The Self Distributing Virtual Machine (SDVM): Making Computer Clusters Adaptive

Jan Haase, Andreas Hofmann, and Klaus Waldschmidt

Technische Informatik
J. W. Goethe Universität
Post Box 11 19 32, 60054 Frankfurt a. M., Germany
{haase|ahofmann|waldsch}@ti.informatik.uni-frankfurt.de[**]

Abstract. The Self Distributing Virtual Machine (SDVM) is a middleware concept to form a parallel computing machine consisting of a any set of processing units, such as functional units in a processor or FPGA, processing units in a multiprocessor chip, or computers in a computer cluster. Its structure and functionality is biologically inspired aiming towards forming a combined workforce of independent units ("sites"), each acting on the same set of simple rules.
The SDVM supports growing and shrinking the cluster at runtime as well as heterogeneous clusters. It uses the work-stealing principle to dynamically distribute the workload among all sites. The SDVM's energy management targets the health of all sites by adjusting their power states according to workload and temperature. Dynamic reassignment of the current workload facilitates a new energy policy which focuses on increasing the reliability of each site.
This paper presents the structure and the functionality of the SDVM.

1 Introduction

In the past, the user's increasing demand for capacity and speed was usually satisfied by faster single processors. Nowadays the increase in clock rates seems to have slowed down. The exploitation of parallelism is one way to enhance performance in spite of stagnating clock speeds. Its use isn't limited to the field of supercomputers; nowadays even Systems-on-Chip(SoC) with a lot of processors, so called MPSoCs, are in production.

Task scheduling and data migration for parallel computers, especially if embodied as a cluster of processing units, are complex problems if solved centralized. The use of biologically-inspired mechanisms can reduce complexity without sacrificing performance. The properties of biological systems like self-organization, self-optimization and self-configuration can be used to ease programming and administration of parallel computing clusters. These properties can be implemented efficiently using a paradigm common in complex biological systems: the collaboration of autonomous agents.

[**] Parts of this work have been supported by the Deutsche Forschungsgemeinschaft (DFG).

Using biologically inspired techniques to implement a parallel computing system is only the means to the end in meeting user requirements. With the introduction of parallel computing, speed is not the only property which users are interested in; others too have come to the fore. In the following, several of those properties are presented. These properties focus on MIMD computer clusters. Such a cluster consists of an arbitrary number of independent processing units called *sites* which are connected using any kind of network.

Despite the performance of parallel computers the computations may take serveral days to finish. For large scale machines like the ASCI-Q machine, the mean time between failures (MTBF) for the whole system is estimated to be mere hours [1]. Thus *system stability* even in the face of failure of single components is an important goal. Parallel systems must therefore detect failures and intercept them transparently and unnoticed by the user. Presently, a system won't be able to repair itself physically, but the other sites should adapt to the changed environment and take over the work from the faulty site. This could be termed "self healing" of a system.

A main cause for the limited use of parallel computers lies in the challenging programmability: For single processors, scheduling in time is sufficient, but for multiprocessor systems, the spatial dimension has to be considered, too. Spatially and timely scheduling of the chunks of a program is a non-trivial optimization problem for the programmer, especially as the parallelism of an application can vary greatly over execution time and depends on the input data. Therefore a possible solution would be to relieve the programmer of the spatial scheduling at all, and let the system decide it at runtime using convenient heuristics automatically. The resulting *transparent parallelization* is similar to the goal of self-optimization, known from the subject of organic computing [2].

Experience shows that the performance demands increase over time. To be cost-effective, it suggests itself to prolong the life-span of a system instead of replacing it with a new system every few years. In the case of a parallel system this can be done by adding new processors or computers to increase its processing power. A parallel computing middleware should therefore support *scalability*. The benefit even increases if the growing and shrinking of the system is possible at runtime to cope with short-time processing power demand peeks.

In the beginning parallel systems were implemented as dedicated clusters. These days they more and more consist of clusters of workstations, multiprocessor embedded systems, or even multicore FPGA-based devices. Thus environmental parameters change frequently and sometimes fast. Configuration by hand of such a dynamically changing system is hard or even impossible. Thus it should *configure itself autonomously*. Concerning parallel systems, for well-founded configuration decisions the sites must be informed about the other sites' load, speed, etc., automatically. This can be denominated as the goal of self-configuration.

In section 2, the concept of the SDVM and its underlying mechanisms are described. After a list of some speedup results in section 3, this paper closes with a conclusion in section 4. The SDVM prototype is implemented in C++ and its complete source code is freely downloadable [3].

