DanceOS: Towards Dependability Aspects in Configurable Embedded Operating Systems*

Horst Schirmeier[1], Rüdiger Kapitza[2], Daniel Lohmann[2], and Olaf Spinczyk[1]

[1] Technische Universität Dortmund, Germany
{horst.schirmeier,olaf.spinczyk}@tu-dortmund.de
[2] Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
{rrkapitz,lohmann}@cs.fau.de

Abstract. Future hardware designs for embedded systems are expected to exhibit an increasing rate of intermittent errors in exchange for smaller device sizes and lower energy consumption. This bears new challenges for system software, especially the operating system (OS), which has to use and provide software measures to compensate for unreliable hardware. Designing, developing, and maintaining such software systems will become increasingly arduous.
The DanceOS project aims at providing dependability by the *fine-grained* and *tailorable* application of software-based fault-tolerance techniques. The vision is to achieve this by separating the implementation of these techniques from the functional parts of the software in a reusable way, allowing for use-case specific static or even run-time application of fault-tolerance measures to critical parts of the *complete* software stack.
This article outlines the main challenges we identify in our OS design undertaking. Among other techniques, we propose aspect-oriented programming and tool-based (semi-)automatic software analysis as means for achieving these goals.

1 Introduction

Emerging from continuously shrinking structure sizes and a high potential for energy saving, a current trend in hardware design for embedded systems is a switch from deterministic to probabilistic hardware components [3,4,9]. Such designs allow for example a multiplication unit to yield wrong results from time to time while being supplied by an unusually low voltage [6]. Ideally, the system software layers should be aware of possible intermittent (soft) errors (Single Event Upset, SEU) and take countermeasures for the sake of their own stability and correctness. The application developer should be enabled to specify dependability requirements on the system's interface level in a goal-oriented manner, driven by the application's fault-tolerance demands.

In this article we state our position in this problem domain and outline first ideas and possible solutions we want to examine in the starting DanceOS[3]

* This work was partly supported by the German Research Council (DFG) focus program SPP 1500 under grant no. KA 3171/2-1, LO 1719/1-1 and SP 968/5-1.

[3] Dependability Aspects in Configurable Embedded Operating Systems

project. The remainder of this article discusses the three main claims we assert, and describes our vision on how dependability measures should be applicable in an application-specific, goal-oriented way.

2 Claims

2.1 Tailored Application of Dependability Measures

Dealing with transient hardware-induced faults by applying software measures, such as replicated execution in space or time, comes at a cost. Applying measures as a one-fits-all solution to the entire software stack (including the OS and application level) wastes an embedded system's scarce resources, limits adaptability to changing environmental and system conditions, and, most importantly, decreases performance. We claim that it is crucial to realize that dependability requirements are highly *application-specific* and may even vary on a system service granularity level: e.g., a specific application may need "five nines" for a *file write* system call but be satisfied with a *sleep* call that is 20% off the mark. Nevertheless, dependability measures may need to be applied on *all* software layers—but in a coordinated manner, well-matched with each other.

These basic assumptions imply the need for a configuration mechanism which is capable of fine-grained tailoring of dependability measures for a specific application scenario (static configuration) or a changing environment (dynamic adaptation). Such **configuration mechanisms** are already state-of-the-art for *functional* properties of embedded operating systems, but need to be adapted and extended for dependability concerns.

2.2 Analysis of Dependability Properties

Dependability measures can be applied at various locations in the software stack from the application down to the OS and hardware layer. Depending on the application's requirements, a subset of these locations—the neuralgic, "weak" spots of the hierarchy—must be chosen. This ensures that, for example, triple-modular redundancy (TMR, [8]) is only applied in places promising the best possible ratio between runtime cost of a measure and its effect on dependability.

As an example, consider a very simple layered system which consists of only two software components A and B on the base layer, and one component C on the upper layer . Induced by possible hardware faults the components have a reduced reliability[4] of $R_A = 0.95$, $R_B = 0.8$, and $R_C = 0.99$. Component C uses input from A and B to compute the final result. Without applying any dependability measures the reliability of the whole system can be calculated as follows:[5]

[4] Here *reliability* is understood as the probabability of computing a *correct* result. Hardware-faults that crash the entire system cannot be handled by software measures only.

