Consistent Reconfiguration Protocols for Adaptive High-Performance Applications

Carlo Bertolli
Department of Computing
Imperial College London
180 Queens Gate, London, SW7 2AZ, UK
Email: c.bertolli@imperial.ac.uk

Gabriele Mencagli and Marco Vanneschi
Department of Computer Science
University of Pisa
Largo B. Pontecorvo, 3, I-56127, PI, Italy
Email: {mencagli, vannesch}@di.unipi.it

***Abstract*—Programming models for Pervasive Computing applications typically include the possibility of specifying software components according to multiple alternative versions, each optimized for a certain class of computing and communication technologies. A main mechanism provided by these programming models permits to dynamically select one of the alternative versions for the execution. This reconfiguration activity may be critical, from a performance point of view, when considering High-Performance Pervasive Computing applications, especially if the reconfiguration must be performed in such a way that the application semantics is respected (i.e. the reconfiguration is consistent). In this paper we show how to introduce consistent reconfiguration protocols for the ASSISTANT programming model, we exemplify two general protocols and we show experimental results for one of them.**

***Index Terms*—High-Performance Computing, Autonomic Computing, Reconfiguration Protocols, Pervasive and Mobile Computing**

I. Introduction

One of the most important issue for *Pervasive Computing* platforms [1], [2] is represented by the presence of several strongly heterogeneous and dynamic computing and communication resources, such as classical server platforms (e.g. clusters), decentralized and mobile nodes (e.g. smart-phones and Personal Digital Assistants) and their interconnection networks.

Therefore High-Performance pervasive applications are supported with the possibility of defining parallel components in multiple alternative versions, each one optimized for a certain class of computing resources. For instance a complex computing component may be defined in two versions: one for traditional HPC architectures as a cluster of workstations and another one for next-generation smart-phone supporting multicore facilities. In some cases it would be necessary to dynamically switch from one version to another one, e.g. to face with node and link failures but also to optimize the application performance w.r.t. the dynamic availability of platform resources or the user's performance sensation [3], [4]. For instance, we may switch between two versions if the newly activated version features a better behavior than the deactivated one according to the actual networking conditions.

There exist programming models [4], [3] offering the possibility to develop components in multiple versions and to dynamically select one of them according to a reactive adaptation logic: the component monitors the actual *context* situation, e.g. the current state of the underlying executionc-platform, and it possibly changes its employed version, i.e. it locally implements the adaptation strategy in what is called an autonomic behavior. As a clear case, in Odyssey [3] it is possible to optimize the user's sensation on the quality of the computation by dynamically selecting a best version for each application component.

Unlike these contributions, in our research work we are mainly interested in *High-Performance Pervasive Computing* (HPPC) applications, examples of which are Emergency Management, Intelligent Transportation and National Defense. For these scenarios it is of paramount importance that certain QoS parameters are respected during the whole execution. For instance, consider an application providing real-time flood forecasting information to a set of operators near an emergency area: in this case, delivering results later than their actual usefulness may lead to disastrous consequences.

To guarantee that QoS parameters are respected, parallel HPPC applications are supported by proper *adaptivity* and *dependability* mechanisms, which costs, in terms of induced performance overhead, must be known. In this paper we consider the notable case in which a component adaptation strategy decides to *switch* from one version to another one. In some cases this "switching" activity (generally speaking called *reconfiguration*) must be performed in such a way that the overall application semantics is not violated, i.e. it runs proper *consistent* reconfiguration protocols.

Consistent reconfiguration protocols may be obtained by modifying existing fault tolerance mechanisms, i.e. *by treating reconfigurations as failures*. Nevertheless, this would avoid the chance of introducing specific reconfiguration optimizations, as well as to introduce other than fault tolerance protocols, such as rollforward ones. Therefore, fault tolerance and reconfiguration protocols are different but they are built up from the same mechanisms which are the common ones in distributed systems, and they aim at different optimizations.

In this paper we introduce a methodology for deriving consistent reconfiguration protocols which are optimized to support HPPC applications during the so-called *version switching* reconfiguration. Our contribution is shown for the ASSISTANT [5], [6] programming model which represents our research framework under which we present general results. The protocols that can be derived from our methodology are shown to be consistent, i.e. they respect the application semantics and their overhead can be statically analyzed by means of proper performance models. These models can be derived by the properties of the chosen programming model and can be used to define adaptation strategies which also consider the cost of a reconfiguration rather than just the performance of the final component version. In this paper we show experimental results assessing the validity of the described performance analysis.

