View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Hertfordshire Research Archive

Compiler-Support for Robust Multi-Core Computing ⋆

Raimund Kirner, Stephan Herhut, and Sven-Bodo Scholz

Department of Computer Science, University of Hertfordshire, Hatfield, United Kingdom
{r.kirner,s.a.herhut,s.scholz}@herts.ac.uk

Abstract. Embedded computing is characterised by the limited availability of computing resources. Further, embedded systems are often used in safety-critical applications with real-time constraints. Thus, the software development has to follow rigorous procedures to minimise the risk of system failures. However, besides the inherent application complexities, there is also an increased technology-based complexity due to the shift to concurrent programming of multi-core systems. For such systems it is quite challenging to develop safe and resource-efficient systems.
In this paper we give a plea for the need of better software development tools to cope with this challenge. For example, we outline how compilers can help to simplify the writing of fault-tolerant and robust software, which keeps the application code more compact, comprehensive, and maintainable. We take a rather extreme stand by promoting a functional programming approach. This functional programming paradigm reduces the complexity of program analysis and thus allows for more efficient and powerful techniques. We will implement an almost transparent support for robustness within the SAC research compiler, which accepts a C-like functional program as input. Compared to conventional approaches in the field of automatic software-controlled resilience, our functional setting will allow for lower overhead, making the approach interesting for embedded computing as well as for high-performance computing.

1 Introduction

In embedded computing it is important to have a high utilisation of resources, as resources tend to be limited. Furthermore, in safety-critical computing it is quite important to provide fault-tolerance for the most severe sources of faults. The classical term of fault tolerance implies that one has to built a fault model for the faults to be tolerated [1]. However, with increasing system complexity it has come to the point where it is very hard to identify the severe faults on a real physical system. As a result, the way to deal with this situation is to make systems robust, basically by enriching them with behaviour patterns that are believed to increase the likelihood of service-sustainability of the system [2]. Actually, robustness is a term still in the beginning of its solid definition, as it needs more research for behaviour patterns of a system that increase its robustness.

⋆ The research leading to these results has received funding from the IST FP-7 research project "Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)".

Besides inclusion of robustness into the system design, it is also mandatory to follow a stringent engineering approach that minimises the risk of unintended system behaviour. For example, in the avionics domain the de-facto standard DO-178b [3] is used and in the automotive domain there is currently an active development of the ISO 26262 [4] standard as a guide for development of embedded software.

With the increasing concurrency within systems, it is also important to improve software development tools so provide a sufficient level of assistance to the developer. For example, in future embedded computing we have the challenge of providing robustness as well as efficient use of multi-core processors.

Our approach to tackle this problem is to develop a compiler that provides support for software-controlled robustness on multi-core architectures. There has been a lot of research in the area of compiler support for fault-recovery, especially in the field of high-performance computing [5–7]. However, there are a lot of technical issues like the overhead of checkpointing and the complexity of recovery.

Our approach is to use the functional programming paradigm, which simplifies many of the issues of fault-recovery, as the state of a component can always be narrowed down to the input data of this component, which allows for an efficient fine-grained recovery strategy. As a concrete compiler framework we use SAC (Single Assignment C), which is a functional programming language that has a syntax quite close to ANSI C [8, 9]. SAC has a special strength on array data structures that allows a compiler to automatically generate concurrent code for that. Technically, SAC compiles to ANSI C code together with library calls to manage the concurrency. Thus, SAC is also a good choice for portability to embedded platforms, even though it has been originally developed for high-performance computing machines.

