Decentralizing Execution of Composite Web Services

Mangala Gowri, Satish Chandra, Vivek Sarkar. Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications

Presented by Xiang Zhang

October 30, 2009

Outline

- Background
- Problem statement
- Solution overview
- PDG construction
- The partition algorithm
- Cost Model
- Experimental Results
- Conclusion

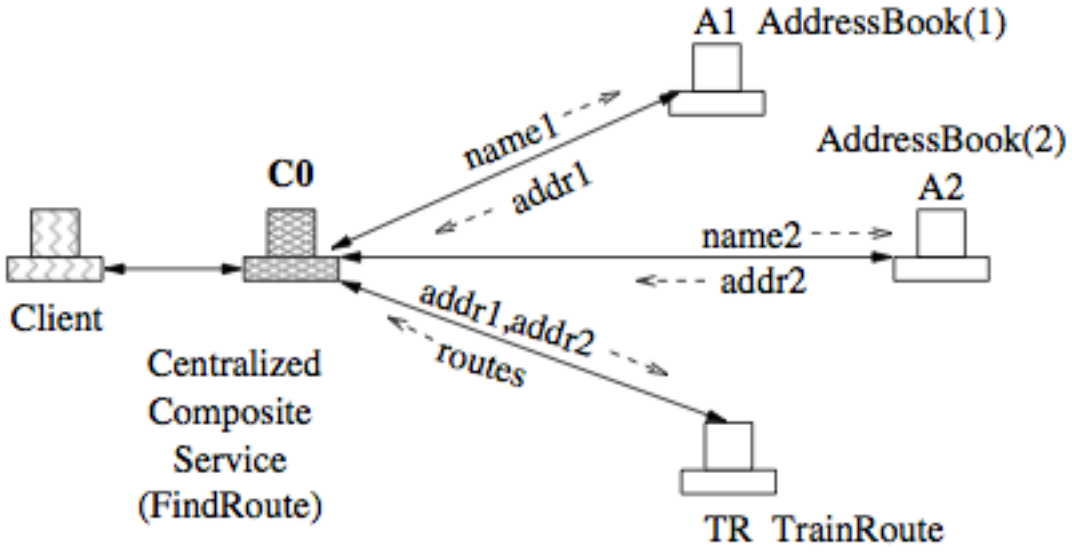
Background

- Distributed enterprise applications today are increasingly being built from services available over the web.
- Business Process Execution Language is now a de facto standard for composing web services.
- The orchestration of web services is typically under centralized control.

Background (cont.)

What's the limitation?

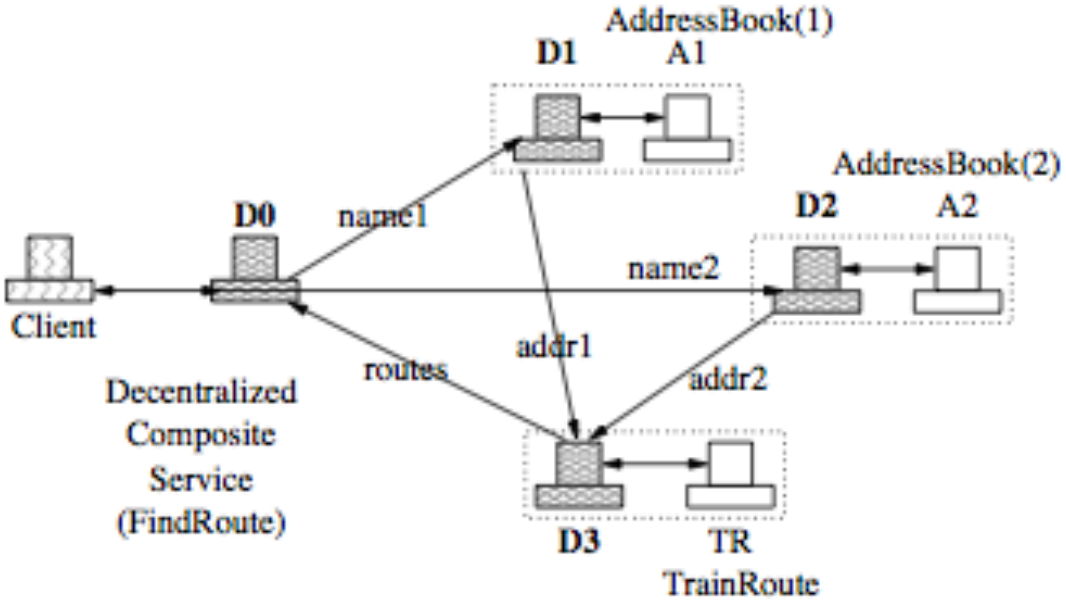
- Centralized engine is a possible performance bottleneck.
- Increased data transfer.



Background (cont.)

Improvement

- Partition a single process into multiple processes running on distributed engines without centralized control.