As a comparison with existing works we consider contributions related to the Grid Computing area. Notable examples are: the CILK programming model [7], which is supported by reconfiguration and fault tolerance mechanisms optimized for its task-parallel programming model; the contributions of [8], based on the concept of *malleable* application, which is able to re-distribute its data units and modify the grain of its tasks. Unlike these contributions we do not focus a specific implementation, but we introduce a methodology to derive reconfiguration protocols for a widespread class of stream-based applications in which compute-intensive elaborations are activated by the reception of a large sequence (i.e. data stream) of independent input tasks.

The outline of this paper is the following: in Section II we introduce the programming model of ASSISTANT. In Section III we present our methodology for deriving consistent reconfiguration protocols and we exemplify two protocols. Finally, in Section IV we show experimental results of one of the protocols and in Section V we give the conclusions.

II. The ASSISTANT Programming Model

ASSISTANT is our research framework for HPPC applications and it is based on *structured parallel programming* [9] to express alternative parallel versions of a same application component. ASSISTANT allows programmers to define parallel and adaptive applications as graphs of interconnected components, each one defined by using a specific programming construct, i.e. the Parallel Module or *ParMod*. Interconnection of ParMods is made by means of *data streams*, i.e. possibly unlimited sequences of typed elements (i.e. a unit of information), which are typically implemented by means of communication channels between the distributed set of processes implementing a ParMod.

The ParMod semantics is characterized by two different logics which interact between themselves (see [10]):

- **Functional Logic** or *Operating Part*: this part encapsulates multiple versions of the parallel module, each one with a different behavior according to several parameters (e.g. memory utilization and expected performance). Only one version at time is allowed to be active;
- **Control Logic** or *Control Part*: this part implements the adaptation strategies by analyzing the current platform and application behavior and by issuing *reconfiguration commands* to the Operating Part.

To express these logics we introduce a new construct, called *operation*, implementing a specific version (i.e. its functional part) and the corresponding adaptation strategy (i.e. its control part) applied when the version is executed. A ParMod includes multiple operations which totally describe its functional and control logics (see a ParMod example in [10]).

For lack of space in this paper we are mostly interested in the interactions between the control and the functional logic, which follow the abstract scheme depicted in Figure 1. The functional logic can be reconfigured in specific *reconfiguration points* implicitly identified by the run-time support system. As an example a typical ParMod reconfiguration point is defined between successive task receiving.

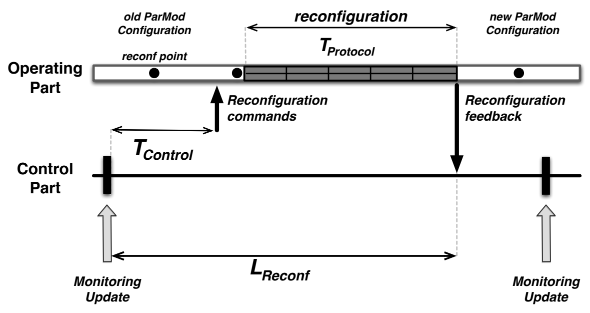

Fig. 1. Interaction scheme between functional and control logics.

When the Control Part receives a *monitoring update* (i.e. updated monitoring data are periodically available from environmental sensors and probing services connected with the ASSISTANT ParMod) it decides the needed reconfigurations by evaluating a specific adaptation algorithm (with an average execution time $T_{Control}$). After that, a set of reconfiguration commands are sent to the Operating Part, which receives and applies them at the following reconfiguration point. For applying the selected reconfigurations the processes implementing the Operating Part execute a consistent reconfiguration protocol, which induces a corresponding overhead during the execution (i.e. with an average completion time $T_{Protocol}$ as shown in Figure 1).

