Runtime System Level Fault Tolerance for a Distributed Functional Language DRAFT*

P.W. Trinder‡ R.F. Pointon‡ H-W. Loidl‡

Abstract

Functional languages potentially offer benefits for distributed fault tolerance: many computations are pure, and hence have no side-effects to be reversed during error recovery; moreover functional languages have a high-level runtime system (RTS) where computations and data are readily manipulated. We propose a new RTS level of fault tolerance for distributed functional languages, and outline a design for its implementation for the GdH language. The design distinguishes between pure and impure computations: impure computations must be recovered using conventional exception-based techniques, but the RTS attempts implicit recovery of pure computations.

1 Introduction

Reliability has been a major issue in the software community for a long time [Tur49], and recent developments greatly increase its importance [Ran00, Lap99]. The industrialised world is becoming increasingly reliant on computer-based systems. Simultaneously, low-cost distributed systems have become commonplace, for example the proliferation of workstation networks. Such systems can not simply be treated as a single stable machine and a more dynamic approach is needed. For such systems, a means of achieving reliability is to introduce distributed fault tolerance.

In theory functional languages have benefits for distributed fault tolerance. One reason is that large parts of a program are *pure*, i.e. without side-effects. Side effects amount to updating the (distributed) global program state, and their absence means that the damage caused by a failing computation is confined: there is no need to undo partial updates in the global state. Moreover, if an error is detected, pure computations can be automatically restarted without the danger of making multiple updates. To tolerate faults in impure

‡Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, EH14 4AS. {HWLoidl,RPointon,Trinder}@cee.hw.ac.uk

*Supported by research grant GR/M 55633 from UK's EPSRC and APART fellowship 624 from the Austrian Academy of Sciences.

computations these automatic and implicit techniques must be augmented with standard explicit mechanisms like exception-based recovery blocks. A second potential benefit of high-level language technology is that fault tolerance is a global property affecting all operations of the runtime system underlying a language, and enforcing such properties is easier with a high-level RTS. Many functional languages have a high-level RTS based on graph reduction, where both code and data are represented as graph and can be easily manipulated. For example computations are easily replicated to facilitate recovery.

We propose a new RTS level of fault tolerance for the GdH distributed functional language. GdH (Glasgow distributed Haskell), is a extension of the standard Haskell lazy functional language [PHA+97], adding facilities to distribute processes to remote processors and communicate with them [PTL00]. The augmented GdH runtime system will detect errors, attempt to recover from them, and provide continued service despite faults. An important feature of our design is that it distinguishes between errors occurring in pure computations and those in impure, typically I/O performing, computations. For an error detected in a pure computation, the RTS will attempt an automatic recovery, for an error in impure computation, it will raise an exception to be handled at the language level.

2 Related Work

Our design combines two technologies: software fault tolerance and distributed functional languages, and we briefly locate our work within these fields.

2.1 Fault Tolerance

We focus on software techniques for fault tolerance within a single application constructed in a conventional language. A fault tolerant system [LA90, Mil90, Lap99] contains mechanisms to cope with faults that remain or develop within it. These faults can be classified into *anticipated faults* and *unanticipated* faults. An anticipated fault is one that has a mechanism in the system to deal with it. An unanticipated fault has no such mechanism, and typically results from poor design. Anticipated faults are typically managed in the following phases.

- **Error Detection.** Faults cannot be directly detected, instead it is the manifestation of a fault that causes *errors*. Error detection is the identification of an erroneous system state. For example a heap overflow or a result timeout may indicate a non-terminating calculation.

- **Damage Confinement.** Because of the delay between the fault occurring, and the detection of an error, many errors may have been introduced, and the extent of the damage to the system must be assessed. In a functional language the damage caused by the failure of a pure computation will be extremely limited because its interaction with other computations is heavily prescribed, i.e. it has no side-effects.

- **Error Recovery** is the process of correcting the erroneous states introduced into the system. Pure computations can be recovered relatively easily because referential transparency means that an expression can be evaluated many times without changing its value. In practice this means that should the evaluation of a pure expression fail, a copy of the expression can safely be reinstated and re-evaluated. Section 3.2 elaborates on this point, and includes a possible mechanism for arranging recovery of failed computations. Such failures and recovery are entirely *implicit*: the semantics of the program remains the same, whether a failure and subsequent recovery occur, or not.

