A Framework for Self-adaptive Collaborative Computing on Reconfigurable Platforms

Michiel W. VAN TOL [a] and Zdenek POHL [b] and Milan TICHY [b] [1]
[a] *Institute for Informatics, University of Amsterdam, The Netherlands*
[b] *Institute of Information Theory and Automation*
Academy of Sciences of the Czech Republic, Prague

AbstractAs the number and complexity of computing devices in the environment around us increases, it is interesting to see how we could exploit that and glue them together to create larger co-operative distributed systems. This paper describes a framework for dynamically aggregating and configuring processing resources in order to meet local requirements and constraints. The capability of this framework is demonstrated by a case study using an adaptive least mean squares filter (ALMS) application. ALMS improves convergence of least mean squares filters at the cost of more resources, and allows us to demonstrate abilities of the framework such as task offloading and run-time adaptation to available resources.

Keywords. distributed systems, collaborative computing, resource management, heterogeneous systems, adaptive systems, SVP, MicroBlaze, FPGA.

1. Introduction

Today's embedded computing is limited to isolated devices bearing some local computation. They consist of traditional microprocessors and application specific accelerators like DSP processors or ASICs, statically programmed to handle for example a user interface, communication or multimedia processing. The number of such embedded devices around us only increases in the future, with each containing processing elements that have their own special capabilities.

The idea that the computer of the future will eventually dissolve into our environment has already been explored two decades ago [1]. And with more and more embedded computing appearing in the objects we use in our everyday life, e.g. in our mobile phones, TVs, toys and cars, we are moving towards this scenario. Can we use a collection of these devices, interconnected with network technologies, as a collaborative computing platform for the future? How do we cope with the heterogeneous nature of such a system, and with devices with reconfigurable hardware? Can we define a way to abstract this and program it transparently?

In this paper we define and demonstrate a framework that supports such an abstraction of heterogeneous platforms. It supports self-adaptation and self-repair, as the envi-

[1]This work was supported and funded by the EU, Project AETHER No. FP6-2004-IST-4-027611

ronment we envisage will not be a static set of resources. The user throws an application onto a collection of resources, regardless of what it contains, and it organizes itself in order to execute the application in the most optimal way. Devices like ASICs cannot change their functionality while microprocessors or FPGAs can be re-programmed or reconfigured. The framework supports changing the configuration of devices to optimize itself for the dispatched applications, potentially migrating components from device to device when devices come and go or need to be reconfigured. The advantage of our framework is the separation of concerns between capturing asynchronous activities in an application and mapping it to a set of parallel resources, which is performed dynamically.

Our approach is based on isolating and distributing software components to devices that are specialized in executing them. We show how our framework acts as a middleware that provides the necessary abstraction layer through both a distributed implementation of the *Sane Virtual Processor*, SVP [2], to manage asynchronous activities and communication, and a *System Environment Place*, SEP [3], protocol implementation to expose the capabilities of the devices and manage resources in the system (Section 2). We present a case study in Section 3 of an adaptive least mean squares filter application applied to a hardware platform that fits the component based approach by offering complex arithmetic operations through an efficient reconfigurable computing pipeline [4,5,6]. We show how the application and the system adapt to both the type and availability of resources. We discuss related work in Section 4 and conclude in Section 5.

2. Framework

In our distributed framework, each node is a *Self-Adaptive Network Element*, SANE [7], which implements the SVP (work delegation) and SEP (resource management) protocols. SVP provides us with a way of expressing groups of asynchronous activities, that can be seen as software components [8], that are started by a *create* action. It represents resources as an abstract *place*, which can be anything where a component can execute, be it in software or hardware. As we apply SVP to a distributed system here, the *create* action can transparently result into a remote call, similar to an Active Message [9], where *place* is used as a parameter to identify both the target node in the system, and the specific resource on that node to execute the requested component. SVP provides us with a way to wait for a component to complete its operation, *sync*, and to interrupt the execution with *kill*. When making remote calls or calling an operation, this means a *create* of the component, local or remote, followed by a *sync* to wait on its completion.

