

A Programming Model and Language Implementation for Concurrent Failure-Prone Hardware

Phillip Stanley-Marbell Diana Marculescu

Department of ECE, Carnegie Mellon

Abstract

We present a programming model and its embodiment in a language implementation, for systems composed of large numbers of failure-prone, resource-constrained elements, interconnected in error-prone networks.

The programming model enables partitioning without replication, of applications, across multiple devices with constrained memory resources. It permits programs to specify the amount of error (value deviation) tolerable in individual variables, as well as tolerable latencies and erasures on communications. The value deviation constraints facilitate compile-time transformations for forward error correction; these transformations enable the value deviations in individual variables to be kept within program-specified bounds, in the presence of an assumed distribution of logic upsets in hardware. To account for situations in which such assumptions may be violated, language constructs enable the change of control-flow in response to tolerance constraint violations. The language model and implementation are targeted primarily at concurrent failure-prone embedded systems, such as sensor networks.

1. Introduction

Programming languages for hardware with large amounts of concurrency have been a topic of interest for many decades, since the advent of multiprocessors. Programming models and language implementations for such *parallel* systems have historically been concerned primarily with performance. The communications between the units of concurrency (processors) in these systems have traditionally been required to be of high performance (low latency), and to be free of both *errors* (e.g., values corrupted in transmission) and *erasures* (e.g., packets wholly lost in transmission).

We are interested in this work, in systems comprising *multiple processing elements* with *severely constrained memory and computation resources* communicating through a *low-performance interconnection network*, and in which both the processing elements and their communications are subject to *high logic-upset rates*, as illustrated in Figure 1. These upsets might be the result of permanent degradation of electric circuits, hostile operation environments, simplistic/low power communication interfaces, or a result of intermittent phenomena such as α -particle hits. One example of such a system is a sensor network comprising hundreds of low-power microcontrollers communicating over a noisy wireless channel. If such a system is deployed in a high-altitude terrestrial,

aviation, or space environment, it will witness a significant α -particle flux, which will result in a high logic-upset rate. Aggressive supply voltage and process technology scaling, will also increase the susceptibility of hardware to such intermittent logic upsets or “soft errors”.

In the application domains of interest in this work, the plurality of hardware resources, e.g., the hundreds of nodes in a sensor network, are not employed to increase performance in the sense in which multiple processors in a multiprocessor or multicomputer are. Instead, the compute resources are spatially distributed to locate them in the environs of physical phenomena which they monitor, and may perform *in-situ* processing on. When sensor systems are required to be embedded pervasively and unobtrusively in environments, advances in semiconductor technology will be harnessed not to make them faster or endow them with more memory, but rather to make them *smaller*.

Today, such systems are programmed by implementing the functionality at each individual node independently [3, 8]. It is desirable to be able to program the whole (possibly unreliable) network as a single computational device, with a single application partitioned for execution on the hardware substrate. Recent research has begun to approach this issue from the point of view of an operating system [7], for networks of PDA-class systems.

An important observation to be made about many application domains of interest in computing today, is that, even when represented by integer values in programs, their computations are often on values derived from, or driving, analog signals in their environments; examples of this include multimedia applications in workstations, and sensor driven applications in embedded systems. In portions of these applications, some amount of numeric deviation of values may be tolerable. For example, variables representing values from a color sensor, or pixels in a video frame, might incur small deviations from their correct values without adversely affecting the results of an algorithm (e.g., in an object tracking or image recognition application). *If such toleration of errors in individual variables can be explicitly encoded in programs, it is possible to introduce error correcting schemes that trade-off correction overhead for amount of value deviation that may occur.*

Programming languages are the *interface* between the algorithms which are of interest to us, and the machine model that executes them. When the abstractions exposed in a programming model accurately reflect properties of interest in the underlying hardware, implementations of the model can be efficient (e.g., high perfor-