  A recent development in functional programming research has been to provide constructs for managing impure operations such as IO, exceptions, and state-manipulation, Monads are one such construct [PW93]. Recovering from failures in impure code entails raising exceptions to be dealt with *explicitly* at the language level.

- **Fault Treatment and Continued Service** entails reconfiguring the system so that the fault does not reoccur. For example if evaluating an expression causes an unanticipated fault resulting in the runtime system crashing on a particular Processing Element (PE) then the remainder of the system must continue to function, dealing with references to the now defunct PE. This implies that each PE must acquire a global picture of the system.

Some related areas outside the scope of this paper are as follows. Fault prevention is the process of engineering the system to avoid and remove faults [LA90]. We do not consider multi-component systems where the failure of one component may induce failures in another, e.g. a database failure may cause a program failure. We do not consider coarse-grain replication of hardware, software, data and computation, as for example in Tandem NonStop technology where processes are duplicated on similar machines [Bar78]. Nor do we consider programming languages with semantics that reduce the faults that can occur, for example ESFP [TT99] which guarantees the termination of computations.

2.2 Fault Tolerance Layers

Fault tolerance is provided at many levels in a system, as depicted in Figure 1. Each layer implicitly uses the fault tolerance provided by lower levels, but may need to intervene explicitly if a fault occurs that cannot be recovered at a lower level. Hardware level detection techniques include parity bits, and an example recovery mechanism is to 'black' (mark as unusable) an area of memory. Operating system level detection techniques include checksums on messages, and an example recovery mechanism is to generate repeated requests for a corrupted message.

Our design is at the application level within a system, and within this level we distinguish between the application language, and it's RTS. High-level or distributed languages often have sophisticated RTS: for example the Java RTS schedules threads and manages

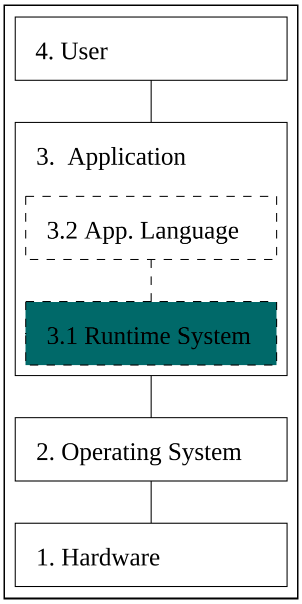

Figure 1: Fault Tolerance Layers

storage, including garbage collection. However, relatively few languages have RTS that tolerate faults. One that does is the SQL database query language where operations are grouped into transactions that may either commit or rollback. The detection of errors and the recovery of SQL transactions is implicitly managed by the underlying database management system.

Most application-level fault tolerance is explicit in the application language. The programmer detects errors using a range of techniques like reasonableness assertions, diagnostic, timeouts, coding, structural and reversal checks [LA90]. The programmer then recovers from errors using techniques like

- Recovery blocks. Programs are structured into modules that check their results and use an alternative method if necessary.

- Deadlines. A variant of recovery blocks, where modules must satisfy services within a given time.

- *N*-Version programming. 'N' instances of a program, where voting determines which result to use [CA78]. Each instance may use the same algorithm, or instances may use independent, but equivalent algorithms.

- Exceptions provide a declarative means for a subprogram to manage errors without excessive complication of the program. A *handler* is declared that will *catch* specific exceptions. On detecting an error, an exception is *thrown* in the subprogram. *Synchronous exceptions* are raised internally by the program, e.g. division by zero, or taking the head of an empty list. *Asynchronous exceptions* are raised by events external to the program, e.g. the programmer typing <Control-C>.

The object orientated community has long been writing distributed applications making use of the benefits afforded by the structure, modularity and flexibility of this language paradigm. These same features are also convenient for implementing fault tolerance in terms of the modularity and structure they possess, as demonstrated in the Arjuna programming system [PSWL95] or at the higher level in scripting, or co-ordination, languages [RSW98].

2.3 Distributed Functional Fault Tolerance

Many distributed functional languages exist e.g. Concurrent Clean [PV98], Kali Scheme [CJK95], Facile [GMP89], and Pict [PT97], but relatively few have addressed fault tolerance.

