Decentralizing Execution of 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

Distributed enterprise applications today are increasingly being built from services available over the web. A unit of functionality in this framework is a web service, a software application that exposes a set of "typed" connections that can be accessed over the web using standard protocols. These units can then be composed into a *composite* web service. BPEL (Business Process Execution Language) is a high-level distributed programming language for creating composite web services.

Although a BPEL program invokes services distributed over several servers, the *orchestration* of these services is typically under centralized control. Because performance and throughput are major concerns in enterprise applications, it is important to remove the inefficiencies introduced by the centralized control. In a distributed, or decentralized orchestration, the BPEL program is partitioned into independent sub-programs that interact with each other without any centralized control. Decentralization can increase parallelism and reduce the amount of network traffic required for an application.

This paper presents a technique to partition a composite web service written as a single BPEL program into an equivalent set of decentralized processes. It gives a new code partitioning algorithm to partition a BPEL program represented as a program dependence graph, with the goal of minimizing communication costs and maximizing the *throughput* of multiple concurrent instances of the input program. In contrast, much of the past work on dependence-based partitioning and scheduling seeks to minimize the *completion time* of a single instance of a program running in isolation. The paper also gives a cost model to estimate the throughput of a given code partition. Experimental results show that decentralized execution can substantially increase the throughput of example composite services, with improvements of approximately 30% under normal system loads and by a factor of two under high system loads.

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors — optimization

General Terms

Languages, Performance

Keywords

Workflow, Business Process, Program Dependence Graph

1. INTRODUCTION

The idea of "software as a service" has recently gained tremendous importance because of standardization of the way in which software may be delivered as a service over the network. For example, the *Web Services* standard [9] provides primitives for communication, messaging, and naming of software applications (or services) available over the Internet. Once software applications are available as a service, a *composite* service can be created by accessing a set of services programmatically using some scripting language. A composite service expresses a *business process*, which captures a particular intra or inter enterprise workflow. This paper is concerned with efficient execution of such enterprise applications.

Business Process Execution Language (BPEL)[4, 11] is a language that is used to specify a composite web service. The language consists of standard flow constructs for sequential, conditional and concurrent execution of *activities* such as invoking a service or standard assignment and arithmetic operations. The language under consideration is summarized in Table 1. BPEL is a standard being developed jointly by IBM, Microsoft, BEA and other companies, and is rapidly gaining importance in the enterprise computing landscape. Note that Web Services and BPEL are particular instances of generic concepts of software-as-service, service composition and a service composition language, and our techniques apply to other instances as well.

Figure 1(a) illustrates BPEL concepts via a small example of a composite service. The composite service, *FindRoute* is built from two address book services, *AddrBook(1)* and *AddrBook(2)*, and a *TrainRoute* service. The *AddrBook* services take as input a name and return the address of that individual. The *TrainRoute* service takes as input two addresses and returns the train schedules from one address to the other. The *FindRoute* service sends, in parallel, *name1* as input to service *AddrBook(1)*, and *name2* as input to service *AddrBook(2)*. This parallelism is expressed using

Table 1: Summary of BPEL constructs and notation

BPEL construct	Description	Notation
	Control Flow Constructs	
`sequence`	sequential flow	`sequence` ... `end-sequence`
`switch`	conditional flow	`switch` ... `end-switch`
`while`	iterative flow	`while` ... `end-while`
`pick`	non-deterministic conditional flow	`pick` ... `end-pick`
`flow`	concurrent flow similar to cobegin-coend	`flow` ... `end-flow`
`link`	wait-notify type of synchronization	`source(linkId)`, `target(linkId)`
	Data Structures	
`variable`	variables include a set of parts analogous to fields	`variableName` { `part1, part2,` ... `partn` }
	Activities	
`invoke`	synchronous (blocking) invocation on a partner P, sending data from an input variable *in* and receiving the response in the output variable *out*	`invoke`(P, *in*, *out*)
`send`[1]	asynchronous (oneway, nonblocking) invocation on a partner P, sending data using an input variable *in* (no response variable)	`send`(P, *in*)
`receive`	blocking receive of data from a partner P into a variable *var*	`receive`(P, *var*)
`reply`	send response to a partner P from a variable *var*	`reply`(P, *var*)
`assign`	assignment. Multiple assignments can be specified in a single `assign` statement, which executes atomically	`var1.p1.g1 = var2.p1.g3`
`compute`	arithmetic or logical operation	

BPEL's `flow` construct; the availability of the flow construct allows a programmer to identify parts of the process that can run concurrently without depending on the implementation to extract the parallelism automatically. The two addresses are returned to the *FindRoute* service, the city and zip code of each address is extracted and then sent to the *TrainRoute* service which returns the train routes from the first address to the second. The BPEL code for the *FindRoute* service is shown in slightly sugared form (C0) in Figure 1(a). This code is interpreted/executed by a *BPEL engine* (such as BPWS4J [5]) that acts as a centralized coordinator for all interactions among the component services. This type of execution is known as *centralized orchestration*.

Note that the requests from and replies to the client are handled by the central server, in this case node C0. Note also that the invoked services, namely *AddrBook(1)*, *AddrBook(2)* and *TrainRoute* are available only on specific nodes A1, A2, and TR respectively. The rest of the code, which may include some business logic, is really "portable" code or "glue" code that in principle can be run anywhere, not necessarily on the central server.

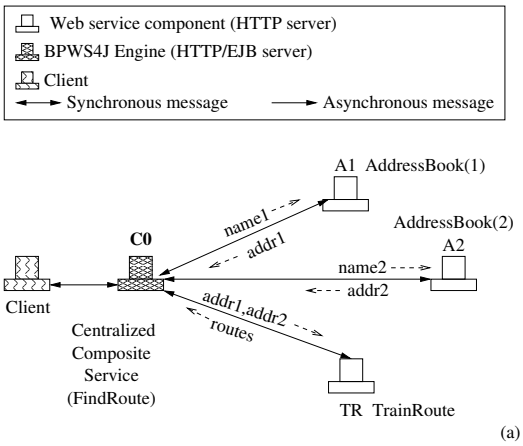
Now consider the same example in Figure 1(b). Here the original BPEL code (C0) has been partitioned into four components that are executed by four distributed engines (D0, D1, D2 and D3). Together, the four engines perform the role of C0. This form of orchestration is termed *decentralized orchestration*. 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 a centralized coordinator. For example, the addresses generated at *AddrBook(1)* and *AddrBook(2)* can be forwarded directly to *TrainRoute* (via D3), as shown in Figure 1(b). Note that while specific services such as *TrainRoute* still reside on the same nodes as before, the glue code is run on different nodes, compared to Figure 1(a).

Decentralization may lead to increased parallelism in executing glue code and reduced message overhead since fewer messages are sent. Decentralized orchestration brings performance enhancements, namely better response time and throughput, to the composite service execution. However, decentralization does require a BPEL engine at all participating nodes. This is a reasonable assumption to make in the context of modern enterprise servers for two reasons. One reason is that the ability of executing BPEL (or another service specification) has become standard software infrastructure in application servers such as WebSphere [1]. The second reason is that the application that the server exports as a web service may itself be implemented as a BPEL program behind the scenes, requiring a BPEL execution environment.

Problem Statement. It is clearly desirable to run composite services in a decentralized manner, assuming of course that the infrastructure supports this execution model. Yet creating and deploying decentralized versions is burdensome. Given a BPEL program for centralized orchestration, can one decentralize it automatically? On the surface, this problem has many similarities with automatic partitioning of programs for multiprocessor execution. However, there are some important differences, as explained in the technical overview below.

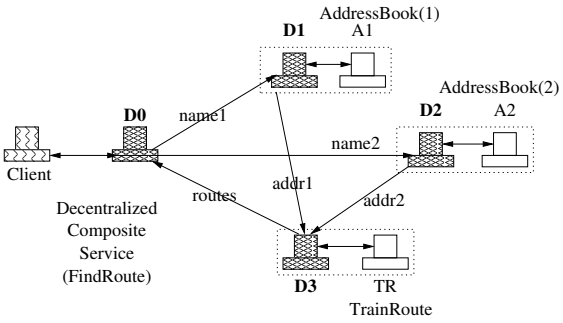
Technical Overview. Figure 2(a) shows the example in Figure 1(a) as a control-flow graph (CFG). We designate

[1]The correct BPEL syntax for oneway messaging is `invoke`(P, *in*). However, in this paper we use the keyword "`send`" to highlight the difference between oneway, nonblocking, asynchronous `invoke` and synchronous `invoke`.



(a)

```
C0
receive(client, c{name1,name2})
flow
   sequence
      n1.name = c.name1
      invoke(A1, n1{name}, a1{ph,street,city,zip})
   end−sequence
   sequence
      n2.name = c.name2
      invoke(A2, n2{name}, a2{ph,street,city,zip})
   end−sequence
end−flow
r.city1 = a1.city
r.city2 = a2.city
r.zip1 = a1.zip
r.zip2 = a2.zip
invoke(TR, r{city1,city2,zip1,zip2}, dir{routes})
reply(client, dir{routes})
```



(b)

```
D0
receive(client, c{name1,name2})
n1.name = c.name1
n2.name = c.name2
flow
    send(D1, n1{name})
    send(D2, n2{name})
end−flow
receive(D3, dir{routes})
reply(client, dir{routes})
```

```
D1
receive(D0, n1{name})
invoke(A1, n1{name}, a1{ph,street,city,zip})
r1.city = a1.city
r1.zip = a1.zip
send(D3, r1{city,zip})
```

```
D2
receive(D0, n2{name})
invoke(A2, n2{name}, a2{ph,street,city,zip})
r2.city = a2.city
r2.zip = a2.zip
send(D3, r2{city,zip})
```

```
D3
flow
    receive(D1, r1{city,zip})
    receive(D2, r2{city,zip})
end−flow
r.city1 = r1.city
r.zip1 = r1.zip
r.city2 = r2.city
r.zip2 = r2.zip
invoke(TR, r{city1,city2,zip1,zip2}, dir{routes})
send(D0, dir{routes})
```

Figure 1: Centralized and Decentralized Architecture

`receive` and `reply` nodes of the CFG as *fixed* nodes that must execute at the central server; `invoke` nodes are also designated as fixed nodes that must be colocated with the corresponding web service; all other activities are designated as *portable* nodes. We use the convention that the fixed nodes are represented by rectangular boxes and the portable nodes by rounded boxes. A program dependence graph (PDG) [6] consists of control dependence and data dependence edges superimposed on the same set of nodes which denote statements and predicate expressions of the CFG. The PDG corresponding to Figure 2(a) is shown in Figure 2(e). Unlike [12], we do not work directly with concurrent PDGs. Instead, a conventional PDG with a few extra edges can represent all the information we need for our purposes. We show how to create these extra edges in Section 2.