SVP provides us with the necessary constructs to implement self-healing and self-adaptive mechanisms. When a program detects a failing software component, it can attempt again to *create* it on a different *place*. A timeout and *kill* can be used to cancel the execution of a component that stopped responding, for example as the resource has become unreachable, or to replace a component that does not deliver the required performance. A different resource is requested and the component is (re)started there.

The SEP protocol provides us with a mechanism to exchange *place* information between nodes, acting as a lookup system to identify which nodes are capable of executing a specific software component. In this paper we discuss an SEP with a single level hierarchy as used in the case study example. However, as suggested in [3], besides a multi-level hierarchy, a distributed peer-to-peer implementation of the protocol is also possible where nodes exchange information that they have about services on other nodes.

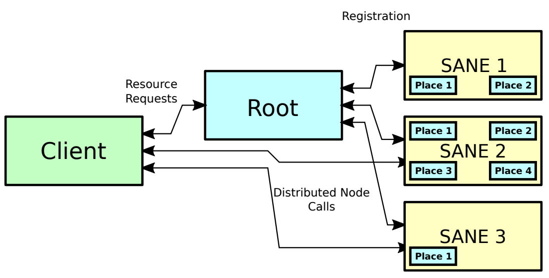

Figure 1. Client-server architecture of the distributed SVP/SEP framework

In a one level hierarchy in the SEP protocol, a SANE announces itself to the *root SANE* when it joins the system, registering the list of services for software components that can be executed on that SANE. *Client* applications requesting a software component will contact the root, which will then negotiate with the appropriate SANEs that offer this service. In this negotiation a cost model is used, which can be based on the efficiency of the software component on a certain SANE, taking into account potential reconfiguration cost, the current workload on the SANE, and the power budget. A SANE is selected and contracted and, if needed, (re)configured for the execution of the software component, and a *place* identifying the contract and the resource is returned to the client. Once the resource has been obtained, the client application can directly use the SVP *create* action to transparently execute the operation on the allocated place. An application is able to acquire multiple places, allowing it to offload and execute many software components in parallel. This client-server architecture of the SEP protocol is illustrated in Figure 1.

The SEP service is run as continuations on top of SVP, where each node has a *place* where the SEP preserves its state, and the SEP is invoked by starting instances of SEP components on these places using *create*. For example, the client makes a request for resources with a *create* of the SEP_request() component, using corresponding arguments, on the *place* of the SEP at the root SANE.

This framework provides us with a dynamic environment for collaborative distributed computing. Nothing prohibits a node from acting both as a client and as a SANE offering services, and both can dynamically join and leave the system. An application is not statically mapped onto resources, but acquires them on an on-demand basis when it wants to execute a certain software component. A *create* to a remote *place* will transparently result into a communication to start the execution remotely.

3. Case Study

3.1. Adaptive Least Mean Squares Filter

To demonstrate the possibilities of our framework, we use an application that implements an *adaptive least mean squares* (ALMS) filter, shown in Figure 2, based on the well-known *least mean squares* (LMS) [10] filter. We send 2000 input samples to four instances of the LMS filter using different *learning rate* coefficients, the initial values $\mu_m(0)$ shown as a vector in equation 1. The learning rate μ determines the step size of the LMS towards the minimum of its cost function, where μ influences the convergence rate and precision of the result. After each batch, the $\mu_m(n)$, for which the LMS reached the

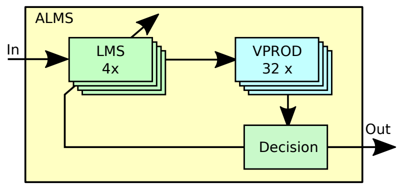

Figure 2. The ALMS filter algorithm

$$\boldsymbol{\mu}^{\mathrm{T}}(0) = [0.015, 0.012, 0.01, 0.009] \quad (1)$$

$$\boldsymbol{\mu}(n+1) = [1, 1.5, 1.2, .9]^{\mathrm{T}}\, \mu_{best}(n) \quad (2)$$

best score is computed as $\mu_{best}(n)$, which is used to generate (equation 2) new learning rates $\boldsymbol{\mu}(n+1)$ for the next batch.