2 Robustness in Embedded Computing

Embedded systems are often used in a safety-critical environment where the correct behaviour of the system is of utmost importance. To achieve dependability, the classic approach is to enrich the system with mechanisms that allow to tolerate a set of explicitly described faults. The prerequisite to design fault-tolerant systems is the definition of a fault model and the intended level of tolerance. Though quite useful as a design concept, fault-tolerant computing is only as good as the level of realism of the fault model is. For example, slightly-off-specification faults can be quite subtle and it is very hard to completely prerecognize them [10]. With this realisation, the focus in dependable computing is currently shifting from fault-tolerance to robustness, where the basic idea of a system to be robust is to maintain an acceptable level of service despite various *unexpected* perturbations [11, 2]. The origin of unexpected perturbations can be manifold, as described by Obermaisser and Kopetz:

> *"Robustness comprises the provision of an acceptable level of service despite the occurrence of transient and permanent hardware faults, design faults, imprecise specifications, and accidental operational faults."* [12]

A robust system is expected to maintain operation but may reduce the level of service during operation, even producing less accurate but still useful output. Robustness

is a system property that is inherently hard to evaluate, as it by definition deals with *unexpected* perturbations. However, there are behaviour patterns that are believed to make a system robust. Among these patterns are resilience, adaptability, recovery, recursive restartability, repairability, anytime computation, or degeneracy [2]. Note that these patterns are not strictly orthogonal, some of them even enforce other.

resilience is the system's ability to compensate for a temporal degradation of the provided level of service.

adaptability is the system's ability to change its behaviour, or internal or external structure in order to compensate for perturbations or changing requirements.

recoverability is the system's ability to detect and remove errors by restoring a correct state of the affected component. Recoverability avoids the accumulation of errors within the system state.

recursive restartability is the system's ability to restart small parts of the system independently in order to remove any erroneous state in these parts after a transient fault. A partial restart is expected to cause less delay and service disruption than a full restart. The "recursiveness" comes from the subdivision of a system into subsystems, where a restart of a (sub)system will require to also restart all its nested subsystems.

repairability is the system's provision of an interface and mechanisms that allow to repair a system after a transient or permanent fault has occurred. After the repair of a system component, the system has to be able to reintegrate the component and its providing services into the system.

anytime computation is an approximating implementation style where the system iteratively refines the result. The benefit of anytime computation is that the intermediate result can be used as an approximation of the final result in case that the system gets short on computational resources [13, 14].

degeneracy is the degree to which a part of the system can take over the functionality of another part of the system. Degeneracy is different from redundancy, as redundancy is meant to add extra resources for the provision of the same service, while degeneracy focuses on the ability of changing the use of a resource by reusing it for the provision of a different service.

In fact, above list of behaviour patterns is not meant to be exhaustive, nor does it mean that a system has to implement all of them to be sufficiently robust, as it is always a matter of application context to decide what level of robustness is sufficient. And of course, it is impossible to implement an absolutely robust system that can withstand whatever kind of perturbations will come.

3 Compiler Support for Robustness

In the following we discuss how development tools like a compiler can support the system developer in developing robust concurrent applications.

There are certain robustness patterns that are best achieved by explicitly programming them at the application level. For example, to deploy the *anytime computation* [13,

14] patterns one has to choose appropriate algorithms that are suitable for anytime computation. Translating a program automatically into anytime computation has not been done so far and it is also questionable whether an automatic translation would result in efficient resource usage. Further, anytime computation is not meant to be applied for individual instructions. Instead the whole algorithm should be trimmed to anytime computation. Thus, anytime computation so far is not considered as a subject for compiler support.

The *adaptability* pattern is also tightly linked to the concrete application semantics. However, the adaptability behaviour is still somehow orthogonal to the logic operations performed by the application. Thus, it would make sense to program adaptability behaviour in a coordination language like S-Net [15, 16]. We are currently working on adding such support for robustness into the S-Net compiler. As some first results we derived a specialised variant of S-Net, which is resource-boundable [17]. To perform system adaptation it is first also necessary to be able to detect the errors to react on. However, this does not necessarily have to be a reactive process. There exists also research towards automatic proactive fault detection [18–20].

