Plan Switching: An Approach to Plan Execution in Changing Environments

Han Yu[1], Dan C. Marinescu[1], Annie S. Wu[1], Howard Jay Siegel[2], Rose A. Daley[3], and I-Jeng Wang[3]

[1]School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, Florida, 32816, USA
{hyu, dcm, aswu}@cs.ucf.edu

[2]Department of Electrical and Computer Engineering and Department of Computer Science
Colorado State University
Fort Collins, Colorado, 80523-1373, USA
HJ@ColoState.edu

[3]Johns Hopkins University
Applied Physics Laboratory
11100 Johns Hopkins Road Laurel, MD 20723-6099
{Rose.Daley, I-Jeng.Wang}@jhuapl.edu

Abstract

The execution of a complex task in any environment requires planning. Planning is the process of constructing an activity graph given by the current state of the system, a goal state, and a set of activities. If we wish to execute a complex computing task in a heterogeneous computing environment with autonomous resource providers, we should be able to adapt to changes in the environment. A possible solution is to construct a family of activity graphs beforehand and investigate the means of switching from one member of the family to another when the execution of one activity graph fails. In this paper, we study the conditions when plan switching is feasible. Then we introduce an approach for plan switching and report the simulation results of this approach.

1. Introduction

A large-scale distributed computing system consists of a collection of heterogeneous systems. Various computing resources and services coexist in a system. The availability of resources and services typically changes quickly over time. The execution of a task cannot be guaranteed when autonomous resource providers are involved and when the goal of the computation is allowed to change. Replanning and plan switching are two basic strategies to overcome this uncertainty. Replanning is a process of either creating a new plan or adapting the existing plan to the current conditions of the computing environment (by using resources or services currently available to the system). Replanning introduces additional time for execution, as the process of replanning and executing a computing task cannot be overlapped. Plan switching assumes that there are a group of activity graphs, or plans, available to perform a computing task. Only one plan is selected for execution. When the execution of the plan fails, we find an alternative plan and migrate the execution of a computing task directly from the current plan to the alternate. If plan switching is successful, we continue the execution without replanning.

This paper addresses the problem of plan switching and presents an approach to this problem. The main idea of the approach is to create alternative plans and locate the execution points from alternative plans for a given task in parallel with the execution of the current plan. When the execution cannot proceed, we continue the execution of a computing task from a selected execution point in another plan. The process of creating alternative plans and finding execution points has a relatively small computational cost and can be performed in parallel with the execution of the computation. Therefore, this approach does not necessarily

increase the execution time of a computing task.

2. Related Work

Much work has been devoted to planning in uncertain environments or to guarantee that the execution of a plan satisfies the conditions or restrictions imposed by real-time systems. These earlier approaches can be classified into two categories: 1) constructing a plan that can react to external events at certain reachable states of plan execution; 2) dynamically modifying an existing plan during plan execution.

Atkins et al. [1] address the problem of generating fault-tolerance plans for systems that have real-time safety requirements. Their approach combines planning with resource allocation and focuses on the interface between these two components. If certain conditions in plan execution cannot be satisfied, backtracking in the current plan is allowed to build an alternative plan. Their approach is implemented upon CIRCA, an intelligent real-time control architecture that combines AI planning techniques to meet the demands in a dynamic real-time computing and control environment [5].

Boutilier et al.'s work [2] is among the many studies of building computational models for planning in uncertain environment. Their model uses a Markov decision process and assumes that a system contains a finite set of states and the probability distribution function determines the transition among the states. They introduce structures for the compact representation of the state space, action space, and value functions of such planning domains, and illustrate how the properties of these structures can be exploited by the planning algorithms.

Pell et al. [6] address the issue of robust planning in uncertain environments in their work to develop an AI system for controlling an aircraft. The problem requires planning in multiple stages; the execution of the current plan is overlapped with the generation of the plan for the next stage. Their approach uses a flexible and abstract planning model that takes into account the complexity of planning and robustness of plan execution.

Wilkins et al. [7] use an asynchronous replanning approach in their design of Cypress, a framework that enables agents to achieve complex goals in a dynamic environment. When a problem occurs during plan execution, a request for replanning is generated while the system continues to execute the portions of the current plan that are not affected by the problem.

In this paper, we present a different approach by assuming that multiple plans are available to perform a computing task and we allow dynamic plan switching during the execution of plans.

3. Problem Formulation

3.1 Assumptions

Plan switching is a function of intelligent middleware (a detailed discussion of the middleware can be found in [8]), based upon several assumptions:

1. A plan, or an activity graph, is a directed graph whose vertices are the atomic activities and directed arcs denote data and control flow dependencies. Concurrent activities are allowed, but iterative execution of activities is not allowed. Only independent activities may be executed concurrently; no communication among concurrent activities is allowed.

2. A family of plans are created in advance. One of the members of the family, the one selected for execution, is called the *current plan*. All other plans serve as backups but still have a chance of execution when the current plan fails. These plans are called *alternative plans*.

3. Once an activity in a plan begins execution, the success of its execution is guaranteed. If, however, an activity cannot be executed, we may either wait until it can be executed or switch the execution of the current plan to an alternative plan.

3.2 Procedure of Plan Switching

The procedure of plan switching consists of four steps: 1) generating alternative execution plans for the computing task; 2) detecting failures in the execution of the current plan; 3) locating an execution point from alternative plans in which the computation can continue from the execution point and still reach the goal of the computation; and 4) migrating the execution after a successful plan switching. Figure 1 shows the procedure of plan switching.

1. Generating Alternative Execution Plans

   The generation of alternative execution plans can be performed either before the execution of a computation task (i.e., before the execution of the current plan starts), or in parallel with the execution of the current plan. These plans are the candidates for plan execution when a new plan needs to be found for plan switching.