By using four LMS filters we are able to improve tracking of the non-stationary model parameters. The adaptation of learning rates is shown in Figure 3. Parameters of the stationary model were estimated up to iteration $n = 50$. After that, the input was switched to a different model, instantaneous changing the estimated parameters. As a result, the learning rates quickly increase after iteration $n = 50$, and as the system adapts to the new condition, the learning rates slowly descend back to their original values.

3.2. *Application Programming Model*

The block diagram of the algorithm for ALMS is shown in Figure 2. Executing one iteration of the ALMS filter can either be done as a single component itself, or, when this is not available, by composing it of four LMS components followed by 32 vector product (VPROD) operations.

The application acts as a client in our framework sending a request to the SEP to determine if the ALMS component is available. If it is not, it sets up four asynchronous operations, using *create*, that request and execute an LMS component from the SEP, and *sync* is used to wait for all four to complete. This sequence of operations is shown in Figure 4; it starts with the blocking *resource request* to the SEP. Once the resource has been acquired, the operation is *called* on that resource. After it finishes, the resource can be released. In the end, the *sync* operation synchronizes all outstanding activities. As these activities are asynchronous, it can use one to four parallel LMS filters and one up to 32 parallel VPROD operations. If either LMS or VPROD is not available, the execution of the user application is suspended until the resources become available. The decision about the best filter result is obtained by the summation over VPROD results and by their comparison. Such solution hides the number of real resources involved in the computation, as the concurrent execution is constrained by the availability of resources. Therefore, the performance can scale up and down dynamically according to the run-time availability of computing places, or we could adapt the quality of our solution.

3.3. *Platform*

In our case study we use a platform based on three Xilinx ML402 prototype boards connected through switched Ethernet. Each has a Virtex2 SX35 FPGA that is configured to contain a MicroBlaze RISC processor [11] on the FPGA fabric. Programmable hardware accelerators [12,4,5,6] are connected to the MicroBlaze to accelerate floating-point DSP algorithms. The possibility to program the accelerators with microcode is one of the key features of our solution; the SANE implementation can provide hardware accelerated functions that can be changed on demand.

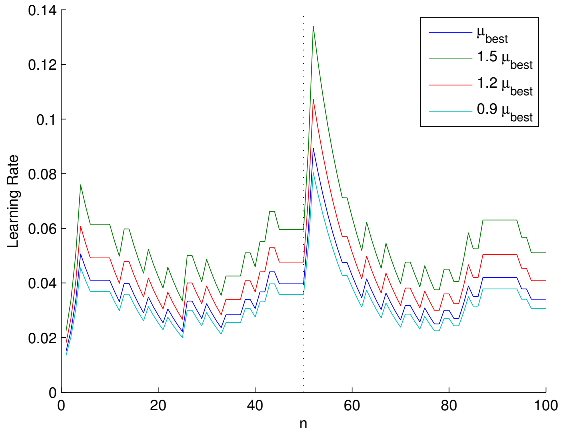

Figure 3. Adaptation of learning rates of the ALMS filter

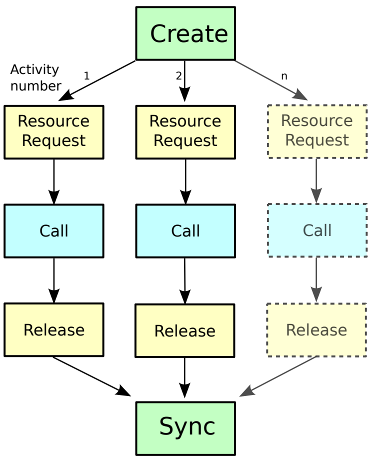

Figure 4. Asynchronous activities using SVP

On the board we run an embedded Linux and use a POSIX threads based implementation of SVP [13], which was extended [8] with a lightweight TCP/IP message protocol to uniformly handle concurrency and communication on our distributed platform. The SEP is implemented on top of this, exposing the accelerators and the MicroBlaze core as *places* offering services. To implement these services we developed accelerator microcode to for both the LMS and the complete ALMS filters, as VPROD was already available. These three are encapsulated as services exposed by the SEP, where each node can support one or more of these services simultaneously. This is discovered by the SEP when a node starts, and the available services are announced to the network.

