock [17].

3 Code Partitioning

As mentioned earlier, a task may be a web service invocation or it may be a piece of business logic. The web service invocation *must* be performed at a specific site whereas the business logic may be performed at any convenient site. We refer to tasks that must be performed at a fixed location as `fixed` tasks and the remaining tasks are referred to as `portable` tasks. Tasks that *must* be performed at the composite service site (for example for access to a local database or to gather user input through a GUI) are also marked as `fixed` tasks (using user annotation).

[1]The syntax has been compressed due to space restrictions.

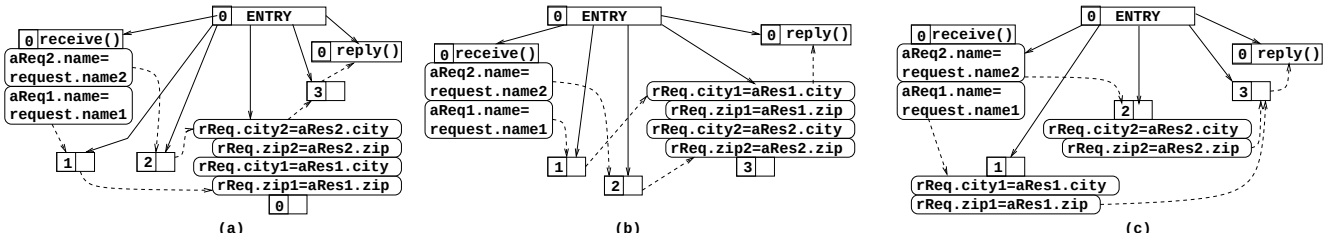

Figure 6. Code partitions for the `FindRoute` Service

3.1 Cost Function

In the TCFG shown in Figure 4 and the corresponding PDG in Figure 5, the `fixed` tasks are shown in rectangles and the `portable` tasks are shown in rounded boxes. The sites are numbered and we refer to the component sites as N_1, N_2 and N_3 and to the composite service site as N_0. We need to determine where the `portable` tasks need to be executed. Figure 6(a) gives a solution where all the `portable` tasks are executed at N_0 (this is equivalent to centralized execution). Note that in this case there are six inter-node messages. The number of inter-node messages in Figure 6(b) and (c) is five. However, in Figure 6(b), the entire `addrResponse` data including `phone` etc. are sent from N_1 to N_3 and from N_2 to N_3, whereas in Figure 6(c), the data on the wire has also been minimized.

Further, consider the program in Figure 7 and its possible partitions. It may appear that partition (d) would give better performance than the partitions (b) or (c), since the critical path has a length of 1, whereas in (b) and (c) the critical path has a length of two. However, preliminary experimental results (discussed in detail in Section 5) in Figure 11 shows that partition (c) actually gives better response time when the size of the messages is small due to the lower synchronization overheads. Partition (d) with its higher level of concurrency brings better performance when the computation time at the nodes becomes significant compared to the synchronization overheads.

Determining the partition which will give the best performance is neither obvious nor trivial, and as can be seen from Figure 11, the difference between two solutions can be of the order of seconds. In general, it is not possible to determine which partition will give the best performance without applying a cost function to evaluate the different topographs. However, determining a cost function for a LAN environment is difficult and more complex still for a WAN environment where additional factors such as geographical distance and bandwidth play an important role. Based on our limited experience, we propose a cost function for the LAN environment as follows: With every node, N_i, we associate a computation cost C_i. For every data message transferred between two components with associate a data transmission cost D_i and for a node with more than one incoming edge we associate a synchronization cost $S_i = (nMsg - 1) * S_c$, where $nMsg$ is the number of incoming edges and S_c is a synchronization cost. Then the cost of a subgraph in a partition is calculated for each path from the root of the subgraph to a leaf node as the summation of the total computation, transmission and synchronization cost along that path. The cost of the subgraph is the maximum cost along any path in the subgraph.

3.2 Partitioning Algorithm

In general, there is an exponential number of ways to distribute the `portable` code amongst the processors. The aim of the code partitioning algorithm is to determine the best site at which each `portable` task must be executed in order to optimize the performance.

The intuitive rationale for the code partitioning algorithm is as follows: If there is a def-use chain in the PDG that starts at a `fixed` node, N_i and terminates at another `fixed` node N_j, we can move all the intervening portable code to either N_i or N_j and generate a communication edge from N_i to N_j. For example, in Figure 5, there is a def-use chain $N_1 \rightarrow$ `rReq.city1=aRes1.city` $\rightarrow N_3$. The intervening code between N_1 and N_3 can be moved to N_1 (Figure 6(c)) or to N_3 (Figure 6(b)). One of the options will be chosen on the basis of which one reduces the data on the wire which in this example turns out to be case (c).

