Semantics of Optimistic Computation

Rick Bubenik
Department of Computer Science
Washington University
St. Louis, Missouri 63130-4899

Willy Zwaenepoel
Department of Computer Science
Rice University
Houston, Texas 77251-1892

Abstract

We address the issue of deriving a semantically equivalent optimistic computation from a pessimistic computation by *application-independent* transformations. Computations are modeled by *program dependence graphs* (pdgs). The semantics of a computation is defined by a mapping from an initial state to a final state, and is realized by a graph rewriting system. Semantics-preserving transformations are applied to the pdgs of the pessimistic computation to produce an optimistic version. The transformations result from guessing data values and control flow decisions in the computation.

We use our transformations to derive an optimistic version of fault tolerance based on message logging and checkpointing. The transformations yield an optimistic version similar to optimistic fault tolerance algorithms reported in the literature, although additional application-dependent transformations are necessary to produce a realistic optimistic implementation.

1 Introduction

Optimistic computations use guesses about their future behavior, and proceed with computation based on those guesses before they can be verified. Output to the external world resulting from computation based on unverified guesses must be kept hidden. If a guess turns out to be incorrect, computation and output based on it are discarded. Otherwise, the output is committed to the outside world. Optimistic computations allow increased parallelism since constraints forcing sequential execution are removed. The performance characteristics of an optimistic computation, compared to an equivalent pessimistic computation, depend on the percentage of correct guesses, the performance gains resulting from correct guesses, and the losses resulting from incorrect guesses. Performance is also affected by the availability of idle resources for optimistic computation and the amount of bookkeeping necessary. Optimistic solutions for several problems have appeared in the literature, including bulk data transfer [4], concurrency control [9], distributed simulation [7], fault tolerance [8, 11], and software maintenance [3].

We explore the extent to which an optimistic computation can be derived by *application-independent* transformations from a pessimistic computation. We model computations by their *program dependence graphs* (pdgs) [6], and define their semantics by means of a graph rewriting system [10]. A number of application-independent, semantics-preserving *transformations* on the pdg transform a pessimistic computation into an optimistic one. These transformations result from guessing data values or control flow decisions before they are known. While the transformations resulting from these guesses are application-independent, where in the computation to make such guesses and what guesses to make in order to obtain performance improvements remain necessarily application-dependent issues.

We illustrate our technique by applying our transformations to fault tolerance based on message logging and checkpointing [1, 8, 11]. Our transformations derive an optimistic algorithm for such fault tolerance that is similar to algorithms presented in the literature [8, 11]. However, in order to achieve a realistic implementation, further application-specific transformations are necessary. We have obtained similar results by applying these transformations to other problems [2].

The rest of this paper is organized as follows. In Section 2, we describe program dependence graphs and their semantics. In Section 3, we present our optimistic transformations. We extend pdgs to accommodate nondeterministic message exchange between deterministic processes in Section 4. In Section 5, we demonstrate our optimistic transformations by applying them to distributed fault tolerance based on message logging and checkpointing. Finally, we draw conclusions in Section 6.

2 Program Dependence Graphs

We specify computations by a variant of the *program dependence graph* (pdg) as defined by Selke [10], with

This work was supported in part by the National Science Foundation under Grants CDA-8619893 and CCR-8716914, and by IBM Corporation under Research Agreement No. 16140046.

extensions to handle inputs and nondeterministic message exchange between deterministic processes. We do not address the problem of mapping other specifications into the pdg model.[1]

The semantics of a program is defined by a mapping from the initial state to the final state, for all possible initial states. The initial state is specified as an input vector, IV, containing $\langle external\_id, value\rangle$ pairs, where *external_id* is an external object identifier and *value* is its associated value. Likewise, the final state produced by pdg evaluation is specified as an output vector, OV, of similar $\langle external\_id, value\rangle$ pairs. This semantics is realized as a system of graph rewriting rules. The rules describe how an *evaluation* of a pdg modifies the graph and produces the output vector.

Pdgs contain nodes, corresponding to operations, and edges that partially order the evaluation of nodes. Internally, the pdg is a dataflow computation, in which the values produced by one node flow along certain edges to other nodes. There is no shared store internal to the pdg. To distinguish multiple values flowing into and out of a single node, values are labeled with an internal identifier (*internal_id*) and nodes reference the particular values consumed and produced by specifying the appropriate internal identifiers. *Input* operations transfer values from the input vector (referenced by *external_ids*) into internally accessible values (referenced by *internal_ids*). *Output* operations transfer internally computed values to the output vector.