In a centralized execution, all portable nodes of a PDG are mapped to the central server, leaving fixed nodes at whichever servers they belong. We would like to explore alternative partitioning of the nodes of the PDG in which some of the portable nodes are assigned to nodes other than the central servers, corresponding to a decentralized execution. For the PDG of Figure 2(e), Figure 2(b) gives the partitioning in which all portable nodes run on the central server (centralized execution), and Figures 2(c) and 2(d) give two possible partitionings in which we have grouped portable nodes with fixed nodes other than the central server (decentralized execution). The data dependence edges in Figures 2(c) and 2(d) refer to messages between partitions, for which `send`/`receive` pairs need to be generated (see code in Figure 1(b) which corresponds to Figure 2(d)). We can see that the number of messages in 2(c) and 2(d) is lower than in 2(b). The experimental results (Section 5) validate that the performance of 2(b) is inferior to the decentralized options (2(c) and 2(d)).

In principle, we could use a PDG-based code partitioning algorithm designed for multiprocessor execution. Such an algorithm creates independently schedulable tasks at the granularity of partitions of a PDG. To reduce overhead, such algorithms try to merge several PDG nodes to create a larger partition, possibly sacrificing parallelism. An example of a merging algorithm that iteratively merges nodes that have the same control dependence condition can be found in [16].

However, the problem of partitioning PDGs for composite services, has an additional constraint that the node merging algorithm must create partitionings such that each partition has exactly one fixed node and zero or more portable nodes. By definition, a fixed node cannot be merged with another fixed node. Portable nodes cannot form a partition without a fixed node, because fixed nodes are the ones that have execution resources. Further, the objective function for composite web services is to maximize throughput, whereas the objective function typically used for multiprocessor execution is to minimize completion time. Due to these differ-

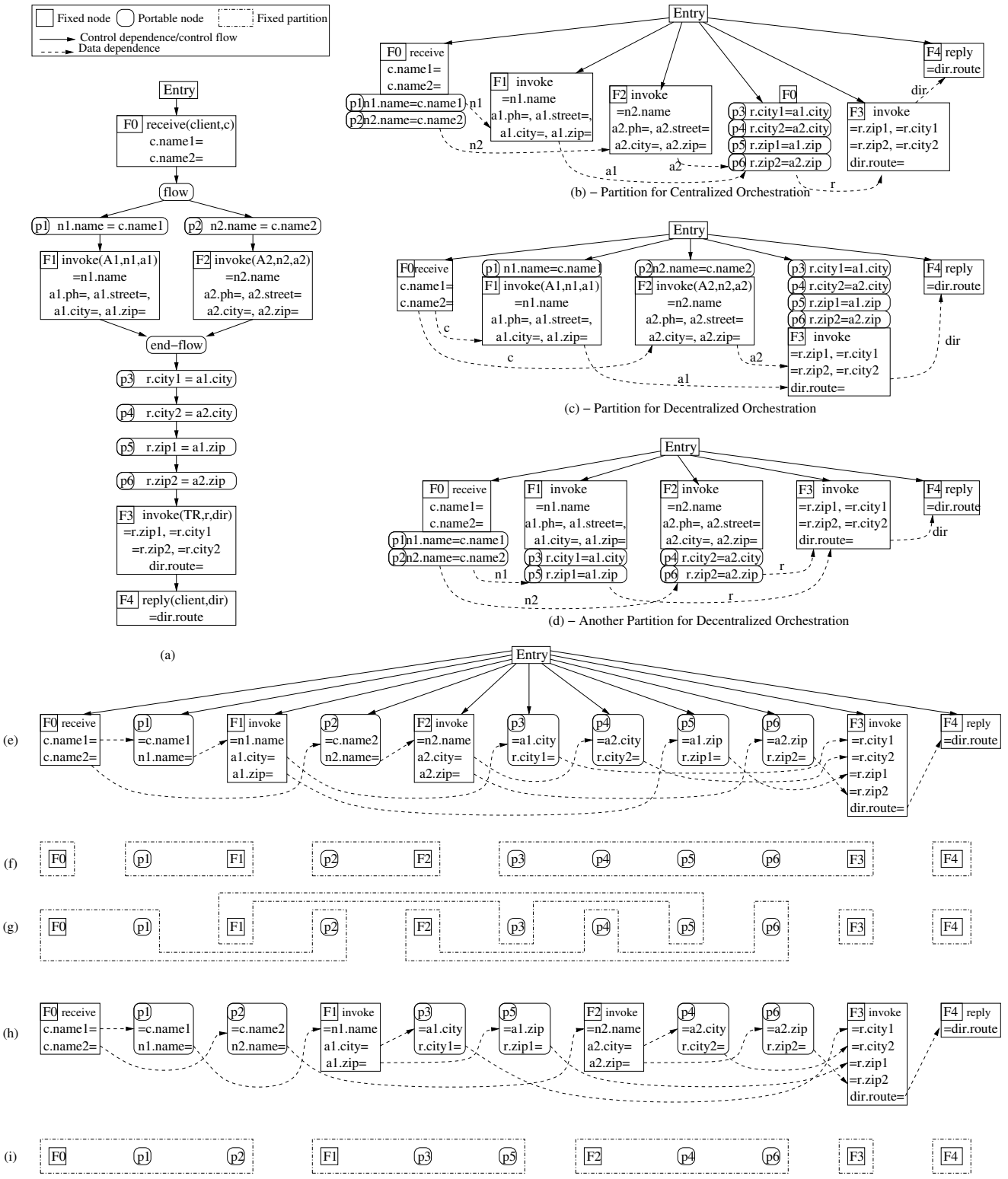


Figure 2: For the example in Figure 1: (a) the Threaded CFG (TCFG); (b) a partitioning of the PDG that corresponds to centralized execution; (c) and (d) two partitionings of the PDG that correspond to decentralized execution; (e) the PDG corresponding to the TCFG; (f) a partitioning generated by merging lexical siblings in the PDG; (g) the desired merging scheme where siblings connected by data dependence edges are merged; (h) the reordered PDG corresponding to (g); (i) the partitioning generated by merging neighboring siblings in the reordered PDG in (h). Note that every partition has exactly one fixed node and zero or more portable nodes.

ences, a node merging algorithm designed for multiprocessor execution may not perform well in the case of partitioning for composite web services.

Returning to our example of Figure 2, while the number of inter-partition edges in both (c) and (d) is the same, it can be seen that the data passed on the wire is larger in Figure 2(c). In Figure 2(d), for example, the `city` and `zip` information is extracted at partition F_1 and F_2 and sent to F_3, whereas in Figure 2(c), the entire `address` is sent from F_1 and F_2 to F_3, and the relevant data is then extracted at F_3. The partitioning of Figure 2(c) corresponds to the partition configuration shown in Figure 2(f) and the partitioning of Figure 2(d) corresponds to the partition configuration of Figure 2(g). In Figure 2(f), each partition merges only siblings in the PDG that are also lexical neighbors, however, Figure 2(g) tries to merge siblings that are not lexical neighbors. A partitioning algorithm that performs only lexical neighbor sibling merges will miss out on this more efficient partitioning. As one might expect, in our experiments we see a noticeable performance difference between (c) and (d) when the size of the entire `address` relative to `city` and `zip` is significant.

Our premise is that merging nodes along flow-dependence edges will result in a more communication-efficient partitioning. Since the source and destination nodes of a flow dependence may not be lexically adjacent, merging along flow-dependence edges is equivalent to first *reordering* the nodes so that the source and destination become adjacent, and then merging the nodes. Consider the PDG in Figure 2(h), in which some of the nodes have been reordered from Figure 2(e). On this new PDG, it is now possible to generate the partitioning of Figure 2(d) just by combining neighboring siblings as shown in Figure 2(i). However, before we generate this partitioning, we need to validate that the reordered PDG is isomorphic [10] with the original PDG.

The main contributions of this paper are as follows:

1. We give a heuristic solution to the problem of decentralization of BPEL programs. To the best of our knowledge, this is the first proposed solution to the decentralization problem for BPEL programs.

2. We introduce a cost model to guide the decentralization algorithm that is based on *throughput* as the primary performance metric, for multiple instances of a program running on a server. In contrast, much of the past work on dependence-based partitioning and scheduling seeks to minimize the *completion time* of a single instance of a program running in isolation.

3. Our experimental results show significant benefits from decentralization for four sample composite services. For the same hardware resource, the decentralized versions performed better than the centralized version across a range of parameters for request rates and message sizes. The results show that decentralized execution can substantially increase the throughput of example composite services, with improvements of approximately 30% under normal system loads and by a factor of two under high system loads.

Organization. The rest of the paper is organized as follows. Section 2 describes how we construct PDGs for our program model (a subset of BPEL), and how we test for legality of node reordering. Section 3 describes our partitioning algorithm and Section 4 describes our cost function. Section 5 contains experimental results for decentralization of four example composite services. Finally, Section 6 discusses related work, and Section 7 contains our conclusions.

2. PDGS AND NODE REORDERING

In this section, we describe how we build the program dependence graph (PDG) representation assumed in our work. We start with a *Threaded Control Flow Graph* (TCFG) representation as shown in Figure 3(a). (We had also seen an example of a TCFG earlier in Figure 2(a).) To obtain a PDG representation of this parallel program representation, we need to insert extra control and data dependences that model the parallel constructs. For the control dependence edges, each `sequence` node representing a parallel section is control dependent on its `flow` node, and each node within a parallel section is control dependent on its `sequence` node. After these control dependence relationships are established, the `sequence` and `flow` nodes are eliminated as follows: let N_c be the node on which the `flow` node is control dependent. Then, every node that is control dependent on a `sequence` node below the `flow` is made control dependent on N_c and the `sequence` and `flow` nodes are eliminated. In this way, we obtain a single integrated PDG from the TCFG, unlike (say) the *Threaded* PDG (TPDG) approach [12] in which PDGs for parallel tasks are represented as separate sub-PDGs embedded in the parent PDG. Our motivation for working with a single integrated PDG is that it enables the partitioning algorithm to treat statements from within and across parallel tasks uniformly. The PDG for the TCFG in Figure 3(a) is shown in Figure 3(b).