However, merging along def-use chains is not always possible because (1) def-use chains do not encapsulate control dependence conditions, and (2) def-use chains may intersect as in Figure 7 giving rise to multiple partition options that need to be evaluated based on the cost function.

Nevertheless, merging along def-use edges is very appealing as it thins out the space of all possible mergings and on that basis we develop our code partition algorithm as follows: Starting at the bottom of the CDG tree we identify sibling nodes that have the same control dependence condition and perform a merge on these nodes. Two sibling nodes in the PDG that have the same control dependence condition may be merged if there is a def-use dependence relationship between them, subject to the condition that the

```
<message name=request>
   <part name="name1" type="xsd:string"/>
   <part name="name2" type="xsd:string"/>
</message>
<message name=addrReq>
   <part name="name" type="xsd:string"/>
</message>
<message name="addrResp">
   <part name="street" type="xsd:string"/>
   <part name="city" type="xsd:string"/>
   <part name="zip" type="xsd:string"/>
   <part name="phone" type="xsd:string"/>
</message>
<message name="routeReq">
   <part name="city1" type="xsd:string"/>
   <part name="zip1" type="xsd:string"/>
   <part name="city2" type="xsd:string"/>
   <part name="zip2" type="xsd:string"/>
</message>
<message name="routeResp">
   <part name="directions" type="xsd:string"/>
</message>
<container name="request" msgType="request"/>
<container name="aReq1" msgType="addrReq"/>
<container name="aRes1" msgType="addrResp"/>
<container name="aReq2" msgType="addrReq"/>
<container name="aRes2" msgType="addrResp"/>
<container name="rReq" msgType="routeReq"/>
<container name="rRes" msgType="routeResp"/>
<sequence>
   <receive partner="caller" container="request"/>
   <flow>
      <sequence>
      <copy from="request.name1" to="aReq1.name"/>
      <invoke partner="addrBook1"
         operation="getAddress"
         inputContainer="aReq1"
         outputContainer="aRes1"/>
      </sequence>
      <sequence>
      <copy from="request.name2" to="aReq2.name"/>
      <invoke partner="addrBook2"
         operation="getAddress"
         inputContainer="aReq2"
         outputContainer="aRes2"/>
      </sequence>
   </flow>
   <copy from="aRes1.city" to="rReq.city1/>
   <copy from="aRes1.zip" to="rReq.zip1/>
   <copy from="aRes2.city" to="rReq.city2/>
   <copy from="aRes2.zip" to="rReq.zip2/>
   <invoke partner="roadRoute"
      operation="getDirections"
      inputContainer="rReq"
      outputContainer="rRes"/>
   <reply partner="caller" container="rRes"/>
</sequence>
```

Figure 3. BPEL specification for the `FindRoute` service

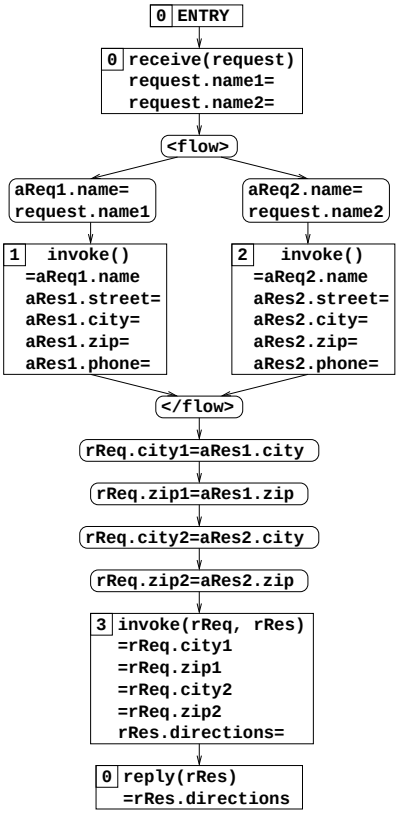

Figure 4. The TCFG for the `FindRoute` Service along with the *definition*s and *use*s at each task.