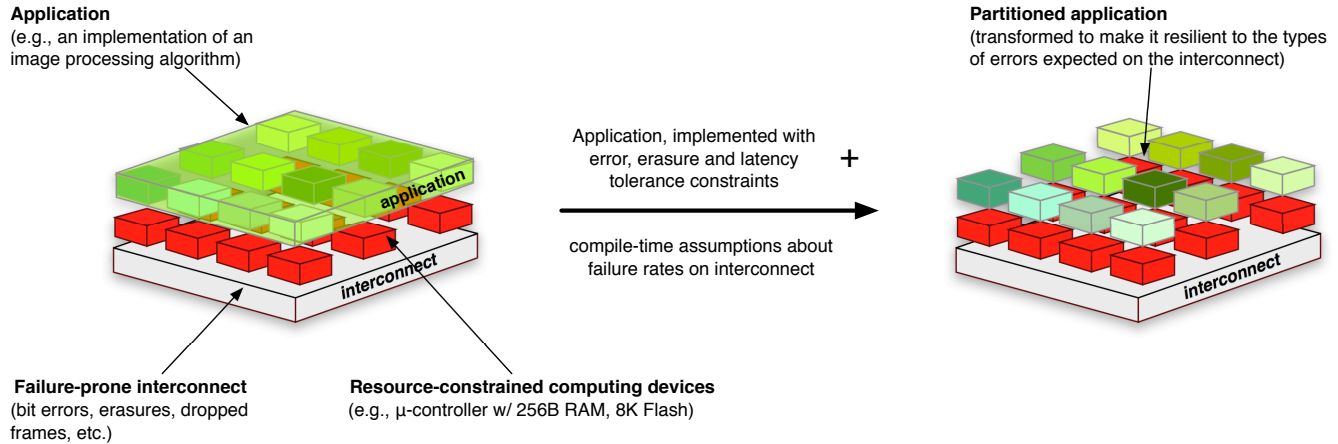


Figure 1. Illustration of the motivating hardware platform of interest in this paper.

mance, low power) and expressive (i.e., pleasant to use by programmers, easily expressing relevant idioms).

This paper introduces a programming model and its implementation, in the form of a small programming language, for systems comprising large numbers of failure-prone elements with constrained resources. The programming model and language implementation are intended to facilitate three primary goals:

1. *Partitioning without replication*, of applications, in order to fit the partitions in devices with very limited memory constraints.
2. (a) *Language constructs for specifying error, erasure and latency tolerance constraints*, and (b) *constraint violation control-flow*. This enables specification of the distribution of numeric errors (deviations) tolerable in individual program variables, and the change of control-flow if these constraints are violated at runtime.
3. *Program transformations that tradeoff performance and reliability* of applications, in the presence of errors and erasures in the underlying hardware.

These goals are of particular relevance in the target application domain of highly resource-constrained embedded systems.

2. The Programming Model

To enable partitioning without replication, programs must be easily broken into distinct pieces that share no state. This contrasts a *single program multiple data (SPMD)* approach, where code is *replicated* on all devices in the system. The reason for our interest in such *pulverization*, is that, we want to be able to reason about algorithms and applications as a single entity, in a high-level language, and subsequently have that application automatically split into a number of disjoint partitions. Each such partition should be able to fit within the memory of our target hardware platforms (of the order of 1KB), and the collection of pieces should work together to implement the desired application, *even in the presence of logic upsets in their computations and in the communications between them*.

The programming model we developed to address these goals is illustrated with the example in Figure 2. Programs are made up of components which we refer to as *name generators*. These can be thought of as analogous to functions, in, say, the C programming language. For example, Figure 2 illustrates an application with three name generators, a, b and c. Unlike functions, which interact by transfer of control-flow (function calls; function returns), name generators interact by communication. Identifiers in programs may represent variables or *channels* on which blocking *send* and *receive* operations may be performed. In this particular sense, name generators are similar to Hoare’s *communicating sequential processes (CSP)*. Section 6 discusses similarities of the proposed model to existing programming models.

Each name generator loaded onto a device in a network, makes visible on the network, an identifier which we refer to as a *name*. These identifiers are analogous to the identifiers representing functions in a program. The collection of these identifiers is referred to as the *runtime name space*. Executing applications may create new identifiers in the runtime name space, with such names tied to variables or channels in programs. For example, in Figure 2, name generator b exposes the channel *ch* and the variable *y* in the runtime name space, using the *chan2name* and *var2name* operators in the language model. These new entries (or any other ones) in the name space can be bound to channels in programs via the *name2chan* operator. Channels in different name generators bound in this manner to the same entry in the runtime name space are effectively connected together.

