Crash Management for distributed parallel systems

Jan Haase   Frank Eschmann

{haase|eschmann}@ti.informatik.uni-frankfurt.de
J. W. Goethe-University, Technical Computer Sc. Dep., Box 11 19 32, D-60054 Frankfurt, Germany*

Abstract: With the growing complexity of parallel architectures, the probability of system failures grows, too. One approach to cope with this problem is the self-healing, one of the *organic computing*'s self-x features. Self-healing in this context means that computer clusters should detect and handle failures automatically. This paper presents a self-healing mechanism based on checkpointing, so that a cluster remains operative even if some sites or the connections between them fail. The proposed method has been implemented and tested on the Self Distributing Virtual Machine (SDVM).

1 Introduction

With the rapidly growing capability of computer architectures their complexity grows as well. More and more parallelism is necessary to provide the needed computing power. Moreover, systems must adapt to changing environments and cope with a breakdown of components. One approach is to incorporate organic features into computer systems. *Organic computers* [VD03] are characterized by self-x properties like self-configuring, self-optimizing, self-healing, and self-protecting.

To make a cluster computer behave "organic", it should possess (among other things) some kind of self-healing feature, which detects and deactivates defective components. The deactivation alone will usually not suffice, though, as in computer clusters data is often stored in a decentralized way. If a node fails, all data stored in its local memory is lost. Therefore concepts have to be developed to store data redundantly and to recover the data in case of a failure. One such concept is described in the following for the *Self Distributing Virtual Machine* (SDVM) [HEKW04].

2 The Self Distributing Virtual Machine (SDVM)

The SDVM is implemented as a daemon that provides a convenient platform for programs to be run on computer clusters. Seen from the architectural point of view the SDVM is based on the concept of *Cache Only Memory Architectures* (COMAs) [HLH92]. COMAs have a distributed shared memory (DSM) in which data migrates automatically to the sites where it is needed. As the memories of a COMA behave like caches, attracting needed data, they are called *attraction memories*. SDAARC (*Self Distributing Associative ARChitecture*) [EKMW02] extends this concept by introducing a code migration mechanism and providing the basic principles for data *and* code migration. The SDVM can be seen as an implementation of SDAARC on computer clusters.
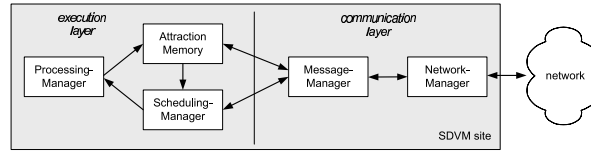

Figure 1: Execution layer and communication layer of one participating site of the SDVM

In this concept program instructions are represented by *microthreads*. A microthread contains a code fragment (block to thread grain), which is executed dataflow oriented—if all parameters are present for a microthread it can be executed. A microthread's parameters are kept in a *microframe* which is stored in the attraction memory. Besides the parameters a microframe holds a pointer to the corresponding microthread and addresses to microframes where the results of the microthread have to be applied to so they can eventually be executed. As soon as a microframe has received all its parameters, it becomes executable, meaning that the corresponding microthread can be executed with the parameters taken from the microframe. Apart from microframes global data objects are stored in the attraction memory as well. Global data in terms of COMA is spread over the sites participating in the SDVM cluster.

Prior to the run time the control flow, the data flow between the microthreads, and the earliest time when a microframe has to be allocated (created) is known. It can be visualized as a CDAG (Controlflow Dataflow Allocation Graph) [KEMW02].

Software architecture

The SDVM is structured in two layers: an execution and a communication layer. Both layers consist of several modules (managers), which communicate using method calls (see Figure 1).

The execution layer represents the core needed to execute SDVM programs on a single machine. It consists of the processing manager, the attraction memory, and the scheduling manager. The communication layer contains all modules which are needed to communicate with other sites. For a detailed introduction of the SDVM's software architecture see [HEKW04].

3   Crashes

One of the mentioned self-x features is self-healing. A system should be able to **notice an injury** and **act on it** to restore its operativeness. In the field of cluster computing, the "injury" is identified usually as a failure of a participating machine (its hardware or the software running on it) or a malfunctioning communication connection. As it cannot be expected from today's computer systems to repair themselves physically, the remaining cluster should try to find a workaround to continue work without the faulty part.