2 The SDVM

The Self Distributing Virtual Machine (SDVM) is a middleware to form an adaptive parallel system which is applicable to different granularities like functional units on an FPGA, processors in a multiprocessor SoC, or a cluster of customary computers(see Figure 1). The SDVM is currently implemented as a prototype in software running as Linux daemons on a workstation cluster.

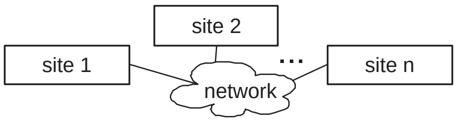
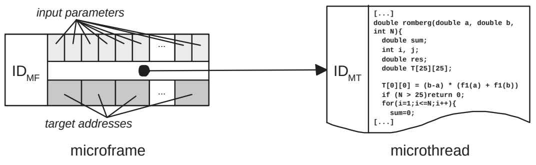


Fig. 1. The SDVM connects processing units (sites) to form a cluster, regardless of the topology of the connection network.

The SDVM actually implements several of the concepts inspired by biological systems, namely the cooperation of somewhat autonomous systems, self-controlled adaptivity to changing environments (as the size of the cluster or its heterogeneity) and decentralization of task scheduling. The sites that build the cluster are basically equal with no master or fixed division of functions. Furthermore, the SDVM supports self-healing by the use of checkpoints, to ensure proper program execution irrespective of failing cluster members.

2.1 The concept

The SDVM can be seen as a dataflow machine augmented with a distributed shared memory: An application to be executed by the SDVM is cut into several chunks of code, the *microthreads.* Each microthread needs certain parameters when run, therefore these parameters have to be collected prior to execution of the microthread. The data container collecting the parameters is called the *microframe* (see Figure 2).



```
[...]
double romberg(double a, double b,
int N){
  double sum;
  int i, j;
  double res;
  double T[25][25];

  T[0][0] = (b-a) * (f1(a) + f1(b))
  if (N > 25)return 0;
  for(i=1;i<=N;i++){
    sum=0;
[...]
```

Fig. 2. Microframe and Microthread

A microframe is filled over time with the parameters it awaits. When all parameters have been received, the corresponding microthread is executed using these parameters and in the process calculates results needed by other microframes as parameters. Microframes can travel throughout the cluster while being filled. As the corresponding microthread is only needed when they are actually executed, the microthread is not included in the microframe to lessen bandwidth consumption when moving from one site to another.

While a microframe is being filled, the SDVM has not yet decided which site will execute this microframe with its corresponding microthread. When a site autonomously decides to execute a microframe locally, it finally needs the corresponding microthread which is then read from the local code cache or copied over the network. In this way the application itself (in terms of its microthreads) spreads automatically throughout the cluster over time—the sites will request just what they need and when they need it.

Microframes are not the only way to exchange data between parts of a program. The entirety of the SDVM provides a distributed shared memory (DSM) like SCI [4] and FLASH [5]. SDVM-programs can allocate and use this memory just like heap memory is used in C/C++. The memory addresses pointing to allocated memory regions can be passed as microframe parameters between microthreads. This global memory consists of the sum of all sites' memories. If a site is shut down (shrinking the cluster) the data stored in its local part of the global memory is pushed out to other sites before.

Any site which has nothing to do will ask other sites for work and will in return get a microframe which is ready for execution, if available. Any new site joining the cluster will just notice that its work queue is empty and act like any site which is out of work. In this way a site autonomously provides itself with work. This is called the *work stealing principle* (also referred as "receiver-initiated load balancing"), as opposed to the *work sharing principle* ("sender-initiated load balancing") where overloaded sites try to push away work to less loaded sites. Nearly all load balancing mechanisms base on work sharing, work stealing, or a combination of both [6]. On heavy loaded clusters work sharing leads to an even higher burden due to unsuccessful load balancing attempts.

As the SDVM provides a way of virtualization, it can connect heterogeneous machines to form a cluster: Several underlying architectures, platform types and operating systems are supported. If a site wants to execute a microthread which doesn't exist in its needed binary format yet, it must be generated somehow. If the SDVM is used as a middleware for computer clusters, it will request the source code and compile it on-the-fly and at runtime using the locally installed compiler (like `gcc`). The results show that the compilation time is fast enough, because the microthreads are small chunks of code and don't have to be linked (this is done automatically by the SDVM when receiving a microthread anyway). When the SDVM is used as a firmware for MPSoCs, techniques like code morphing can be used to translate the binary of the microthreads.