2. Detecting Failures in the Execution of the Current Plan

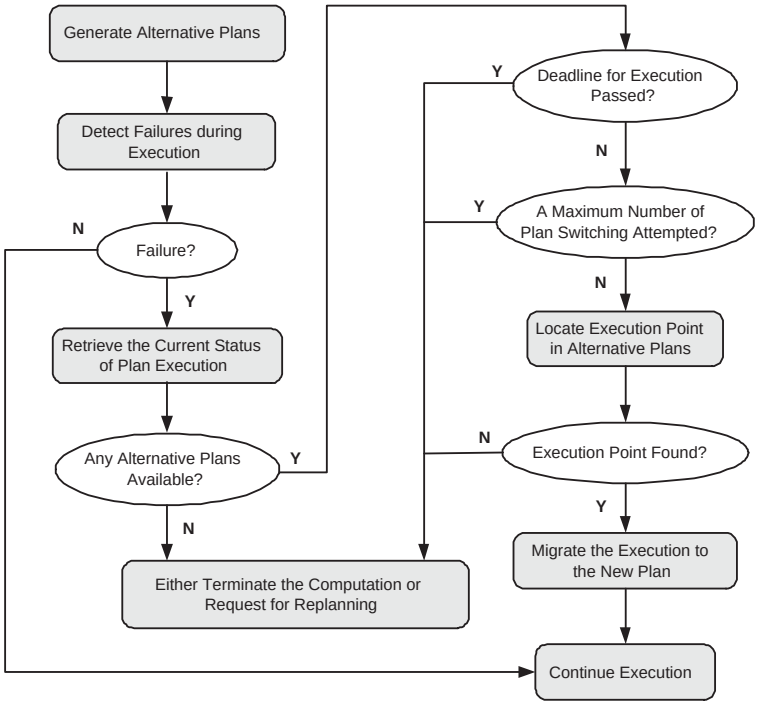

Figure 1. The procedure of plan switching.

Failure in the plan execution may occurs for a variety of reasons, e.g., a failure in computing nodes that supports the execution, the temporary unavailability of computing resources. When failure occurs, we retrieve the current status of the computation, collecting all intermediate data that were generated from the execution of the current plan.

3. Locating an Execution Point from Alternative Plans

   After we have obtained the knowledge regarding the current status of the computation, we decide whether plan switching can be performed. The decision of plan switching is determined by three conditions: 1) whether there is no alternative plan available to perform the same computing task; 2) whether the deadline for the computing task, if specified by a user, has already passed; and 3) whether a maximum number of plan switchings have been attempted. If any one of the above conditions is satisfied, we do not perform plan switching, either terminating the computation or requesting for replanning. Otherwise, we apply the plan switching algorithm, which will be discussed in Section 4, and locate an execution point in alternative plans so that the computation can continue from that point.

4. Migrating the Execution

   If an execution point can be found, we migrate the execution of the computation from the current plan to the new plan. The process of migration consists of three steps: 1) scheduling the execution of all subsequent activities from the execution point in the new plan; 2) transferring all intermediate data to the corresponding computing nodes where the activities are assigned; and 3) starting the execution of the computation on the new plan. If, however, plan switching fails, we either terminate the computation or request for replanning.

3.3 Definitions

We now provide several definitions necessary to formulate our plan switching problems. The term "snapshot" has been used to determine the progress of multiple processes running on distributed systems [3]. We use the same term to determine the progress of a plan execution. We next introduce the concept of "congruent snapshot," which is the basis of the plan switching approach.

Definition 1. A *single snapshot* is a partial description of the progress of plan execution. A single snapshot can be defined on either a pair of consecutive activities $\{a, b\}$ in a plan, denoting that activity a has finished execution while activity b is still pending, or between an activity and a dummy activity, if the activity has no precedent or subsequent activities.

We can annotate a plan by adding single snapshots in three cases: 1) between every two consecutive activities, 2) before all activities that have no precedent activities, and 3) after all activities that have no subsequent activities. Figure 3 shows an annotated version for the plan in Figure 2.

Definition 2. The *subsequent activities* of a single snapshot are the set of all activities that should be executed after the snapshot is reached. We use the function $subs(s)$ to denote the subsequent activities of a given single snapshot s. For instance, in Figure 3, $subs(s_1) = \{a_1, a_2, a_3, a_4, a_5, a_6\}$, $subs(s_5) = \{a_5, a_6\}$, and $subs(s_9) = \phi$. The *preceding activity* of a single snapshot is a set that contains the most recent executed activity, if it exists, before the snapshot is reached. We use the function $prec(s)$ to denote the preceding activity of a given single snapshot s. For instance, in Figure 3, $prec(s_1) = \phi$, $prec(s_5) = \{a_2\}$, and $prec(s_9) = \{a_6\}$.

Definition 3. A pair of single snapshots, s_a and s_b, is independent if $prec(s_a) \cap subs(s_b) = \phi$ and $prec(s_b) \cap subs(s_a) = \phi$. For instance, snapshots s_2 and s_3 in Figure 3 are independent snapshots, while snapshots s_2 and s_5 are not. A set of single snapshots S is independent if every pair of single snapshots in S is independent. For instance, in Figure 3, $S = \{s_2, s_3, s_4\}$ is a set of independent snapshots.

Definition 4. A *composite snapshot* is a combination of single snapshots. We use a pair of square brackets "[" and "]" to denote the operation of combining single snapshots. For instance, $[s_2, s_3]$ denotes a composite snapshot that combines s_2 and s_3. A composite snapshot may contain single snapshots that are not independent of each other.

The above notions for a single snapshot can also be applied to composite snapshots. The *subsequent activities* of a composite snapshot are the union of the sets of subsequent activities of all single snapshots. The *preceding activities* of a composite snapshot are the union of the sets of preceding activities of all single snapshots. Two composite snapshots, s_a and s_b, are independent if $prec(s_a) \cap subs(s_b) = \phi$ and $prec(s_b) \cap subs(s_a) = \phi$. For instance, in Figure 3, $subs([s_2, s_3]) = \{a_2, a_5, a_6\} \cup \{a_3, a_5, a_6\} = \{a_2, a_3, a_5, a_6\}$, $prec([s_2, s_3]) = \{a_1\}$, and snapshots $[s_2, s_3]$ and s_4 are independent.