2.1 Graph Description

Formally, a pdg is tuple $\langle N, E, IV, OV\rangle$, where N is a set of nodes, E is a set of edges, IV is the input vector, and OV is the output vector. Nodes in the pdg are uniquely labeled and have the form:

$$\begin{aligned} node \ ::= \ & \langle assignment, e, IID\rangle \mid \langle valve, IID\rangle \mid \\ & \langle define, c, IID\rangle \mid \langle input, EID, IID\rangle \mid \\ & \langle output, IID, EID\rangle \mid \langle decision, p\rangle \end{aligned}$$

where e is an *expression* over the domain of *internal_ids*, p is a *predicate* over the domain of *internal_ids*, c is a set of *constants*, IID is a set of *internal_ids*, and EID is a set of *external_ids*.

Edges in the pdg have the form:

$$edge \ ::= \ \langle flow, i, j\rangle \mid \langle control, i, j, b\rangle$$

where i is the head node, j is the tail node, and b is a boolean, either **T** (true) or **F** (false).

[1]The problem of mapping programs to pdgs is discussed by Selke [10], who proves that a pdg preserves the sequential semantics implied by the textual representation of a program.

An *assignment* node computes a value for each *internal_id* $x_i \in IID$ by evaluating the expression e. Evaluation of this expression is deterministic in that it produces the same values if all incoming *internal_ids* are bound to the same values when evaluation begins.

The *valve* node is a distinguished type of *assignment* node, in which each *internal_id* $x_i \in IID$ is assigned its own value. As discussed by Cartwright and Felleisen [5], only one value is allowed to flow into a node for each *internal_id*. This restriction is enforced by inserting *valve* nodes in the branches of a decision construct in which no assignment is made to a particular *internal_id*, if the *internal_id* is assigned to in the other branch.

The *define* node assigns a constant from the set c to each *internal_id* $x_i \in IID$.

Input nodes assign to each *internal_id* $x_i \in IID$ the value read from an external object referenced by a corresponding *external_id* $y_i \in EID$. *Input* nodes do not have any incoming *flow* edges since they do not reference values computed previously in the pdg.

Output nodes modify the output vector OV by transferring the value referenced by each *internal_id* $x_i \in IID$ to a corresponding external object referenced by $y_i \in EID$. *Output* nodes do not have any outgoing edges since no values are passed from *output* nodes to other nodes in the pdg.

Decision nodes have a predicate, along with **T** *control* edges leading to all nodes that are evaluated *only if* the predicate evaluates to **T** and **F** *control* edges leading to all nodes that are evaluated *only if* the predicate evaluates to **F**.

Flow edges order nodes based on data constraints. A *flow* edge is placed from node i to node j if i produces a value labeled x and j consumes this value.

Our pdg model lacks a specific construct for modeling iteration. Loops are represented by the infinite expansion of the loop into nested decision constructs.

When exactly one value for any identifier is allowed to flow into each node, the pdg is *deterministic* in that it always generates the same output vector for a given input vector, assuming each *external_id* is output only once. For now, we assume all pdgs are deterministic. In Section 4, we augment pdgs to accommodate nondeterministic message exchange between deterministic processes.

2.2 Graph Rewriting Rules

During each rewriting step, some *enabled* node i is evaluated, and i and all its outgoing edges are removed from the pdg. A node becomes enabled when all incoming edges have been removed. At any time, all enabled nodes may be rewritten in parallel. Additional nodes and edges may be removed from the graph or modified during a rewriting step, based on the node type, according to the following rules:

• $i = \langle \textit{assignment}, e, IID\rangle$

$$\begin{aligned} N' &= N - \{i\} - \{j \mid \langle i,j\rangle \in E\} \\ &\quad \cup \{j[IID/\mathcal{V}_l[\![e]\!]] \mid \langle i,j\rangle \in E\}^{2} \\ E' &= E - \{\langle i,j\rangle \mid \langle i,j\rangle \in E\} \end{aligned}$$

The notation $j[IID/\mathcal{V}_l[\![e]\!]]$ denotes a substitution in node j, where the identifiers $IID = \{x_1, \ldots, x_n\}$ appearing in j's expression or predicate are replaced with the values computed by the evaluation function $\mathcal{V}_l[\![e]\!]$. The rewriting of an *assignment* node is shown in Figure 1.