As a middleware the SDVM connects several machines. In contrast to client/server concepts like CORBA [7], the machines are treated equally, though. The SDVM cluster consists of the entirety of all sites, which are SDVM daemons running on participating machines. The number of sites, their computing power, and the network topology between them is irrelevant, as the SDVM automatically adapts to any cluster it is run on, even when the cluster grows or shrinks at runtime by adding or removing sites [8].

The SDVM daemon consists of several managers with different fields of responsibility. Some deal with the execution of code fragments, some attend to communications with other sites, some are concerned with the actual decision-making (see Figure 3). The latter implement the self-x features of the SDVM. They are described in the next sections.

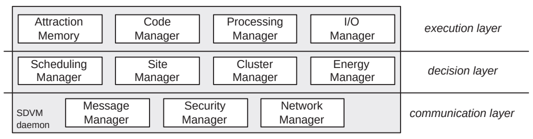


Fig. 3. An SDVM daemon consists of several managers.

2.2 The execution layer

The execution layer is responsible for the handling and execution of the code and data. Furthermore it provides I/O virtualization.

Microframes waiting for more parameters as well as global memory objects are kept in the *attraction memory.* If a data object is requested, it is first sought locally. In case of a miss the site it actually resides on is determined and then the data object is moved or copied to the local site.

The microthreads are only requested when they are to be executed locally. The local caching of microthreads and the compilation of microthreads, if needed, is done by the *code manager.*

The *processing manager* executes the microthread/microframe pair. To accomplish this, it provides an interface for the microthread to read the parameters of its microframe. When the execution has finished the processing manager deletes the no longer needed microframe. To hide network latencies when e.g. an access to a remote part of the global memory is needed, the processing manager may execute several microthread/microframe pairs concurrently. Test runs suggest that a number of 5 parallel processing manager threads are a good value for applications having much communication between the microframes.

The *input/output manager* manages user interaction and accesses local resources like hard disks or printers.

2.3 The communication layer

The communication layer manages sending and receiving of messages between sites. The *message manager* is the central communication hub for all other managers. It generates serialized data packets to be sent to other sites, adds information about the local site and determines its address before optionally passing them to the *security manager.* This manager may then encrypt and sign the data packets to avoid e.g. eavesdropping and spoofing. On the receiving site it will validate the signature and decrypt the message, if necessary, before passing it to the message manager.

The *network manager* is the part of the SDVM which is responsible for the actual transportation of the data packets. For the currently existing cluster realization it uses TCP/IP to send data to other sites. For an implementation of the SDVM on SoCs or multiprocessor chips it would have to use the on-chip network to pass data to the receiving site.

2.4 The decision layer

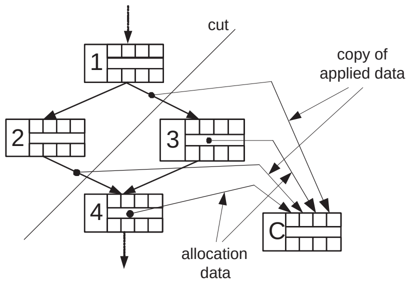
While the responsibilities of the managers in the execution and communication layers are more or less usual in computer systems, the decision layer implements the more sophisticated parts and the self-x-properties of the SDVM.

The SDVM features distributed scheduling which is done by the *scheduling manager*. Most scheduling methods assume a central calculation of the execution order, combined with a centrally managed load balancing. They take advantage of the accord that all information is collected on one site and thus good scheduling decisions can be made. However, in big clusters this central machine may become a bottleneck or even a single point of failure.

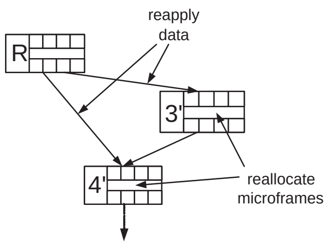
The SDVM works without client-server concepts as far as possible. Therefore the scheduling is done autonomously by each site. The sites therefore don't have knowledge about the current global execution status of the application, but only about the locally available executable microframes. Some information can be extracted in advance, though: The dataflow graph of the application contains all microthreads and therefore the critical path of an application and regions of high data dependencies can be detected. These parts will then be executed with higher priority resp. executed preferably on the same site.