Definition 5. A snapshot is *consistent* if it is either a single snapshot or a composite snapshot of a set of independent snapshots. A consistent snapshot s in a plan P is a *global consistent snapshot* if there does not exist a single snapshot s' in P such that s' is not included in s and s' is independent of all snapshots in s. In contrast to a single snapshot, a global consistent snapshot gives a complete view of the status of a plan execution. For instance, the composite snapshot $[s_2, s_3, s_4]$ in Figure 3 is a global consistent snapshot. This snapshot describes a status of plan execution in which activity a_1 has finished execution while activities a_2, a_3, and a_4 are pending execution followed by a_5 and a_6.

Definition 6. A global consistent snapshot s_1 in plan P_1 is *congruent* to a global consistent snapshot s_2 in plan P_2 if we are able to switch execution from s_1 to s_2, execute the subsequent activities of s_2, and finish the computing task. The subsequent activities of s_1 and s_2 do not have to be the same across the two plans; these activities within each plan must be able to collectively complete the task. For instance, Figure 4 shows two available plans, P_1 and P_2, to perform a computing task. If the task can be finished by executing $\{a_1, a_2, a_3, a_4\}$ from P_1 and $\{b_3, b_4, b_5\}$ from P_2, the global consistent snapshot $[s_3', s_4']$ in Plan P_2 is congruent to the global consistent snapshot $[s_5, s_6, s_7]$ in Plan P_1. We use the symbol "$\sim$" to denote the relation of congruency. If a snapshot s_1 is congruent to s_2, $s_1 \sim s_2$.

Definition 7. The *optimal congruent snapshot* for a given snapshot is the one whose subsequent activities incur minimal execution cost among all congruent snapshots.

3.4 Plan Switching between Congruent Snapshots

We formulate the problem of plan switching as follows: if the execution of the current plan P_{curr} cannot proceed from a global consistent snapshot s, find a congruent snapshot s' of s from alternative plans and continue the execution from s' in the plan to which s' belongs. Figure 4 shows an example of switching between two plans, P_1 and P_2. Initially, P_1 is the current plan. When the execution of P_1 cannot continue in snapshot $[s_5, s_6, s_7]$, a congruent snapshot $[s_3', s_4']$ in Plan P_2 is found, and the plan execution is switched to P_2 from this congruent snapshot. When the execution finishes, the complete set of activities having been executed is $\{a_1, a_2, a_3, a_4\}$ from P_1 and $\{b_3, b_4, b_5\}$ from P_2. As we allow plan switching to occur multiple times during the execution of a computing task, a plan that fails during execution may still have an opportunity of

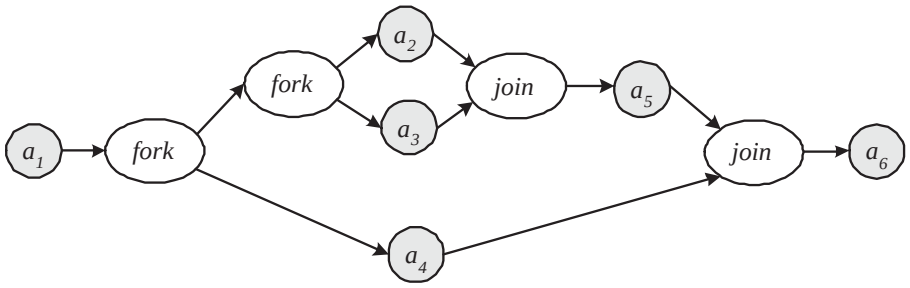

Figure 2. An example plan that contains six activities.

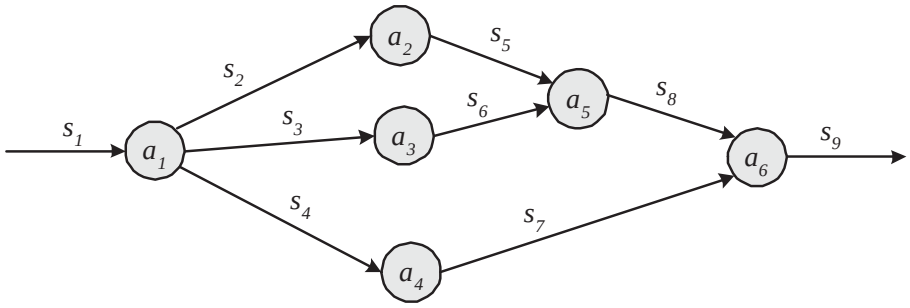

Figure 3. An annotated version of the plan shown in Figure 2. Nine single snapshots are added.