2.3.1 ERLANG

ERLANG was developed by Ericsson for use in the telecommunications industry. It is a simple impure functional language designed for concurrent, real-time, distributed fault tolerant systems [Wik94, Wik96]. It provides primitives for the creation of local and remote threads and of channels, as well as mechanisms for monitoring remote operations. A typical application constructed is Mnesia [MNW99], a distributed robust database management system.

Unfortunately ERLANG omits many features of modern pure functional languages, such as currying, higher-order functions and lazy evaluation, to name a few. More importantly it is an untyped language which allows many programmer errors to go unnoticed at compile time, although a type system has been proposed [MW97]. An ERLANG programmer builds fault tolerance, typically N-Version, using a few low level constructs, including timeouts, and the messaging system which processes error signals together with the normal messages.

2.3.2 OZ

OZ the language of the MOZART system, provides multi-paradigm programming, in that it combines functional, object-oriented, and logic styles. It provides a variety of primitives for distribution and fault tolerance. Also much work has gone into planning and providing semantics for fault-prone systems [Van99].

Essentially it is the use of exceptions that allows an OZ programmer to deal with faults[HVS97, HVBS98]. Lazy error detection, by handlers, enables the management of synchronous exceptions. Eager error detection, by watchers, enables the management of asynchronous exceptions. Fault tolerance is provided throughout the RTS down to the level of distributed garbage collection.

2.3.3 Concurrent Haskell

Concurrent Haskell [PHA+97] is a state-of-the-art concurrent lazy functional language that uses `forkIO` to create new processes, and the polymorphic semaphore (`MVar`) as primitives for communication and synchronisation of processes. Concurrent Haskell has recently evolved to provide error handling using synchronous and asynchronous exceptions that permit the transmission of higher-order values [PRH+99, MPM00].

2.3.4 GpH

Glasgow parallel Haskell (GpH) is a small extension to sequential Haskell constructed by the authors [THM+96]. It supports multiple threads on multiple PEs, but has only a single process. Threads are introduced and synchronised using parallel and sequential composition operators. Evaluation strategies abstract over the compositions to allow polymorphic and higher-order description of parallel behaviours [THLP98], e.g. `parList` creates a thread to evaluate every element of a list.

2.3.5 GdH

The language we propose to add fault tolerance to is GdH [PTL00], a superset of both Concurrent Haskell and GpH. GdH is a non-strict functional language that provides distributed concurrency i.e. remote threads and communication channels, as well as parallelism using distributed graph reduction. PEs can be identified in the language, meaning that computations, including impure computations like I/O, can be explicitly distributed. It possesses a distributed shared heap and supports both synchronous and asynchronous exceptions as in Concurrent Haskell. Although exceptions enable explicit faults handling, there are currently very limited error detection mechanisms, and no error recovery or continued service facilities exist in the RTS.

In contrast to Erlang which is a simple, impure, and untyped language with explicit low-level fault tolerance mechanisms, GdH is a complete, polymorphically-typed, pure functional language with features like currying and higher-order functions. Moreover, our fault tolerance design provides higher-level explicit recovery using exceptions. On the other hand OZ is elaborate: combining functional, logic and object-oriented paradigms, with explicit high-level exception-based fault tolerance. In comparison, the simpler language model of GdH, combined with the fact that programs consist of mainly pure computations, means implicit automatic recovery can be used alongside high-level explicit recovery of impure computations.

3 Adding Fault Tolerance to GdH

Our GdH fault tolerance design has three components: error detection, error recovery and continued service. A novel feature of our design is that we provide an additional level of fault tolerance at the runtime system level (3.1 in Figure 1). At this level we are able to use information about the virtual machine to detect errors, recover from them, and provide continued service.

In order to facilitate error recovery we distinguish between two kinds of errors based on the location where they occured:

- When an error is detected in *pure code* the RTS will attempt an automatic backward recovery using the techniques outlined below. There are cases where this automatic recovery may not be possible, or may not succeed, for example if a computation repeatedly fails on several PEs. In such a case we raise an exception to be caught and handled at the language level. In effect we attempt to recover at the RTS-level (3.1 in Figure 1) before resorting to programmer-supplied recovery mechanisms at the language level (3.2 in Figure 1). We do not intend to add any new language constructs to GdH to support fault tolerance, although we will define some new classes of exceptions to report unrecoverable errors.