• $i = \langle \textit{valve}, IID\rangle$. Analogous to the *assignment* node rewriting step, with $\mathcal{V}_l[\![e]\!]$ replaced by the values of each *internal_id* in IID.

• $i = \langle \textit{define}, c, IID\rangle$. Analogous to the *assignment* node rewriting step, with $\mathcal{V}_l[\![e]\!]$ replaced by c.

• $i = \langle \textit{input}, EID, IID\rangle$. Analogous to the *assignment* node rewriting step, with $\mathcal{V}_l[\![e]\!]$ replaced by $IV(EID)$, assigning the values read from the external objects specified in the set EID to the *internal_ids* in IID.

• $i = \langle \textit{output}, IID, EID\rangle$.

$$\begin{aligned} N' &= N - \{i\} \\ OV' &= OV[EID/IID] \end{aligned}$$

The notation $OV[EID/IID]$ denotes the assignment of the value of each *internal_id* $x_i \in IID$ to the corresponding *external_id* $y_i \in EID$.

• $i = \langle \textit{decision}, p\rangle$. Assume $b = \mathcal{V}_b[\![p]\!]$, where $\mathcal{V}_b$ is the predicate evaluation function.

$$\begin{aligned} N' &= N - \{i\} - \{j \mid \langle i,j,\bar{b}\rangle \in E\} \\ E' &= E - \{\langle i,j\rangle \mid \langle i,j\rangle \in E\} \\ &\quad - \{\langle j,k\rangle \mid \langle i,j,\bar{b}\rangle \in E\} \\ &\quad - \{\langle k,j\rangle \mid \langle i,j,\bar{b}\rangle \in E\} \end{aligned}$$

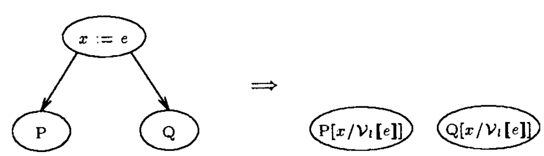

Figure 1 *Assignment* Node Rewriting, $IID = \{x\}$.

[2]For simplicity, we frequently include only the *head node* and *tail node* fields in edge specifications, assuming that the omitted field(s) can be bound to any legal value. If a specific binding is required, additional fields are specified.

The rewriting of a *decision* node is shown in Figure 2, where the predicate evaluates to **T** (*Control* edges are shown dotted). All nodes reachable by **F** edges from i are removed from the graph, as well as edges into and out of these nodes, and all **T** edges from i are removed.

3 Optimistic Transformations

Two main transformations, together with some auxiliary transformations, are used to derive optimistic computations from pessimistic ones.

1. The *data guess transformation* results from guessing the values produced by an *assignment* node so that these values can be used in the optimistic evaluation of subsequent nodes.
2. The **T** (**F**) *branch prediction transformation* results from guessing the value of the predicate of a *decision* node to be **T** (**F**) so that nodes reachable from the *decision* node by **T** (**F**) *control* edges can be evaluated optimistically.

Until the correctness of a guess is verified, the transformations prevent values computed by optimistically executed nodes from flowing into the rest of the computation, or from being written into the output vector. The values are discarded, if the guess is incorrect.

Optimistic transformations allow certain nodes in the pdg to be evaluated earlier than they would be without the transformations. The transformations remove *flow* and *control* edges leading into these nodes, thus increasing the available parallelism in the computation. The transformations are application-independent since they are applied to all pdgs identically, without regard to the operations appearing in the pdgs. However, the guesses that drive the transformations are necessarily application-dependent.

For the example given in this paper (Section 5), the branch prediction transformation suffices. The data guess transformation is similar, and a detailed description of it is omitted for brevity. A description appears elsewhere [2].

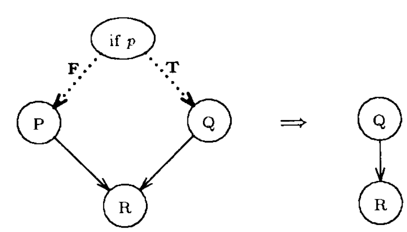

Figure 2 *Decision* Node Rewriting, $\mathcal{V}_b[\![p]\!] = \mathbf{T}$.