For the data dependence edges, we must preserve the ordering constraints implicit in the TCFG. To accomplish this task, we first compute all data dependences (flow dependences, output dependences and anti dependences) necessary to preserve correct execution order within each parallel section. Next, we insert additional dependence edges to capture the ordering constraints across parallel sections. There are two cases that require insertion of these extra edges in the PDG:

1. Dependences with statements outside a flow construct — these dependences occur between a statement in a parallel section and a statement outside its flow construct that precedes or succeeds the flow construct containing the parallel section.

2. Dependences among parallel sections — these dependences capture explicit synchronization-based ordering between two parallel sections (as specified by the BPEL `link` construct).

We discuss these two cases in more detail below. These extra dependence edges are analogous to the *constraint edges* introduced by Horwitz et al [10] in the context of merging variants of PDGs into a single PDG, and to the *synchronization edges* introduced by Sarkar [17] for extending PDGs to parallel programs. Our code partitioning algorithm (Section 3) relies on the correctness of these extra dependence edges, because it attempts to reorder nodes in a region of a PDG in search of a more efficient partitioning and it must ensure that all reorderings considered are legal.

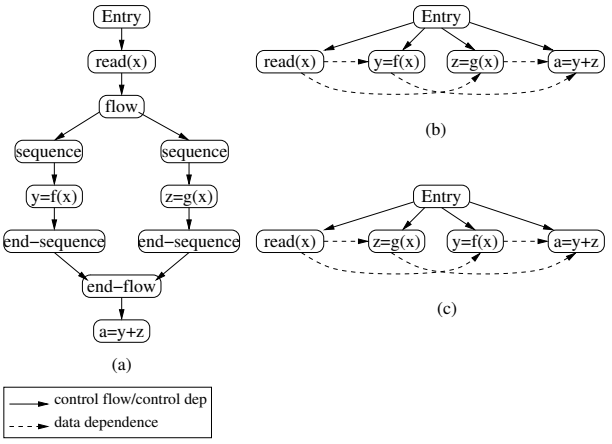


Figure 3: (a) A Threaded Control Flow Graph (TCFG), (b) its PDG representation where parallel sections have been merged into a single PDG without parallel sections, (c) a PDG isomorphic to that shown in (b).

Dependences with statements outside a flow construct. Consider a statement S_j in a parallel section. If there is a statement S_i outside the flow construct containing the parallel section, such that S_i precedes the flow construct and S_i and S_j perform interfering (write-read, read-write, or write-write) accesses on the same variable, then a corresponding (flow, anti or output) dependence edge is inserted in the PDG from S_i to S_j. Likewise, if there is a statement S_k outside the flow construct containing the parallel section, such that S_k follows the flow construct and S_j and S_k perform interfering (write-read, read-write, or write-write) accesses on the same variable, then a corresponding (flow, anti or output) dependence edge is inserted in the PDG from S_j to S_k. In Figure 3(b), we see an example of four such dependence edges being inserted in the PDG, due to dependences on variables x, y, and z. Figure 3(c) shows another legal ordering of nodes of this PDG.

Dependences among parallel sections. By default, parallel sections execute independently and there is no need to insert any dependence edges between two parallel sections in the same flow construct. However, dependence edges do need to be inserted to capture explicit synchronization-based ordering between two parallel sections, as specified by the BPEL `link` construct. Specifically, if there is an explicit synchronization link from `source` in parallel section θ_1 to `target` in parallel section θ_2, then the semantics imply that every node that precedes the `source` node must execute before any node that follows the `target` node. There is no inter-process ordering implication for nodes that follow the `source` node or precede the `target` node. To enforce legal orderings, we first insert a *synchronization edge* from `source` to `target`. Then, for every definition of x that precedes `source` in θ_1 we create a dependence edge to a use or definition of x that follows `target` in θ_2; and for every use of x that precedes `source` in θ_1 we create a dependence edge to a definition of x that follows `target` in θ_2. For example, in Figure 4(a) `source` $\rightarrow$ `target` is a synchronization edge. (Note that in the PDG, we do not differentiate between the

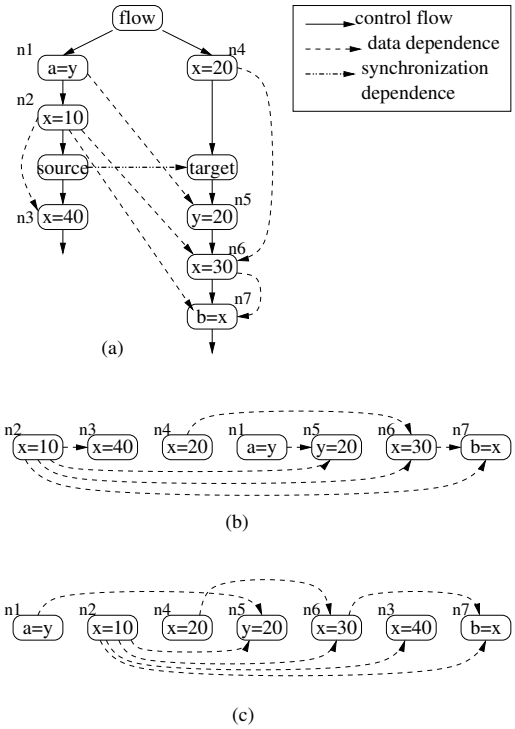


Figure 4: (a) Threaded CFG showing Synchronization Dependence, (b) and (c) Isomorphic PDG variants produced after merging the parallel sections

different forms of dependence edges for the purpose of determining legal orderings.) The semantics of synchronization enforces the following dependence edges across parallel sections: $n_1 \rightarrow n_5$, $n_2 \rightarrow n_6$, and $n_2 \rightarrow n_7$. In Figure 4(b), n_1 is legally topologically ordered after n_2 as this does not violate any dependence edges. Figure 4(b) and (c) show two topological orderings where the value of x at n_7 is 30 and 40 respectively.

One question that may arise is: what should be done in cases when two parallel sections have interfering accesses to a variable that are not ordered by an explicit synchronization link? As mentioned earlier, the semantics of parallel sections does not require us to insert a dependence edge in this case. This case represents a *nondeterministic* parallel program, where the program can exhibit different behaviors depending on the relative execution order of the interfering accesses. Note that, since the BPEL specification states that each `assign` activity is an atomic activity akin to a *synchronized* block in Java, this case does not represent a data race. If an `assign` activity contains multiple `copy` statements, the BPEL specification dictates that the entire set of assignments will be performed atomically.

3. CODE PARTITIONING

In general, there is a bounded but exponential number of ways to distribute the portable code amongst the partitions. Hence, an exhaustive search algorithm that tries every possible placement of portable nodes is intractable. In this section we first describe a simple heuristic called the *merge-by-def-use* heuristic. The aim of the *merge-by-def-use* partitioning algorithm is to determine the best partitions

at which each portable task must be executed in order to optimize the throughput of the decentralized program.

As mentioned in the introduction, our code partitioning algorithm is based on the idea of merging tasks along loop-independent flow dependence edges. This idea is appealing not only for its performance implications, but also because it serves as a heuristic to prune the space of possible solutions. However, merging along flow dependence edges must take into account several considerations including: (1) not all combinations of merges will be legal (a legal combination must not create a dependence cycle among partitions), and (2) a flow dependence edge between two nodes with different control dependences cannot be merged (without introducing guards/predicates).

Starting at the bottom of the control dependence 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, provided the following conditions hold - (1) the reordering along the flow dependence edge does not violate any other dependences, and (2) each partition has at most one fixed node. Using a cost function, we exhaustively evaluate all possible merges along flow dependence edges and compute a local minimum for each region. Once a region has been evaluated, the algorithm is recursively applied to the parent node and its siblings.

Merging Portable Code. An informal description of the merging algorithm is as follows:

1. Locate a control node, T_c in the PDG whose child nodes are all leaf nodes. For all nodes that have the same control dependence condition on T_c repeat steps 2 through 8. Continue till all control nodes have been processed.

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 tasks of the edge. The resultant dependences of the merged task is the union of the component tasks.

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.

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. To determine whether the merge may violate a dependence condition, we check if the merge can introduce a dependence cycle.

5. Exhaustively consider all merging configurations of siblings that can be generated by merging some subset of the flow dependence edges in E. Since the size of E for a single region is usually small, this exhaustive search is usually feasible in practice. (Later we describe a more complex heuristic that further reduces the number of partitionings evaluated.)

6. Choose the merging configuration from step 5 that is likely to yield the best overall throughput value, using the cost model discussed in Section 4. Though in Section 4, the cost function is defined for a complete partition, it can be adapted for an intermediate partition in which some portable tasks have yet to be merged with fixed tasks. This is a greedy heuristic that gives a locally optimal solution, but does not necessarily guarantee global optimality.

7. Any remaining portable tasks that are not merged with a fixed task are merged with the parent. At this point, the parent has only fixed tasks (if any) for children. The parent node is now marked as a leaf node.

8. 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 dependences of the merge is a union of all dependences in the child nodes as well as the parent node.

Example. Figure 5(a) gives the CFG for an example Loan-Approval BPEL service, where the client sends in his profile and the required loan amount. If the amount is less than $10,000, the web service sets a `risk` factor to zero, else it invokes web service `F1` to get a risk assessment. Then the web service sends the `risk` and `amount` to two banks `F2` and `F3` which return the `rate` of interest. Additionally `F2` returns a document with details about the loan scheme. Finally the web service returns the lower rate and the loan scheme information (which may be null) to the client. The PDG for the example is shown in Figure 5(b). All fixed tasks are labeled `F`i and portable tasks are labeled `p`i. We follow the convention that a partition is referred by the label of the fixed task it contains (if it contains a fixed task) or by the label of the portable task with the lowest index.

We give here a partial walk-through of the algorithm. The merging algorithm first merges nodes at the bottom of the control dependence tree.

- For the nodes control dependent on `p1`: `F1` and `p3` have the same control dependence condition. The algorithm starts with `p3`, finds that there is a dependence edge `F1` → `p3` and merges `F1` and `p3`. No other merges are possible. `p2` has no siblings and so it gets merged with the parent node `p1`.

- For nodes control dependent on `p5`: `p6` and `p7` have no siblings and hence get merged with `p5`.

The resultant intermediate partitioning is shown in Figure 5(c). The set of edges in and out of `p1` is the union of edges in and out of `p1`, `p2`, `F1` and `p3`, and similarly for `p5`.

Next we consider the nodes that are control dependent on `Entry`.

- Let the algorithm start with the portable task `p5` which is source or destination in the def-use edges `F2` → `p5`, `F3` → `p5`, and `p5` → `F4`. Let us merge `p5` with `F3` as shown in Figure 5(d1).

- Then the algorithm considers `p4` which has the option of merging with one of `F2`, `p1` or `F0`. Let us merge `p4` with `F2` as shown in Figure 5(d2).