Resilience is an emerging behaviour pattern that is driven by others. For example, *resilience* can be realised based on *recovery*. Given that the system state is saved on some regular basis, whenever an error is detected a recovery can be evoked by restoring the last saved correct system state [21, 22]. Depending on the type of fault and the system architecture, it might be sufficient to restore the state only partially, which would be more efficient than a global state restoration [23]. However, such a selective recovery is more complex to realise. If there is a permanent fault, a reconfiguration might be necessary for proceeding the operation. Till having finished the recovery process the system's level of service may temporarily degrade. The placement of checkpoint code and recovery code can be controlled by the compiler, as it has a relatively rich knowledge about the program code.

Whenever there is a faulty local state detected in part of the system, *resilience* might be also achieved by restarting this part of the system, thus bringing it back to a correct initial state. In general, restarting is a more simple technique than recovery as it does not require to regularly store the system state for recovery. However, in complex calculations a recovery might be more beneficial as a restart, as the restart will cause to loose any intermediate result having been calculated so far by this part of the system. The compiler support for recursive system restart is similar to the second part of system recovery, the detection of erroneous states and their removal.

The *Repairability* pattern makes the overall design choices to achieve robustness explicit. Accepting that it is not always possible to avoid all faults or to mask them, one has to provide efficient solutions to regain a correct system state. A repair action typically consists of the following actions [2]:

1. Error detection
2. Location of the erroneous part of the system
3. Analysis to direct the repair action
4. Repair action
5. Reintegration of the service

The specific repair action can be one of the already discussed measures like restart, rollback, etc. It is important that the runtime system provides the support for the reintegration of the repaired component. This reintegration might be an issue for the compiler is case that the reintegration is done at the level of a runtime layer that is generated by the compiler.

Degeneracy is an import design criterion of robustness, that also offers a potential for strong benefit from compiler support. The compiler can generate the code of a service implementation for different platforms such that the runtime system can migrate a service to a different hardware component. In less resource-constrained systems the recompilation for a different hardware itself may be also performed during runtime, adding further flexibility to this robustness pattern.

4 Robustness in a Functional Setting

As detailed in the previous section, we consider recoverability, recursive restartability and degeneracy as the key aspects of robustness that should be tackled, at least partially, at the compiler level. All three of these require that a thread of execution can be stopped and its state captured. In the context of recoverability, it suffices to store the state of the entire system or application, commonly referred to as checkpointing, such that the system can be resumed from there later. However, the main overhead in checkpointing in modern systems stems from storing the associated state [21]. Therefore it is beneficial to minimise the state that is to be stored. Rich type systems as they are commonly found in functional programming languages can be of great value in this context. By exploiting structural information on heap objects derived from their types, the size of the checkpoint data can easily be reduced by up to 70% [22].

Rich type information becomes even more valuable in the context of recursive restartability. Here, a global checkpoint does not suffice, as this would only allow to restart the entire computation. Instead, checkpoints for each sub-computation need to be generated and stored. Reducing checkpoint size is therefore even more important. Another key aspect in this setting, however, is to compute the containment of a sub computation, i.e., those parts of the global state that are required to restart the computation and those that are modified by the computation. Again, functional programming languages have key advantages here. Due to their call-by-value semantics, which ensure that arguments to functions are immutable, and explicit modelling of side-effects, containment of function state is well defined in a functional setting. Only the arguments to a function are required to restart its computation and the function itself does not modify any global state. Therefore, support for reversing and repeating a computation in a compiler for functional languages is simple to implement compared to the imperative setting [24].

Lastly, rich type information and the strict containment of function state can be exploited in the context of degeneration, as well. A key requirement to be able to tolerate the failure of a hardware component or subsystem is the ability to migrate the program and its state from the failed component to another component of the system. In today's heterogeneous systems, this migration might involve, apart from capturing the state, a conversion of the data. In weakly typed imperative settings like C, migrating