Identifying crashes

The three possible problems (defective hardware, crashed software and broken connection) can be treated essentially as the same damage: A site cannot be reached anymore. Of course, a broken connection can be circumvented by rerouting, so that other (redundant)

connections are used. But if the rerouting fails to re-enable the machines to communicate, it makes no difference whether a site or the connection to it is failing.

However, it is an undecidable problem to distinguish between a crashed site and a very slow (or extremely busy) site. Therefore a limited period of time has to be defined in which the site must answer, or else it will be treated as crashed.

The SDVM identifies crashes only when messages or requests passed from one site to another can't be delivered because the target site doesn't reply.

Crash management

If a site crashes, all data which was residing on this site is lost irrevocably. Therefore, this data has to be copied in advance, so that a redundant copy can be fetched and used to continue work.

An obvious but bad solution for this problem is to simply duplicate all data ever created in the SDVM, so that in the case of a crash only some data is lost which exists twice anyway. However, to accomplish this, a copy of all data must be sent immediately over the network to avoid a crashing site holding both duplicates, and therefore much communication overhead can be expected. Another approach may be to keep all applied data in memory until all microframes using these parameters are executed and have on their part applied new data to other microframes, so in case of a data loss the missing executions could be done again. This would require a powerful (and expensive) mechanism to identify data which can be deleted, though.

Another solution is to write down all data at specific points in time, making a *checkpoint*, known i.a. from the subject of databases (see e. g. [Ul82]). To simplify matters, the following description assumes running only one program on the SDVM-cluster at a time—the mechanism works for several programs running concurrently on the same or different sites, too, though.

Part 1 – Checkpointing: The checkpointing mechanism introduces a new kind of microframe into the CDAG, the *checkpoint frame*. The CDAG is cut at any desired position (though preferably cutting only few edges). Information about every already allocated microframe is sent to the checkpoint frame, and every edge crossing the cut is duplicated, the duplicate being a parameter to the checkpoint frame (see Figure 2(a)). As the original edges are not altered, the program will run on undisturbed.

When eventually the last data on crossing edges is applied and therefore the checkpoint frame has got all its parameters, it becomes executable like any microframe. When executed, it writes all data about the checkpoint to a disk file on the machine executing it and terminates.

Several cuts are supported, each of them resulting in a checkpoint written to a file. The cuts may even cross each other. However, to keep the size of the checkpoint frame and file small, e. g. Ford and Fulkerson's Minimal Cut algorithm [FF62] can be used to determine good positions for cuts.

Part 2 – Recovery: When all execution of the crashed program has been stopped, all sites are asked about their last *complete* checkpoint of the concerned program. When the most recent (and accessible) one is identified, the site holding it will create another sort of microframe, a *recover-frame* (see Figure 2(b)). The restart-counter of the program is

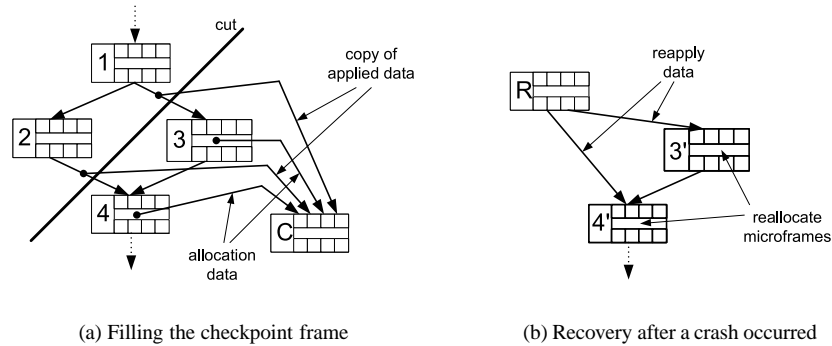(a) Filling the checkpoint frame   (b) Recovery after a crash occurred

Figure 2: The checkpointing mechanism

incremented so that some data or microframes which are discovered later to belong to the crashed program run can be identified and deleted. This can occur if e.g. a slow connection was busy sending a belonging microframe just while both its sender and target site were deleting the concerned data. Furthermore, this way an erroneous duplication of the program in case that a crash was detected wrongly is avoided. However, if in a network topology with two subclusters having only one connection between them and just this connection breaks, both parts will recognize a crash and restart the program, resulting in a double re-execution of the program on two now separated subclusters.