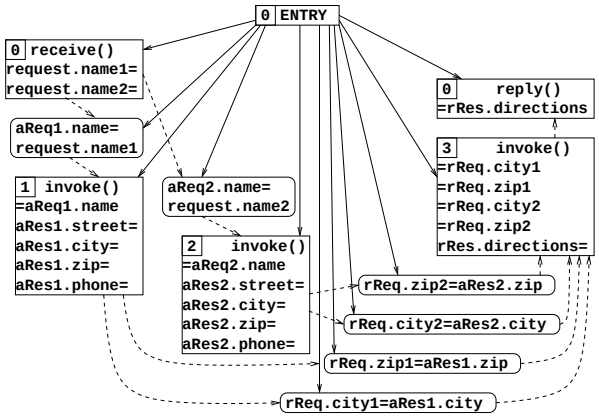

Figure 5. The PDG for the `FindRoute` Service

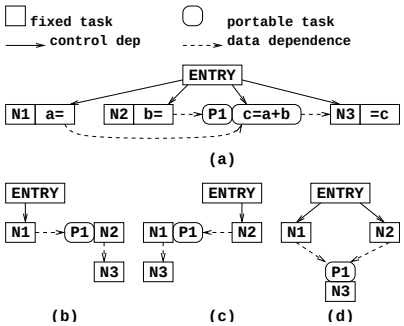

Figure 7. Code partitions

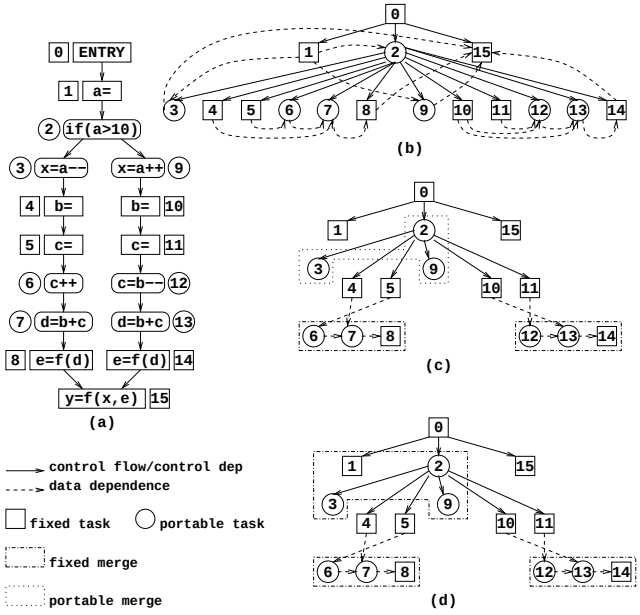

Figure 8. Merging Portable Tasks

reordering along the def-use edge does not violate any other dependences. We exhaustively evaluate all possible merges along def-use edges and compute a local minimum for each subgraph. Once a subgraph has been evaluated the parent node may be merged with its siblings.

Merging Portable Code In this section we show how to merge within a subgraph of the CDG.

1. Locate a control node, T_c in the CDG whose child nodes are all leaf nodes in the CDG. For all nodes that have the same control dependence condition on T_c repeat steps 2 through 6.

2. For each `portable` task, t_i identify all siblings that it is data dependent on as well as all siblings that are data dependent on t_i. Exhaustively try merging t_i with