3.4. Experiments

In our experiment, we show the behavior of the application running on the framework when resources are entering and leaving the system at run-time. Figure 5 shows the presence of resources and the development of the speedup of the application compared to running on the bare minimum of resources it requires to successfully execute (one LMS, one VPROD).

At the start of the run, the user application can not execute its computations because there are no available resources. It starts immediately when a SANE, providing one LMS and one VPROD enters the system. When another SANE providing these services enters the system at iteration $n = 10$, the application starts to use both in parallel and the performance is improved. Now all three nodes are in action. One with the user application and two with SANEs, each containing one place. After that, around $n = 20$ we withdraw one SANE from the system again, and replace it with a new SANE which contains four places able to execute four LMS or VPROD in parallel. The application starts to use four LMS filters in parallel. At the same time, the application uses all five available VPROD operations. When the second SANE with one computing place is replaced by the SANE with four places at $n = 32$, the number of VPROD operations in use reaches eight. However, in this case the number of used LMS filters remains at four as the application can not use more LMS operations in parallel. At this point the performance of the system

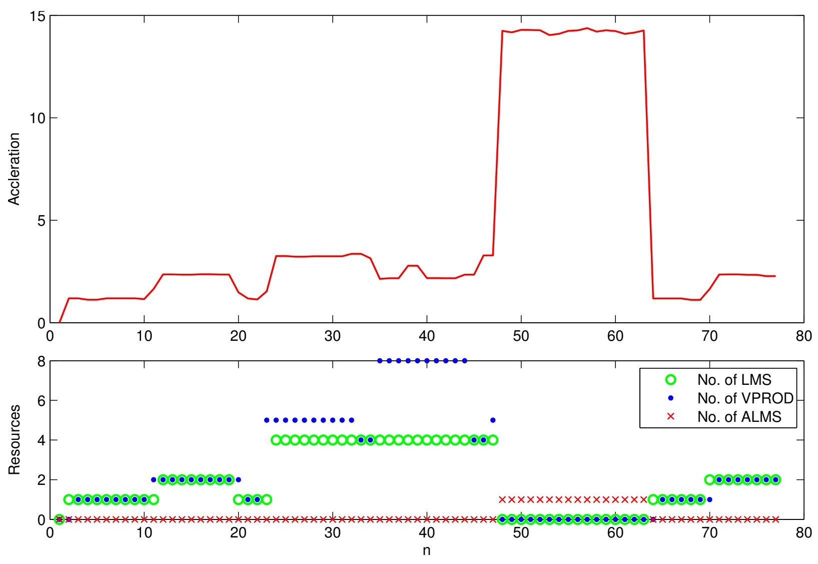

Figure 5. Adaptation of the ALMS application to the available resources

decreases because the overhead of executing 8 remote VPROD operations. Finally, the SANE capable of computing the complete ALMS filter appears at $n = 47$. Since the user application prefers to use this implementation, the ALMS is used instead of the composition of LMS and VPROD. This configuration shows the best performance. When the ALMS capable place leaves the system again at $n = 74$, the backup solution based on using the LMS and VPROD operations is restored.

Further measurements on our framework implementation showed that the minimum overhead for calling a remote function is 16ms on a 1 Gbit Ethernet network connection from a PC to our FPGA boards, and as low as 0.3ms between two PCs on a 1 Gbit Ethernet connection. This means that the software components that are distributed across the network need to be of sufficient granularity to hide the latency of the network overhead. However, we measured that a single iteration of the ALMS software implementation takes 6.8ms on the MicroBlaze on a node, and 0.7ms when executed in the hardware accelerator. Taking into account that the requests to the SEP take multiple remote calls, the granularity of the distributed software components turned out too small compared to the framework overhead. This can also be observed in the result of the previously explained experiment when there were 8 VPROD operations available on the network, the overall performance of the application would decrease due to network overhead.