When the *name2chan* operator is applied to names which represent an implementation of a name generator (such as a, b or c in the example), they cause the activation of a new instance of the name generator, with its own stack / activation record. The channel obtained as the result of such a *name2chan* operation is a communication path to that particular activation of the name generator. Within each name generator, its identifier (e.g., a or b or c in Figure 2) is a valid channel variable, and behaves just as though the identifier were explicitly bound to a name in the runtime name space via *name2chan*. Thus, when read, it blocks until a matching write is performed on a channel tied to the name generator’s identifier, by the entity that caused the name generator to begin executing. It

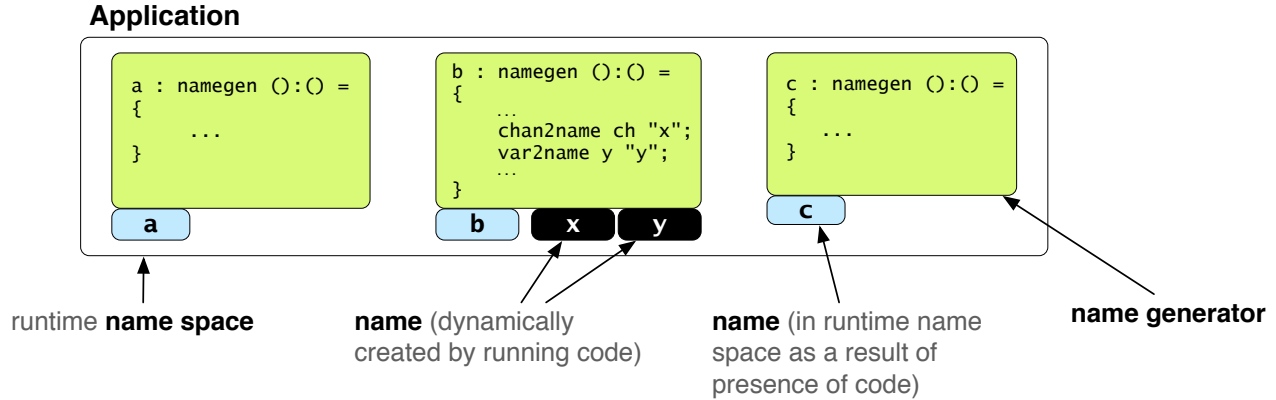


Figure 2. Organization of applications into *name generators* which interact via *names* in a *runtime name space*.

is in this manner that "callers" and "callees" are uniquely linked. The main benefit of these and related constructs in the language model, is to (1) enable the construction of applications with constructs for modular programming and (2) to make the sharing of state between these units of modularity (name generators) explicit, and through a structure that is a runtime entity. This is fundamental to enabling breaking applications into disjoint partitions.

3. Error/erasure/latency tolerance constraints

The programming model provides *tolerance constraints* as part of the type annotation for identifiers (variables and channels). These tolerance constraints may be constraints on *error* (value deviation), *erasure* (or lost communicated values) and *latencies* (on communication operations on channels). We have developed fundamental analytic techniques to enable the transformation of the bit-level layout of variables to satisfy the program-specified constraints on tolerable value deviation [14]. These transformations use as input the error-tolerance constraints, along with compile-time *assumptions* about the environment in which the application will be deployed.

Because it is never possible to perfectly predict the distribution of upsets induced by a particular environment, it is possible that the assumptions made about logic upsets in the deployment environment, used as inputs to an analysis, may be overly optimistic. In such situations, the transformed program may still violate the program-level constraints, despite program transformations performed at compile time to ensure otherwise. For this reason, the language model includes the concept of *tolerance violation control-flow*. These are constructs that enable the re-direction of control-flow, as a result of violations of tolerance constraints. In the remainder of this paper, we will discuss only the error-tolerance (and not the erasure- or latency-tolerance) constraints in more detail.

We term the difference between the correct and erroneous values, as a result of logic upsets in computation or communication, the *error magnitude*. The idea of the error magnitude is illustrated in Figure 3. An *error magnitude tolerance* constraint on a variable in a program specifies the acceptable distribution of error magnitude (value deviation) for that variable. For example, a plain English specification of a simple constraint could be "ensure that

the error magnitude *exceeds 2.0, at most one out of every thousand times* that the variable is read or written". As might be evident from Figure 3, satisfying such a constraint will depend on (1) knowledge of the probability of logic upsets at each bit position, (2) knowledge of the probability distribution of values taken on by program variables¹, and (3) a means for changing the bit-level layout of variables (e.g., by adding redundancy), such that, under known distributions of values, in the presence of logic upsets, the constraints are met.