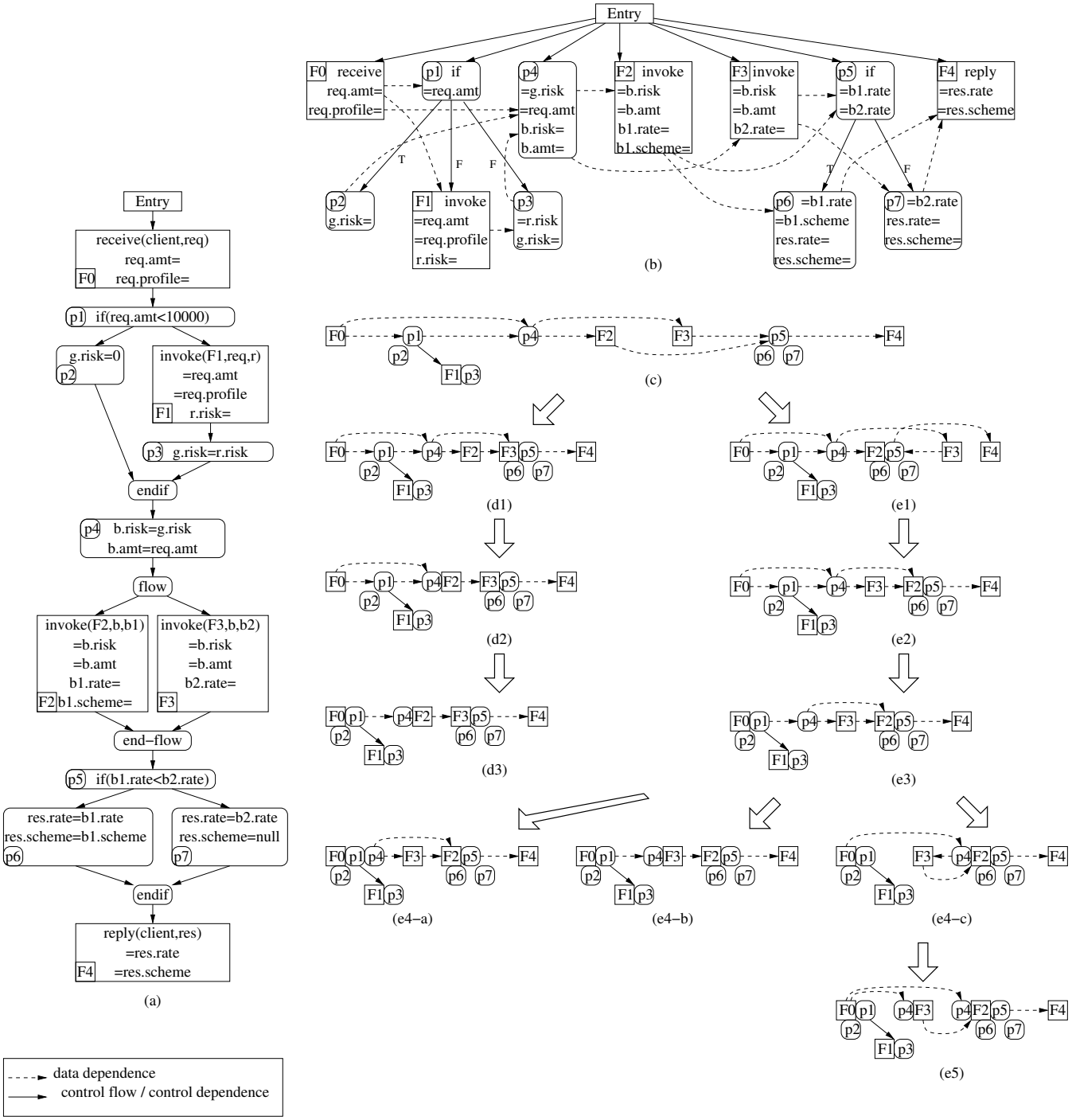


Figure 5: LoanApproval Example - Merging Portable Tasks

- Finally the algorithm merges `p1` with `F0` as shown in Figure 5(d3).

Now there are no more portable nodes left and so this is one of the final partitionings. (We happened to use only lexical siblings in this partitioning.)

As part of enumerating all the merging configurations outlined in step 5, the algorithm backtracks from Figure 5(d1). We do not show all the steps performed by this recursive algorithm but show only some interesting combinations generated.

- Let the algorithm merge `p5` with `F2` (instead of `F3`) as shown in Figure 5(e1).
- Even though this partitioning has an edge `F3` → `F2` which points right-to-left, it is legal because it has not introduced a dependence cycle. To make the acyclic structure clearer, we reorder the partitions topologically, resulting in Figure 5(e2). This partition has the interesting property that the edge `F2` → `p5` is internalized within a partition. Thus the assignment of `res.scheme` in task `p6` happens (if it does) at the same node as `F2`, avoiding a large message. Note that this partitioning could not have been created by lexical merges alone. By contrast, in the partitioning shown in Figure 5(d3), `F2` → `p5` is an inter-partition edge.
- Next let the algorithm merge `p1` with `F0` resulting in Figure 5(e3). To complete the partitioning, `p4` can be merged with one of `F0`, `F3` or `F2` as shown in Figure 5(e4-a), (e4-b) and (e4-c) respectively. Of these, (e4-a) and (e4-b) are valid partitionings, but (e4-c) is not valid as it has generated a dependence cycle. Hence, the partitioning of Figure 5(e4-c) will be discarded as an infeasible PDG.

The cycle detected in the example in Figure 5(e4-c) can be broken by the judicious use of code replication. In the example the cycle arises because `p4` needs to be executed before either `F2` or `F3` executes. Therefore `p4` is replicated at both `F2` and `F3`, giving the partitioning in Figure 5(e5). This code may be replicated if it is a pure computation and hence has no side effects. Note, however, that code replication may not improve performance.

Complexity and Heuristics. An exhaustive search algorithm that tries every possible placement of portable nodes would have a complexity of $O(f^p)$, where p is the maximum number of portable nodes that are siblings in the PDG and have the same control dependence condition; and f is the corresponding number of fixed nodes. The *merge-by-def-use* algorithm described in this section applies a heuristic that attempts to reduce this search space while trying to reduce the data on the network. The complexity of the *merge-by-def-use* algorithm is $O(e^p)$, where e is the maximum number of def-use edges that enter or exit a portable node, and p is the maximum number of portable nodes that are siblings in the PDG and have the same control dependence conditions. For example, in the program in Figure 2, there are 6 portable nodes, each has two def-use edges and so the number of possible partitionings are 2^6 which is 64. The program in Figure 5(c) has portable nodes `p1` with 2 edges, `p4` with 4 edges, and `p5` with 3 edges. Hence the number of possible partitionings is $2 * 4 * 3$ which is 24. Many BPEL programs

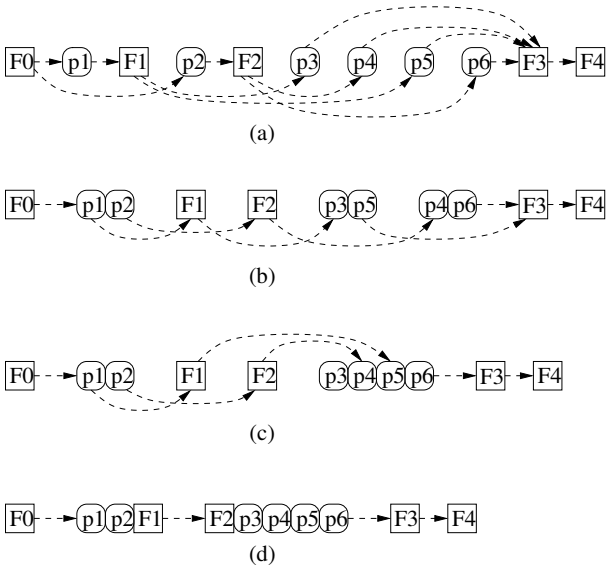


Figure 6: Applying the *pooling* heuristic to the example of Figure 2 - (a) The PDG of Figure 2; (b), (c) Two possible *pooling*s; (d) A partitioning generated from (c) by merging the pool (`p1`, `p2`) with `F1` and the pool (`p3`, `p4`, `p5`, `p6`) with `F2`.

are very small and hence they can be analyzed exhaustively. However, we did sample some BPEL programs that could not be analyzed in a reasonable amount of time without applying heuristics. We apply two more heuristics - (1) the *greedy-merge* heuristic is a refinement of the *merge-by-def-use* heuristic that further tries to minimize the data on the network and (2) the *pooling* heuristic tries to minimize the total number of messages.

In the *greedy-merge* heuristic we examine every portable node, p_i, that has exactly one incoming def-use edge from p_d and one outgoing def-use edge to p_u. Then p_i is merged with p_d if the volume of data between p_i and p_d is less than the volume of data between p_i and p_u, else it is merged with p_u. This halves the options for p_i. Applying this heuristic to the example in Figure 2, reduces the number of partitionings to exactly one and the partitioning generated is the one shown in Figure 2(i) which we will show experimentally to be the best. Applying this heuristic to the example in Figure 5, reduces the number of partitionings to 12.

The *pooling* heuristic is as follows: if two or more portable nodes have the same def-use source or if two or more portable nodes have the same def-use destination, then "pool" them together first and treat them as a single portable node with the combined dependencies of the merge. Then apply the *merge-by-def-use* algorithm as before along with the *greedy-merge* heuristic. The pooling is, of course, subject to the condition of correctness. This heuristic tends to combine all data before entering or exiting a node and hence minimizes the inter-component messages. Consider the example of Figure 2. The PDG is redrawn in Figure 6(a). The possible first level poolings are shown in Figure 6(b) and 6(c). In Figure 6(b), `p1` and `p2` are pooled as they have a common source `F0`; `p3` and `p5` are pooled as they have a common source `F1`; and `p4` and `p6` are pooled as they have a common source `F2`. In Figure 6(c), `p1` and `p2` are pooled

as they have a common source `F0`; and `p3`, `p4`, `p5` and `p6` are pooled as they have a common destination `F3`. These are the only possible "poolings" of portables for this example. The number of initial portable nodes has reduced from 6 in Figure 6(a) to 3 in Figure 6(b) and 2 in Figure 6(c), thus reducing the overall complexity of the algorithm. The possible number of merges is $3 * 1 * 1$ for Figure 6(b) and $3 * 3$ for Figure 6(c), giving $3 + 9 = 12$ partitionings, and a total of 13 partitionings including the partitioning generated by the *greedy-merge* heuristic. This is a vast reduction from the original 64 partitionings. Note the partitioning in Figure 6(d) which is derived from the initial pooling in Figure 6(c) and which minimizes the number of communication edges at the expense of sending extra data. In this case `F0` sends both the names to `F1`. `F1` uses the first name to get the first address and sends the first address along with the second name to `F2`. `F2` obtains the second address and sends both the addresses to `F3`. Thus the input for `F2` is pipelined through `F1` and the output of `F1` is pipelined through `F2`. At low message rates this is quite an efficient method of communication due to the lack of synchronization overheads.