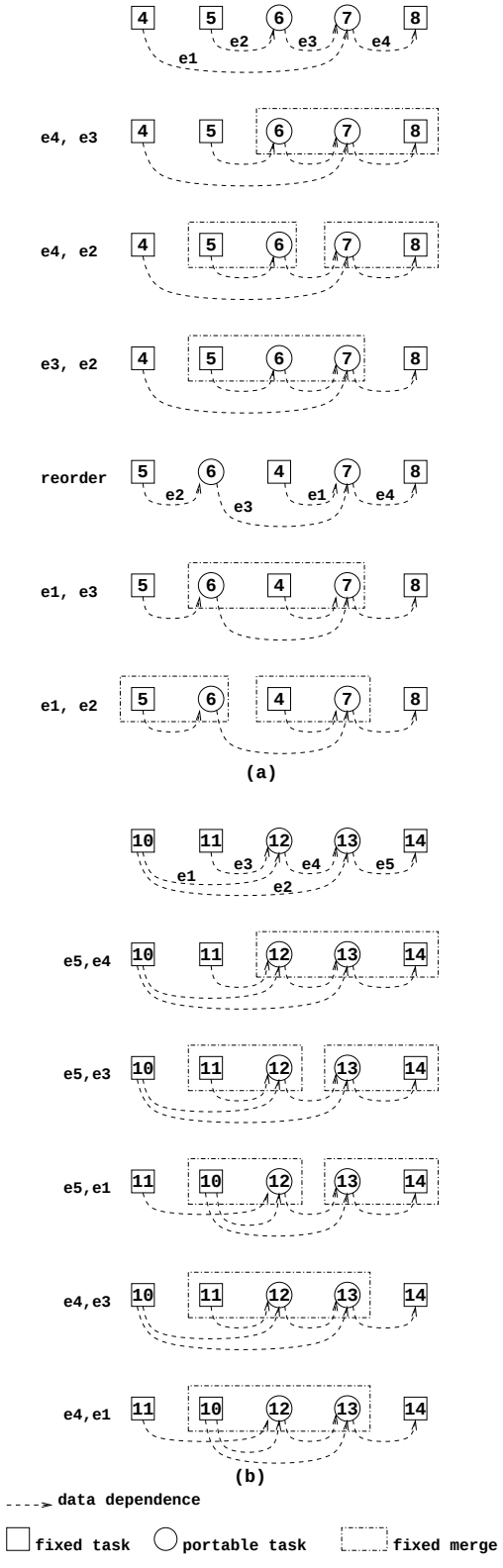

Figure 9. Merging Sibling Tasks

each such sibling. Delete one control dependence edge from the merged nodes.

3. A `portable` task that gets merged with a `fixed` task becomes part of the `fixed` task. When a `portable` task gets merged with another `portable` task the combined task is also marked as a `portable` task.

4. When no more portable tasks can be merged, all the remaining `portable` tasks are merged with the parent. At this point, the parent has either no children left or only `fixed` tasks for children. The parent node is now marked a leaf node. The cost of this local partition can now be computed as the maximum cost of any path from T_c to a leaf node in the partition.

5. When a node is merged with a sibling that is not its lexical neighbor, we need to ensure that no data dependence conditions are violated. To determine whether the merge may violate a dependence condition, we reorder tasks in the same way as we did to fix "link" edges in the parallel sections and check if all the dependences are still in the left-to-right direction in the new ordering. If this is not possible, then the merge is not performed.

6. Once a subgraph of the CDG has been merged, we treat the whole subgraph as a single node for the purpose of merging at the next higher level. The data dependences of the merge is a union of all dependences in the child nodes as well as the parent node.

Example: Consider the example in Figure 8. The algorithm starts by analyzing the subtree rooted at the task labeled 2 (we refer the shorthand T_i to refer to tasks). The tasks T_3, $T_4, \ldots T_8$ have the same control dependence conditions and will be merged first. We show the details of the different merge solutions in Figure 9(a). (T_3 is omitted from the picture as it has no dependence relationship with any of the siblings and hence does not participate in any merge).

Observe that T_7 may merge along the edges e_4, e_3 or e_2, while T_6 may merge along the edges e_3 and e_2. We exhaustively try every combination. In the first case, T_7 merges along e_4, so T_7 and T_8 merge into a fixed task. Then T_6 merges along the edge e_3 to generate the merged fixed task $\langle\, T_6, T_7, T_8 \,\rangle$.

Merging T_7 with T_4 along the edge e_1, causes e_2 to point right-to-left in the sequence. So we reorder the tasks, as shown and then apply the merge.

There are a total of five possible merge solutions. The cost computed for the first option is maximum of $D_4 + C_4 + e_3 + S_n$ and $D_5 + C_5 + e_2 + S_n$, where D_4 and D_5 is the cost of data transmission from T_2 to T_4 and T_5 respectively. The cost for each of the other solutions can be computed in a similar manner. Note, that the cost of a merge solution can be computed only after the entire block has been merged since the cost depends not only on the inter-component edges but also the incomimg edge count at a component.

Figure 9(b) gives the merge options for the set of tasks $T_{10} \ldots T_{14}$. In this case T_{13} can merge along the edges e_5, e_4 and e_2. The edges e_5 and e_4 represent dependences between lexical neighbors and hence are feasible. However, it is not possible to merge T_{13} with T_{10} as the dependence conditions require that T_{12} must precede T_{13} (edge e_4) and succeed T_{10} (edge e_1) which is not possible if T_{13} merges with T_{10}. Thus although the total number of edges in the graph has increased the actual merge possibilities has not increased.

At the end of the merging exercise one of the merge solutions is chosen based on cost computed and as shown in Figure 8(c). Now the tasks T_3 and T_9 which have not been merged with `fixed` tasks are merged with the parent node T_2, T_2 can now be marked a "leaf" node and the algorithm will continue the merging process for the tasks T_1, T_2 and T_{15}. A possible solution is shown in Figure 8(d).

Notes At the end of the analysis, any `portable` task that has not been merged with a `fixed` task is performed at the composite node N_0.

When merging along data dependence edges, we do not consider loop-carried data dependences. This is because, a merge is always performed only between two nodes that have the same control dependence conditions. A node n_i that is loop-carried data dependent on a node n_j may be lexically control dependence on the same node n_c but they are actually dependent on different instances of n_c and hence cannot be merged.