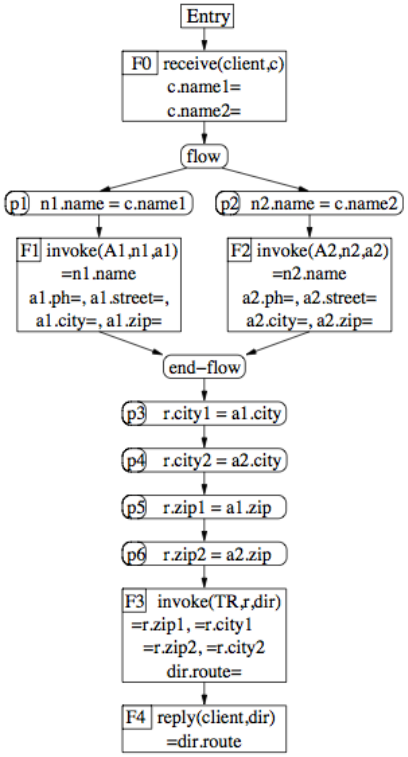


Problem Statement

For a CFG representation of a BPEL program, the problem is which nodes run on which servers.

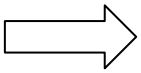
Key observations:

1) Services are only available on certain servers
2) BPEL program properties must be preserved
3) The number of ways to distribute the nodes is exponential



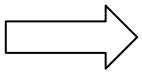
Solution Overview

Services are only available on certain servers



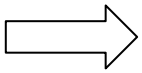
Divide the nodes into two categories: fixed nodes and portable nodes

BPEL program properties must be preserved



Construct a program dependence graph for a BPEL program

The number of ways to distribute the nodes are exponential



Propose a merge-by-def-use partitioning algorithm to prune the space of possible solutions

Classifying nodes

- Fixed nodes:

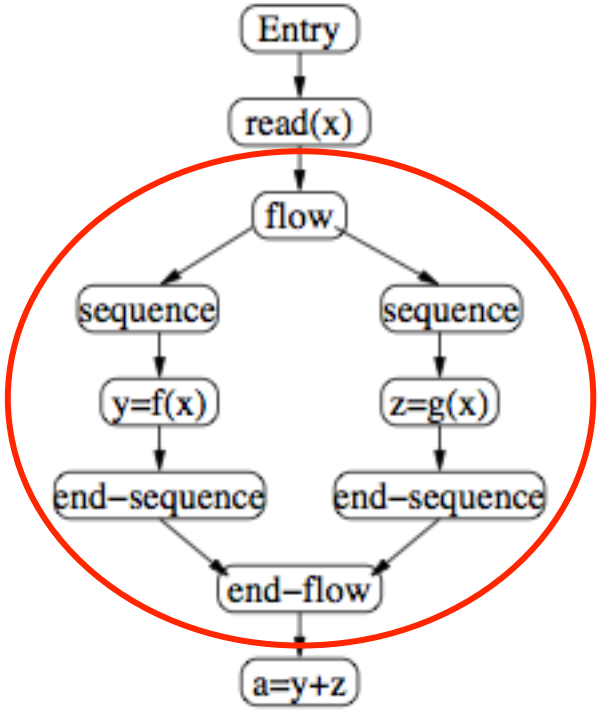
  Fixed nodes are those nodes can only executed on certain servers, e.g. *invoke*, *receive*, *reply*.

- Portable nodes:

  Portable nodes are those nodes can be executed on any servers, e.g. *compute*, *assignment*.

PDG Construction

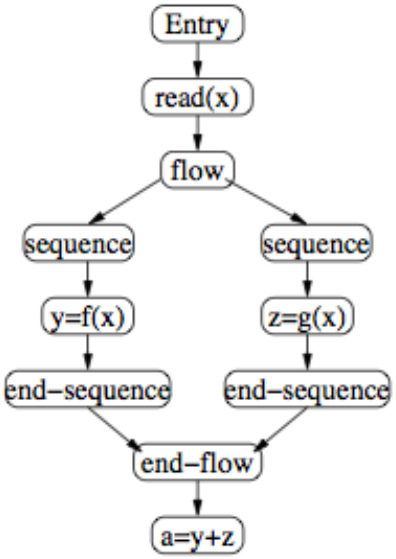
- Besides the traditional PDG, extra control and data dependences need be inserted to model the parallel constructs.

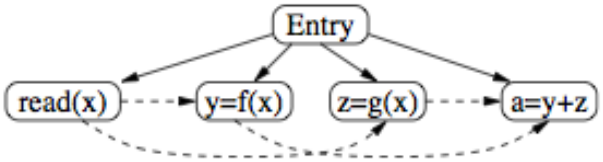


PDG Construction (Cont.)

Insert extra control dependence edges:

- Step 1: each *sequence* node is control dependent on *flow* node, and each node within a parallel section is control dependent on its *sequence* node
- Step 2: eliminate *flow* and *sequence* node by making every node which is control dependent on *flow* control dependent on the node on which the *flow* node control dependent.