Concerning the Operating Part, in this paper we consider two general structured parallel computations: task-farm and data-parallel (see [10]). In the former structure an input stream of tasks is parallelized on a replicated set of *workers*, where each task is solved sequentially by one of them. In the implementation, task scheduling and result collection are respectively performed by two processes, i.e. the *emitter* and the *collector* . In data parallel programs we partition each received task amongst a set of workers (i.e. each task gives place to a complex state which is partitioned amongst the set of workers) which concurrently participate to its elaboration possibly according to some communication stencils. The state distribution and result collection are respectively performed by *scatter* and *gather* processes. For both structures we denote with the term *parallelism degree* the number of used workers.

III. Consistent Reconfiguration Protocols

Consider the case of an operation (i.e. version) switching from a *source* to a *target* operation. To simplify the discussion we assume that tasks can be independently executed both for task-farm and for data-parallel computations, that they are deterministic, in the sense that a task execution gives always the same result, and that they are idempotent.

Broadly, consistent reconfiguration protocols are applied to the following situation: a ParMod consumes from a set of input streams and produces results to a set of output streams. If we take a computation snapshot, we can see that there are: (i) a set of tasks T_{IN} which are stored in the input streams; (ii) a set of tasks T_P currently in execution on the ParMod; (iii) and a set of task results T_{OUT} previously produced onto the output stream by the ParMod. *In this sense the goal of a consistent reconfiguration protocol is to properly manage the set* T_P.

We formalize the concept of consistency implemented by a reconfiguration protocol. In this section we consider two definitions:

- **Weak Consistency**: all elements received on input streams are processed and their results are delivered to the intended consumers.

Note that this definition does not admit to lose a result but it permits their replication. The second definition forbids also replication:

- **Strong Consistency**: all elements produced onto input streams are processed and their results are delivered to the intended consumers *at most one time*.

We introduce a formalization methodology of consistent reconfiguration protocols which is based on a proper modeling tool enabling us to define protocols in terms of tasks and results.

A. Formalization of Reconfiguration Protocols

We introduce a modeling tool enabling us to uniquely identify stream elements and which requires that such elements may be recovered at any instant of the computation by accessing streams with proper identifiers.

The tool is inspired by the *Incomplete Structure* (or I-Structure) data structure, introduced with other purposes in data-flow programming models [11]. An I-Structure is a possibly unlimited collection of typed elements each labeled with a sequence identifier, or a integer *position*. There are two ways of accessing an I-Structure:

- we can read the element stored in a given position. This operation is denoted with *get(position, element)* and, in case the provided position is empty, it blocks until a value is produced on the position;
- we can write a value to a given position. This operation is denoted with *put(position, element)* and it features the following write-once property: it is not possible to perform a put more than once on each I-Structure position.

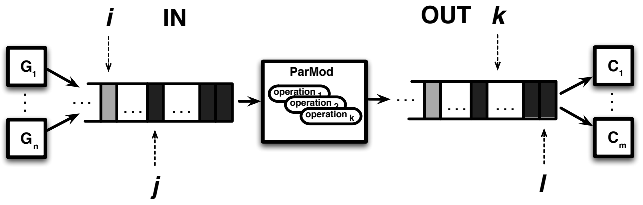

Fig. 2. I-Structure model of ParMod input and output streams, respectively inter-connected to a set of generators $(G_1, \ldots, G_n)$ and clients $(C_1 \ldots, C_m)$. I-Structure black colored elements denote tasks which have been consumed, while elements colored with lighter gray denote tasks produced onto the stream but not yet consumed.

In Figure 2 we show how the I-Structure tool is used: all input streams of a ParMod are mapped onto a single I-Structure denoted with *IN*, and all output streams are mapped onto a further I-Structure denoted with *OUT*. When accessing IN or OUT, the producers, consumers and ParMod processes must arrange the generation of positions in such a way that both the write-once and application semantics are respected. Note that IN and OUT are not necessarily ordered between them.

In this model we can identify, for each I-Structure, two indexes: that of the last consumed element (e.g. j, l in Figure 2); that of the last produced element (e.g. i, k). By using these indexes we can precisely characterize the task sets described above: for instance T_P can be defined as the set of elements with indexes on the input stream from 0 to j, to which we have to subtract all elements whose results have been produced to the output stream, i.e. result elements with indexes on the output stream from 0 to k. This index-based modeling is especially useful when proving the correctness of consistent reconfiguration protocols (which are not presented here for space reasons).