This algorithm generates the PDG of Figure 6(c) when applied to the composite service of Figure 5.

4 Code Generation

Once the code has been partitioned, we generate a *Parallel Program Graph* (PPG) [14] which is essentially the partitioned PDG from which we remove redundant control dependence edges. Once this graph is generated, control flows along the edges of the PPG. On the basis of the PPG, we generate code for "local" segments (the partitioned boxes in Figure 8(d) and finally we handle external incoming and outgoing edges at each component. This requires "global concurrency" analysis which has been described in an earlier paper [10] and which we briefly summarize in the appendix for completeness.

4.1 The Parallel Program Graph

The implication of a control dependence edge from N_i to N_j is that N_i must complete execution before N_j may execute. A data dependence has the same implication. If a node N_i is control dependent on a node N_c and data dependent on a node N_j such that N_j is a sibling of N_i in the PDG (*i.e.*, it has the same control dependence condition as N_i) or N_i is data dependent on a descendant of N_j, then the control dependence edge from N_c to N_i can be removed. The reason is simple - since N_j is also control dependent on N_c (and its descendants are transitively control dependent on N_c), it is clear that N_j (or its descendants) cannot execute before N_c. Hence the data dependence edge subsumes the control dependence edge.

Local Code Generation Local code is generated for a component, N_i, by traversing the subset of the PPG that has been assigned to N_i and outputting corresponding code. An edge $t_i \rightarrow t_j$ for a task t_i in N_i and a task t_j in N_j generates a `<send/>` at t_i and a `<receive/>` at t_j.

Global Code Generation Global code generation requires concurrency and synchronization analysis of the intercomponent messages [10]. This has been covered in an earlier paper, but for completeness, we include a brief description in the appendix. The basic issue is explained with an example below.

Consider the PPG in Figure 10. There are two incoming edges $N_1 \rightarrow N_3$ and $N_2 \rightarrow N_3$ at N_3. This form of concurrency is labeled β concurrency and the ways in which data synchronization may be achieved are enumerated below

- Direct Deposit: N_1 and N_2 send data directly to N_3.
- Request Response: N_1 and N_2 inform N_3 when data is ready and N_3 pulls the data when it is ready.
- Combined Direct Deposit and Request Response (CDDRR): N_1 sends data to N_2, N_2 sends data to N_3 and N_3 pulls data from N_1. This possible since there is a path in the PPG from N_1 to N_2.
- Pipeline: N_1 pipes data to N_3 through N_2.

There are performance tradeoffs in the different orchestrations based on which one of the orchestrations must be chosen. The corresponding code fragments are also shown in Figure 10.

β_2 concurrency brings in new complications. The data generated at N_1 may get killed along some path to N_3 and hence data from N_1 cannot be forwarded to N_3 until N_2 has executed. This is not an issue in local code generation, since in local code each task has access to all the local containers that are being updated and messaging passing is not required to transfer data from one task to another. In β_2 concurrency, additional code is required at N_2 to inform N_1 of the value of the conditional evaluated. N_1 waits for a message from N_2 and forwards data to N_3 if the condition evaluated to false, else it discards the data. At N_3, two messages are expected, so N_2 sends a dummy message if the condition evaluates to true to complete the count. The receiving `container` should be a union of data that may come along any edge along with an identifier indicating the source of the data so that N_3 knows how to process the data.

The code generated for a Direct Deposit orchestration of β_2 concurrency is shown in Figure 10. The possible orchestrations for all forms of concurrency are given in the appendix.

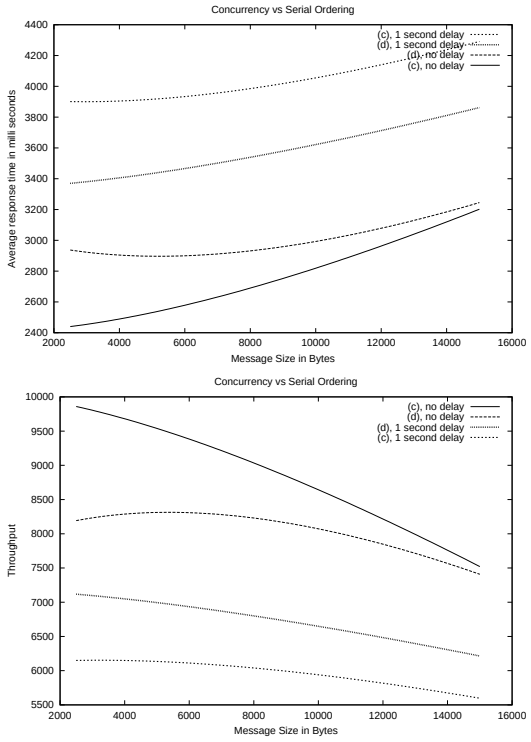

Figure 11. Performance of Partition (c) and (d) of Figure 7.