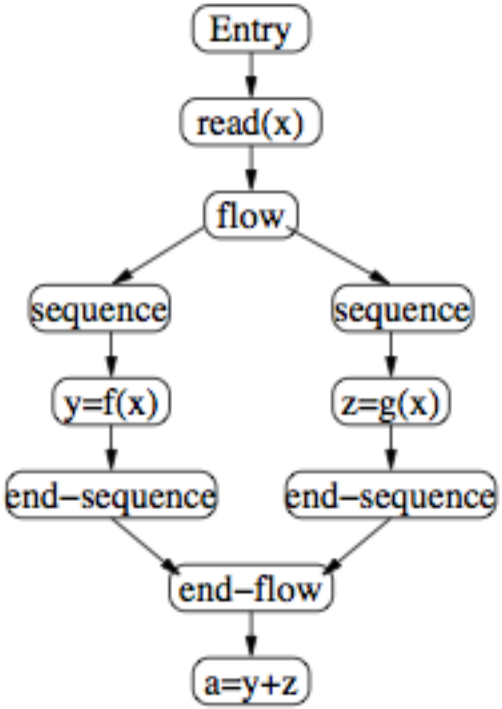




PDG Construction (Cont.)

Insert extra data dependence edges:

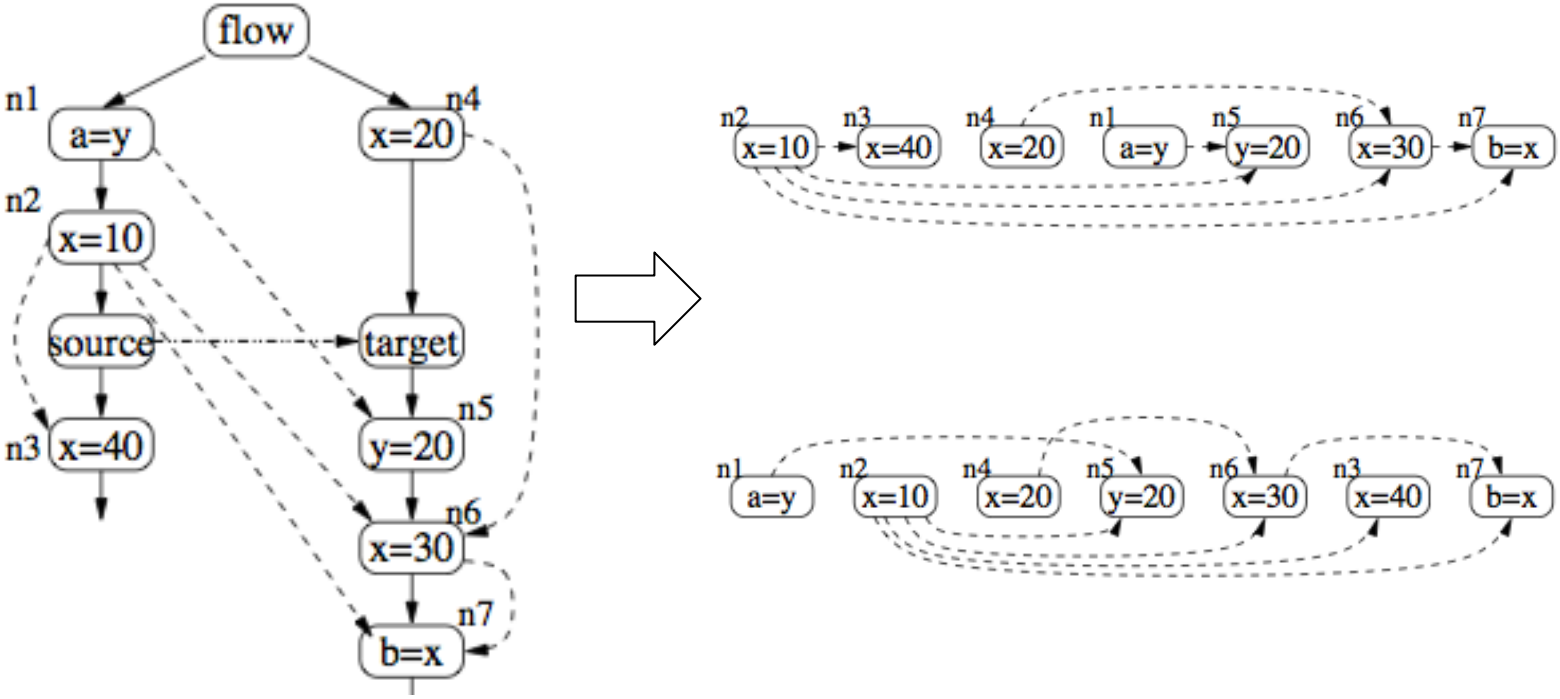
- Step 1: Determine the dependences with statements outside a flow construct.
- Step 2: Determine the dependences among parallel sections.



PDG Construction (Cont.)