Note, that the set of partitionings generated by the *greedy-merge* heuristic is a subset of the partitionings generated by the *merge-by-def-use* heuristic, but the partitionings generated by the *pooling* heuristic are different from those generated by the *merge-by-def-use* heuristic. Neither is a subset of the other and they may have a non-null intersection. In general, the number of partitionings generated by the *pooling* heuristic is fewer than the number of partitionings generated by the *merge-by-def-use* heuristic.

4. COST MODEL

Much of the past work on partitioning and scheduling has focused on minimizing the *completion time* of a single instance of a program running in isolation. Application servers instead use multiple threads to overlap the execution of multiple instances of one or more programs, and are usually more concerned with optimizing the *throughput* performance that can be delivered for a given a *hardware capacity*. In contrast, completion time is only of interest as a quality of service threshold, and can be ignored as an optimization metric in cases when the request rates on server nodes can be satisfied by available capacities. Therefore, the cost model developed in our work is focused on throughput as its objective function.

As in standard queueing system models, if R is the number of requests sent to a service per unit time, the overall throughput of the service can be modeled as a function $T(R)$ representing the average number of requests processed per unit time. Typically, $T(R)$ ramps up with increases in R until a steady-state plateau is reached when one or more resources is fully utilized; eventually a "breakdown" phase is reached when a backlog accumulates and the system throughput may decline dramatically. Real systems generally use some form of admission control to avoid the breakdown situation.

Our system model for decentralized execution is a system consisting of a set of communicating server nodes, $S = \{S_1, \ldots, S_k\}$, each of which implements a portion of the overall service as dictated by the task partition. The throughput delivered by each individual server node contributes to an upper bound on the overall throughput as follows,

$$T(S) \leq \min(T(S_1), \ldots, T(S_k))$$

For convenience, we use the same notation, S_i to refer to both a server node, and the request rate that it receives. It is therefore important to *balance* the throughput across the nodes, because the overall throughput will be bounded by that of the slowest node. (This is akin to the importance of balancing stages in a pipelined system.) Consequently, we need a model to figure the rate at which each participating server can process (its portion of) client requests. This rate depends on the "capacity" of the server as well as the amount of work it is required to carry out per client request.

Notice that in the cost model, we explicitly consider the fact that multiple instances of the same application, by way of concurrent requests, are running on the same server. We factor in the presence of other independent processes by assuming a reduction in available "capacity" – which reflects the fraction of a server dedicated to (all instances of) a given application. In practice, application servers partition resources statically among independent applications.

Participating servers are assigned work in the following manner. The `receive` and `reply` fixed nodes, generally F_0 and $F_{\max}$ are mapped to the central server. The `invoke` fixed nodes are mapped to their corresponding servers, except in the centralized execution in which all the nodes, fixed or portable, are located on the central server. A portable node is mapped to the same server as the fixed node in its partition. Figure 7 gives work assignment to servers as well as the data dependencies corresponding to three different partitioned configurations of the loan approval process (Figure 5): Figure 7(a) shows the centralized partition configuration and Figures 7(b) and 7(c) show configurations for two decentralizations, corresponding to Figure 5(e4-a) and Figure 5(e4-b) respectively. The straight arrows depict messages across servers while the squiggly arrows show wait times within the thread that is working on (that node's portion of) a particular client request. A client request arrives at F_0 on the central server (node C0 for the centralized orchestration in Figure 7(a) and node D0 or D0′ for the decentralized orchestrations of Figure 7(b) and 7(c)) and works its way through the servers following inter-partition edges. It may fork activity along two edges concurrently (shown as α and β in the figure). The request terminates at $F_{\max}$ on the central server, which responds to the client.

In computing the load per request on a server, we do not need to account for thread waiting time, because it will not have an impact on throughput — the server would instead switch to processing its part of another client request, for which the assigned thread is ready to run. Therefore the load is simply the aggregation of all activity at each server. With this reasoning, we can abstract out the wait times from the pictures in Figure 7, considering each node simply as a collection of fixed and portable tasks. Only the aggregate compute and messaging costs at each server are relevant for the estimation of throughput. Since our cost function is an *upper bound* on throughput, we assume both sides of a conditional are executed when estimating the cost of conditionals. Further refinement of the cost functions to use execution profile information is a subject for future work.

The cost of computation per client request required at a server includes the cost of running all the fixed and portable tasks stationed at the server. (Note that the service itself

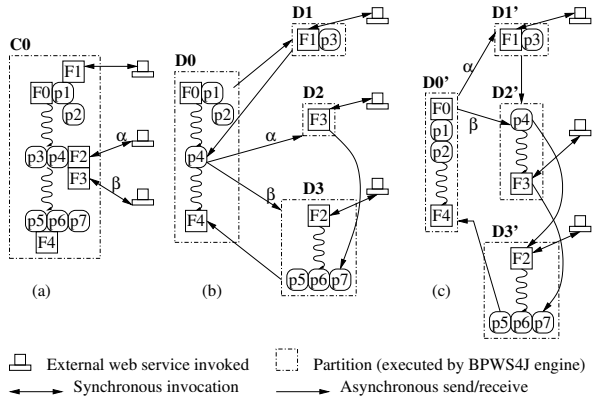


Figure 7: Partition configurations of the Loan-Approval Example - (a) The centralized orchestration; (b), (c) Decentralized orchestrations corresponding to Figure 5(e4-a) and Figure 5(e4-b) respectively.

accessed by an `invoke` may not be adding to the cost in case it is implemented by another "backend" server.) For each portable task, P_i, we assign a cost c_{P_i}. The cost of the fixed task `receive`, c_R, includes in addition to the cost of data handling, the cost of setting up a new process if it is the first `receive` or the cost of correlation for every subsequent `receive`. The cost of a `reply`, c_L, includes only data handling cost for sending over an existing channel. A synchronous `invoke` fixed task needs communication with a backend server, as shown by double-headed arrows in Figure 7. The cost of an `invoke`, c_I, involves message setup in addition to marshalling and unmarshalling data sent and received.

At places where flow dependence edges have their source in one partition and destination in another, decentralization introduces asynchronous communication, which is shown by single-headed arrows in Figure 7. This communication is handled by introducing a `send` at the source partition and a `receive` at the destination partition. These tasks have no direct correspondence with the original code, but are *generated* to support decentralization. Thus at the source of an inter-partition edge we need to add the cost of a `send` and at the destination of an inter-partition edge we need to add the cost of a `receive`. The cost of a `send`, c_S includes only the cost of message setup plus marshalling data, and the cost of a `receive`, c_R, includes the cost of unmarshalling data plus correlation overheads. The cost of `send` or `receive` is typically less than that of an `invoke`.

The cost of each task depends on various factors including the size of data being handled and the complexity of data being manipulated. For example, the cost of an `assign` is independent of the size of the data if it involves moving simple strings, but the cost of an `assign` is extremely sensitive to data size if it involves an XPath expression. The cost of `invoke`s, `send`s and `receive`s are dependent on size of data as well as complexity of data since these tasks involve marshalling and unmarshalling of data. These costs need to be determined empirically using micro-benchmarking, which is explained in greater detail in Section 5.

Example. Given the cost of each task, the peak rate at node C0 in Figure 7(a) is

$$\frac{Capacity_{C0}}{c_R+c_L+c_{P_1}+c_{P_2}+...+c_{P_7}+3*c_I}$$

As another example, consider the cost of node D3 in Figure 7(b). In addition to the obvious cost of fixed node F2 (which is an `invoke`) and the portable tasks p5, p6 and p7, we need to add the cost of two `receive`s (for the two incoming inter-partition edges) and the cost of one `send` (for the outgoing inter-partition edge). Hence the peak rate at D3 is

$$\frac{Capacity_{D3}}{c_{P_5}+c_{P_6}+c_{P_7}+c_I+c_S+2*c_R}$$

Likewise, the peak rates for other nodes in Figure 7(b) and (c) can be computed. The minimum rate across nodes would be the upper bound on the throughput that this configuration would sustain. In this example we can see that the critical node is D0 for (b) and node D0′ for (c). Assuming all portable tasks have the same cost c_p:

$$\text{peak rate(C0)} = \frac{Capacity_{C0}}{c_R+c_L+7*c_p+3*c_I}$$

$$\text{peak rate(D0)} = \frac{Capacity_{D0}}{c_R+c_L+3*c_P+3*c_S+2*c_R}$$

$$\text{peak rate(D0}') = \frac{Capacity_{D0'}}{c_R+c_L+2*c_P+2*c_S+c_R}$$

Assuming C0, D0 and D0′ have the same capacity, the peak rate of the configuration in Figure 7(c) is clearly higher than that in Figure 7(b). However, the relative peak rate of C0 and D0 is not immediately obvious and in Section 5.3 we show how to determine the values of the individual costs (c_R, c_L, etc.) using micro-benchmarking. In this case, the peak rate of Figure 7(b) turns out to be higher than that of Figure 7(a). The experimental results are given in Section 5.1.

Note that although in our example programs, the portable tasks seem to be trivial assignments, the assignment statements actually represent extraction of data from containers in BPEL, because in general messages need to be reconstituted as they flow between fixed nodes. This can, in turn, involve complicated operations such as XPath queries. Thus, they are more expensive than assignment statements in standard programming languages.

Finally, we outline how our cost function can be extended to determine the bounds on throughput that occur when a server is unable to consume the available compute capacity due to other reasons. These cases can occur if the available capacity such as the number of threads in a thread pool on the server node is too low. The cost function discussed thus far estimates an upper bound of the throughput on a single server node S_i as $T(S_i) \leq Capacity/Cost$, where *Capacity* is the raw compute capacity of node S_i and *Cost* is the total amount of work that needs to be performed on node S_i for a single request. The bound on throughput due to limited number of threads in the server can be modeled as

$$T(S_i) \leq Number\ of\ Threads/CritPath$$

where *CritPath* is the thread occupancy time along the critical path. Then we can estimate an upper bound of the

throughput, $T(S_i)$ on a single server node S_i as follows,

$$T(S_i) \leq \min(\frac{Capacity}{Cost}, \frac{Number\ of\ Threads}{CritPath})$$

Similarly, there may exist other bounds based on bandwidth availability or memory usage.