5 Preliminary Results

We have implemented a workflow agent for decentralized composite service execution. Each node has one such agent, capable of receiving and sending *asynchronous* messages over HTTP. At runtime the agent interprets a decen-

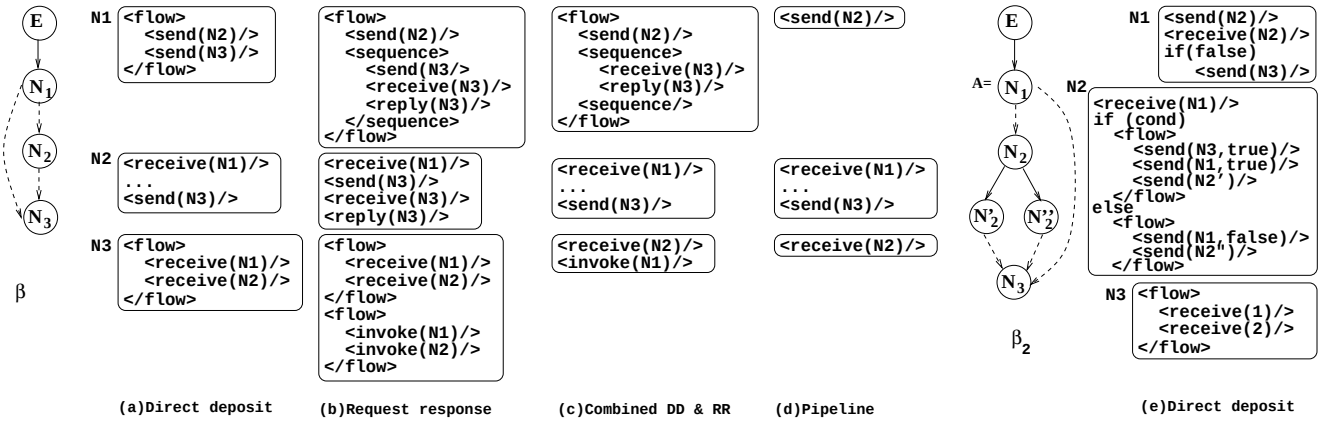

Figure 10. Code generation for β concurrency and β_2 concurrency

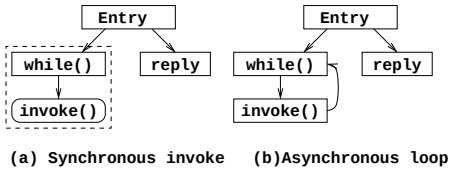

Figure 12. Synchronous vs Asynchronous Loops

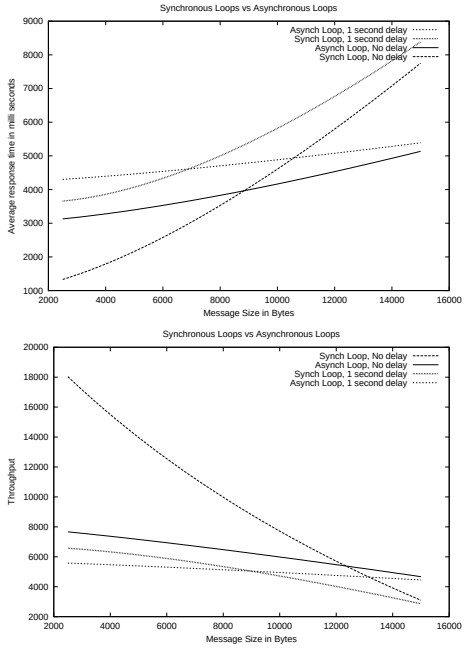

Figure 13. Experimental Analysis of Synchronous vs Asynchronous Loops

tralized specification coded in a subset of BPEL.[2] Requests are initiated by clients that send a *synchronous* HTTP message to a composite service. The composite service holds the client's connection and *asynchronously* dispatches data to the relevant component service(s). The last component asynchronously returns a response to the composite service, which forwards the response to the client over its pending connection. We conducted experiments in a LAN environment with 80 concurrent clients. All the web services run on standard Intel-based machines running `tomcat` on Linux.

Delays in the web services were introduced by making it sleep for a given amount of time, and the length of the messages was artificially enlarged by introducing garbage into the legitimate response.

The response time was calculated at the composite service as the round trip from the time the composite service receives a request to the time it receives the response. The throughput was calculated as the number of requests processed in a ten minute interval.