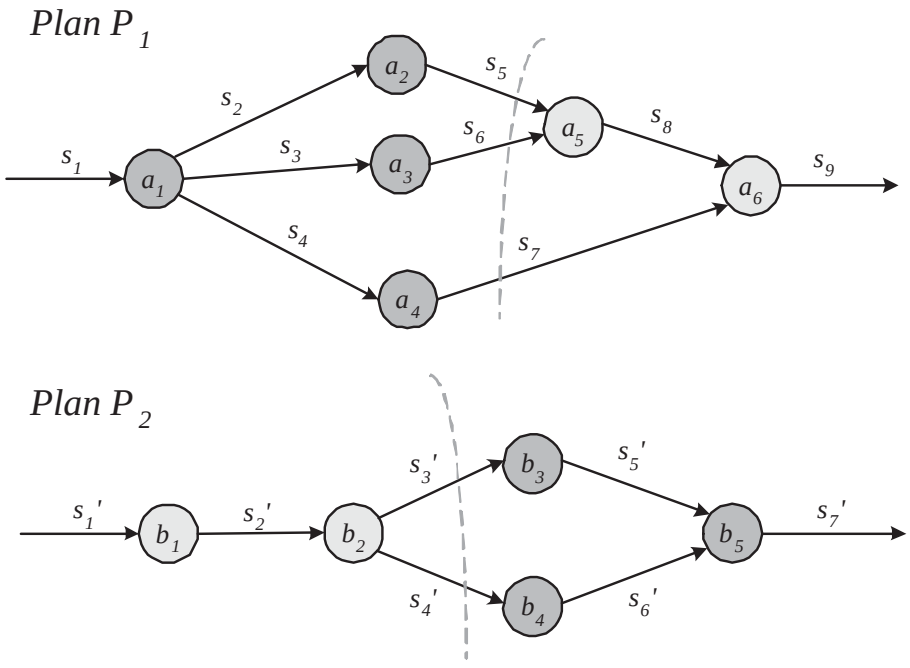

Figure 4. An example of execution switching between two plans. Snapshot $[s_3', s_4']$ in Plan P_2 is congruent to snapshot $[s_5, s_6, s_7]$ in Plan P_1.

execution if the execution fails again in the new plan.

4. Algorithm Design

The process of finding a congruent snapshot for a given snapshot consists of three steps: 1) generate a set of global consistent snapshots for each plan; 2) identify congruent snapshots from the set of global consistent snapshots; and 3) select a congruent snapshot from the identified congruent snapshots.

1. Finding Global Consistent Snapshots

   To find a set of global consistent snapshots for a plan, we first set S to be the set of all single snapshots in the plan. Then we repeat the every step in Figure 5 to update the snapshots in S until S cannot be further updated. The final set S is the set of global consistent snapshots of a plan. Note that S does not necessarily contain all global consistent snapshots of a plan.

2. Locating Congruent Snapshots for a Given Snapshot

   Locating congruent snapshots from alternative plans allows a plan executor to easily switch task execution from the current plan to an alternative plan, when temporary or permanent failure occurs in current plan execution. During the execution of the current activity(ies), we try to find the congruent snapshots for the snapshot after the execution of the current activity(ies) finishes. There are two criteria for a snapshot to be congruent: 1) all its subsequent activities can be executed; 2) the execution of the subsequent activities is able to produce the data that satisfy all goals for the computing task. One way to verify a congruent snapshot is based on the preconditions and postconditions of every subsequent activity (refer [8] for detailed information). Figure 6 shows the pseudo code for locating congruent snapshots.

3. Selecting a Congruent Snapshot

   Once we have identified congruent snapshots in step two, we are able to estimate the computational cost of all subsequent activities for each congruent snapshot. If the computational costs of activities are not given or are difficult to estimate, a rough estimation that simply counts the number of subsequent activities for a congruent snapshot is applied. The congruent snapshot to be selected is the one that incurs the lowest computational cost among all congruent snapshots found in step two.

What should we do if there does not exist a congruent snapshot when plan switching is requested? There are three options: 1) send out a request for replanning; 2) terminate the execution of the current plan completely and execute an alternative plan from the beginning; and 3) roll back the execution of the current plan to the previous global consistent snapshot and try to find a congruent snapshot for that snapshot. We discuss the third option in detail.

Rollback is a process of backtracking the computation to a previous saved point when a failure occurs, so that the whole computation does not have to be resumed from the beginning [4]. We say an activity is *reversible* if we can roll back the execution of the activity completely to the snapshot right before it is executed. In order to roll back the execution of activities, we need to record every global consistent snapshot that has been reached and an ordered list of all activities that have been executed in the current plan. When a plan execution cannot proceed and there is not a congruent snapshot for the current snapshot, we try to roll back the execution of the last executed activity. If the activity is not reversible, we have to choose one of the first two options, either perform replanning or choose another plan to execute from the beginning. Otherwise, we roll back the execution of the activity, regress the computation to the previous snapshot, and attempt to find a congruent snapshot for this snapshot. If a congruent snapshot exists, we switch the execution of the computation to another plan. If, however, a congruent snapshot is still unavailable, we repeat the preceding steps and roll back the execution of previous activities, until we have successfully reached a snapshot that has a congruent snapshot, or the execution of the computation cannot be further rolled back.

5. Simulation Study

We perform a simulation study to evaluate the effectiveness of plan execution. The simulation environment consists of a number of randomly generated plans. Both the sizes of plans (denoted by the number of activities) and the number of subsequent activities of each activity in a plan may be different but follow Gaussian distributions. In average, a plan contains ten activities and each activity has two subsequent activities. A maximum ten plan switches can occur during the computation of an entire task. If this limit is reached, we terminate the process and mark the plan execution as a failure. We also assume that all activities have the same computational costs.

We test different cases by varying the success rate of activity execution, the probability of a global consistent

```
Begin.
1. For each snapshot s in S, find all single snapshots that are independent of s.
2. If s has at least one independent snapshot, do
   a. Remove s from S.
   b. For each of its independent snapshots s', do
      (1) Combine s with s'.
      (2) If the combined snapshot is not in S, include it in S.
   c. End for.
3. End if.
End.
```

Figure 5. The procedure of finding global consistent snapshots for a plan.

```
Begin.
1. P-curr = the currently executing plan.
2. A-curr = the set of currently executing activities.
3. s-curr = the global consistent snapshot after all activities in A-curr finishes.
4. S = {}. /* initially, the set of congruent snapshots is empty */
5. For each alternative plan P, do
   a. For each global consistent snapshot s in P, do
      (1) If subs(s) can be executed from s-curr and the execution of subs(s)
          produces the data that satisfy all goals for the computing task, include
          s in S.
   b. End for.
6. End for.
End.
```

Figure 6. The procedure of locating congruent snapshots for a given snapshot in the current plan.

snapshot being a congruent snapshot, and the number of available plans to a computing task. We test each case 50 times, each time using a set of randomly generated plans. We use the algorithm shown in Figure 5 to generate the set of global snapshots of each plan. We evaluate the performance with two criteria: the success rate of plan execution (i.e., the number of runs out of 50 runs that a computation task can successfully finish) and, for the successful runs, the average number of plan switchings performed. Table 1 lists the parameter settings for the experiment.