- When an error is detected in *impure code*, typically I/O performing code, we raise an exception, to be handled at the language level.

Finally, the implementation will have the ability to log all errors detected, all recovery actions, and all continued service actions undertaken for subsequent analysis.

3.1 Error Detection

Currently in Concurrent Haskell unrecoverable errors at the hardware and operating system cause an exception to be raised that is either handled by the programmer at the language level, or by a default handler [PRH+99]. In the distributed context of GdH, our network layer (PVM) detects, and attempts to recover from, corrupt messages, lost messages and lost communication links [GBD+94]. At the RTS-level we will utilise distributed virtual machine information to detect, and subsequently recover from, the following *errors* using the techniques identified.

- The network layer message corruption checks will be augmented with a higher-level virtual machine check: the sending PE includes a 'fingerprint' of the program/data graph sent in a message, i.e. an abstract description of the graph as a reasonableness check. At the recipient the unpacked graph is checked against the fingerprint.

- The failure of a PE will be detected lazily when a request for data times out. The laziness implies that if a PE fails, but has no data required for the program to complete we need never detect its failure or attempt a recovery. Preserving laziness also implies that we must adapt the eager PE failure mechanism provided by the PVM network layer [GBD+94]. Intermittent PE failure is managed because whenever a PE detects a failed PE, it broadcasts the failed PE's identity, and all PEs ignore the failed PE thereafter. We also plan to implement a mechanism whereby the failed PE may reinitialise and subsequently rejoin the computation.

- Single communication link failure. Our communication model is fully connected, i.e. any PE can communicate with any other PE. In such a model the failure of communication link(s) to a single PE is indistinguishable from the failure of that PE.

- Multiple communication link failure. The failure of links to a group of PEs partitions the network. In GdH, one PE is distinguished as the *MainThread PE*, and both starts and terminates a program. The partition containing the MainThread PE will restart the pure computations from the 'lost' partition and may be able to complete the program. The programmer must provide mechanisms for recovering impure computations in the event of a partition.

It may be possible to use similar techniques to detect the failure of computations to terminate within a system-wide time limit, by associating a timeout with every computation. We do not intend to consider malicious or security-related errors. RTS-level deadlock and livelock detection mechanisms are not included because they cannot arise in pure computations, and we offer no advances over standard techniques for impure computations.

3.2 Error Recovery

In order to recover from errors in the distributed system we plan to extend the runtime-system, so that it is possible to restart pure computations. This requires several extensions in the communication mechanism of the current system and the introduction of new data structures for keeping track of the status of the network. Figure 2 summarises these changes. For details of the existing implementation the interested reader is referred to [THM+96].

Backup heap: Most importantly, the existing heap will be complemented with a *backup heap* of thunks that have been sent to another processor (the lower half in Figure 2). Since normal form closures (i.e. data items) are copied, rather than moved, they do not have to be copied into the backup heap. The existing *global indirection table (GIT)*, which maps the global addresses (GAs) of thunks to their local addresses on the processor will be extended to also contain the local addresses of the backup thunk*. After data has been sent off to another processor (right hand side of Figure 2) a global indirection (Fetchme) is then left on the original PE. The Fetchme points into the GIT and thereby connects to its backup thunk. Whenever a thread requires the data represented by a Fetchme, a message is sent, adding a time-stamp into the GIT.

In this design it is easy and cheap to find the backup thunk of a given GA, and the garbage collection of backup thunks can be performed with the same mechanism as for the standard heap: if no references from other processors exist and a closure has not been reached during local garbage collection, both the closure and the backup closure can be eliminated. However, this design also involves significant additional costs during garbage collection (the evacuation of closures has to be mirrored by evacuating the same closure in the backup heap, always going through the GIT in the process). Similarly, the GIT has increased in size which further increases the management overhead.

One major challenge will be to reduce the additional memory consumption imposed by the backup heap. To this end it will be important to reconcile a partially evaluated graph

*A thunk represents computation in our abstract machine model.