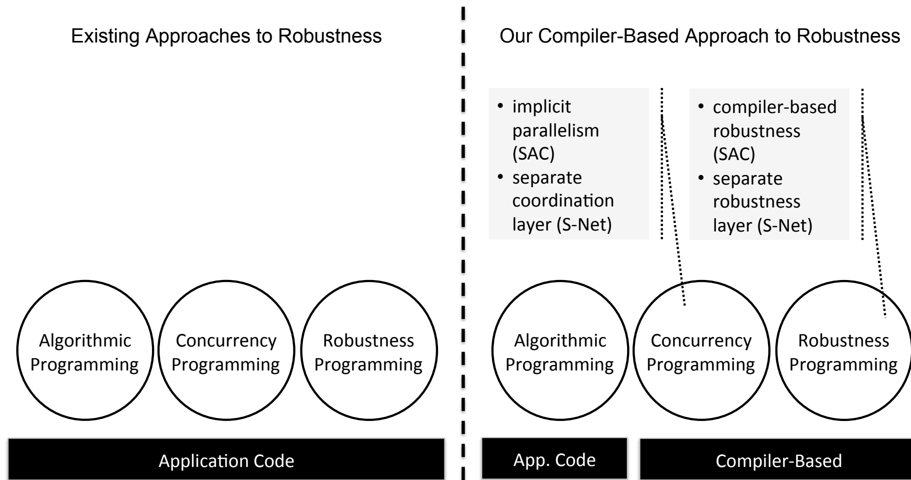

Fig. 1. Overview of our approach to compiler based robustness versus classical approaches.

state across platforms requires sophisticated compiler support and involves significant runtime costs [25]. The two key challenges involved, capturing the state of a computation and analysing the data-layout, are greatly simplified in a functional setting. We have argued for the former before. The latter, data-layout analysis, is supported greatly by the rich type annotations. It is common practise in functional languages to generate boiler-plate code like serialisers and representation transformations automatically [26].

A general overview of our approach and the divide between programmer supported tasks and compiler supported tasks in robust concurrent programming is given in Figure 1. In classical approaches, as shown on the left-hand side, the full responsibility lies with the application programmer. In the source code the arithmetic programming is typically mixed up with concurrency programming and, if needed, also robustness programming. We shift the responsibility for concurrency and robustness, at least in part, to the compiler, as shown on the right-hand side of Figure 1. We see concurrency programming ideally as an implicit activity to simplify concerns, or even separate it by using a coordination language like S-Net for it. Robustness programming should also be kept separate from the algorithmic programming of the application. Robustness programming would also fit quite will into the concept of a coordination language like S-Net. Further, we see a significant contribution of robustness programming can be done automatically by the compiler, with optional guidance by the developer.

4.1 SAC - Data-parallel Functional Programming

To investigate the techniques and key challenges in compiler support for robustness in the setting of functional languages, we have chosen to use Single Assignment C [27, 8] and its compiler[1] as test environment.

[1] The compiler is available for download from the project's website at `http://www.sac-home.org`

Single Assignment C, or SAC for short, is a first-order applicative functional programming language designed initially with high-performance and high-productivity programming in mind. It has a syntax similar to C, combined with a programming model that resembles that of MATLAB.

The focus on high programmer productivity shows on two levels in SAC. Firstly, SAC programs are specified using a rather high level of abstraction compared to classical high-performance languages in the imperative domain. To allow for this, we have developed compiler technology that maps programs specified at high levels of abstraction to efficient low-level implementations on various architectures.

Secondly, those complexities of programming that do not contribute to the actual algorithm design, i.e., memory management, concurrency and scheduling, are handled implicitly by the compiler, as opposed to being done by the programmer like in most other languages. This, on the one hand, simplifies program specification and reduces the likelihood of programming errors. On the other hand, and even more importantly in the context of compiler supported robustness, it gives the compiler full control over the actual realisation of those aspects.