The following definition is used in the description of the transformations. Let N denote the node set of G and E denote the edge set of G. A *region* $\mathcal{R}$ of a pdg G is a subgraph of G containing a set of nodes $N_{\mathcal{R}} \subseteq N$ and a set of edges $E_{\mathcal{R}}$, where $\langle i,j \rangle \in E_{\mathcal{R}}$ if and only if $\langle i,j \rangle \in E \wedge (i \in N_{\mathcal{R}} \vee j \in N_{\mathcal{R}})$. Additionally, regions have no outgoing *control* edges and no paths along outgoing edges that lead back into the region. Examples of regions include:

- One or more *assignment*, *valve*, *define*, *input*, or *output* nodes, plus interconnecting edges (if any).
- A *decision construct*, consisting of a *decision* node i, an **F**-*region* (the portion evaluated only if i's predicate is **F**), and a **T**-*region* (the portion evaluated only if i's predicate is **T**).

3.1 Branch Prediction Transformation

The **T** (**F**) *branch prediction transformation* allows nodes reachable by **T** (**F**) *control* edges of a *decision* node to be evaluated optimistically, before the *decision* node is evaluated. Figure 3 shows the effect of a **T** branch prediction transformation. All nodes reachable by **T** *control* edges (region $\mathcal{T}$), minus the *output* nodes (region $\mathcal{T}_{out}$), can be evaluated before the predicate outcome at node i is determined. *Valve* nodes prevent the optimistically computed values from flowing into region $\mathcal{R}$ before the correctness of the guess is verified, and *control* edges prevent the optimistically computed values from being output.

Correctness (Sketch) If the branch prediction is incorrect, none of the optimistic evaluations can affect the output vector or the remainder of the pdg, region $\mathcal{R}$, since the *output* nodes are removed and the optimistically computed values are discarded by removing the *valve* nodes. Region $\mathcal{F}$ executes the same in the original and the transformed pdg. Hence, the same values flow into region $\mathcal{R}$, and region $\mathcal{R}$ evaluates the same in both pdgs. If the branch prediction is correct, region $\mathcal{T}$ evaluates the same in both pdgs since the same values flow into $\mathcal{T}$ from region $\mathcal{P}$. Consequently, the same values flow into region $\mathcal{R}$, and therefore $\mathcal{R}$ evaluates the same in both pdgs. Therefore, regardless of the correctness of the prediction, pdg evaluation is the same before and after this transformation.

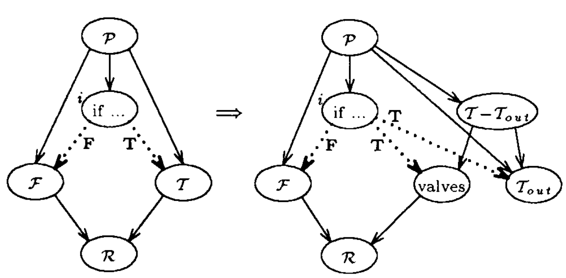

Figure 3 **T** Branch Prediction Transformation.

3.2 Region Copy Transformation

The *region copy transformation* copies a region $\mathcal{R}$ following the decision construct such that one copy of $\mathcal{R}$ is placed in the **F**-*region* and the other copy is placed in the **T**-*region*. This transformation is shown in Figure 4. A region copy transformation may be followed by an additional branch prediction transformation to allow nodes that were originally following the decision construct to be evaluated optimistically.

Correctness (Sketch) Region $\mathcal{P}$, the *decision* node i, and region $\mathcal{T}$ or $\mathcal{F}$ (depending upon the outcome of i) all evaluate the same in the original and the transformed pdgs since the values flowing into these regions are unaltered. In either case, we are left with a single copy of region $\mathcal{R}$ followed by $\mathcal{R}'$. Since all other parts of the pdg have evaluated the same both in the original and in the transformed pdg, the same values flow into $\mathcal{R}$ and $\mathcal{R}'$, and therefore both regions evaluate the same in the original and in the transformed pdg.