5. EXPERIMENTAL RESULTS

5.1 Runtime Performance

Experimental Setup. Our experimental setup for testing decentralized orchestration is as follows. We use a cluster of Intel Pentium based Linux machines (2.2 GHz, 2 GB RAM) connected by a 100Mb/s LAN. For the results reported in this paper, two of the machines were used as clients, and up to four machines were used as servers.

The clients execute multithreaded Java programs that run a total of 10 to 200 threads generating a steady request rate of 1 request/second (or 60 requests/minute) to 20 requests/second (1200 requests/minute). The test message sizes varied from 256 bytes to 24 KB.

In the centralized setup (as shown in Figure 1(a)), the clients send requests to an HTTP server that hosts a BPWS4J engine. The requests are synchronous and sent using SOAP over HTTP. Each component web service is deployed on an HTTP server running on a separate machine. The BPWS4J server internally invokes the component web services using SOAP over HTTP. While the centralized setup uses only HTTP servers, the decentralized setup uses a combination of HTTP and EJB servers. EJB servers are necessary as HTTP servers cannot handle asynchronous messaging.

In the decentralized setup (as shown in Figure 1(b)), the clients send requests to an EJB server that hosts a BPWS4J engine. The requests are sent using SOAP over JMS. However, the component web services continue to run on HTTP servers, each running on a separate machine. An EJB server hosting a BPWS4J engine is co-located with each of the web services.

Test Examples. 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 our algorithm on all examples that come with the BPWS4J distribution as well as several other applications culled off the web. Although there are many possible partitionings for each example, for visual clarity we show only a few illustrative partitionings for each example.

We first present performance results for the LoanApproval example introduced in Section 3 and the FindRoute example introduced in Section 1. Then we show results for two more examples. The TranslateArticle service locates an article of interest, gets it translated into the required language, formats both the original and translated article and returns it to the client. The NearestRestaurant service receives a restaurant preference from a client along with his mobile cell number and a radius parameter. The service locates a restaurant of the specified cuisine, gets the location of the client using a GPS service, then invokes a map service that takes the client location, the list of restaurants and shortlists the restaurants that are within the specified radius of the client. This list is returned to the client[2].

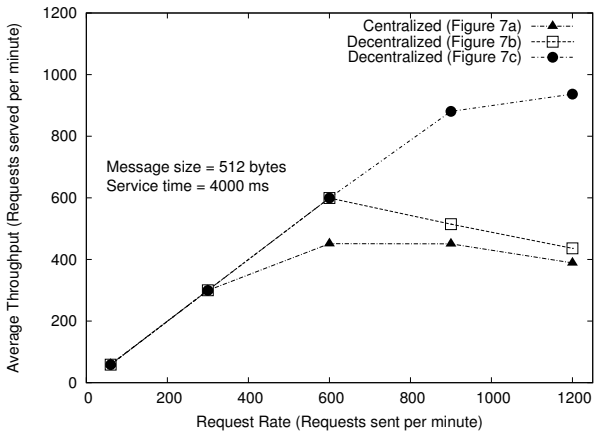


Figure 8: LoanApproval - Throughput Variation with Request Rate.

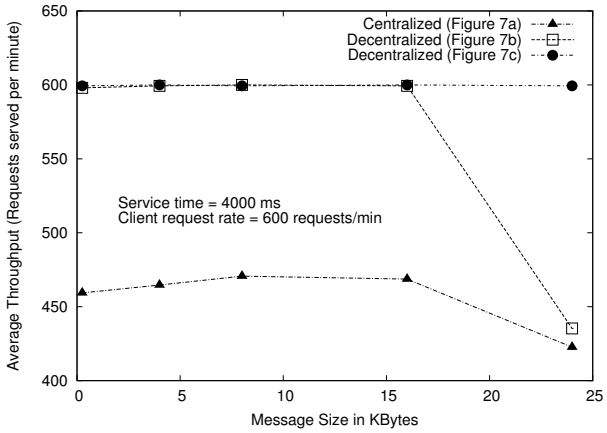


Figure 9: LoanApproval - Throughput Variation with Message Size.

LoanApproval Example. In Figure 8, we show the throughput observed for the three partitionings of the LoanApproval example shown in Figure 7, as a function of the request rate. This example uses three component web services which were accessed using the standard SOAP over HTTP protocol. For Figure 8, the service time for each web service was fixed at 4000 ms, and the message size fixed at 512 bytes. We varied the client request rate from 60 requests/minute to 1200 requests/minute. At lower rates the requests do not exceed the capacity of the system and hence the throughput is equal to the request rate. As mentioned in Section 4, the partitioning in case (a) was the first to reach its capacity limit, followed by case (b), with case (c) delivering the best throughput. In each case, the throughput begins to decline when the request rate exceeds the capacity.

In Figure 9, we show how the throughput for the LoanApproval example varies with message sizes of 512B, 4KB, 8KB, 16KB, and 24KB. To highlight the differences among

[2]All the BPEL examples are available on request from the authors

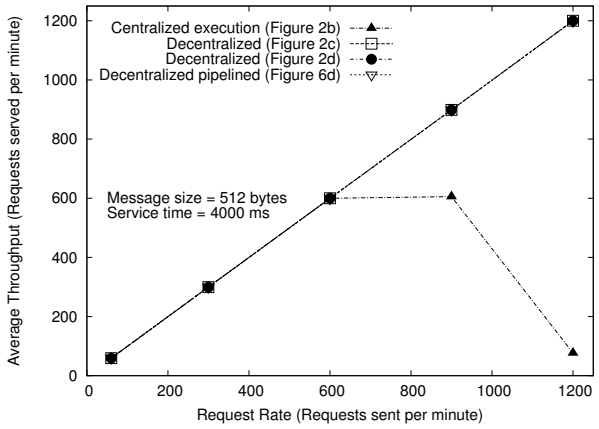


Figure 10: FindRoute - Throughput Variation with Request Rate. The plots for cases 2c, 2d and 6d coincide.

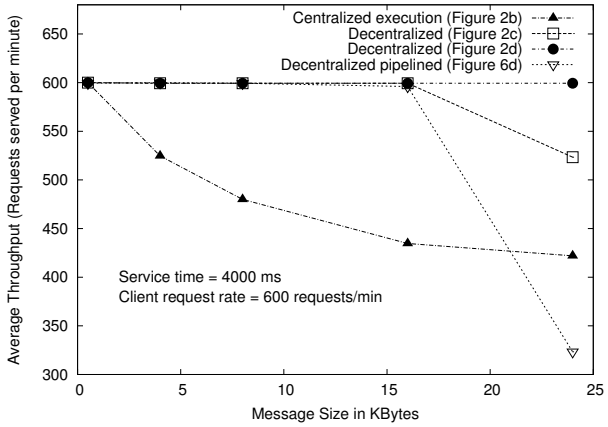


Figure 11: FindRoute - Throughput Variation with Message Size

the three cases, we set the y-axis origin for Figure 9 to 400 requests/minute instead of zero. In this experiment, we fixed the client request rate at 600 requests/minute at which case(a) and case(b) are just within their capacity limits while case(c) is well within its capacity limits (see Figure 8). The service time was fixed at 4000 ms as before. Increasing the message size has a different impact for the three different partitionings. In case (a), C0 has to send/receive six messages per request, but the relative impact of message size is low compared to the load imbalance of a centralized partitioning. In case (b), D0 has to send/receive five messages per request, and increasing the message size from 16KB to 24KB causes the throughput to fall due to a decrease in processing capacity. In case(c), D0′ has to send/receive three messages per request, but the request load did not cause its capacity to fall short within the parameters set by this experiment.

FindRoute Example. For the FindRoute example Figure 10 shows throughput results for variation in request rate and Figure 11 shows throughput results for variation with message size. We show performance results for the three partitionings labeled (b), (c) and (d) in Figure 2 and also for the partitioning in Figure 6(d) where the number of intercomponent messages has been minimized but some data gets pipelined causing excess data to flow in the network. Recall that partitioning 2(b) corresponds to centralized orchestration, whereas 2(c) and 2(d) are two decentralized orchestrations. We stated that 2(d) is likely to give better performance since it communicates less data on the network. Experimentally we observe that when the message size is small, all the decentralized orchestrations (including the pipelined case) perform equally well (Figure 10), but show variations when the message size increases (Figure 11). In our experiments we ensure that the fraction of the address required by the TrainRoute service is very small. Partitioning 2(c) puts the entire data onto the network and hence shows a deterioration in performance as the message size increases. Partitioning 2(d) puts only the required fraction of data on the network and hence is unaffected by increasing size of addresses. The pipelined version (which carries the maximum data) deteriorates the most.

TranslateArticle Example. The BPEL code, the PDG and two partitionings for the TranslateArticle service are given in Figure 12. The partitionings have been selected as case (c) that minimizes data on the network (using the *greedy-merge* heuristic) and case (d) that minimizes the number of hops on the network (using the *pooling* heuristic). Both the decentralized versions perform well (Figure 13) compared to the centralized version. The partitioning generated by the *pooling* heuristic performs marginally better than the partitioning generated by the *greedy-merge* heuristic as the *greedy-merge* version has higher synchronization overheads. Neither orchestration shows deterioration (Figure 14) in performance with increasing message size within the experimented range.

NearestRestaurant Example. The BPEL code, the PDG and two partitionings for the NearestRestaurant service are given in Figure 15. Here also the partitionings have been selected as case (c), a partitioning generated by the *greedy-merge* heuristic and case (d), a partitioning generated by the *pooling* heuristic. Both the decentralized versions perform well (Figure 16) compared to the centralized version. The partitioning in case (c) minimizes data on the network and performs marginally better than the partitioning of case (d) as case (d) has higher data overheads without much benefit of reduced synchronization overheads. Neither orchestration shows deterioration (Figure 17) in performance with increasing message size within the experimented range. However although we do not show the results here, we observed that the response time in case (c) was substantially better than that in case (d) due to the higher parallelism.