The *site manager* collects data about the local site, e.g. processing speed, current load, number of applications the site works on, etc. This information is then passed (piggyback on other messages) to other sites' *cluster managers*, measuring the current network latency between these sites on the way. The cluster manager then possesses performance data about any site it directly works together with. Thus it can provide hints on which microframes to pass to which site. For example, a slow site with long network latencies will not be given a microframe which lies in the critical path of the application—another microframe which will be needed a bit later and therefore can afford to be calculated slower would be a better choice.

Another job of the cluster manager is the crash management. If a site does not respond to messages anymore, it is (after a while) regarded as crashed. The cluster is informed about the crash, then the applications which were executed on this site are determined by the other sites, as these applications have to be restarted. To avoid a whole restart of an application the SDVM features a checkpointing mechanism: Any microthread may not only apply its calculation results to the microframe awaiting them but also to a special microframe, the checkpoint frame (see Figure 4(a)). When a crash occurs, the site holding the youngest complete checkpoint frame is determined. This site then creates a recovery frame which recreates the not-yet executed microframes and reapplies



(a) Information about microframes and the data applied to them gets copied to the checkpoint frame.



(b) After a crash occured, the recovery frame is generated and executed. It recreates the stored microframes and reapplies the stored data.

Fig. 4. The checkpointing mechanism works on the CDAG (controlflow dataflow allocationflow graph) [9] of an application

the parameters to them (see Figure 4(b)). The application then runs on from that point undisturbed.

2.5 Freedom of adaptivity

The optimization success of an application's execution depends on how the current environment properties can be dealt with. Therefore an application which doesn't make too many restricting assumptions before runtime is more easily optimized at runtime. Typical assumptions are e.g. the platform type the application will be run on, the performance needed, the size of the cluster, the degree of parallelism, etc. The later those degrees of freedom are exploited and actual information taken into consideration, the more this information will be accurate with regard to the execution environment—and thus the system be made adaptive and the optimization improved.

In order to cope with the mentioned degrees of freedom, the SDVM acts as a virtualization layer which hides most properties of the underlying hardware from the applications. Therefore the SDVM may decide single-handedly where and when to execute specific microframes. In the area of reconfigurable hardware, the SDVM may even decide to resize the cluster by configuring additional processors and thus react to performance demand peeks. Based on available space and application requirements microthreads themselves can be configured as hardware at runtime and thus executed much faster.

The support for heterogeneous hardware architectures and varying cluster sizes makes it possible to upgrade hardware while the software runs on: Add new hardware and shut down the old.

2.6 Reliability and dynamic power management

The SDVM features another interesting concept which can be useful to enhance the reliability of a cluster or better yet of a multiprocessor chip it runs on.

The *energy manager* monitors the current load of the whole cluster and decides whether more processing power than needed is available. In this case it will send some sites a signal to work in a slower mode or even shut down completely. This reduces energy consumption and avoids overheating of processors. In case the load increases sites will get a signal to recur from sleep or shutdown mode.

Since energy management has an impact on the reliability of a system [10], the reliability can be further enhanced by introducing a new energy management policy. Unlike usual strategies which try to minimize energy consumption or reduce it without sacrificing performance, the new policy aims towards a minimal number of temperature changes. Thermal cycles induce mechanical stress which is a major contributor to chip failure [11]. Thus reducing thermal cycles reduces mechanical stress and therefore prolongs lifetime.

The SDVM is well suited for this kind of energy management policy, because the workload distribution adapts automatically to the changing performance of each site. Sites which fail to request work are not slowed down immediately in order to reduce thermal cycling. Similarly, sites having high load levels are not put to a higher performance level immediately if there are still underworked sites present in the cluster.

A method where any site may freely decide for itself its energy status may result in a situation where all sites simultaneously decide to shut down; therefore, as a mitigation of the distributed paradigm, the energy managers use an election algorithm to define a master which then is the only one to decide. The master may even decide to shut down its own site or to quit being the master; then the election is simply started again among the remaining sites.

3 Results

In this section some results are shown for a simple application, namely the Romberg numerical integration algorithm [12]. This algorithm partitions the area to be measured into several portions of constant width. Those can be measured independently and the results added eventually. The first microthread will generate a target microframe where the results are finally added and then, in our example, 100 or 150 other microframes containing the Romberg algorithm, which can be run in parallel.

The SDVM needs a lot of calculations and communication to distribute code and data. Therefore a question is whether the additional overhead is small enough to maintain the concept.

First, it shall be demonstrated how much overhead is generated by using the SDVM. To show this, run times on a stand-alone SDVM site are compared with the run times of a corresponding sequential program (see Figure 5). This overhead appears to be about 2%, even if the microthreads have to be compiled before execution.