Although SAC was designed for large-scale high-performance applications, many of its features prove beneficial in the embedded domain, as well. In large-scale high-performance applications, efficient use of resources is of similar importance as in embedded systems. For instance, the memory footprint needs to be minimised to ensure that applications can run within existing resource bounds. Using deferred heap management by means of garbage collection, which is predominantly used in functional programming languages, therefore is not an option in SAC. To minimise memory usage, frequent garbage collection phases would be required. Those, however, have a detrimental effect on program runtime. SAC therefore uses non-deferred heap management via reference counting. Using reference counting, the size of the heap can be constantly kept at a minimum without periodically disrupting program execution. Instead, reference counting operations are explicitly inserted throughout the program code, leading to a predictable runtime cost. The latter property of reference counting, i.e., its predictable runtime behaviour and cost, is also important in the context of real-time systems.

As mentioned before, functional programming languages in general use call-by-value semantics, i.e., state is not shared across function boundaries but instead arguments are copied. However, for performance reasons, this property is often weakened in practise. For larger data-structures like arrays, where copying comes at a significant runtime cost, references and mutable data-structures are used instead. This, however, impedes static analyses and reasoning on programs. In SAC, all data-structures are immutable on the language level. Thus, once defined, their value cannot change regardless of the context they are passed into. Updating a value, at least conceptually, always produces a fresh copy. This rather rigorous setting allows us to apply advanced program rewriting techniques. Even more importantly, it gives the SAC compiler ultimate freedom to place objects in memory, distribute them and schedule their computation, as different threads of computation cannot share state but merely values. Of course, to achieve good runtime performance, at runtime mutable objects are used. However, the important distinction here is that these are introduced by the compiler as opposed to by the programmer.

Finally, the last feature that is of particular importance in the context of compiler support for robustness is the implicit nature of concurrency in SAC. By design, SAC puts forward a programming model that supports the programmer in expressing algorithms such that they naturally contain implicitly parallel operations. These are then mapped by the SAC compiler to the concurrency resources of the target platform. The mapping thereby can make use from large degrees of freedom, allowing the compiler to compute a range of schedules, from fully sequential to fine-grained massively parallel execution. Combined with the platform independent nature of SAC, this allows us to generate code for a range of systems from standard platforms like symmetric multiprocessors to very restricted targets like GPGPUs.

4.2 Support for Robustness with SAC

Our roadmap for adding robustness support to the SAC research compiler currently lists three major sub projects. Firstly, we will explore different fault detection techniques like software-based replication [6] and timeout-based failure detection. For the former, we are particularly interested to find what advantages a functional setting offers with respect to replicated execution. Apart from the simplified separation of state, we expect easier code generation for checking the validity of states in a coarser grained fashion.

Supplying interfaces to the underlying operating system and hardware to cater for proactive fault detection would be an interesting addition here. However, we do not expect an advantage from the functional setting in this case and will thus leave this as an optional future addition.

With this concept we hope to address single-event upsets as well as permanent local faults of hardware components. However, a thorough investigation of possible fault sources and techniques how to detect or tolerate them is another crucial component of this step.

Once we have the techniques in place to detect faults, a next step will be to adapt code generation such that faults can be tolerated and program execution can recover. In this setting, we plan to focus on adapting the scheduling of concurrent tasks in SAC to a robust setting. As SAC features fully implicit memory management and concurrency, we have a tight grip on the actual implementation of these by the compiler. Other than in C, for instance, where the communication pattern and scheduling is usually hard-wired using POSIX threads or MPI, in SAC the compiler has more freedom. If the failure of one computation is detected, the computation can be restarted. Here, the functional nature of SAC and the use of immutable data structures in SAC turns out to be very beneficial. By using our existing liveness analysis for reference counting, we can easily detect the active heap objects that take part in a computation. Furthermore, functions and parallel operations seem to be a natural guide for the granularity of recovery. As found previously in the context of ZPL [28], checkpoints are best placed in between parallel sections of execution. These optimal code positions are easily identifiable in SAC. Other than the previous work, however, in SAC we will implement recovery using a much finer granularity. Due to the side-effect free nature of parallel computations in SAC, we are able to restart each thread of computation and recover on the level of partial results. For this, we borrow techniques originally developed in the setting of software transactual memory for functional languages [29].