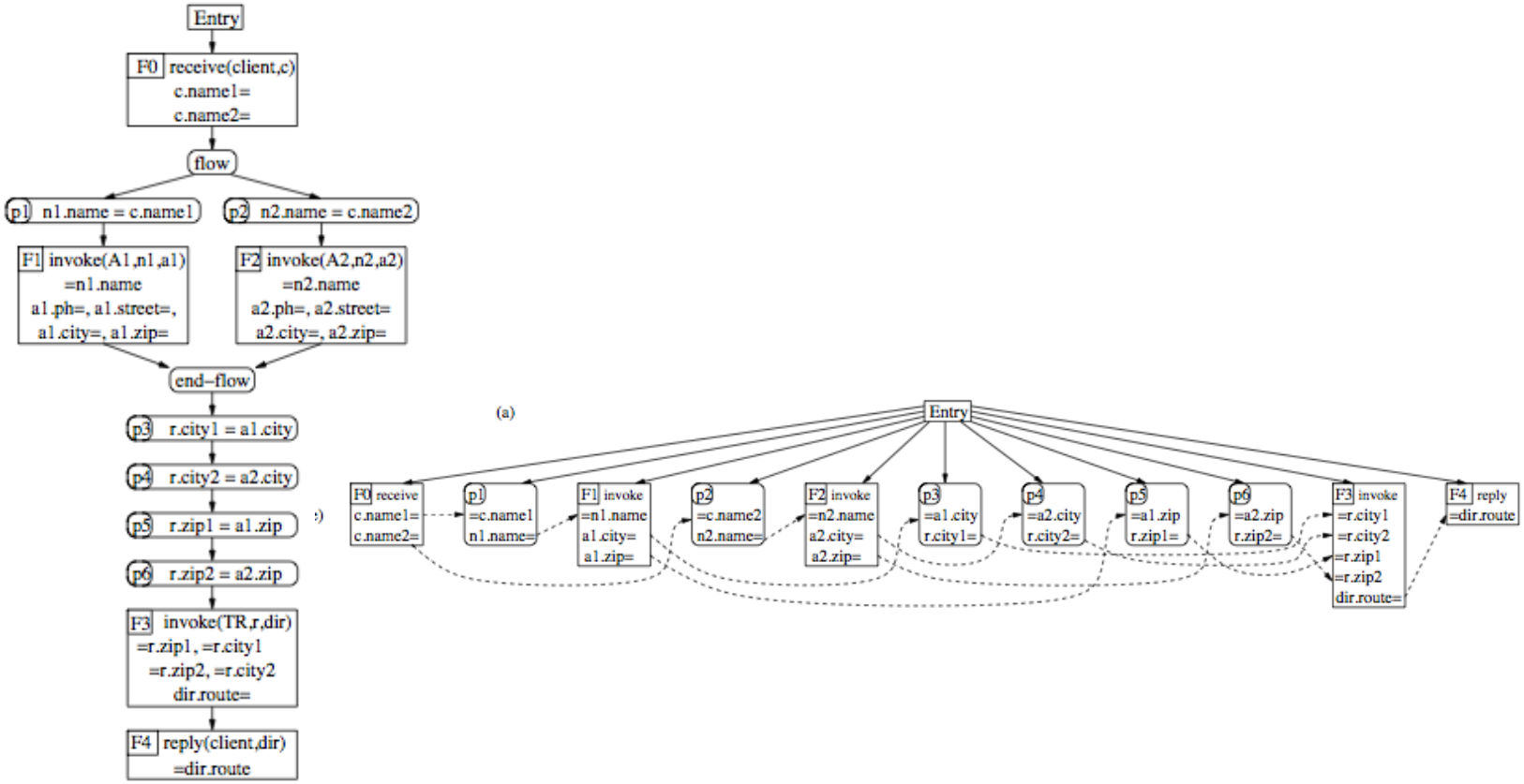
Insert extra data dependence edges:

- Step 1: Determine the dependences with statements outside a flow construct.
- Step 2: Determine the dependences among parallel sections.



PDG Construction (Cont.)

An example:

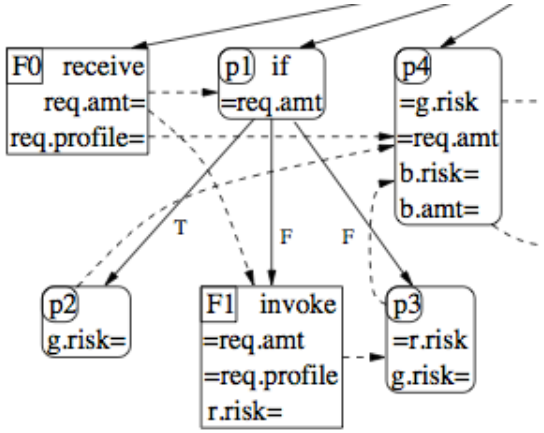


The Partitioning Algorithm

- The algorithm is based on the idea of merging tasks along flow dependence edges for two reasons:
  1) Performance implication
  2) Effectively prune the space of possible solutions

The Partitioning Algorithm

- Step 1: locate a control node T_c whose child nodes are all leaf nodes. Repeat step 2 through step 8 for these leaf nodes. Continue all control nodes have been processed.
- Step 2: Identify the set of flow dependence edges, E, that pertain to a flow dependence between siblings with the control dependence condition chosen in step 1, such that at least one of the siblings is a portable task. Pick an edge in E and merge the source and destination task of the edge. Union the dependences.