[5] We ignore that the calculation in C could theoretically yield a correct result even though the input by A and B was wrong.

$$R_{ABC} = R_A \cdot R_B \cdot R_C = 0.95 \cdot 0.8 \cdot 0.99 = 0.7524 \tag{1}$$

If the user specified a reliability goal of 75% for the output of C, no action would be required. However, if the goal was, for instance, 80%, a dependability measure such as TMR had to be applied. TMR can improve the reliability of a module by means of redundant execution and voting [8]:

$$R_{\mathrm{tmr}(M)} = 3R_M^2 - 2R_M^3 \tag{2}$$

For a system designer there is a straightforward solution: The software stack is executed three times and a voter discards wrong results, i.e. TMR is applied on the top level of the system. This is an effective strategy for dealing with transient errors. It raises the resulting system reliability to a value above the requested 80% mark:

$$R_{\mathrm{tmr}(ABC)} = 3R_{ABC}^2 - 2R_{ABC}^3 \approx 0.8464 \tag{3}$$

The obvious drawback is that the computation time increases by a factor of three. Even if the time factor is hidden by offloading redundant execution onto a different CPU core, the additional energy consumption and RAM usage cannot be neglected in many scenarios. Alternatively, TMR could be applied to any subset of $\{A, B, C\}$, or even multiple subsets. For instance, the overall system reliability for TMR on component B (only) is $R_{A\,\mathrm{tmr}(B)C} = R_A \cdot (3R_B^2 - 2R_B^3) \cdot R_C = 0.95 \cdot 0.896 \cdot 0.99 \approx 0.8426$. This is almost as good as $R_{\mathrm{tmr}(ABC)}$ and better than the 80% mark, too. At the same time this solution would be much more efficient, because only component B has to be executed in a redundant manner. For the other possible system configurations the results are worse: $R_{\mathrm{tmr}(A)BC} \approx 0.7862$ and $R_{AB\,\mathrm{tmr}(C)} \approx 0.7598$.

The example shows that a dependability goal can be achieved in different ways. A good solution can only be found by considering dependability *and* resource consumption of each possible solution in a holistic manner. Furthermore, the most cost efficient solution depends on the actual user-specified goal. For instance, a requirement of 78% could also be fulfilled by $\mathrm{tmr}(A)BC$. More variability comes into play when other dependability measures with different characteristics than TMR are taken into account, too. Furthermore, the dependability goals for different parts of the system might vary. This complicates the process of finding the best solution dramatically, because the parts might have overlapping dependencies in the module structure.

In order to be able to take specific and problem-oriented action, a **dependability model** for the system layers—foremost but not exclusively the hardware and OS layers—is needed. Parnas' "uses" hierarchy [10], which models a system as a kind of dependency graph, is a promising foundation. For efficiency reasons and to keep chances for human error low, tool support for **static code analysis** should at least partially automate the model creation.

```
aspect TMR {
   // virtual pointcut: "where" this aspect should have effect
   // defined in a concrete specialization of the aspect
   pointcut virtual where() = 0;
   // advice: "what" should be done when the pointcut matches
   // "around" advice replace the original function call
   advice execution(where()) : around() {
      // advice body: TMR implementation
      Result a = tjp->proceed(); // call original function
      Result b = tjp->proceed();
      if (a == b) { return a; }
      return tjp->proceed(); // default to third try
   } };
```

Fig. 1. An "aspect" that implements triple-modular redundancy in a highly generic way, thus providing *separation of concerns* for an otherwise crosscutting problem.

2.3 Separation of Concerns

Once the system designer knows *where* to (application-specifically) apply *what* software dependability measures, actual code implementing the measures must be added to the software layers. Unfortunately, implementing an inherently crosscutting concern such as dependability "in-line" by means of the C preprocessor leads to classic modularization problems such as *scattering* (distribution of a concern implementation across multiple implementation artifacts) and *tangling* (many different concerns implemented in a single implementation module)—commonly entitled the "#ifdef hell". Especially in the case of *configurable* embedded system software, tailored dependability measures would be a *second* configuration dimension, leading to an explosion of the source code variant space.