4 Send/Receive Nondeterminism

We extend the deterministic pdg model to accommodate a restricted form of nondeterminism, arising from message exchange between deterministic *processes*. In this extended model, a computation is defined as a collection of processes, and a *process* is defined as a deterministic pdg containing *send* and *receive* nodes. Each process is uniquely labeled with a process identifier *pid*. Values are passed between processes in *messages*, by means of

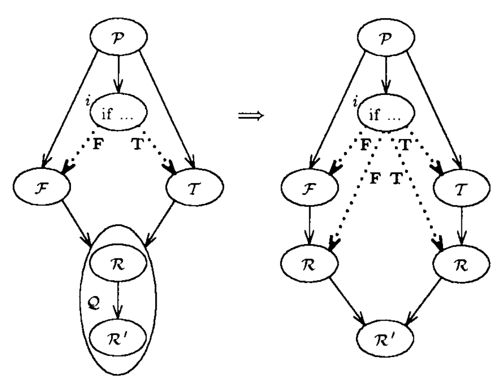

Figure 4 Region Copy Transformation.

send and *receive* nodes. Each process is *deterministic* in the sense that if it is given an input vector IV and an ordered set of messages received by the *receive* nodes, then the process produces the same output vector OV and sends the same sequence of messages to other processes during evaluation.

4.1 Definitions

The *send* and *receive* node types are defined as follows:

$$node \quad ::= \quad \langle send, m\rangle \mid \langle receive, M, R, IID\rangle$$

where m is a *message*, M is a *message set*, R is a *received-message set*, and IID is a set of *internal_ids*. The *send* node passes message m to all *receive* nodes that are directly reachable from the *send* node by a *flow* edge, placing the message in the *receive* node's message set M. The message contains a unique *send identifier* SID, the *pids* of the source and destination processes, and possibly other values.

All *receive* nodes of a given process are totally ordered. A *receive* node has a distinguished incoming *flow* edge, called the *enable* edge. The value flowing into the *receive* node over this edge is the received-message set R containing the $SIDs$ of all messages previously received by this process. A *receive* node is enabled when the incoming enable edge and all incoming *control* edges have been removed. When a *receive* node is enabled, it chooses any message in the message set M whose SID is not in the received-message set R and whose destination *pid* matches the *pid* of the process performing the receive. If no such message exists, evaluation of the *receive* node is suspended until an appropriate message becomes available. When evaluation of the node is resumed by the arrival of a message, the *internal_ids* $x_i \in IID$ are assigned the values in the incoming message, and these values flow over the appropriate *flow* edges to subsequent nodes, as in an *assignment* node evaluation. After *receive* node evaluation completes, an *assignment* node adds the SID of the received message m to the received-message set R, and passes this new set to the subsequent *receive* node over the subsequent node's incoming *enable* edge.

4.2 Discussion

Since computations can be nondeterministic, several possible output vectors can be produced for a given input vector. We define the semantics of a nondeterministic computation as the *set* of possible output vectors.

The optimistic transformations are essentially the same as in Section 3. *Send* and *receive* nodes are transformed in the same way as *assignment* nodes. However, we prevent the region copy transformation from copying a *receive* node if there is a path from the *receive* node to the *decision* node where the region copy transformation is applied (indicating a cycle in the graph). This restriction is necessary for correctness [2].

5 Fault Tolerance Using Message Logging and Checkpointing

We demonstrate the use of our optimistic transformations by applying them to a pessimistic fault-tolerant computation. We show elsewhere [2] how optimistic variants of distributed simulation, concurrency control, and the *make* program can be derived.

5.1 Pessimistic Algorithm

With fault tolerance methods based on pessimistic message logging and checkpointing, each message received by a process is logged *before* the process is allowed to act on that message. Processes are occasionally checkpointed, but no coordination is needed between the checkpoints of different processes. After a failure, a process is restarted from its latest checkpoint and the sequence of messages it received since that checkpoint are replayed from the log. Duplicate messages sent during recovery are ignored.