The Partitioning Algorithm (cont.)

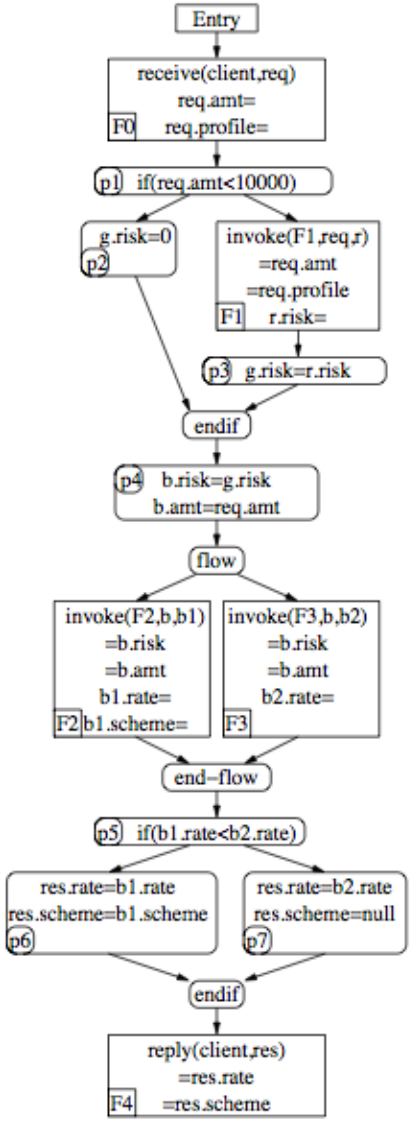
- Step 3: when a portable task gets merged with a fixed task the combined task is a fixed task. When a portable task gets merged with another portable task the combined task is also marked as a portable task.
- Step 4: When a node is merged with a sibling that is not its lexical neighbor, we need to ensure that no dependence conditions are violated by checking if the merge can introduce a dependence cycle.
- Step 5: Exhaustively consider all merging configurations of siblings that can be generated by merging some subset of the flow dependence edges in E.

The Partitioning Algorithm (cont.)

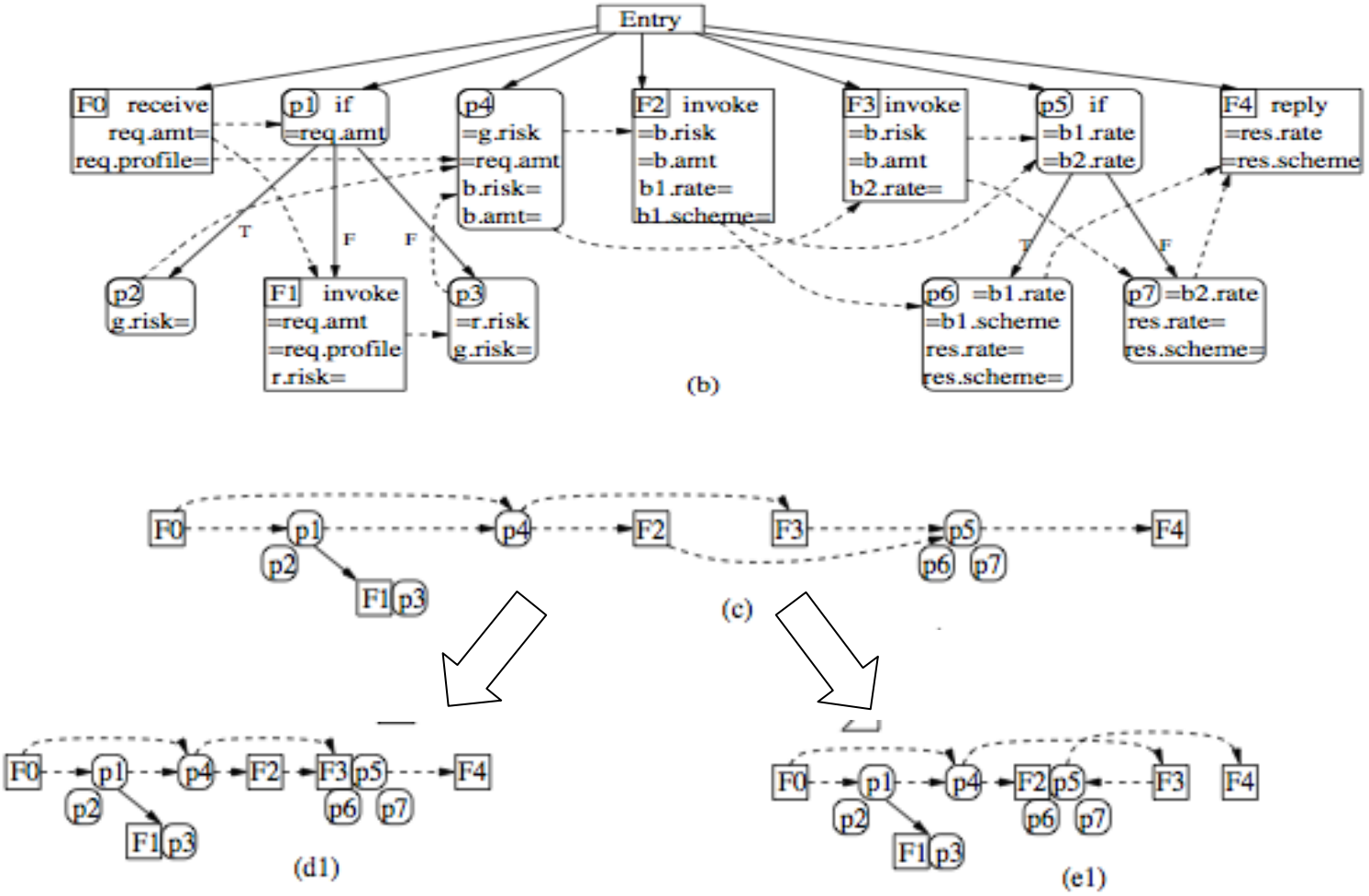
- Step 6: Choose the merging configuration from step 5 that is likely to yield the best overall throughput value, using the cost model discussed later.
- Step 7: Any remaining portable tasks that are not merged with a fixed task are merged with the parent.
- Step 8: Once a region has been merged, we treat the whole subgraph as a single node for the purpose of merging at the next higher level. Union the dependence.