We first test the case in which there are three available plans (i.e., one current plan and two alternative plans) and the probability of congruent snapshots is fixed at 0.1. We vary the success rate of a computing activity between 0.4 and 0.9. No activity is allowed to roll back its execution. Figure 7 shows the number of successful runs in each case and the minimum, average, and maximum number of plan switchings in the successful runs. The results indicate that a lower success rate of computing activities increases the possibility and occurrences of plan switchings. As a congruent snapshot cannot always be found when plan switching is requested, a lower success rate of computing activities results in a higher probability of failure in plan execution.

Next, we evaluate the impact of the probability of congruent snapshots on the success of plan switching. We test the cases in which only 1% and 5% of the global consistent snapshots are congruent snapshots. Again, the success rate of a computing activity is set between 0.4 and 0.9, and no activity is allowed to roll back its execution. The simulation results, shown in Figures 8 and 9, indicate that the probability of congruent snapshots has profound effect on the success of plan switching. A lower probability leads to a lower possibility of finding a congruent snapshot, and thus reduces the success rate of plan execution. When the probability of congruent snapshots is reduced to 0.01, the failure of plan execution, in most cases, is due to inability to find congruent snapshots rather than overreaching the maximum allowed number of plan switches. This result indicates that allowing rollback of plan execution might be an effective cure to plan execution when the probability of congruent snapshots is low.

Figure 10 shows the results in additional 10 runs when rollback of plan execution is enabled and all activities in a plan are reversible. The probability of congruent snapshots is kept as low as 0.01. Obviously, allowing rollback of activity execution gives more opportunities for finding congruent snapshots and thus increases the probability of a successful plan switch. Comparing Figures 9(a) and 10(a), only for some of the different probabilities of a successful activity does rollback improve the percentage of successful runs. However, Figure 11 indicates that in those failed runs, allowing rollback offers more chances for plan switching among alternative plans. Some runs fail solely because a maximum number of plan switches have already been attempted.

In the third case, we study whether the number of alternative plans affects the success of plan switching. We repeat the above tests by setting the probability of congruent snapshots to 0.05 but increase the number of plans to six. Activities are not allowed to roll back their execution. The simulation results, shown in Figure 12, demonstrate that having more alternative plans definitely improves the performance of plan switching, especially in cases where the probability of a successful activity is low (≤ 0.6). When there are six available plans, the success of plan switching is less likely to rely on the success rate of activity execution, as more global consistent snapshots (hence more congruent snapshots) are available from alternative plans.

6. Conclusions and Future Work

This paper focuses on the problem of plan switching: switching the execution of a computing task among multiple plans. We formulate the problem of plan switching and present an approach to the problem. This approach introduces the concept of congruent snapshots that allow transition from the execution of one plan to another. The main idea of the approach is to find congruent snapshots from alternative plans so that when the execution of the current plan fails, we continue the execution of the task from a selected congruent snapshot in another plan.

A simulation study on this approach indicates that a high probability of congruent snapshots, a high success rate of computing activities, and more alternative plans can improve the performance of plan switching. In addition, allowing rollback of activity execution offers additional opportunities to find congruent snapshots, thus is also benefiting plan switching.

Future work will address the problem of improving the process of selecting the congruent snapshot for continuing the plan execution. Currently, the congruent snapshot to be selected is the one that has the lowest cost of subsequent activities among all candidates. While this is a simple approach, it may result in switching to a plan that contains inexecutable activities so that another request of plan switching may be inevitable. Other heuristics can be embedded into this process to balance both the execution cost of a computing task and the success rate of plan execution.

Parameter	Value
Number of Runs	50
Number of Plans	3, 6
Avg. Number of Activities	10
Maximum Number of Plan Switches	10
Avg. Number of Subsequent Activities	2
Prob. of an Successful Activity	0.4 - 0.9, with increase step of 0.1
Prob. of a Congruent Snapshot	0.01, 0.05, and 0.1

Table 1. Parameter settings for the experiment.

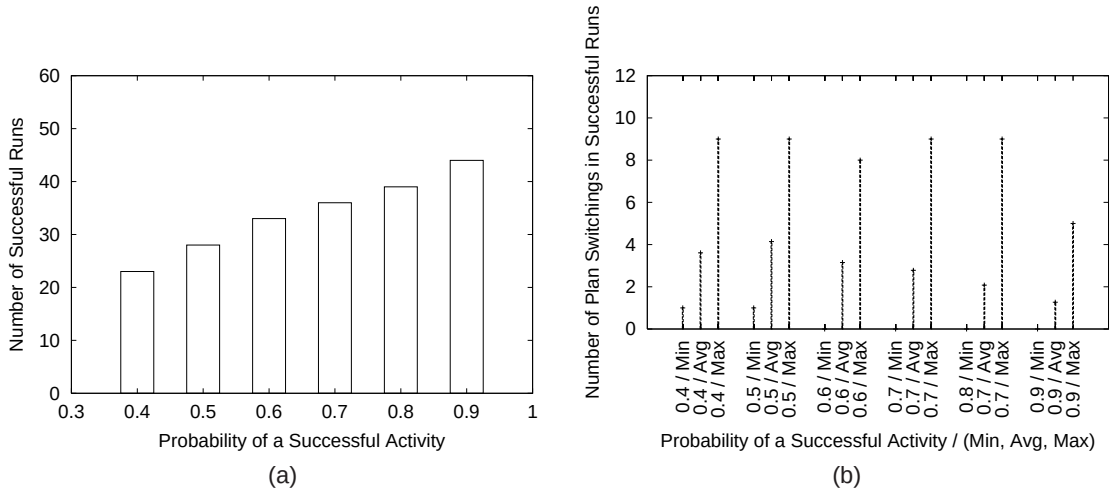

Figure 7. The simulation results on the effect of the success rate of a computing activity to the success of plan switching. (a) The number of successful runs out of 50 runs. (b) The average, minimum, and maximum number of plan switches in successful runs.