Figure 11 shows the average response time and throughput for the partitions labeled (c) and (d) in Figure 7. Partition (c) which has a longer critical path, less concurrency and lower synchronization gives better (lower) response time and (higher) throughput than partition (d) when the delay is small. As the delay increases, the benefits of concurrency overcome the overheads of synchronization and then partition (d) performs better.

There are other factors too that affect the performance. A loop that contains only one invocation performs better in synchronous mode (Figure 12(a)) rather than asynchronous mode (Figure 12(b)) when the data is less and the time to process is small. As the data size increases the asynchronous solution overtakes the synchronous solution. As

[2] Although our current implementation is not strictly BPEL, we use the term BPEL for convenience. We are in the process of migrating to a complete BPEL grammar.

the time to compute increases the cross-over takes place earlier - *i.e.*, the asynchronous solution becomes better at lower data lengths. It is interesting to note that at high data loads, the asynchronous solution with high computation time performs better than the synchronous solution with low computation time.

A cost function needs to take into account such details and, in general, this is a non-trivial exercise.

6 Related Work

Much work has been done on automatic parallelization of sequential programs based on PDGs e.g., [1, 4]. In contrast, the focus on this paper is on the use of PDGs in partitioning of concurrent composite web service applications. There are many references in the literature that are relevant to partitioning and clustering algorithms for parallel programs e.g., [16, 2, 12, 15, 13, 6, 18]. Though we plan to leverage the results from past work on program partitioning, we observe that there are some key characteristics that distinguishes our problem statement from the problem statements considered in past work. Specifically, our work is focused on decentralization of composite web services. This problem has some unique features compared to other partitioning problems, such as the presence of fixed tasks and portable tasks. In addition, most previous work on partitioning was focused on minimizing some objective function (such as execution time) for a single instance of a parallel program. The goal of this work is instead to maximize the throughput for the case when multiple instances of the parallel program (composite web service) are executed.

7 Conclusion

In this paper we have given a novel code partitioning algorithm which is applicable to decentralization of composite web services. We show how to generate code for the decentralized service based on the code partitions and the concurrency analysis of the global graph. We identify some of the factors that go into developing a comprehensive cost function that can help generate better partitions.

References

[1] W. Baxter and I. H. R. Bauer. The program dependence graph and vectorization. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1989.

[2] S. H. Bokhari. Partitioning Problems in Parallel, Pipelined, and Distributed Computing. *IEEE Transactions on Computers*, C-37:48–57, January 1988.

[3] Business Process Execution Language. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[4] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[5] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers; ISBN: 1558604758, November 1998.

[6] A. Gerasoulis, S. Venugopal, and T. Yang. Clustering Task Graphs for Message Passing Architectures. *Proceedings of the ACM 1990 International Conference on Supercomputing*, pages 447–456, June 1990. Amsterdam, the Netherlands.

[7] S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, and R. Neyama. *Building Web Services with Java: Making sense of XML, SOAP, WSDL and UDDI*. Sams; ISBN:0672321815, 2001.

[8] J. Krinke. Static slicing of threaded programs. In *Program analysis for software tools and engineering (PASTE 98)*, pages 35–42. ACMSOFT, 1998.

[9] J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. ACMSIGPLAN, May 1999.

[10] M. G. Nanda and N. Karnik. Synchronization analysis for decentralizing composite web services. In *Proceedings of the ACM Symposium on Applied Computing*, 2003. To appear.

[11] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACMSIGSOFT, 1984.

[12] V. Sarkar. *Partitioning and Scheduling Parallel Programs on Multiprocessors*. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, MA, 1989.

[13] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35:779–804, Sep/Nov 1991.

[14] V. Sarkar. A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence Graphs (Extended Abstract). *Springer-Verlag Lecture Notes in Computer Science*, 757:16–30, 1992. Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, Yale University, August 1992.

[15] V. Sarkar and D. Cann. POSC — a Partitioning and Optimizing Sisal Compiler. *Proceedings of the ACM 1990 International Conference on Supercomputing*, pages 148–163, June 1990. Amsterdam, the Netherlands.

[16] V. Sarkar and J. Hennessy. Partitioning Parallel Programs for Macro-Dataflow. *ACM Conference on Lisp and Functional Programming*, pages 202–211, August 1986.

[17] V. Sarkar and B. Simons. Parallel Program Graphs and their Classification. *Springer-Verlag Lecture Notes in Computer Science*, 768:633–655, 1993. Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, Oregon, August 1993.