The Partitioning Algorithm (cont.)

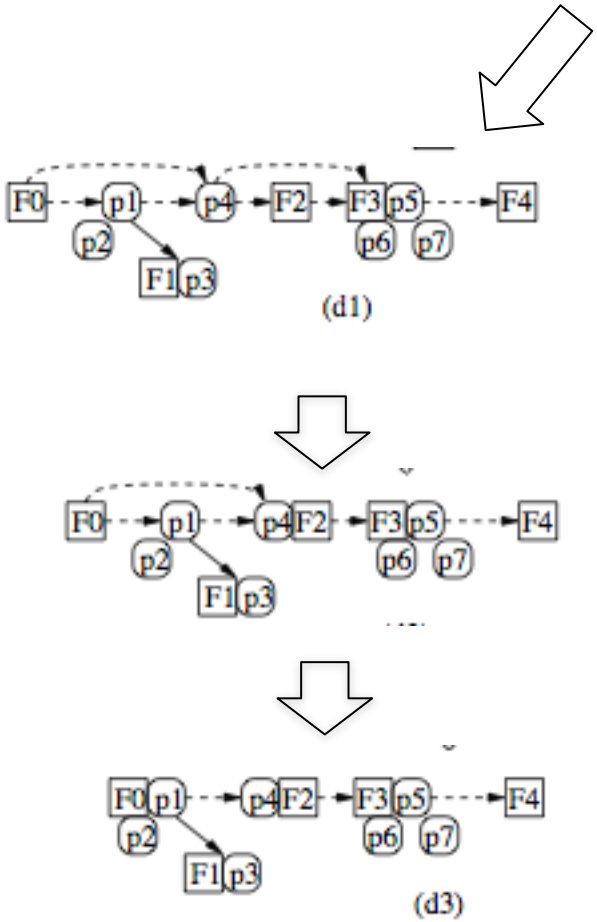
An example:



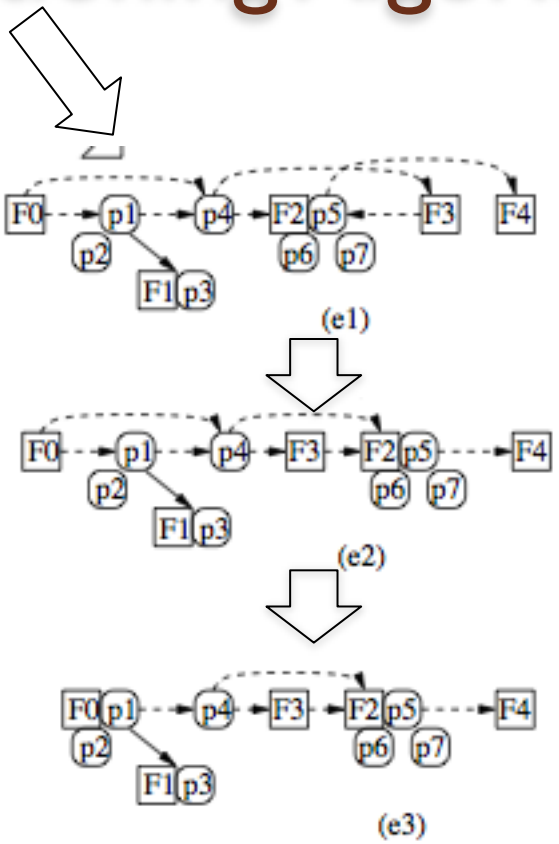
The Partitioning Algorithm (cont.)