5.2 Compile-time Performance

We evaluated three different algorithms - one which does an exhaustive search of the space of all possible merges, one which uses the *merge-by-def-use* heuristic of merging along def-use edges and one which combines the *greedy-merge* and *pooling* heuristics described in Section 3. For the three algorithms we report in Table 2, the total number of configurations that each algorithm explores and the number of configurations that are valid. In Table 3 we report the time taken to run the algorithms. In these tables, we include one

F0 receive(client,req{lang,item})
p1 f2.lang=req.lang
p2 f1.item = req.item
F1 invoke(F1, f1{item}, f1Res{article})
p3 f2.article=f1Res.article
F2 invoke(F2, f2{lang,article}, f2Res{translated})
p4 f3.translated=f2Res.translated
p5 f3.article=f1Res.article
F3 invoke(F3, f3{article,translated}, f3Res{output})
p6 res.output = f3Res.output
F4 reply(client,res{output})

(a)

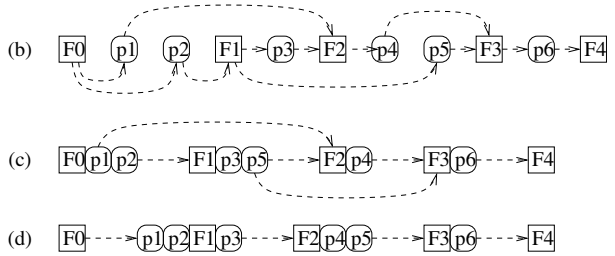


Figure 12: TranslateArticle - (a) The CFG, (b) The PDG, (c) and (d) Two topologies

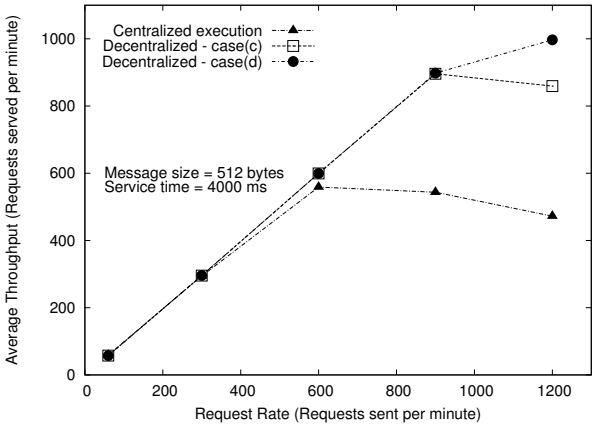


Figure 13: Throughput Variation with Request Rate

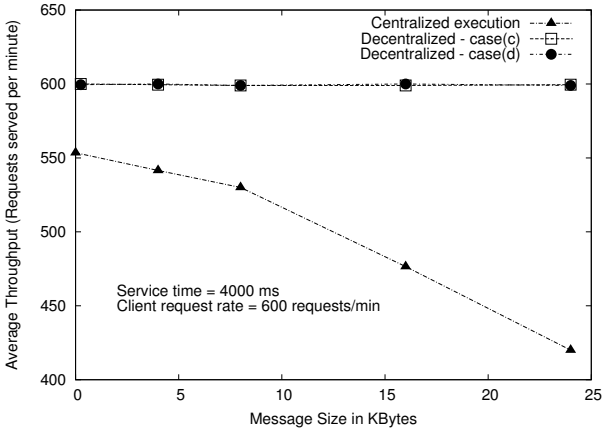


Figure 14: Throughput Variation with Message Size

F0 receive(client,req{cuisine,id,radius})
p1 f1.cuisine=req.cuisine
p2 f2.id = req.id
flow
F1 invoke(F2, f2{id}, f2Res{location})
F2 invoke(F1, f1{cuisine}, f1Res{list})
end−flow
p3 f3.radius=req.radius
p4 f3.list=f1Res.list
p5 f3.location=f2Res.location
F3 invoke(F3, f3{list,budget,location}, f3Res{list})
p6 res.list = f3Res.list
F4 reply(client,res{list})

(a)

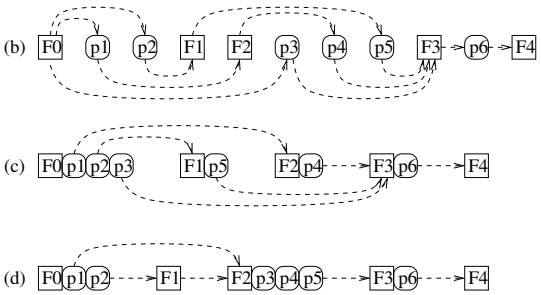


Figure 15: Nearest Restaurant - (a) The CFG, (b) The PDG, (c) and (d) Two topologies

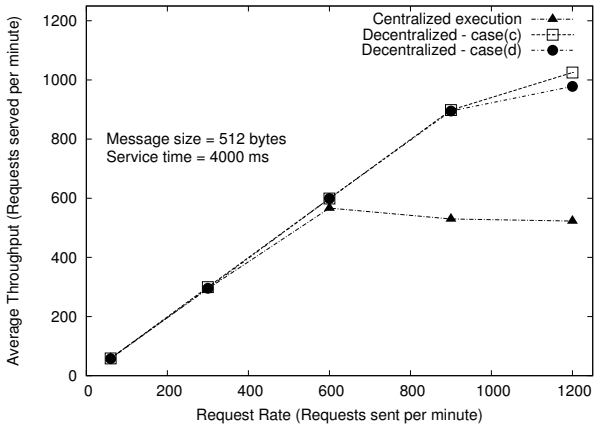


Figure 16: Throughput Variation with Request Rate

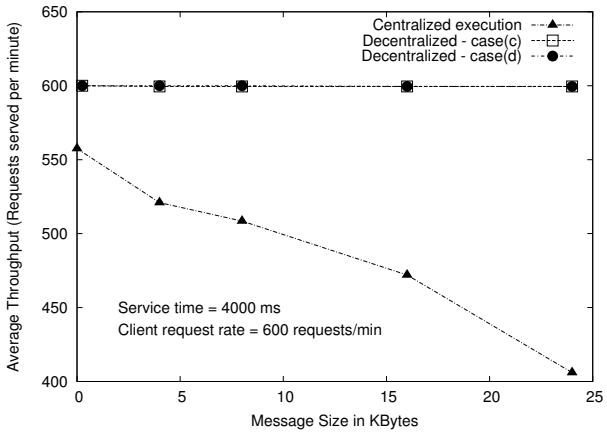


Figure 17: Throughput Variation with Message Size

Table 2: Number of Configurations Generated. "Total" is the total number of configurations explored, "Valid" is the number of valid configurations found. (LA=LoanApproval, FR=FindRoute, TA=TranslateArticle, NR=NearestRestaurant, BA=BioAnnotator)

Test Case	Exhaustive search		Merge by def-use		Greedy-merge + Pooling	
	Total	Valid	Total	Valid	Total	Valid
LA	64	11	24	11	12	11
FR	78125	736	64	64	13	10
TA	15625	144	64	64	27	12
NR	15625	448	64	64	24	14
BA	-	-	65536	65536	1	1

Table 3: Compute Time in milliseconds. (LA=LoanApproval, FR=FindRoute, TA=TranslateArticle, NR=NearestRestaurant, BA=BioAnnotator)

Example	Exhaustive search	Merge by def-use	Greedy-merge + Pooling
LA	3039	2983	2975
FR	6576	2265	1959
TA	4080	2028	1932
NR	4185	2050	1954
BA	-	71880	3554

extra example, the BioAnnotator composite service. BioAnnotator creates a chain of web services, in which each web service adds some set of annotations to an input file and passes the annotated file to the next annotator web service. This is a fairly large program but with a completely linear structure.

An exhaustive search is clearly very expensive compared to the heuristics presented in this paper. For the BioAnnotator example, we were unable to run the exhaustive search algorithm. The *merge-by-def-use* heuristic generated more than 65000 partitionings, but the *greedy-merge* and *pooling* heuristics together generated exactly one partitioning which in fact has the best performance characteristics. The *merge-by-def-use* heuristic generates many partitionings that are similar and hence generates a much larger number of partitionings. The time to compute the exhaustive search algorithm is also very large compared to the heuristics. For smaller programs, the time to run the *greedy-merge* and *pooling* heuristic algorithms is comparable with the time to run the *merge-by-def-use* heuristic, but as the program size increases as in the case of BioAnnotator, the time to compute the *merge-by-def-use* heuristic is much larger.

5.3 Micro-benchmarking

Micro-benchmarking is required to compute the cost of primitive activities used in BPEL programs. As mentioned in Section 4 these costs need to be determined empirically for different data sizes as well as for different operations on data. For brevity, we show the cost variance with data size only for the following four activities: `receive`, `reply`, `assign` and `invoke`. To compute the cost of these activities

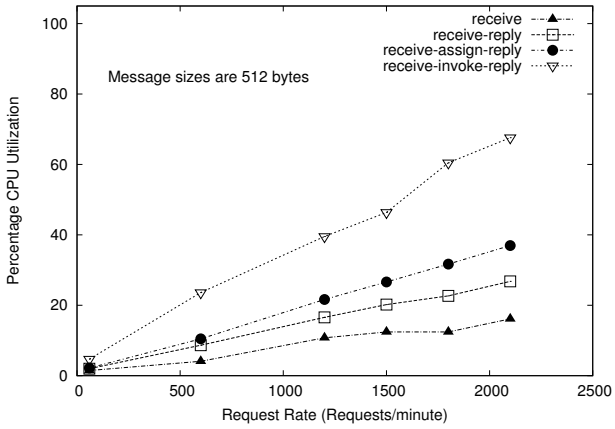


Figure 18: Micro-benchmarking: CPU Utilization for each micro-benchmark program at varying request rates for messages of size 512 Bytes.