Each process in such a fault-tolerant computation consists of a number of receives, each followed by a check to determine if the message is logged, and, if so, by the necessary computation in response to the received message. Figure 5(a) shows, in outline, the pdg for such a process. *Flow* edges are labeled with the values that flow over these edges. The region labeled $\mathcal{C}$ represents the computation occurring as a result of an incoming message. Computation $\mathcal{C}$ is an arbitrary function of the process's *state vector* S and the incoming message m_1, and produces a new value of the state vector S. The node labeled "if m_1 logged" is a *decision* node that evaluates to **T** if the message is logged before a failure and to **F** otherwise. We refer to this *decision* node as the *if-logged* node. The *msg_set* is the set of messages received so far by the process. The newly received message gets added to *msg_set* only in the **T** branch of the *if-logged* node. Region $\mathcal{R}$ contains subsequent receive/log/computation intervals, similar to the one shown. The pdg for the entire fault-tolerant computation consists of a number of similar pdgs, one for each process, with *send* and *receive* nodes appropriately connected.

5.2 Optimistic Transformations

Optimistic fault tolerance methods based on message logging and checkpointing [8, 11] guess that messages are logged before a failure, and allow the processes to act on a message before it is guaranteed that the message has been logged.

We initially restrict our attention to a process that does not perform any outputs or any sends, and receives

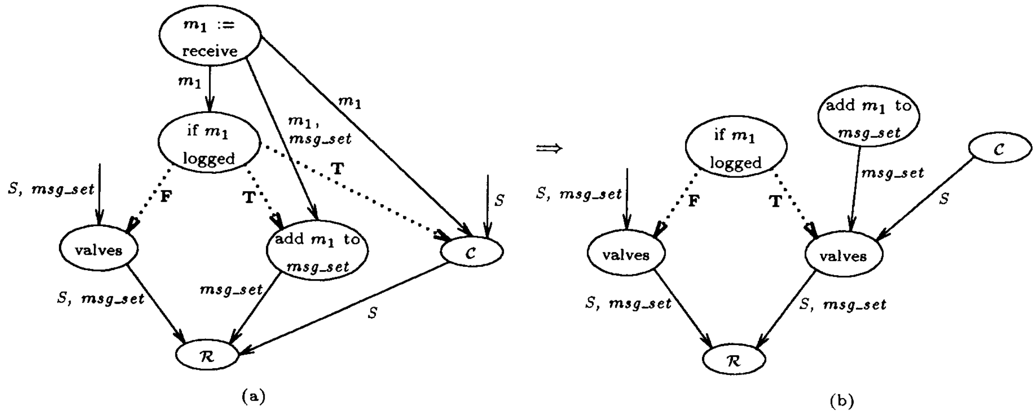

Figure 5 Fault-Tolerant Computation Pdg With Transformations.

no messages that have been sent optimistically as the result of a guess that has not been verified. In essence, we are restricting the optimism to a single process and do not let it "spread" to other processes. We will remove these restrictions shortly.

Figure 5(b) shows the pdg of a process after receiving a message and after a **T** branch prediction transformation has been applied to the *if-logged* node. While in the pessimistic version computation $\mathcal{C}$ cannot be executed until the preceding *if-logged* node returns **T**, $\mathcal{C}$ is enabled in the optimistic version, regardless of the execution of the *if-logged* node.

If the message gets logged before a failure occurs, the *if-logged* node evaluates to **T**. The **T** *control* edges from the *if-logged* node are removed, and the *valve* nodes reachable by these **T** *control* edges can pass the values of S and *msg_set* on to the region $\mathcal{R}$. The *valve* nodes reachable by **F** edges from the *if-logged* node are removed.

Otherwise, if there is a failure before the message gets logged, the *if-logged* node returns **F**. The *valve* nodes reachable by **T** edges from the *if-logged* node are removed and the values of S and *msg_set* computed by the optimistic computation based on the receipt of that message cannot reach region R. The **F** *control* edges from the *if-logged* node are removed, and the values of S and *msg_set* prior to this *receive* node flow into region $\mathcal{R}$.

To allow continued optimistic evaluation beyond region $\mathcal{C}$, we first apply several region copy and valve deletion transformations,[3] copying some or all of the nodes in $\mathcal{R}$ into both branches of the *if-logged* node. Then, we apply a branch prediction transformation, allowing nodes in the **T** branch to be evaluated optimistically based on the guess that message m_1 gets logged. If region $\mathcal{R}$ contains *receive* nodes, additional branch prediction transformations can be applied, as above.

We now allow $\mathcal{C}$ to contain *output* and *send* nodes. If $\mathcal{C}$ contains *output* nodes, then, as a result of the branch prediction transformation, a **T** *control* edge from the corresponding *if-logged* node remains directed at those *output* nodes, preventing their execution until the message has been logged.