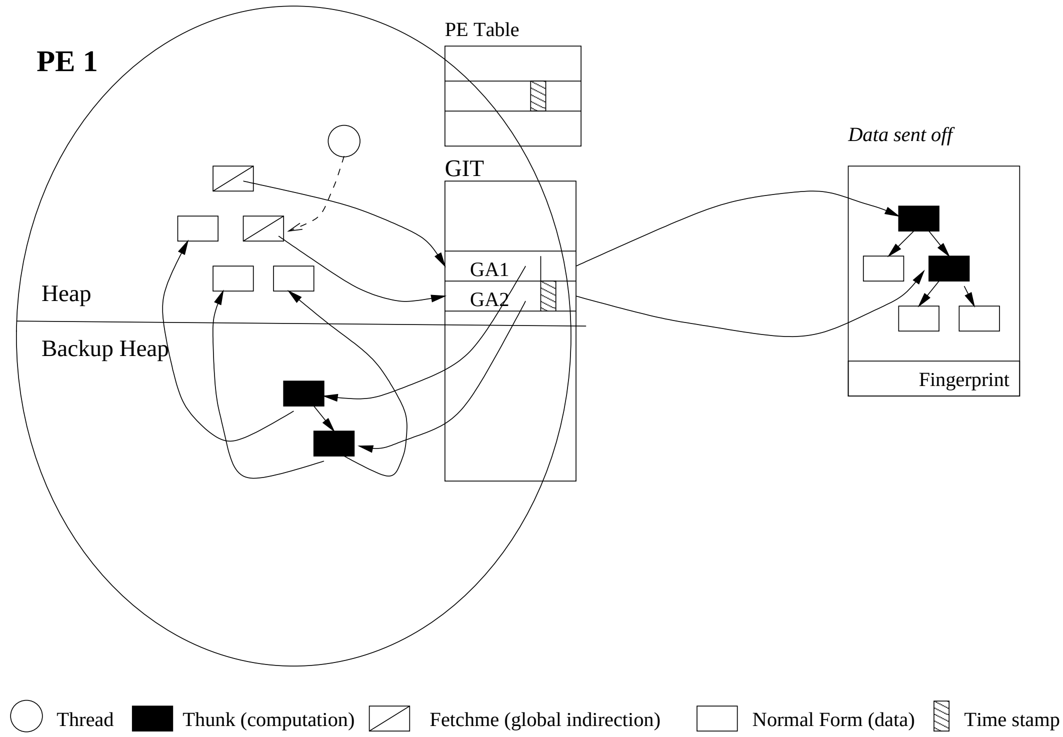

Figure 2: Changes in the communication mechanism to support fault tolerance

with its original unevaluated form. In the current system, whenever a partially evaluated graph for a locally existing thunk arrives, the thunk is overwritten with an indirection to the new graph. This reflects that the imported graph has proceeded further in its evaluation than the original one. Reflecting this overwrite in the backup heap is safe at this point. Of course, introducing pointers from the backup heap into the normal heap further complicates the garbage collector.

PE table and timeouts: In order to detect errors we plan to use *timeouts* on requests for remote data. Whenever a request for a global address is sent to a remote processor, a time stamp will be added to the corresponding entry in the GIT. Whenever a message is received from another processor a time stamp will be added to a newly introduced *PE table*. This PE table would hold various useful information on each PE, such as time stamps or the load on this PE. Having time stamps in both tables achieves a finer grain picture of the status of a remote PE. For example it is possible to detect that a remote data item is not available, e.g. because of a crashed thread, but that the PE itself is still alive.

In general the checks of the status of the network will be performed during garbage collection. If at that point the time stamp for a PE is older than a given value and a request has been sent in the meantime a timeout occurs. In this approach the system would “blacken” this PE and try to revert all required data that is known to reside on that PE. Information about a failure on the PE has to be propagated to the other PEs in the system. Depending on the nature of the failed code, it may be possible to automatically recover from this error, by retrieving the closure from the backup heap. Once it is available, the requesting thread enters this closure and evaluation of the thunk will now be performed locally. This assumes that timeouts are only checked for data that is actually demanded from another PE (*lazy recovery*). This might be augmented with a system where status information is collected via other (status) messages. If these status messages detect an error it is possible to recover backup closures referring to a faulty PE even before this data is demanded (*eager recovery*). Such behaviour should improve response time but might also create significant unnecessary work if no data is actually needed from the faulty PE.