[18] T. Yang and A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

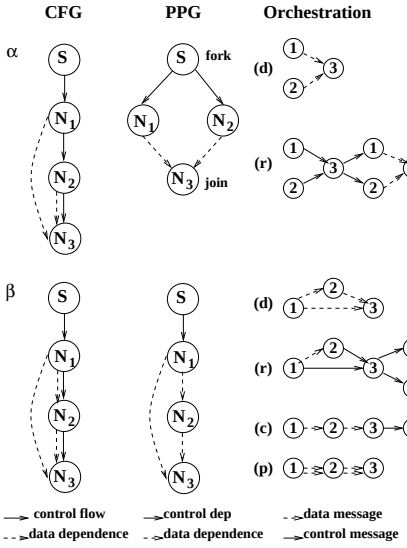

Figure 14. Synchronization protocols (d) for Direct Deposit, (r) for Request Response, (c) for Combined Direct Deposit and Request Response, and (p) for Pipeline, for PPGs generated from sequential code.

A Concurrency analysis

The classification scheme analyzes two incoming edges $N_1 \rightarrow N_3$ and $N_2 \rightarrow N_3$ at a node N_3. An α concurrency is differentiated from a corresponding β concurrency by the fact that in β concurrency there is a path in the PPG from N_1 to N_2 and no such path in the case of α concurrency. Sequential code in a CFG generates α and β concurrency, and code with conditionals / iteration generate *o*, α_1, β_1, α_2 and β_2 concurrency. *o* concurrency is a case where there is actually no concurrency since at runtime either $N_1 \rightarrow N_3$ or $N_2 \rightarrow N_3$ will be traversed and not both. (*o* concurrency is detected by finding the closest common ancestor of N_2 and N_3 in the PPG. If the ancestor is a `switch/if` node then we have *o* concurrency at N_3.)

In α_1 and β_1 concurrency N_3 does not post-dominate N_1 in the CFG and in α_2 and β_2 concurrency, N_3 does post-dominate N_1 in the CFG, but the value computed at N_1 may be killed along some path to N_3. Based on this classification there are different ways in which the nodes may communicate which have been graphically depicted in Figures 14 and 15.

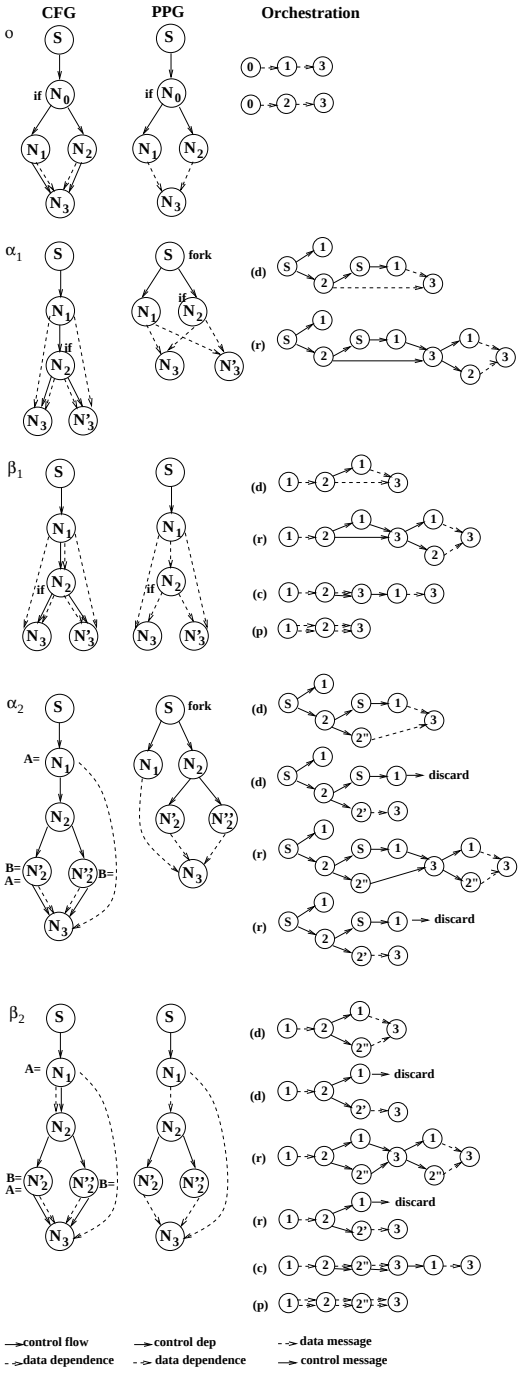

Figure 15. Synchronization protocols (d) for Direct Deposit, (r) for Request Response, (c) for Combined Direct Deposit and Request Response, and (p) for Pipeline, for PPGs generated from branched code.