Figure 6 depicts the case in which a *send* node appears as part of $\mathcal{C}$. In Figure 6(a), the graph has been transformed to the point where the *send* node is enabled in the sending process. In Figure 6(b), several region copy transformations have been applied to the appropriate *receive* node in the receiving process, and to the region $\mathcal{R}'$ following the *receive* node. In Figure 6(c), a **T** branch prediction transformation and a valve deletion transformation have been applied, allowing the *receive* node and region $\mathcal{R}'$ to execute optimistically.

Assume the *send* node depends on a single unconfirmed guess (as in Figure 6). If the guess is incorrect, the nodes reachable by the **T** *control* edges from the *if-logged* node are removed, and hence the optimistic execution of the *receive* node and all further nodes in region $\mathcal{R}'$ do not modify the output vector and do not flow beyond region $\mathcal{R}'$. The copy of the *receive* node and all further nodes following it in the **F** branch of the *if-logged* node become enabled. These nodes receive the state vector S and *msg_set* prior to the receipt of the message. If the guess is correct, all nodes in the **F** branch are deleted and all **T** *control* edges from the *if-logged* node are removed. If the *send* node depends on multiple unconfirmed guesses, *control* edges from each of the *if-logged* nodes at which a guess was made are di-

[3]The valve deletion transformation is an auxiliary transformation that removes the superfluous *valve* nodes [2].

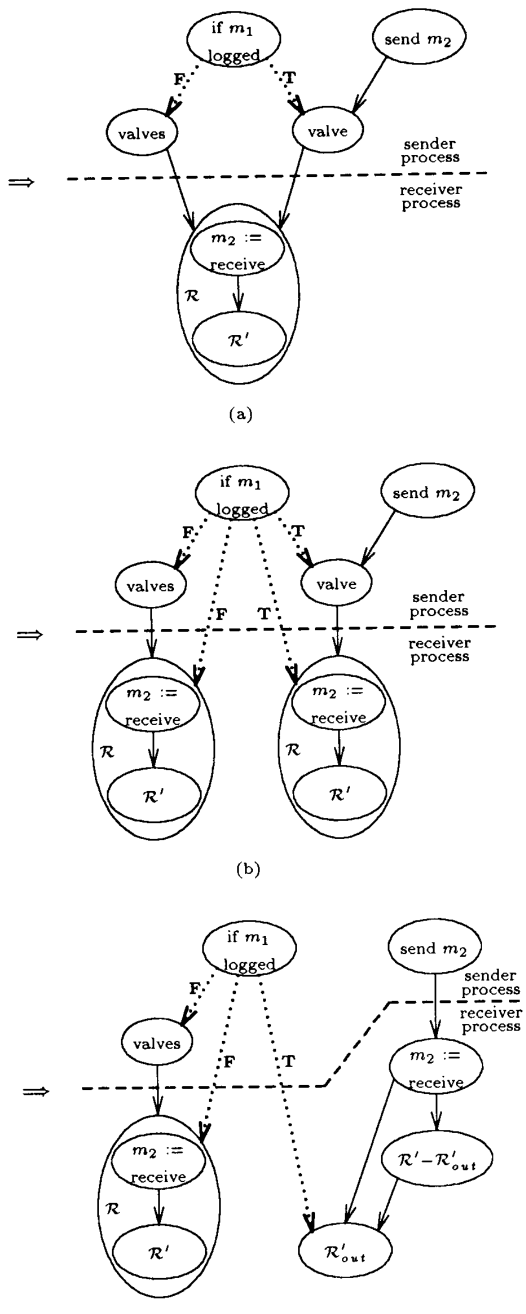

Figure 6 Additional Transformations Applied to Fault-Tolerant Computation Pdg.

rected at the nodes dependent on those guesses. Since the *control* edges have a conjunctive effect, the correctness of all guesses has to be verified. As soon as any one of these fails, the node is removed.

5.3 Towards A Realistic Implementation

The creation of nodes in the **F** branch of the *if-logged* node (as a result of the region copy transformation), along with the values of S and *msg_set* that flow into the *valve* nodes in the **F** branch of the *if-logged* node, represent the saving of the process's state, prior to the receipt of the corresponding message. In an implementation, this is the state that is restored in the event of a failure by restarting the failed process and replaying the logged messages in the order they were received. The nodes of the pdg that are evaluated optimistically after the transformations are applied correspond to the continuing execution using the message just received. After the message has been logged, the removal of all nodes reachable by **F** *control* edges from the *if-logged* node corresponds to the garbage collection of the saved state, which is no longer needed. Removal of the **T** *control* edges leading to *output* nodes corresponds to the commitment of output.