B. Implementation

We show reconfiguration protocols for a distributed implementation of ASSISTANT (see [10]) based on networks of distributed processes and communication channels, in which streams are implemented as channels and each element is mapped onto a message. At this level the I-Structure abstract model requires that stream elements (or messages) can be recovered at any time during the computation. Typically, in a channel implementation, when a message is received (i.e. extracted) from the message queue its content can be overwritten by successive messages. Therefore we can guarantee recovery of messages in two ways: (1) by supporting communication channels with message logging techniques; (2) by requiring all application components to re-generate elements under request. The protocols described in this paper refer to the latter solution. Nevertheless, they may be applied also to the former case, by changing the way in which message re-generation is obtained.

The implementation of the I-Structure model for streams is shown in Figure 3. All input and output streams are respectively mapped onto the same CH_{IN} and CH_{OUT} channels. For

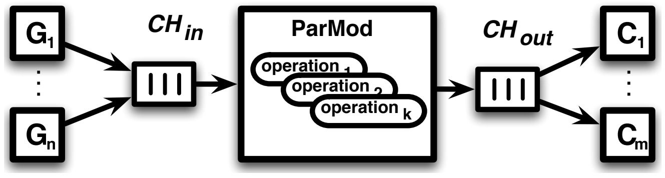

Fig. 3. Implementation of the I-Structure model: input and output streams are respectively implemented with two asymmetric communication channels.

the purpose of re-generation, messages are labeled with their input stream sequence identifier, i.e. when performing a put (or a send to CH$_{IN}$) the sent message includes the element itself along with the input stream sequence identifier. The input stream identifier is also preserved when the corresponding task result is produced onto the output stream.

Finally, we define the notion of *Vector Clock* (VC) to map element positions from different I-Structures. For re-generation purposes we are required to map output stream identifiers onto corresponding input stream identifiers (i.e. result identifiers onto related input task identifiers). A generic VC includes a set of pairs of the form $(1, k_1); (2, k_2); \ldots, (N, k_N)$. The first element of each pair is the sequence identifier of result on the output stream (from 1 to N); these are related to the corresponding sequence identifier of input task generating the results (from k_1 to k_N).

In a VC we can identify the *maximum contiguous sequence identifier* MC, in the set $k_1, \ldots, k_N$, as the maximum integer included in the input stream identifiers for which all its predecessors are included in the set $k_1, \ldots, k_N$. For instance, if we have the following VC $(1, 12); (2, 3); (3, 15); (4, 1); (5, 20); (6, 2);$, MC $= 3$, as the mapped identifiers include all numbers from 1 to 3, and the successive included number is 12.

C. Consistent Operation Switching Protocols

There are at least two ways of implementing a consistent operation switching protocol for an ASSISTANT ParMod:

- we can wait for the source operation to perform all tasks in T_P and then activate the target operation from the first task which has not been consumed by the source operation. When the source operation is notified of the reconfiguration, it first needs to stop to receive input tasks from IN. Next, the target operation must know the position of the last value consumed by the source operation. We denote these kinds of protocols as *rollforward*;
- when notified the source operation can immediately stop its execution. The target operation has to re-obtain the tasks in T_P and re-start their execution. We denote these kinds of protocols as *general rollback*, because they can be supported independently of the actual parallel computations performed by the source and the target operations. The design of these kinds of protocols requires the target operation to characterize the T_P set, possibly sharing information with the source operation. A solution consists in analyzing the IN, OUT streams and a VC relating them.

In this section we describe the rollforward and generic rollback protocol. In the description we assume that the target operation has been previously deployed on the target platform and it is ready to start its execution.

a) Rollforward Protocol: a rollforward protocol is based on “flushing out” all tasks in T_P in the source operation and then starting the execution of the target operation from the last un-consumed element from the input stream. To do so, we can simply connect both operations to the input and output channels CH$_{IN}$ and CH$_{OUT}$ and implement the following sequence of actions after ParMod Operating Part has been notified of the operation switching command:

- the processes on the source operation go on with their computation, except that they stop receiving tasks from the input stream;
- when the all tasks are executed and their corresponding results are delivered, they signal to the Control Part their termination;
- the Control Part notifies the target operation to start its execution;
- these processes simply start to receive input tasks from CH$_{IN}$, whose message queue contains the last un-received tasks.