Lastly, we will investigate support for degeneracy in the SAC compiler. Our current compiler already features many of the required techniques. For instance, due to the high-level nature of SAC, we are able to generate code for a variety of platforms from a single source specification. Thus, we can support truly portable migration by using the compiler alone. As an example, consider a setup with a legacy computing node equipped with a special purpose co-processor like a GPU. Initially, we would emit code for the special purpose processor to maximise throughput and reduce energy costs. However, if such processor fails, we could automatically fall back to a generic implementation on the legacy node.

We are currently extending this approach to runtime adaptive codes [30]. The idea here is to recompile the program at runtime to adapt to a changing environment. Our current implementation focuses on adapting to changes in the input data, however this could be extended to fault-tolerance scenarios, as well.

5 Discussion

One might wonder whether the functional paradigm has any potential impact in the field of embedded computing. But, for example, with the programming language HUME there exist already well-suited functional programming environments for the embedded domain [31–33]. Furthermore, in safety-critical computing, people use code guidelines like MISRA [34] in the automotive domain, that provide rules that restrict conventional imperative programming languages almost to functional programming patterns. Thus, whenever it comes to safety-critical systems, the functional computing paradigm has a realistic impact potential.

The approach presented so far just highlights the potential for software-controlled robustness. However, we have already identified some special benefits of deploying a functional setting for software-controlled robustness. Exploiting the rich static knowledge about resource usage and the structure of a system's state, combined with the side-effect free nature of computation in the functional setting, has the potential to find more simple and efficient solutions as have been developed so far for conventional imperative programming paradigms.

We think that increased tool support for software-controlled robustness and multi-core deployment will become essential for the development of future embedded applications in the avionic and automotive domain. With the current rather inflexible development approaches it will become increasingly challenging to maintain efficiency and robust behaviour for future development platforms.

References

1. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing **1**(1) (Jan. - Mar. 2004) 11–33
2. Mikolasek, V.: Dependability and robustness: State of the art and challenges. In: Proc. Workshop on Software Technologies for Future Dependable Distributed Systems, Tokyo, Japan (Mar. 2009)

3. RTCA: Software considerations in airborne systems and equipment certification. RTCA/DO-178B (1992)
4. ISO/DIS: Road vehicles – functional safety. ISO/DIS standard 26262
5. Treaster, M.: A survey of fault-tolerance and fault-recovery techniques in parallel systems. ACM Computing Research Repository (CoRR) **abs/cs/0501002** (2005)
6. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software implemented fault tolerance. In: Proc. 3rd International Symposium on Code Generation and Optimization (CGO). (Mar. 2005)
7. Chang, J., Reis, G.A., August, D.I.: Automatic instruction-level software-only recovery. IEEE Micro **27**(1) (2007) 36–47
8. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34**(4) (2006) 383–427
9. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15**(3) (2005) 353–401
10. Ademaj, A.: Slightly-off-specification failures in the time-triggered architecture. In: Proc. 7th IEEE International Workshop on High Level Design Validation and Test, Cannes, France (Oct. 2002) 7–12
11. Mikolasek, V.: Robustness in complex systems - state of the art report. Research Report 26/2008, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria (2008)
12. Obermaisser, R., Kopetz, H.: From ARTEMIS requirements to a cross-domain embedded system architecture. In: Proc. Embedded Real Time Software and Systems, Toulouse, France (May 2010)
13. Horvitz, E.J.: Reasoning about beliefs and actions under computation resource constraints. In: Proc. Workshop on Uncertainty in Artificial Intelligence, Seattle, Washington (1987)
14. Boddy, M., Dean, T.: Solving time-dependent planning problems. In: Proc. 11th International Joint Conference on Artificial Intelligence. (Aug. 1989)
15. Grelck, C., Scholz, S.B., Shafarenko, A.: A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. Parallel Processing Letters **18**(2) (2008) 221–237
16. Shafarenko, A., Scholz, S.B., Grelck, C.: Streaming networks for coordinating data-parallel programs. In Virbitskaite, I., Voronkov, A., eds.: Perspectives of System Informatics, 6th International Andrei Ershov Memorial Conference (PSI'06), Novosibirsk, Russia. Volume 4378 of Lecture Notes in Computer Science., Springer Verlag (2007) 441–445
17. Kirner, R., Scholz, S.B., penczek, F., Shafarenko, A.: PS-NET - a predictable typed coordination language for stream processing in resource-constrained environments. In: Proc. 1st Int'l Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking. (Nov. 1010) (submitted).
18. Vallee, G., Engelmann, C., Tikotekar, A., Naughton, T., Charoenpornwattana, K., Leangsuksun, C., Scott, S.L.: A framework for proactive fault tolerance. In: Proc. 3rd Int'l Converence of Availability, Reliability and Security, Barcelona, Spain (May 2008) 659 – 664
19. Lee, C., Lee, D., Koo, J., Chung, J.: Proactive fault detection schema for enterprise information system using statistical process control. In: Proc. Conference on Symposium on Human Interface 2009, Berlin, Heidelberg, Springer-Verlag (2009) 113–122
20. Wang, C., Mueller, F., Engelmann, C., Scott, S.L.: Proactive process-level live migration in hpc environments. In: Proc. ACM/IEEE conference on Supercomputing (SC'08), Piscataway, NJ, USA, IEEE Press (2008)
21. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Compututing Surveys **34**(3) (2002) 375–408