Propagating status information: We plan to add an additional class of messages to the existing system in order to exchange system information. The existing implementation has been primarily tailored for efficient parallel execution but lacks support for reactive systems. In such systems a fast response to dynamic changes in the system and to user input is paramount. Therefore *system messages*, carrying information about the status of the individual PEs, will be sent at regular intervals during the computation, even if that involves some performance penalty due to an increased amount of communication even if no fault occurs.

As an optimisation to this approach it might be possible to propagating system information via existing messages, such as fish messages. These messages are sent when a processor becomes idle and performs work stealing from other PEs. In the process of looking for work a fish might reach several PEs and thereby can accumulate information about the global state of the system.

Programmer controlled recovery: Not all errors in the system can be handled automatically. In such a case the programmer must use suitable techniques, e.g. recovery blocks, and suitable exception handlers to recover from anticipated failures in impure computations.

3.3 Continued Service

The implementation will provide continued service using the following techniques.

- A computation is never restarted more than a (small) fixed number of times before raising an exception. This prevents an erroneous computation systematically destroying all PEs.

- A message is never retransmitted more than a (small) fixed number of times before raising an exception, preventing livelock.

- A system administrator can also add new PEs to replace failed PEs. The new (logical) PE may be a failed machine that has been restarted.

The PE failure recovery mechanism described above ensures that as long as the MainThread PE continues to function and there are no unrecoverable communication errors, then a computation that terminates on a single PE will also terminate on a network of PEs despite failures. The programmer can use exception handlers to recover from unrecoverable communication errors.

4 Future Work

We intend to implement the RTS-level fault tolerance outlined above. We then intend to build a number of demonstrators to exercise the error detection, recovery and continued service facilities. We will construct a long-running pure parallel program, like a large text searching engine, to demonstrate that continued service features enable such programs to complete even in the event of failures. We will also construct a program combining both pure and impure parts, like a a web-accessible parallel server, to demonstrate the combination of error recovery mechanisms: automatic RTS-level recovery of pure computations on the parallel machine, combined with programmer-specified recovery of impure webserver computations.

We plan to use the demonstrators outlined above to evaluate GdH's fault tolerance. In particular we will measure the runtime cost of fault tolerance. Measuring the cost of error detection in the absence of faults will be facilitated by constructing the RTS in such a way that it can be built with or without fault tolerance. We will also measure the varying costs as fault rates, both PE and communication, are increased.

We also intend to compare our functional fault tolerance with conventional languages. We hope to show reduced development and maintenance times because faults are mainly managed implicitly, requiring minimal additional code from the programmer. The metrics for the comparison will include lines of code dealing with faults. We will investigate the possibility of abstracting over exception handling to provide programmers with high-level descriptions of fault-tolerant behaviour. It may be possible to provide higher-order functions that capture common fault-tolerant constructs, rather like algorithmic skeletons capture common parallel constructs [Col88].

5 Conclusion

Distributed functional languages are finding uses in application areas like telecommunications where fault tolerance, in particular continued service is important. Examples include the use of Erlang in the new Ericsson AXD 301 ATM switch, and in the implementation of the Mnesia distributed database [MNW99]. We have outlined a new RTS-level fault tolerance for a distributed functional language that can be adopted to enhance the dependability of this class of applications. It is unusual to provide fault tolerance at the RTS level, and the new level augments the fault tolerance already provided by hardware and operating systems. Our design uses the high-level graph reduction machine underlying

GdH to characterise and replicate computations and data. The design also distinguishes between pure and impure computations, attempting implicit recovery of pure computations before resorting to programmer-supplied recovery mechanisms.

References

[Bar78] J.F. Bartlett. A 'NonStop' Operating System. In *The 11th Hawaii International Conference on System Sciences (HICSS'78)*, pages 103–117, Honolulu, USA, January 1978.

[CA78] L. Chen and A. Aviziensis. N-Version Programming: a fault-tolerance approach to reliability of software operation. In *FTCS-8 — 8th IEEE Int. Symp. on Fault-Tolerant Computing*, pages 3–9, Toulouse, France, 1978.

[CJK95] H. Cejtin, S. Jagganathan, and R. Kelsey. Higher-order distributed objects. *ACM Trans. On Programming Languages and Systems (TOPLAS)*, 17(1), September 1995.