Note that this protocol implements the *strong consistency* definition.

The application of this protocol depends, from a performance viewpoint, on the time available to perform the operation switching and the time needed to execute the protocol $T_{Protocol}$. If we suppose that all the communication channels have been implemented with an asynchrony degree equal to one, upper bounds to the time spent on executing the rollforward protocol can be estimated as:

- if the source operation is a task-farm, at the worst case we have to consider the parallel execution of all tasks possibly executed on N workers (each costs T_W) plus the time needed to schedule T_E, execute T_W, collect and deliver T_C the last task received by the emitter. The time needed to flush out this last task ($T_E + T_W + T_C$) can be partially overlapped to the time needed to collect all results of previous tasks currently executed on the workers, i.e. at the worst case $N \cdot T_C$. Therefore we can define:

$$T^{farm}_{rollfwd} \leq \begin{cases} 2T_W + T_E + T_C & \text{if } T_E + T_W \geq N \cdot T_C \\ T_W + N \cdot T_C & \text{otherwise} \end{cases}$$

- if the source operation implements a data-parallel, we have to consider the termination of the task currently in execution (in T_{Ws} time) possibly plus the time needed to scatter T_S, execute T_{Ws} and gather T_G a task (i.e. the one remaining on the scatter process). Therefore:

$$T^{dp}_{rollfwd} \leq \begin{cases} 2 \cdot T_{Ws} + T_S + T_G & \text{if } T_S + T_{Ws} \geq T_G \\ T_{Ws} + 2 \cdot T_G & \text{otherwise} \end{cases}$$

b) Generic Rollback Protocol: in a rollback protocol we stop the execution of the source operation and we immediately switch to execute the target one. To do so, the Control Part sequentially signals the processes in the Operating Part of both operations (i.e. it first stops the source operation processes and then it signals the start to the target operation processes).

The target operation needs to obtain all tasks in T_P from the related generators. To do so, it needs to access the Vector Clock mapping output onto input sequence identifiers, which can be provided directly from the source operation. We can characterize two types of information passed from the source to the target operation:

- the *MC* value: in this case the target operation will request generators to re-generate elements with sequence identifiers greater than *MC*. Note that some elements may have been previously executed and their results sent to the output stream, hence their execution will be duplicated. Therefore this protocol implements the *weak consistency* definition;
- the entire VC: in this case the target operation guides the re-generation of elements in T_P by requesting to generators only those elements corresponding to the missing sequence identifiers, which are found by scanning from *MC* to the maximum sequence identifier of submitted results in the VC. Note that this protocol avoids result duplication, hence it implements the *strong consistency* definition.

In both cases, the information can be provided by the source operation, or can be obtained by channel analysis and cooperation between the target operation, the generators, and the clients.

As in the rollforward protocol the target operation, after recovering and re-executing all tasks in T_P, can simply restart to receive messages from the input channel. To avoid replication of input tasks we can, for instance, clear the input channel CH_{IN} or delete their sequence identifiers from the VC before passing it to generators.

The choice of applying this kind of protocol depends on the amount of work which we can lose. We can quantify this work depending on the parallel computation performed by the source operation:

- if the source operation is a task-farm, we have to re-execute at most $N + 2$ tasks: N for the workers and 2 for emitter and collector. In addition, in the first version of the protocol we have to sum up also all tasks whose results have been delivered in an un-ordered way to the output stream;
- if the source operation is a data-parallel, we have to re-execute at most three tasks: one on the scatter, one currently executed by workers, and one on the gather. No more tasks have to be re-executed because the data-parallel guarantees ordering on input and output streams.

c) Protocol Comparison: The choice of the best protocol depends on the characteristics of the involved source and target operations (e.g. their performance), and the specific reconfiguration case. For instance, the rollforward protocol can be used if we can guarantee that the source operation will be running and available for the whole reconfiguration time. If this is not the case, then we have to run the rollback protocol which minimizes or nullify the participation of the source operation. From a performance viewpoint we select the rollback protocol if the target operation has a higher performance w.r.t. the source one, or the rollforward one otherwise.