22. Choi, S.E., Deitz, S.J.: Compiler support for automatic checkpointing. In: Proc. 16th Annual International Symposium on High Performance Computing Systems and Applications, Washington, DC, USA, IEEE Computer Society (2002) 213
23. Dinan, J., Singri, A., Sadayappan, P., Krishnamoorthy, S.: Selective recovery from failures in a task parallel programming model. In: Proc. IEEE International Symposium on Cluster Computing and the Grid, Los Alamitos, CA, USA, IEEE Computer Society (2010) 709–714
24. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, NY, USA, ACM (2005) 48–60
25. Ramkumar, B., Strumpen, V.: Portable checkpointing for heterogeneous archtitectures. In: Proc. 27th International Symposium on Fault-Tolerant Computing (FTCS'97), Washington, DC, USA, IEEE Computer Society (1997) 58
26. Hinze, R.: A new approach to generic functional programming. In: Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, ACM Press (2000) 119–132
27. Scholz, S.B.: Single Assignment C — efficient support for high-level array operations in a functional setting. Journal of Functional Programming **13**(6) (2003) 1005–1059
28. Choi, S.E., Deitz, S.J.: Compiler support for automatic checkpointing. In: HPCS '02: Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications, Washington, DC, USA, IEEE Computer Society (2002) 213
29. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2005) 48–60
30. Grelck, C., van Deurzen, T., Herhut, S., Scholz, S.B.: An Adaptive Compilation Framework for Generic Data-Parallel Array Programming. In: 15th Workshop on Compilers for Parallel Computing (CPC'10), Vienna University of Technology, Vienna, Austria (2010)
31. Patai, G., Hanák, P.: Embedded functional programming in Hume. In: IASTED on Software Engineering, Innsbruck, Austria, ACTA Press (2007) 328–333
32. Hammond, K., Michaelson, G.: The design of Hume: A high-level language for the real-time embedded systems domain. In Lengauer, C., Batory, D.S., Consel, C., Odersky, M., eds.: Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers. Volume 3016 of Lecture Notes in Computer Science., Springer (2003) 127–142
33. Hammond, K., Michaelson, G.: Hume: A domain-specific language for real-time embedded systems. In Pfenning, F., Smaragdakis, Y., eds.: Proc. 2nd International Conference on Generative Programming and Component Engineering. Volume 2830 of Lecture Notes in Computer Science., Springer (Sep. 2003) 37–56
34. MISRA, T.M.I.S.R.A.: MISRA-C 2004: Guidelines for the Use of the C Language in Critical Systems. MISRA (Oct. 2004) ISBN: 0-9524156-4-X (pdf version).