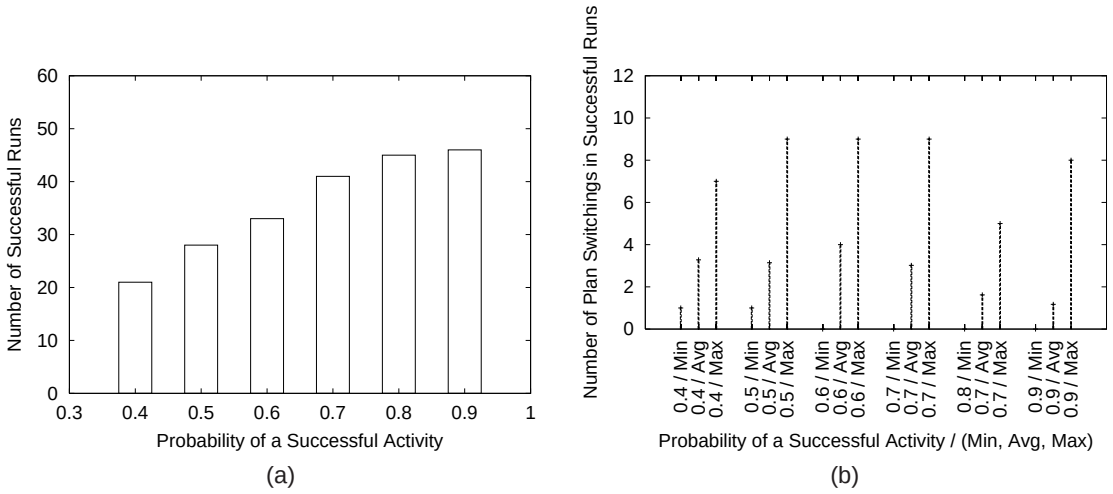

Figure 8. The simulation results when 5% of the global consistent snapshots are congruent snapshots. (a) The number of successful runs out of 50 runs. (b) The average, minimum, and maximum number of plan switches in successful runs.

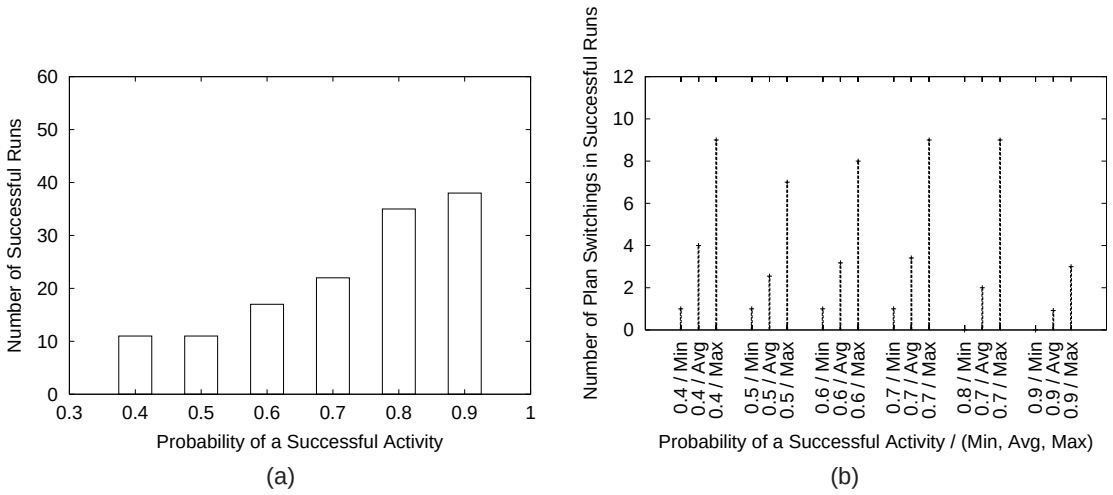

Figure 9. The simulation results when 1% of the global consistent snapshots are congruent snapshots. (a) The number of successful runs out of 50 runs. (b) The average, minimum, and maximum number of plan switches in successful runs.

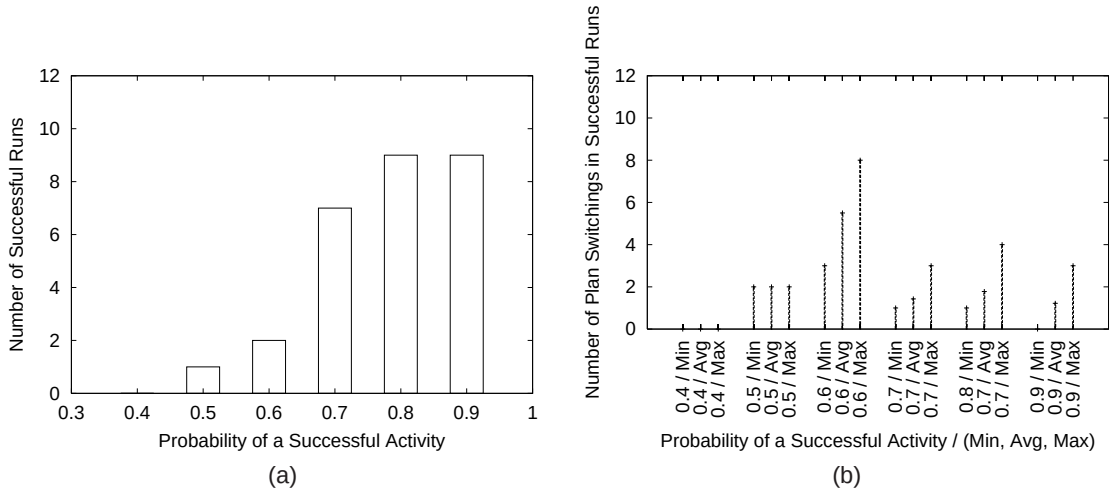

Figure 10. The simulation results showing the effectiveness of allowing rollback in plan execution when all activities are reversible and 1% of the global consistent snapshots are congruent snapshots. (a) The number of successful runs out of 10 runs. (b) The average, minimum, and maximum number of plan switches in successful runs.