If processes execute at different sites, the confirmation of a guess can be implemented by a message sent from the site where the guess was made and verified to the site where the guess was "inherited" by the receipt of the message. One may view the **T** *control* edge emanating from the *if-logged* node as the channel over which this confirmation message is sent. The **F** *control* edges can be viewed as the channels on which to send a message notifying the receiver that the guess was incorrect. Here the mapping to a real implementation is not as direct since the failure typically causes the process to lose knowledge of which processes it sent messages to based on the receipt of messages that had not been logged.

Several application-specific transformations can be performed to improve the efficiency of optimistic message logging, and to provide notification of incorrect guesses. The information as to which guesses a particular node's execution depends on can be much more efficiently encoded by a dependency vector [8, 11]. With every receive, and thus with every guess, a counter in the receiving process, the *state interval index*, is incremented. With every message sent, the state interval index of the sender at the time of sending the message is included. This effectively summarizes all guesses on which the sending of this message is based. It suffices to include the last such guess, since an incorrect outcome of any of the earlier guesses is sufficient to render all subsequent guesses incorrect. The receiver of the message records the highest state interval index it has received in a message from each other process in a dependency vector. The dependency vector indicates the last guess of each other process that the receiver depends on. When messages get logged, the receiver of these messages can, for instance, announce this information in a message, and other processes can decide which optimistically executed nodes are now confirmed, by inspection of their dependency vectors.

In order to handle notification of failures, Strom and Yemini [11] increment a process's *incarnation number* after every failure. This incarnation number is sent along with all messages. When a process receives a message with a later incarnation number, it deduces that the sender has failed, and asks for the initial state interval index of the new incarnation This informs the process of those guesses of the failed process that were rendered incorrect by the failure. The process then discards any of its execution based on such incorrect guesses.

6 Conclusion

We have derived optimistic computations from equivalent pessimistic ones by performing optimistic transformations on the program dependence graph of the pessimistic computation. These optimistic transformations result from guessing data values or control flow decisions before they are known, and preserve the semantics of the pessimistic computation. While the transformations are application-independent, the guesses remain application-dependent. We have used our transformations to derive an optimistic version of fault tolerance based on message logging and checkpointing. Additional application-specific transformations are necessary to derive an efficient optimistic version. Our work improves on earlier work by Strom and Yemini [12] on optimistic transformations, in that we have identified application-independent transformations and in that we have shown that these transformations preserve the semantics of the computation.

Acknowledgements

We would like to thank several of our colleagues for their contributions to this work and their helpful comments on earlier drafts of this paper. Special thanks goes to John Carter, Corky Cartwright, Elmootazbellah Elnozahy, Matthias Felleisen, Jerry Fowler, Dave Johnson, Pete Keleher, Mark Mazina, and Becky Selke.

References

[1] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[2] R. Bubenik. *Optimistic Computation*. PhD thesis, Rice University, March 1990.

[3] R. Bubenik and W. Zwaenepoel. Performance of optimistic make. In *Proceedings of the ACM SIGMETRICS and PERFORMANCE '89 International Conference on Measurement and Modeling of Computer Systems*, pages 39–48, May 1989.

[4] J. Carter and W. Zwaenepoel. Optimistic implementation of bulk data transfer protocols. In *Proceedings of the ACM SIGMETRICS and PERFORMANCE '89 International Conference on Measurement and Modeling of Computer Systems*, pages 61–69, May 1989.

[5] R. Cartwright and M. Felleisen. The semantics of program dependence. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 13–27, June 1989.

[6] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[7] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[8] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181, August 1988.

[9] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Data Base Systems*, 6(2):213–226, June 1981.

[10] R.P. Selke. A rewriting semantics for program dependence graphs. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–24, January 1989.

[11] R.E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[12] R.E. Strom and S. Yemini. Synthesizing distributed and parallel programs through optimistic transformations. In *Current Advances in Distributed Computing and Communications*, pages 234–256. Computer Science Press, 1987.