IV. Experiments

We have performed experiments to evaluate the cost of the described rollforward protocol to assess the performance models described in the previous section. The target application is related to a flood emergency management situation described in [10], [5], [12], and it practically consists in a ParMod solving tri-diagonal systems of linear equations. For this component two distinct parallelization schemes have been implemented by using the MPI (*Message Passing Interface*) communication library:

- a task-farm structure, in which an emitter process is responsible for scheduling each input task (i.e. in this case a data-structure representing a tri-diagonal system) to a set of replicated parallel executors. Each input task is scheduled to an available worker according to a fair and load-balanced on-demand distribution (each worker notifies the emitter whenever it is available for receiving a new task). Results collection is performed by a further collector process;
- a data-parallel structure, in which a scatter process performs for each received system a partitioning of the input data-structure among a set of worker processes that compute the sequential elaboration on their own partition. In this algorithm data dependencies imply a necessary communication phase between workers that need to exchange local data at each step of the computation. This interaction is also called *communication stencil*. Finally a gather process collects the workers results filling the output data-structure that will be transmitted to out-going application modules.

The experiments are related to the situation in which we are switching from a task-farm operation executed on a distributed-memory architecture (a cluster of workstations in this case), to a data-parallel operation mapped onto a shared-memory multicore platform. The cluster includes 30 nodes Pentium III 800 MHz with 512 KB of cache, 1 GB of main memory and interconnected with a 100 Mbit/s Fast Ethernet; the multicore is a Intel Xeon E5420 Dual Quad Core multicore processor, featuring 8 cores of 2.50 GHz, 12 MB L2 Cache and 8 GB of main memory.

In the experiments we monitor the time spent on applying the reconfiguration protocol $T_{Protocol}$, which is evaluated as the time passing from the notification from the Control to the Operating Part of the operation switching to the instant in which the target operation starts delivering results to the client.

As the single task execution time on the source operation influences the rollforward cost, we show the experimented times for different task grains, which in this application corresponds to the size of resolved systems: for 8MB $T_W = 4.0436$ sec.; for 16MB $T_W = 8.4194$ sec.; for 32MB $T_W = 17.4813$ sec.

The results presented in Figure 4 show the L_{reconf} behavior w.r.t. the task grain and the task-farm parallelism degree of the source operation. By comparing T_W values we can see that

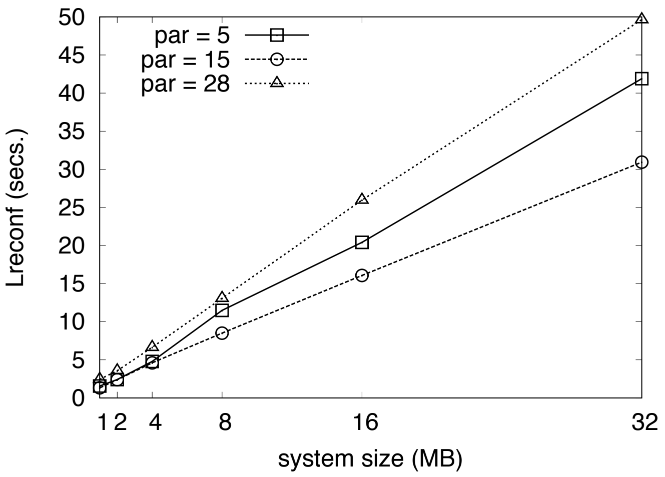

Fig. 4. Switching overhead (L_{reconf}) of the rollforward protocol by varying the task grain (i.e. solved system size).

L_{reconf} can be broadly approximated to the double of T_W, especially when the optimal parallelism degree (see below) is used (in this application T_E and T_C are negligible).

Figure 5 shows the L_{reconf} behavior for a wider set of task grains by varying the parallelism degree p. As we can note, the case of $p = 15$ is the optimal one w.r.t. the cases $p = 5$ and $p = 32$, as it best fits the optimal value for the task-farm parallelism degree.