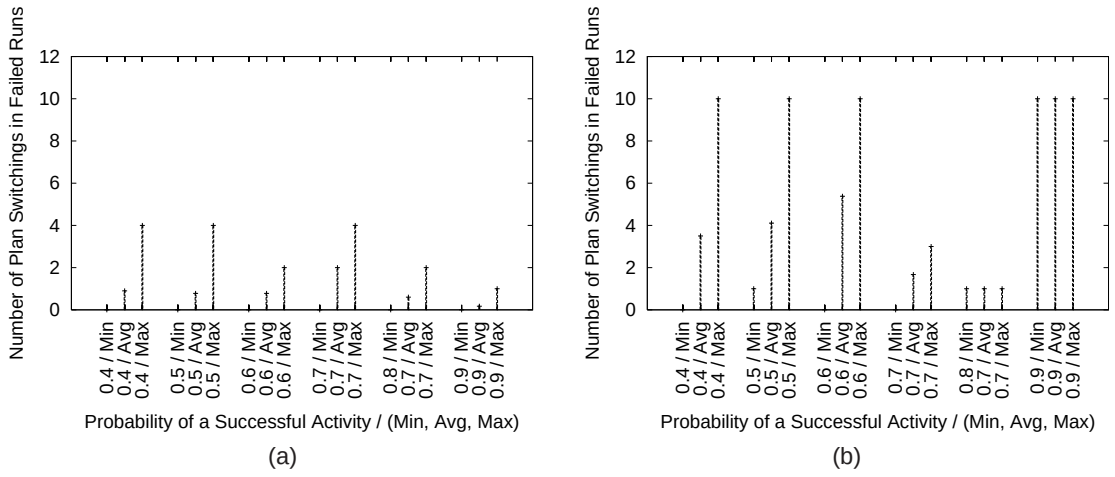

Figure 11. The minimum, average, and maximum number of plan switches before the plan execution fails. (a) Rollback of execution is not allowed. (b) Rollback of execution is allowed.

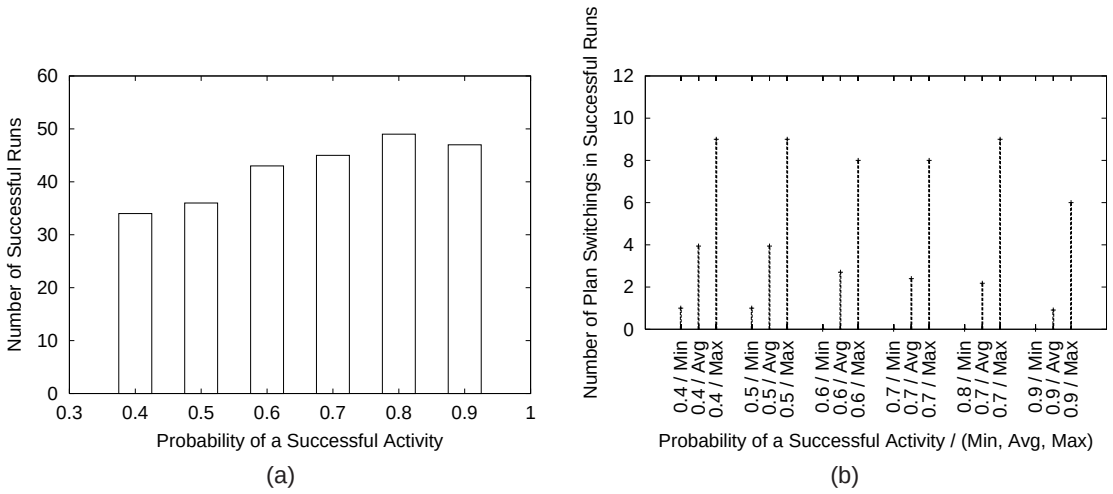

Figure 12. The simulation results for cases in which six plans are available for execution and 5% of the global consistent snapshots are congruent snapshots. (a) The number of successful runs out of 50 runs. (b) The average, minimum, and maximum number of plan switches in successful runs.

In addition, the algorithm shown in Figure 5 can only find a subset of all global consistent snapshots of a plan. Another approach to improving the success of plan switching is to design an algorithm that finds all global consistent snapshots of a plan so that more congruent snapshots can be located when plan switching is requested.

Acknowledgments

This research was supported by National Science Foundation grants MCB9527131, DBI0296035, ACI0296035, and EIA0296179, the DARPA Information Exploitation Office under contract No. NBCHC030137, by the Colorado State University Center for Robustness in Computer Systems (funded by the Colorado Commission on Higher Education Technology Advancement Group through the Colorado Institute of Technology), and by the Colorado State University George T. Abell Endowment. Approved for public release, distribution unlimited.

References

[1] E. M. Atkins, T. F. Abdelzaher, K. G. Shin, and E. H. Durfee. Planning and resource allocation for hard real-time, fault-tolerant plan execution. *Journal of Autonomous Agents and Multi-Agent Systems*, 4:57–78, 2001.

[2] C. Boutilier, T. Dean, and S. Hanks. Planning under uncertainty: structural assumptions and computational leverage. In *New Directions in AI Planning*, pages 157–172, 1996.

[3] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[5] D. J. Musliner, E. H. Durfee, and K. G. Shin. World modeling for the dynamic construction of real-time control plans. *Journal of Artificial Intelligence*, 74(1):83–127, 1995.