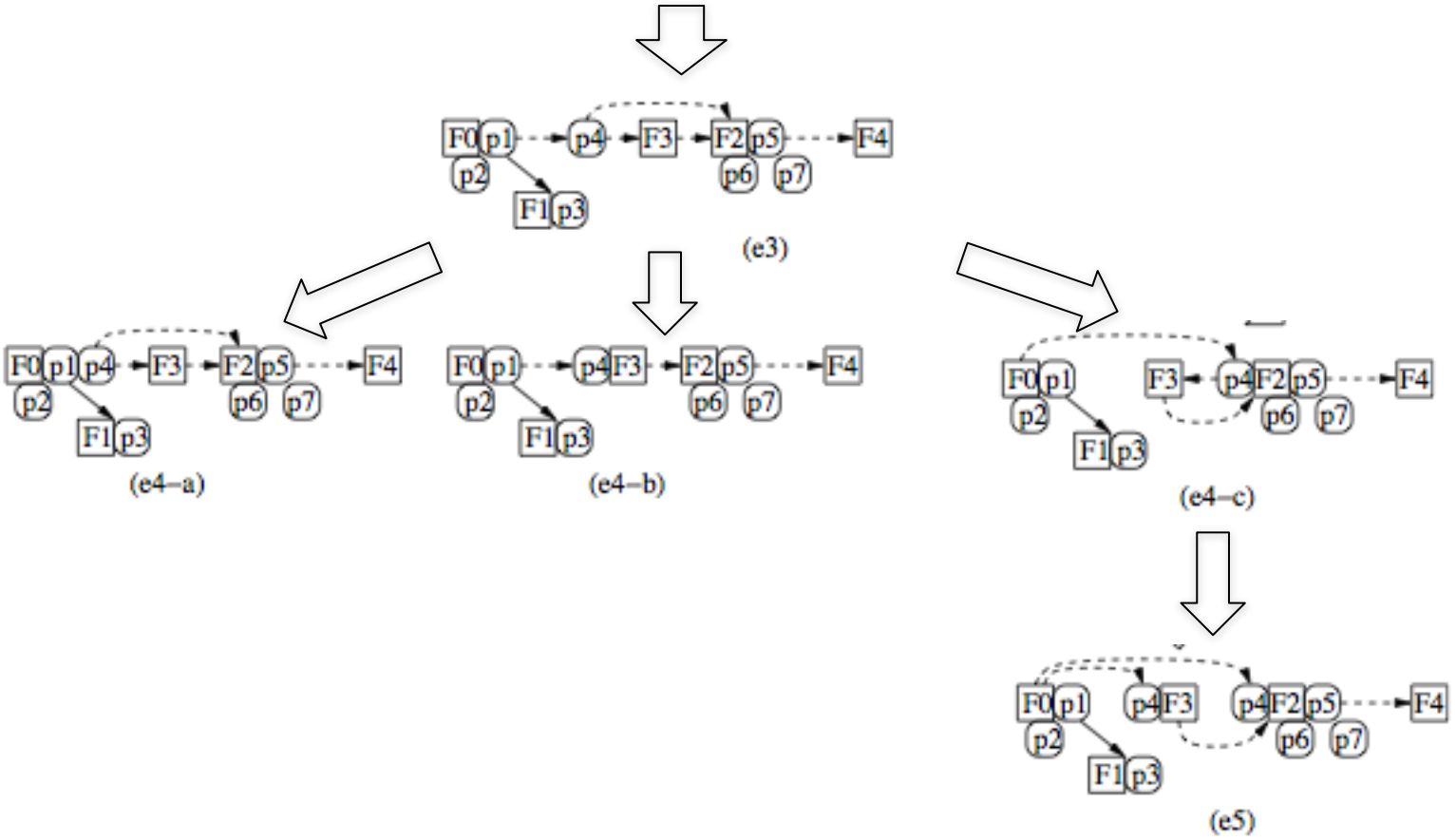
The Partitioning Algorithm (cont.)



The Partitioning Algorithm (cont.)



The Partitioning Algorithm (cont.)



Cost Model

- The cost model developed is focused on throughput as its objective function.
- Places where time is consumed:

  1) Service invocation (C_I)

  2) Receive (C_R)

  3) Reply (C_L)

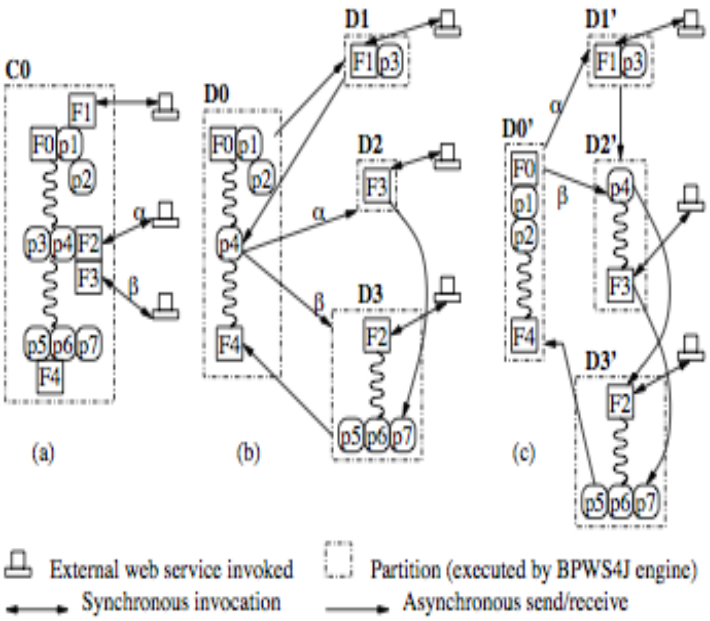
  4) Portable tasks (C_{pi})

  5) Send (C_S)

- The load at one server is simply the aggregation of all activities at that server.