4. Related Work

There are many different techniques that have been developed to transparently access heterogeneous resources and services in distributed systems. Well-known technologies are RPC [14], Corba [15] or Java RMI [16] which implement interfaces that can also deal with dynamic resources. However, they do this from a classic sequential execution viewpoint with blocking communication primitives. Furthermore, they are not very concerned with resource management, with the exception of load balancing. This differs from our framework where there is a clear distinction of responsibility between delegation and allocation of work, which can all happen asynchronously. Such a coarse-grained dataflow execution model is supported by Legion [17], which is another distributed object system presenting an arbitrary heterogeneous pool of resources as a single virtual computer. The main differences with the work we present here, is that it is object based whereas our framework is based on services, and that all communication is hidden in Legion.

More coarse grained distributed environments like Grid and Cloud computing are less suitable for fast changing collaborative computing environments. Service oriented computing on the Cloud [18], using for example SOAP [19], offers very coarse grained web services that have their own contained functionality. They are not small reusable components from which we construct our applications dynamically in our framework. The Grid often uses techniques such as MPI [20,21] for communication which only deals well with embarrassingly parallel applications on a static set of resources, and leave resource management to Grid schedulers which only adapt on a very large timescale. Tool-kits like Globus [22] provide several services to build a dynamic infrastructure, authentication, resource allocation, storage, and a communication layer. XtreemOS [23] is similar approach integrating the functionality into the Linux kernel. Most of this is taken care of in an SVP implementation, which makes our framework closer to Legion, though it still shows explicit communication by mapping execution to places.

The framework that we presented in this paper is a suitable dynamic *computing fabric* which can be ideally targeted with other component oriented languages such as S-Net [24], when we can map the components directly on the services provided on the fabric. On the related side, we have also successfully integrated another hardware platform [25] with our framework, combining it with an S-Net runtime and application that transparently used the dynamically available hardware components.

5. Conclusion

We have shown that we can use the described technologies for creating a collaborative distributed computing system, in which we can encapsulate already available components as services and share them within our environment. It supports the dynamic nature of such a system where nodes can join and leave the network using a simple set of primitives. Therefore this framework can be a viable basis for further future research in the direction of collaborative distributed computing, including, but not limited to, research on the cost model and mapping of applications, as well as the SEP network organization.

A case study was shown of the proposed framework based on FPGA prototyping boards running Linux. We used it to show how our system supports run-time allocation of resources and adaptation by dynamic reconfiguration of computing nodes driven by application needs and resource availability. We experimentally confirmed how an application can use this dynamic and heterogeneous nature of the system with the implementation of the ALMS filter, which dynamically used software or hardware accelerated components to adapt to the available resources in the system.

We showed that our software component based distributed system approach needs to be of sufficient granularity to amortize the network overhead. However, this approach is useful for self-healing systems as they can be easily re-instantiated on other nodes if there is a failure in the system. It allows the application programmer to express requests for functionality, without having to worry how and where this is actually executed.

References

[1] M. Weiser, “The computer for the 21st century,” *Scientific American*, vol. 265, pp. 94–104, Sept. 1991.

[2] C. R. Jesshope, “A model for the design and programming of multi-cores,” *Advances in Parallel Computing*, vol. High Performance Computing and Grids in Action, no. 16, pp. 37–55, 2008.

[3] C. R. Jesshope, J.-M. Philippe, and M. W. van Tol, “An architecture and protocol for the management of resources in ubiquitous and heterogeneous systems based on the SVP model of concurrency,” in *SAMOS '08: Proc. of the 8th Int. workshop on Embedded Computer Systems*, (Berlin, Heidelberg), pp. 218–228, Springer-Verlag, 2008.

[4] J. Kadlec, M. Danek, and K. L., “Proposed architecture of configurable, adaptable SoC,” in *The Institution of Engineering and Technology Irish Signals and Systems Conference, ISSC*, 2008.

[5] M. Danek, J. Kadlec, R. Bartosinski, and L. Kohout, “Increasing the level of abstraction in FPGA-based designs,” in *Int. Conf. on Field Programmable Logic and Applications, FPL 2008*, pp. 5–10, Sept. 2008.

[6] J. Kadlec, “Design flow for reconfigurable MicroBlaze accelerators,” in *4th Int. Workshop on Reconfigurable Communication Centric System-on-Chips* (J. M. Moreno and et. al., eds.), July 2008.