[6] B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith. Plan execution for autonomous spacecraft. In *Proceedings of AAAI-96 Fall Symposium*, pages 109–116, 2004.

[7] D. E. Wilkins, K. L. Myers, and J. D. Lowrance. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227, 1995.

[8] H. Yu, X. Bai, G. Wang, Y. Ji, and D. C. Marinescu. Metainformation and workflow management for solving complex problems in grid environments. In *Proceedings of the 13th Heterogeneous Computing Workshop (HCW)*, 2004.

Biographies

Han Yu received his Ph.D. in Computer Science from the University of Central Florida (UCF) in 2005. He received BS degree from Shanghai Jiao Tong University in 1996 and MS degree from the University of Central Florida in 2002. His research area includes grid

and distributed computing, evolutionary computation, and AI planning.

Dan C. Marinescu is Professor of Computer Science at University of Central Florida. He is a Provost Research Professor and Scientific Director of the Interdisciplinary Information Science and Technology Laboratory at UCF. From 1984 till 2001 he was a Professor in the Department of Computer Science at Purdue University. He is conducting research in parallel and distributed computing, computational structural biology, and quantum computing. He has co-authored more than 160 papers published in refereed journals and conference proceedings. He is the author of the book *Internet-based Workflow Management* published by Wiley in 2002, has co-edited *Process Coordination and Ubiquitous Computing* published by CRC Press in 2002. His most recent book *Approaching Quantum Computing* was published by Prentice Hall in 2004. He is a member of the editorial board of several journals and member of the Program Committee for international conferences. He was the keynote speaker and tutorial lecturer at scientific meetings in US and abroad. He has consulted for industry and government. See http://www.cs.ucf.edu/∼dcm for additional information.

Annie S. Wu is an Associate Professor in the School of Electrical Engineering and Computer Science and Director of the Evolutionary Computation Laboratory at the University of Central Florida (UCF). Before joining UCF, she was a National Research Council Postdoctoral Research Associate at the Naval Research Laboratory. She received her Ph.D. in Computer Science & Engineering from the University of Michigan.

Howard Jay Siegel holds the endowed chair position of Abell Distinguished Professor of Electrical and Computer Engineering at Colorado State University (CSU), where he is also a Professor of Computer Science. He is the Director of the CSU Information Science and Technology Center (ISTeC). ISTeC a university-wide organization for promoting, facilitating, and enhancing CSU's research, education, and outreach activities pertaining to the design and innovative application of computer, communication, and information systems. Prof. Siegel is a Fellow of the IEEE and a Fellow of the ACM. From 1976 to 2001, he was a professor in the School of Electrical and Computer Engineering at Purdue University. He received a B.S. degree in electrical engineering and a B.S. degree in management from the Massachusetts Institute of Technology (MIT), and the M.A., M.S.E., and Ph.D. degrees from the Department of Electrical Engineering and Computer Science at Princeton University. He has co-authored over 300 technical papers. His research interests include heterogeneous parallel and distributed computing, parallel algorithms, parallel machine interconnection networks, and reconfigurable parallel computer systems. He was a Coeditor-in-Chief of the Journal of Parallel and Distributed Computing, and has been on the Editorial Boards of both the IEEE Transactions on Parallel and Distributed Systems and the IEEE Transactions on Computers. He was Program Chair/Co-Chair of three major international conferences, General Chair/Co-Chair of six international conferences, and Chair/Co-Chair of five workshops. He is a member of the Eta Kappa Nu electrical engineering honor society, the Sigma Xi science honor society, and the Upsilon Pi Epsilon computing sciences honor society. He has been an international keynote speaker and tutorial lecturer, and has consulted for industry and government. For more information, please see www.engr.colostate.edu/∼hj.

Rose Daley is a member of the Senior Professional Staff at the Johns Hopkins University Applied Physics Lab (JHU/APL). She holds a B.S. in Electrical Engineering from Rensselaer Polytechnic Institute and an M.S. in Computer Science from the John Hopkins University, specializing in Distributing Computing. She has over twenty years experience architecting and implementing software systems, including both distributed large-scale tactical systems encompassing multiple operating systems and communication protocols, and enterprise systems with large databases on internal Intranets. She is the PI for a project for adaptive middleware in large-scale computing environments as well as two internal efforts for developing specialized architecture frameworks. She has led numerous efforts in architecture and modeling, resource management, determining tactical mission sensitivity to network resource attacks and casualties, including development of several tactical and enterprise DoD systems.

I-Jeng Wang is a Principal Professional Staff with the Research and Technology Development Center at the Johns Hopkins University Applied Physics Laboratory. He has a joint appointment with the Johns Hopkins University Computer Science Department as a Research Assistant Professor. He received the M.S. degree from Penn State University in 1991 and the Ph.D. degree from Purdue University in 1996, both in Electrical Engineering. From 1996 to 1997, he was a postdoctoral fellow with the Institute for Systems Research at the University of Maryland, where he conducted research in intelligent control and stochastic approximation. Since October 1997, he has been with JHU/APL where he manages and directs internal research in developing scalable algorithms for solving large-scale DoD problems in areas including resource allocation, wireless

networking and pattern recognition. He is the Co-PI of a project on mission-oriented resource management for a Total Ship Computing Environment funded by the DARPA IXO ARMS program. He was the PI of a project on adaptive information control to develop efficient resource allocation techniques for dynamic QoS provisioning over distributed and disparate networks, funded by the DARPA AICE program. He was a Co-PI of a project on autonomous internetworking funded by the Army Collaborative Technology Alliance, under which he leads a multi-organization team to develop scalable and dynamic routing protocols. His current research interests include stochastic optimization and control, sensor networks, wireless networking, and distributed Bayesian learning and inference. He is an associate editor for the *IEEE Transactions on Automatic Control*.