The recover-frame does not need any parameters to be executable and thus is executed immediately. It reads the locally stored checkpoint data and reallocates the microframes contained in the checkpoint data (and thus recreates the nodes of the CDAG). Then it reapplies all parameters stored in the checkpoint data (recreating the edges of the CDAG) and terminates. After that, the scheduling manager will again hold some executable microframes of the restarted program, which can be run normally. As the other sites are idle, they will call for work in the cluster again, and get some of those executable microframes (work stealing principle [BL94]). In this manner the restarted program will soon be spread over the cluster again, running normally.

As the newly recreated microframes and global data objects will probably not be placed at the exact same position in the global memory (see Figure 3), all addresses pointing to the old positions will have to be recalculated and corrected in the recreation process. The information needed for this is stored in the checkpoint as well, so it is merely a matter of calculation.

Another problem which may occur is a second crash happening while the recovery process is not yet finished. This will affect the program being restarted only if the site actually doing the recovery crashes. In this case the crash of this site will have to be recognized and then the next remaining most recent checkpoint will have to be found, continued by a normal recovery process as mentioned above.

Moreover, if a node which was identified as crashed recovers (e.g. if the network connection to it was temporarily cut off and then reconnected) the crash management will not stop. The restart from the last complete checkpoint will be carried out nevertheless.

659

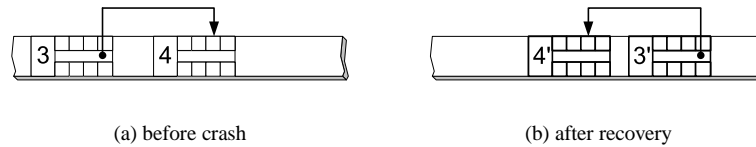(a) before crash   (b) after recovery

Figure 3: Example: The memory position of microframes 3 and 4 may vary after recovery and therefore the apply address in microframe 3' will have to be recalculated to point to the correct position.

First results

The checkpointing mechanism obviously produces overhead, especially since all but the most recent of the checkpoints made are probably never needed again. First test runs show that short test programs with three checkpoints were less than one percent slower than without checkpoints. This is because the checkpointing itself is automatically run in parallel as well as the program itself.

4   Conclusion and future Prospects

A mechanism to cope with possible crashes in computer clusters (making them in a way self-healing) was proposed. The checkpointing principle seems to fulfill the requirements and doesn't produce too much overhead, as shown for the SDVM, so it can be used for parallel applications.

At the moment, reasonable cut positions have to be found by the programmer. Therefore it would be helpful to let this be done by a compiler, ideally when splitting the program into microthreads and therefore creating the CDAG anyway.

References

[BL94]   Blumofe, R. and Leiserson, C.: Scheduling multithreaded computations by work steal-ing. In: *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico*. S. 356–368. November 1994.

[EKMW02] Eschmann, F., Klauer, B., Moore, R., and Waldschmidt, K.: SDAARC: An Extended Cache-Only Memory Architecture. *IEEE micro*. 22(3):62–70. 05/06 2002.

[FF62]   Ford, L. and Fulkerson, D.: *Flows in Networks*. Princeton University Press. 1962.

[HEKW04] Haase, J., Eschmann, F., Klauer, B., and Waldschmidt, K.: The SDVM: A Self Dis-tributing Virtual Machine. In: *Organic and Pervasive Computing – ARCS 2004: International Conference on Architecture of Computing Systems*. volume 2981 of *Lecture Notes in Computer Science*. Heidelberg. 2004. Springer Verlag.

[HLH92]   Hagersten, E., Landin, A., and Haridi, S.: DDM — A Cache-Only Memory Architec-ture. *IEEE Computer*. 25(9):44–54. 1992.

[KEMW02] Klauer, B., Eschmann, F., Moore, R., and Waldschmidt, K.: The CDAG: A Data Struc-ture for Automatic Parallelization for a Multithreaded Architecture. In: *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP 2002)*. Canary Islands, Spain. January 2002. IEEE.

[Ul82]   Ullman, J.: *Principles of Database Systems*. Computer Science Press. 1982.

[VD03]   VDE/ITG/GI-Arbeitsgruppe Organic Computing: Organic Computing, Computer- und Systemarchitektur im Jahr 2010. Technical report. VDE/ITG/GI. 2003.