[7] M. Danek, J.-M. Philippe, P. Honzik, C. Gamrat, and R. Bartosinski, “Self-adaptive networked entities for building pervasive computing architectures,” in *ICES '08: Proc. of the 8th Int. Conf. on Evolvable Systems: From Biology to Hardware*, (Berlin, Heidelberg), pp. 94–105, Springer-Verlag, 2008.

[8] M. W. van Tol and J. Koivisto, “Extending and implementing the self-adaptive virtual processor for distributed memory architectures,” *CoRR*, vol. abs/1104.3876, April 2011.

[9] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active messages: a mechanism for integrated communication and computation,” in *ISCA '92: Proc. of the 19th annual Int. Symp. on Computer architecture*, (New York, NY), pp. 256–266, ACM, 1992.

[10] B. Widrow and S. D. Stearns, *Adaptive Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1985.

[11] Xilinx UG081 (v6.0), *MicroBlaze Processor Reference Guide*, June 2006.

[12] J. Kadlec, R. Bartosinski, and M. Danek, “Accelerating MicroBlaze floating point operations,” in *Int. Conf. on Field Programmable Logic and Applications, FPL 2007*, pp. 621–624, August 2007.

[13] M. W. van Tol, C. R. Jesshope, M. Lankamp, and S. Polstra, “An implementation of the SANE Virtual Processor using POSIX threads,” *Journal of Systems Architecture*, vol. 55, no. 3, pp. 162–169, 2009.

[14] A. D. Birrell and B. J. Nelson, “Implementing remote procedure calls,” *ACM Trans. Comput. Syst.*, vol. 2, pp. 39–59, February 1984.

[15] Object Management Group, “CORBA component model,” specification v4.0, April 2006.

[16] A. Wollrath, R. Riggs, and J. Waldo, “A distributed object model for the Java system.,” in *COOTS*, USENIX, 1996.

[17] A. S. Grimshaw and W. A. Wulf, “The Legion vision of a worldwide virtual computer,” *Commun. ACM*, vol. 40, no. 1, pp. 39–45, 1997.

[18] Y. Wei and M. B. Blake, “Service-oriented computing and cloud computing: Challenges and opportunities,” *IEEE Internet Computing*, vol. 14, pp. 72–75, 2010.

[19] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Frystyk Nielsen, S. Thatte, and D. Winer, “Simple object access protocol (SOAP) 1.1.” W3C, Note NOTE-SOAP-20000508, May 2000.

[20] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir, “MPI-2: Extending the message-passing interface,” in *Euro-Par'96 Parallel Processing* (L. a. e. Bougé, ed.), vol. 1123 of *LNCS*, pp. 128–135, Springer Berlin / Heidelberg, 1996.

[21] I. Foster and N. T. Karonis, “A grid-enabled MPI: message passing in heterogeneous distributed computing systems,” in *Supercomputing '98: Proc. of the 1998 ACM/IEEE Conf. on Supercomputing*, (Washington, DC), pp. 1–11, IEEE Computer Society, 1998.

[22] I. Foster and C. Kesselman, “The Globus project: A status report,” in *Proc. of the 7th Heterogeneous Computing Workshop*, HCW '98, (Los Alamitos, CA), pp. 4–, IEEE Computer Society, 1998.

[23] C. Morin, “XtreemOS: A grid operating system making your computer ready for participating in virtual organizations,” in *Proc. of the 10th IEEE Int. Symp. on Object and Component-Oriented Real-Time Distributed Computing*, ISORC '07, (Washington, DC), pp. 393–402, IEEE Computer Society, 2007.

[24] C. Grelck, S.-B. Scholz, and A. Shafarenko, “A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components,” *Parallel Processing Letters*, vol. 18, no. 2, pp. 221–237, 2008.

[25] J.-M. Philippe, B. Tain, and C. Gamrat, “A self-reconfigurable FPGA-based platform for prototyping future pervasive systems,” in *Evolvable Systems: From Biology to Hardware* (G. Tempesti and et. al., eds.), vol. 6274 of *LNCS*, pp. 262–273, Springer Berlin / Heidelberg, 2010.