[Col88] M.C. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* PhD thesis, University of Edinburgh, 1988. Also published in book form by Pittman/MIT, 1989.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[GMP89] P. Giacalone, P. Mishra, and S. Prasad. Facile: a symmetric integration of concurrent and functional programming. In *Tapsoft89*, LNCS 352, pages 181–209. Springer-Verlag, 1989.

[HVBS98] S. Haridi, P. Van Roy, P. Brand, and C. Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

[HVS97] S. Haridi, P. Van Roy, and G. Smolka. An overview of the design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, pages 176–187, Maui, Hawaii, USA, July 1997. ACM Press.

[LA90] P.A. Lee and T. Anderson. *Fault Tolerance Principles and Practice.* Springer-Verlag, 2nd edition, 1990.

[Lap99] J.C. Laprie. *Dependable Network Computing.* Kluwer Academic Publishers, 1999.

[Mil90] A. Mili. *An Introduction to Program Fault Tolerance.* Prentice Hall, 1st edition, 1990.

[MNW99] H. Mattsson, H. Nilsson, and C. Wikstrom. MMnesia - A distributed robust DBMS for telecommunications applications. In *First International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, 1999.

[MPM00] S. Marlow, S.L. Peyton Jones, and A. Moran. Asynchronous exceptions in Haskell. In *ICFP'00*, 2000. Submitted.

[MW97] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 136–149, Amsterdam, The Netherlands, June 9–11 1997. ACM Press.

[PHA+97] J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A.D. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S.L. Peyton Jones, A. Reid, and P. Wadler. *Report on the Programming Language Haskell (Version 1.4)*, April 1997.

[PRH+99] S.L. Peyton Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *Proc Programming Languages Design and Implementation (PLDI'99)*, Atlanta, 1999.

[PSWL95] G.D. Parrington, S.K. Shrivastava, S.M. Wheater, and M.C. Little. The Design and Implementation of Arjuna. *USENIX Computing Systems*, 8(3), 1995.

[PT97] B.C. Pierce and D.N. Turner. Pict: A Programming Language Based on the Pi-Calculus. Technical report, Indiana University, 1997.

[PTL00] R.F. Pointon, P.W. Trinder, and H-W. Loidl. GdH Design and Implementation. In *IFL*, Aachen, September 2000. To be submitted.

[PV98] R. Plasmeijer and M. Van Eekelen. *Concurrent Clean - Language Report*. High Level Software Tools B.V. and University of Nijmegen, version 1.3 edition, 1998.

[PW93] S.L. Peyton Jones and P. Wadler. Imperative Functional Programming. In *POPL'93 — Symposium on Principles of Programming Languages*, pages 71–84, Charleston, South Carolina, January 1993. ACM Press.

[Ran00] B. Randell. Facing up to faults. *The Computer Journal*, 43(2):95–106, 2000.

[RSW98] R. Ranno, S.K. Shrivastava, and S.M. Wheater. A Language for Specifying the Composition of Reliable Distributed Applications. In *The 18th International Conference on Distributed Computing Systems (ICDCS '98)*, Amsterdam, The Netherlands, May 1998.

[THLP98] P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[THM+96] P.W. Trinder, K. Hammond, J.S. Mattson, A.S. Partridge, and S.L. Peyton Jones. GUM: A portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation (PLDI'96)*, Philadelphia, USA, May 1996.

[TT99] A.J. Telford and D.A. Turner. Ensuring Termination in ESFP. In *15th British Colloquium in Theoretical Computer Science*, page 14, Keele, April 1999.

[Tur49] A.M. Turing. Checking a large routine. In *Report on a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge University Mathematical Laboratory, Cambridge, 1949.

[Van99] P. Van Roy. *On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in Mozart.* World Scientific, Tohoku University, Sendai, Japan, July 1999.

[Wik94] C. Wikstrom. Distributed programming in Erlang. In *PASCO'94 - First International Symposium on Parallel Symbolic Computation*, Linz, September 1994.

[Wik96] C. Wikstrom. Implementing Distributed Real-time control systems in a functional language. In *IEEE Workshop on Parallel and Distributed Real-Time Systems*, Honolulu, Hawaii, April 15-16 1996.