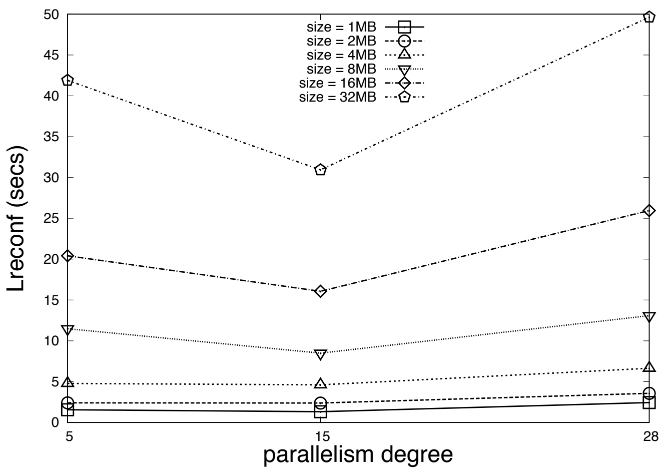

Fig. 5. Switching overhead (L_{reconf}) of the rollforward protocol by varying the parallelism degree.

V. Conclusion

In this paper we have shown how consistent reconfiguration protocols can be supported for ASSISTANT applications during operation (i.e. version) switching. We have introduced a methodology based on the abstract I-Structure model and we have exemplified two protocols, respectively based on a "graceful" switching rollforward mechanism and a fast rollback strategy. We have described the experimental results for the rollforward protocol, showing how the performance behavior of the parallel computation performed by the source ParMod operation influences the overhead induced by the protocol.

In our next research works we will provide experimental results also for the generic rollback protocol. Especially for such class of reconfiguration protocols optimizations are certainly possible, as the possibility to start the computation of the T_P task set from partially computed results instead from the beginning. The applicability of such optimizations, which certainly depend on the specific pair of source and target operations, will be investigated in our future works.

References

[1] V. Hingne, A. Joshi, T. Finin, H. Kargupta, and E. Houstis, "Towards a pervasive grid," in *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 207.2.

[2] T. Priol and M. Vanneschi, *Towards Next Generation Grids: Proceedings of the CoreGRID Symposium 2007*. Springer Publishing Company, Incorporated, 2007.

[3] B. D. Noble and M. Satyanarayanan, "Experience with adaptive mobile applications in odyssey," *Mob. Netw. Appl.*, vol. 4, no. 4, pp. 245–254, 1999.

[4] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, "Project aura: Toward distraction-free pervasive computing," *IEEE Perv. Comp.*, vol. 1, no. 2, pp. 22–31, 2002.

[5] C. Bertolli, D. Buono, S. Lametti, G. Mencagli, M. Meneghin, A. Pascucci, and M. Vanneschi, "A programming model for high-performance adaptive applications on pervasive mobile grids," in *Proceeding of the 21st IASTED International Conference on Parallel and Distributed Computing and Systems*, November 2009, pp. 38–54.

[6] C. Bertolli, G. Mencagli, and M. Vanneschi, "A cost model for autonomic reconfigurations in high-performance pervasive applications," in *Proceedings of the 4th ACM International Workshop on Context-Awareness for Self-Managing Systems*, ser. CASEMANS '10. New York, NY, USA: ACM, 2010, pp. 3:20–3:29. [Online]. Available: http://doi.acm.org/10.1145/1858367.1858370

[7] R. D. Blumofe and P. A. Lisiecki, "Adaptive and reliable parallel computing on networks of workstations," in *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1997, pp. 10–10.

[8] T. Desell, K. E. Maghraoui, and C. A. Varela, "Malleable applications for scalable high performance computing," *Cluster Computing*, vol. 10, no. 3, pp. 323–337, 2007.

[9] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Comput.*, vol. 30, no. 3, pp. 389–406, 2004.

[10] C. Bertolli, D. Buono, G. Mencagli, and M. Vanneschi, "Expressing adaptivity and context-awareness in the assistant programming model," in *Procs. of the Third Intl. Conf. on Autonomic Comp. and Comm. Syst.*, vol. 23, September 2009, pp. 32–47.

[11] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: data structures for parallel computing," *ACM Trans. on Progr. Lang. and Syst.*, vol. 11, no. 4, pp. 598–632, 1989.

[12] C. Bertolli, G. Mencagli, and M. Vanneschi, "Adaptivity in risk and emergency management applications on pervasive grids," in *ISPAN '09: Proceedings of the 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 550–555.