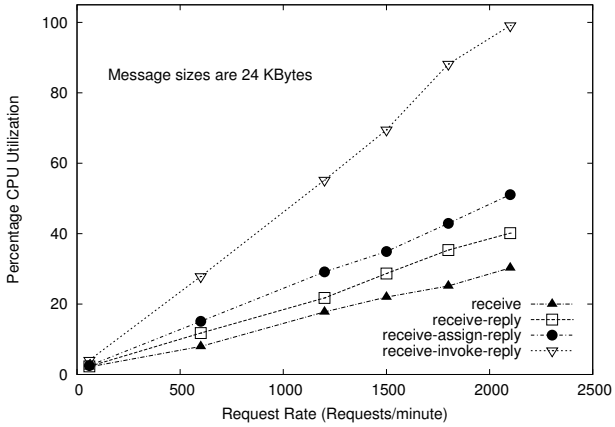


Figure 19: Micro-benchmarking: CPU Utilization for each micro-benchmark program at varying request rates for messages of size 24KB.

we ran four micro-benchmark programs - (1) the `receive` BPEL program which contains exactly one `receive` activity and nothing else; (2) the `receive-reply` BPEL program which receives a string and echoes it back; (3) the `receive-assign-reply` BPEL program which receives a string in one variable, copies it to another variable and sends it back; and (4) the `receive-invoke-reply` BPEL program which receives a string in one variable, invokes a web service using the same data and returns the response to the client. The invoked web service is an "echo" service. Figure 18 gives the percentage CPU utilization for each program as a function of request rate when the data sizes are all 512 bytes and Figure 19 gives the percentage CPU utilization for each program as a function of request rate when the data sizes are 24 KB. Since CPU utilization varies linearly with request rate we can compute the cost of an activity from the slope of the plot. For the `receive-assign-reply` benchmark for 512 byte messages, from the slope of the corresponding CPU utilization plot we find that the cost is 0.019 work units – computed as CPU utilization (40%) divided by request rate (2100). Similarly, the cost of the

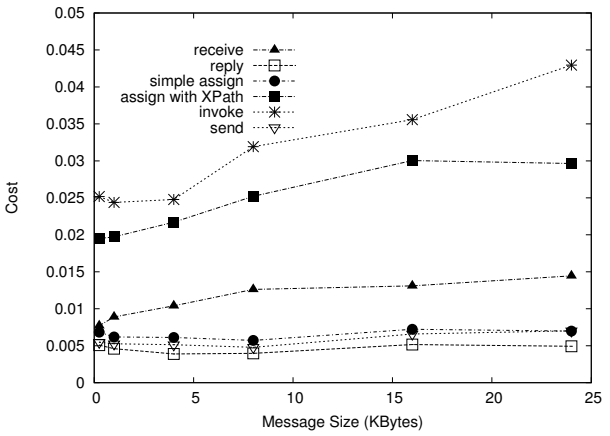


Figure 20: Micro-benchmarking: Costs of different activities at varying message sizes.

`receive-reply` BPEL program is 0.012 units. Hence the cost of an `assign` is $0.019 - 0.012 = 0.007$ units. Similarly, the cost of each activity can be benchmarked and computed. The cost of an `assign` is 0.007 at message sizes of 24KB showing that simple `assign`s are not affected by message size. When the `assign` contains an XPath expression, the cost is significantly higher and also sensitive to message size. The cost of `invoke` however is 0.025 for 512 bytes and 0.045 for 24KB messages, indicating that `invoke`s are sensitive to message sizes. The costs have been plotted for message sizes of 256 bytes, 1KB, 4KB, 8KB, 16KB and 24KB in Figure 20.

Once the micro-benchmark costs have been computed, they can be used to predict the throughput of a given partition as follows: The available capacity (*e.g.*, $Capacity_{C0}$ in the example in Section 4) is set to 100 units if 100% CPU is alloted to the process else appropriately to a smaller value. Then our cost function computes the request rate at which a partitioning will deliver peak throughput and beyond which throughput is expected to deteriorate.

Example. If we assume a 90% capacity availability, then applying the cost values in Figure 20 to the example in Section 4, we get the peak rates in Requests/minute as:

$$\text{peak rate(C0)} = \frac{90}{.007+.005+7*.20+3*.25} = 396$$

$$\text{peak rate(D0)} = \frac{90}{.007+.005+3*.20+3*.006+2*.007} = 865$$

$$\text{peak rate(D0}') = \frac{90}{.007+0.005+2*.20+2*.006+.007} = 1267$$

While these values as absolute values are not important, they are *indicative* of the relative performance characteristics of different partitions.

Discussion and Limitations. We have benchmarked several Linux machines with different CPU, memory and kernel configurations. The trends in all the machines is the same though the actual values differ. However, machines with the same configuration give the same results. Thus it is important to benchmark only one of each type of machine in the system. We have not benchmarked machines running different operating systems.

The cost function works on the assumption that the size of the data handled by each activity is known a priori. A WSDL (Web Services Description Language [9]) document is a signature of a web service, providing input/output details of the web service. Data size information can sometimes be estimated by the WSDL descriptors of the BPEL programs and the WSDL descriptors of the invoked web services. However, in general, this information needs to be gathered by profiling.

Our micro-benchmarking computes an upper bound based on CPU and memory utilization. However, as mentioned in Section 4, there may be other upper bounds: the number of available threads in the server or network bandwidth. We have not benchmarked these parameter. For this paper, we assume that plentiful threads are available, although real application servers obviously limit the number of threads that can be configured. We also assume that network bandwidth is not a bottleneck which is not unrealistic for LAN scenarios.

6. RELATED WORK

Much work has been done on automatic parallelization of sequential programs based on PDGs e.g., [2, 6]. In contrast, the focus in this paper is on the use of PDGs in partitioning of composite web service applications for decentralized orchestration. There are many references in the literature that are relevant to partitioning and clustering algorithms for parallel programs e.g., [3, 15, 16, 7, 23]. Though we 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, decentralization of composite web services presents a partitioning problem with the additional constraint that tasks can be either fixed or portable. In addition, most previous work on partitioning focused on minimizing the completion time of a single instance of the program or for parallel tasks. The works by Graham [8] and Reiter [14] show how to determine bounds on execution times and throughput in acyclic and cyclic dependence graphs respectively for parallel computation. The goal of this work is to maximize the throughput for the case when multiple instances of the parallel program (composite web service) are executed.

Our work uses and expands on techniques for merging PDGs [10]. Singhai [19] uses a similar idea in the area of loop fusion, where two loops are merged subject to the condition that the merge does not violate dependences. His algorithm also uses the notion of maximizing dependence edges within a fused loop to increase reuse of cached variables. However neither of them consider the dependence constraints that must be applied when merging two explicitly parallel sections of a program.

Work by Subhlok *et al* [20, 21] merges adjacent tasks in a task graph and solve the problem of assignment of tasks to a set of processors. Their problem is different in that they need to determine the optimal number of processors for an application based on certain constraints and cost models. In our case, the number of processors is pre-determined, some tasks (fixed tasks) are preallocated to specific processors and the remaining tasks need to be assigned. Their applications are typically pipelines of tasks where the output of one task feeds into the input of the next and hence it suffices to merge

only adjacent tasks. Our applications have much more complex communication patterns and hence benefit from task reordering.

Partitioning problems in distributed computing have been tackled in many ways. Singh and Pande [18] give a solution to code migration based on mobile agents. A mobile agent represents a single flow of control (albeit decentralized) that determines what code is executed at what location. Our decentralized solution may have many parallel threads of execution that interact and synchronize and hence the problems we solve are different. Zhou *et al* [24] give a static analysis solution for method partitioning. However, their solution lies in partitioning the message handling code between the sender and receiver. While we do something similar, we also generate and evaluate different partition configurations, whereas they do not attempt to change the topology of a given decentralized application.

Tilevich and Smaragdakis [22] give a related partitioning algorithm that uses the notion of "anchored" and "mobile" tasks. Certain classes are anchored to fixed locations while others may be allowed to migrate. Their algorithm generates proxies to access the migrated classes and determines an efficient partitioning of the code. However, the final orchestration is centralized over RMI calls unlike our orchestration which is decentralized with asynchronous messaging.

Finally, there exist other techniques for partitioning workflows such as state and activity chart-based techniques to enable distributed execution according to the original semantics [13]. These techniques do not alter the invocation order of activities, and also do not consider load balancing issues in mapping activities to workflow servers. Our work does not have these limitations because we model decentralized execution as a general partitioning of a program dependence graph.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have given a new code partitioning algorithm that is applicable to decentralization of composite web services. The algorithm depends on a technique for testing for legality of reordering of PDG nodes, and on a technique to estimate the throughput of a network of servers executing a business process. Our experimental results show that decentralization can increase the throughput of example composite services substantially, easily doubling it under high system load.

In the near future we plan to build a feedback control mechanism that will determine the correct runtime values to the parameters in the cost function. Depending on the feedback we will enable switching between different partition configurations based on runtime conditions.

From an algorithmic standpoint, we plan to enhance our algorithm in the future to consider merges of fixed nodes *i.e.*, when multiple fixed nodes are placed in the same partition.

8. ACKNOWLEDGMENTS

We would like to thank Girish Chafle, Sunil Chandra and Vijay Mann of IBM India Research Laboratory for their their suggestions, ideas, and for their enthusiastic support in implementing the tool. We would also like to thank the BPWS4J team at IBM T. J. Watson Research Center for their support in using the BPWS4J engine.

9. REFERENCES

[1] WebSphere Application Server. http://www-3.ibm.com/software/info1/websphere/index.jsp?tab=products/appserv.

[2] 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.

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

[4] Business Process Execution Language for Web Services Version 1.1. http://www.ibm.com/developerworks/library/ws-bpel/.

[5] BPWS4J: Java Run Time for BPEL4WS. http://www.alphaworks.ibm.com/tech/bpws4j.

[6] 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.

[7] 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.

[8] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[9] 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.

[10] S. Horwitz, J. Prins, and T. Reps. Integrating Non-Interfering Versions of Programs. *Conf. Rec. Fifteenth ACM Symposium on Principles of Programming Languages*, pages 133–145, January 1988.

[11] R. Khalaf, N. Mukhi, and S. Weerawarana. Service-Oriented Composition in BPEL4WS. In *Proceedings of the Twelfth International World Wide Web Conference*, 2003.

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

[13] P. Muth, D. Wodtke, J. Weissenfels, D. A. Kotz, and G. Weikum. From centralized workflow specification to distributed workflow execution. *Journal of Intelligent Information Systems (JIIS)*, 10(2), 1998.

[14] R. Reiter. Scheduling Parallel Computations. *Journal of the ACM*, 4(15):590–599, 1968.

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

[16] V. Sarkar. Automatic Partitioning of a Program Dependence Graph into Parallel Tasks. *IBM Journal of Research and Development*, 35(5/6), 1991.

[17] V. Sarkar. A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence

Graphs. *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.

[18] A. Singh and S. Pande. Compiler optimizations for Java aglets in distributed data intensive applications. In *Proceedings of the ACM Symposium on Applied Computing*, pages 87–92, 2002.

[19] S. Singhai and K. McKinley. Loop fusion for data locality and parallelism. In *Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems*, 1996.

[20] J. Subhlok, D. O'Hallaron, T. Gross, P. Dinda, and J. Webb. Communication and memory requirements as the basis for mapping task and data parallel programs. In *Proceedings of Supercomputing*, pages 330–339, Washington, DC, November 1994.

[21] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architecture (SPAA)*, Padua, Italy, June 1996.

[22] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of European Conference on Object Oriented Programming (ECOOP'02)*, 2002.

[23] 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.

[24] D. Zhou, S. Pande, and K. Schwan. Method partitioning - runtime customization of pervasive programs without design-time application knowledge. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, 2003.