Only strict separation of concerns can avoid a maintenance nightmare: The dependability concern implementation therefore mandatorily has to be separated from other, functional system concerns. We are convinced that **aspect-oriented programming** (AOP) [5] is an appropriate means to satisfy this requirement: Based on a description of the application's dependability goals and the dependability model, which was motivated in the previous section, we derive *aspect code* defining *where* ("pointcut") dependability measures (*what:* "advice") should be applied. A so-called "weaver" automatically takes care of intermixing the concern implementations. To illustrate the expressiveness of aspect-oriented programming languages, Fig. 1 shows a (slightly simplified) AspectC++ implementation of TMR, modularized in a *generic* aspect, applicable to arbitrary functions. The only restriction is that the return type must be copyable and comparable with the "==" operator.

AOP techniques have already been successfully applied as a means to modularize fault-tolerance mechanisms [1,2]. We intend to advance the field by integrating advanced static-analysis methods into the weaver in order to be able to offer far more powerful language constructs on the AOP-language level.

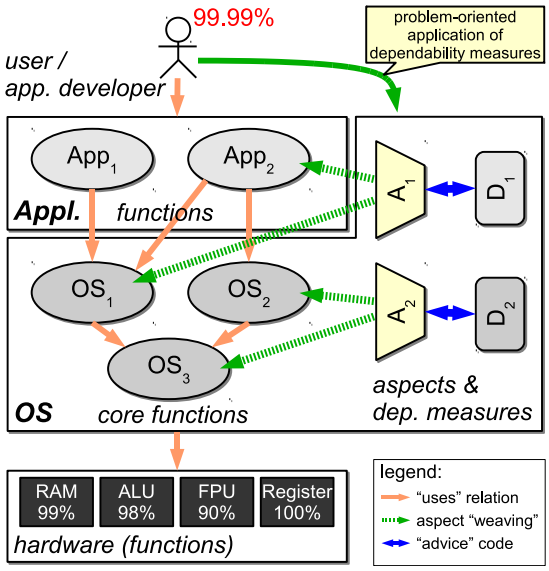

Fig. 2. Vision: OS and application components (**OS**, **App**) are accompanied by dependability aspects (**A**), which are able to "weave" various dependability measures (**D**) into arbitrary software components throughout the whole software stack.

3 Vision and Future Work

Our general idea is to achieve software-based dependability cost-efficiently (with respect to other non-functional properties, such as performance, memory footprint, or energy consumption) by means of dedicated *operating-system support* and *tailoring*. The system-structure diagram in Fig. 2 sketches the vision of the DanceOS project: Subject of the *tailoring* are the *OS* in general and its *dependability aspects* in particular, which—and this is the core concept—implement and *encapsulate* well-known and novel software-dependability measures as reusable modules that can take effect *across* the whole software stack (including the application) by means of aspect orientation. Thereby, dependability measures can be tailored and composed on the granularity of functions and expressions, driven by the actual dependability requirements of the application and the reliability properties of the underlying hardware instance.

We aim at verifying our claims and approaching the envisioned system software architecture. We will extend our CiAO OS product line [7], which is implemented in the AOP language AspectC++, in order to reflect the described architecture. Open research questions include the design of an appropriate dependability model, the required static analysis techniques, novel (fault-tolerance specific) AOP language features and OS dependability measures. Another crucial issue is to design a model and provide tool support for mapping goals to aspects that affect the "neuralgic spots" in system and application code at minimal costs.

References

1. F. C. Afonso. *Operating System Fault Tolerance Support for Real-Time Embedded Applications*. Dissertation, Universidade do Minho, Escola de Engenharia, Jan. 2009.
2. R. Alexandersson and P. Öhman. Implementing fault tolerance using aspect oriented programming. In *LADC '07: Proceedings of the Third Latin-American Symposium on Dependable Computing*, volume 4746 of *Lecture Notes in Computer Science*, pages 57–74. Springer-Verlag, 2007.
3. S. Y. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.
4. M. Duranton, S. Yehia, B. de Sutter, K. de Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, N. Navarro, A. Ramirez, O. Temam, and M. Valero. The HiPEAC vision. Technical report, Network of Excellence on High Performance and Embedded Architecture and Compilation, 2010.
5. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.
6. M. S. K. Lau, K.-V. Ling, and Y.-C. Chu. Energy-aware probabilistic multiplier: design and analysis. In *Proceedings of the 2009 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 281–290. ACM Press, 2009.
7. D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX Association.
8. R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, 6:200–209, April 1962.
9. V. Narayanan and Y. Xie. Reliability concerns in embedded system designs. *IEEE Computer*, 39(1):118–120, 2006.
10. D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, Mar. 1976.