In the next step, it has to be shown that the speedup is in expected regions. On a cluster of identical machines (Pentium IV, 1.7 GHz), a value for the

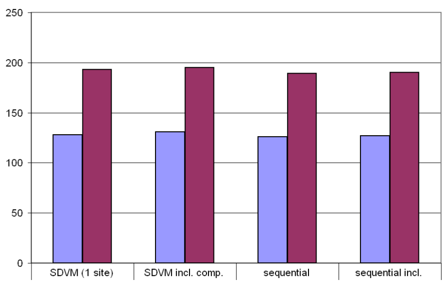


Fig. 5. Romberg algorithm: Comparison of the run times (in seconds) of a sequential program and the SDVM with one site. Values are given with and without compilation time, respectively, for width 100 and 150.

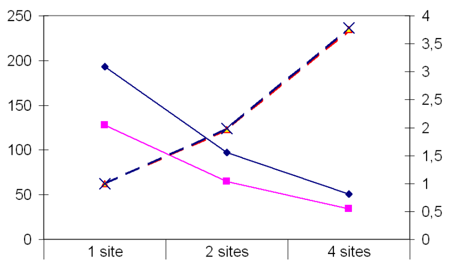


Fig. 6. Romberg algorithm: Run times and speedup depending on the number of sites

	1 site	2 sites	4 sites
width 100	128	65	34
width 150	193	97	51
speedup width 100	1	1.97	3.76
speedup width 150	1	1.99	3.78

speedup is shown in Figure 6. It reaches roughly the number of participating sites, which is a good result.

4 Conclusion

The Self Distributing Virtual Machine is a middleware which connects any functional units to form an adaptive parallel computing system. Both structure and functionality are biologically inspired as it is built from autonomous interacting units, features decentralized decision making and supports self-healing from cluster member faults. The SDVM detects failed members, removes them from the cluster and enables applications to efficiently recover from failure by the use of checkpointing.

The SDVM is self-organizing as a new SDVM-enabled unit which wants to join only needs a communication channel to a site which is already part of the cluster. As sites may join or leave at runtime without disturbing the execution of running applications, the cluster may grow or shrink to any convenient size, moreover regardless of the sites' operating systems, hardware or even the network topology between them. The cluster scales automatically.

It is self-optimizing as it automatically distributes data and program code to sites where it is needed, thereby dynamically balancing the workload of the whole system. Furthermore, this vastly facilitates a hardware upgrade while the system is running by shutting down old hardware and signing on new hardware—the applications will be relocated automatically and continue to run nonetheless. Similarly, resources can be added temporarily to cope with short term peeks in computing power demand.

The distributed scheduling of the SDVM provides the foundation for a new energy management policy which can improve the reliability of the participating systems. It differs from usually applied policies in its focus to reduce the number of thermal cycles of the system while minimizing the negative impact on performance. The tradeoffs between performance and reliability, and number of thermal cycles and mean temperature levels are currently investigated.

A prototypical implementation of the SDVM has been created and evaluated for the area of cluster computing. The prototype and its full source code is freely downloadable [3]. The SDVM is currently being adapted to multi-core processor systems.

References

1. George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18. IEEE Computer Society Press, 2002.
2. VDE/ITG/GI-Arbeitsgruppe Organic Computing. Organic Computing, Computer- und Systemarchitektur im Jahr 2010. Technical report, VDE/ITG/GI, 2003.
3. The SDVM homepage, 2006.
4. *SCI: Scalable Coherent Interface, Architecture and Software for High-Performance Compute Clusters*. Springer-Verlag, 1999.
5. Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 485–496. ACM Press, 1998.
6. Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, New York, 1994.
7. Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 2.5 edition, September 2001.
8. Jan Haase, Frank Eschmann, Bernd Klauer, and Klaus Waldschmidt. The SDVM: A Self Distributing 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.
9. Bernd Klauer, Frank Eschmann, Ronald Moore, and Klaus Waldschmidt. The CDAG: A Data Structure 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.
10. K. Mihic, T. Simunic, and G. De Micheli. Reliability and power management of integrated systems. In *DSD - Euromicro Symposium on Digital System Design*, pages 5–11, 2004.
11. Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, Jude Rivers, and Chao-Kun Hu. Ramp: A model for reliability aware microprocessor design. In *IBM Research Report, RC23048 (W0312-122)*, December 2003.
12. G. Dahlquist and A. Bjorck. *Numerical Methods*. Prentice Hall, Englewood Cliffs, NJ, 1974.