Cost Model (Cont.)

An example:

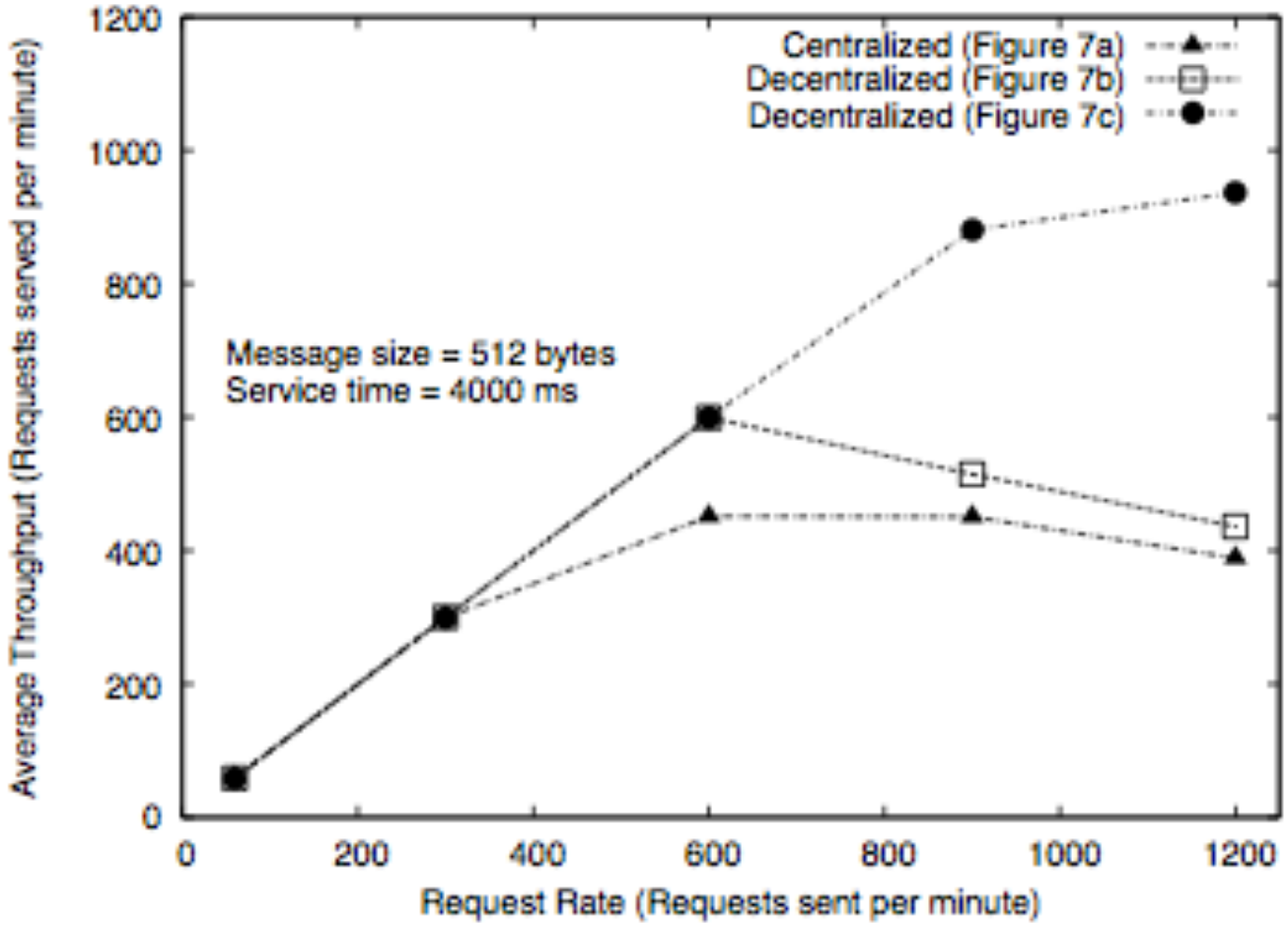


Peak throughput(C_0)=
Capacity(C_0)/(C_R+C_L+7*C_P+3*C_I)

Peak throughput(D_0)=
Capacity(D_0)/(C_R+C_L+3*C_S+2*C_R)

Peak throughput(D_0')=
Capacity(D_0')/(C_R+C_L+2*C_P+2*C_S+C_R)

Experimental Results



Conclusion

- PDG construction and node reordering
- A technique to estimate the throughput of a single node.
- A new code partitioning algorithm that is applicable to decentralization of composite web services.

Thank you!