4. Example

To illustrate the ideas presented thus far, we present a small example program in the language we are implementing, *M*, that realizes a simple image processing algorithm, *edge detection*. This specific example was chosen because its variants are relevant across a variety of domains, from their use in workstation-class applications such as desktop publishing, to embedded applications such as object recognition, object tracking and cluster formation. Since the algorithm processes values obtained from the environment (e.g., images), we can also use it as a vehicle to demonstrate the role of language-level error magnitude tolerance constraints.

```

1 EdgeDetect : proctype
2 {
3   READ    : const true;
4   WRITE   : const false;
5   img_row : namegen (bool, int,
6                   byte epsilon(2.0, 0.01)): (byte);
7   init    : namegen ():(args: list of string);
8 }
9
10
11 init =
12 {
13   x, y      : int;
14   image     : array [64] of chan of
15             (bool, int, byte epsilon(2.0, 0.01));
16
17   # Instantiate several name generators to hold dynamic
18   # data structures across devices on network
19   for (i := 0; i < 64; i++) {
20     image[i] = name2chan img_row "img_row" 4E-6;
21     out_image[i] = name2chan img_row "img_row" 4E-6;
22   }

```

¹For example, if all bits are always set (logic 1), and all logic upsets are always 0 → 1 transitions, then all upsets will always be masked, and the error (value deviation) will always be 0.

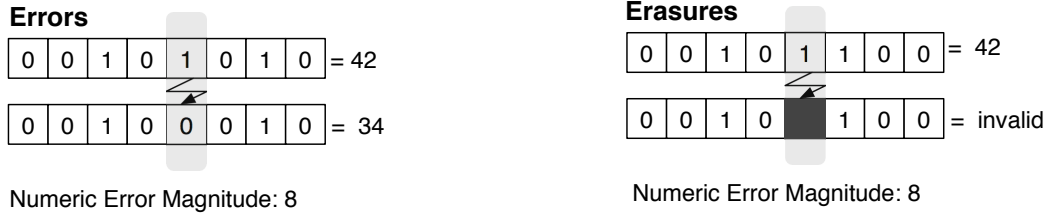


Figure 3. *Numeric error magnitude*, the result of runtime errors and erasures on integer and floating point approximate real values.

```

23
24 # Obtain a channel to a hardware image sensor device
25 # which implements a name generator in hardware.
26 sensor := name2chan S.imgsensor "mem@0xA0FF";
27
28 # Read in image from sensor
29 for (x = 0; x < 64; x++) {
30   for (y = 0; y < 64; y++) {
31     sensor <-= (x, y);
32     image[x] <-= (WRITE, y, <-sensor);
33   }
34 }
35
36 # Now, loop over image and perform convolution
37 for (x = 0; x < 64; x++) {
38   for (y = 0; y < 64; y++) {
39     matchseq {
40       y == 0 || y == 64 => sum = 0;
41       x == 0 || x == 64 => sum = 0;
42     }
43
44     # Core of loop elided for clarity.
45
46     # Write result pixels to output image
47     out_image[x] <-= (WRITE, y, 255 - sum);
48   }
49 }
50 }
51
52
53 img_row =
54 {
55   row := array [64] of byte;
56
57   for (;) match {
58     <-img_row => {
59       (Op, idx, val) := <-img_row;
60       match {
61         op == WRITE => row[idx] = val;
62         op == READ  => img_row <-= row[idx];
63       }
64     }
65   }
66 }

```

Syntactically, programs are collections of implementations of name generators. Such a collection may implement a particular *interface*, called a *program type*. A program type is a unit of modularity that defines a set of types, constants, and name generators. The unit of compilation of programs is a single program type and its implementation. This single complete program input to the compiler is used to generate one or more compiled outputs, corresponding to the pieces of the partitioned application. Partitioning at the level of name generators is straightforward, since they share no state. Due to the structure of the language, it is possible to further partition a single name generator further into smaller pieces. The reason for this ease of partitioning is as follows: any component of a program can be made visible in the runtime name space through constructs provided in the pro-

gramming model; as a result, arbitrary *cuts* can be made in the data-flow graph, *projecting* live variables at a given point into the runtime name space, and projecting entries from the runtime name space back into programs using a complementary set of constructs.

The example above implements the Sobel edge detection algorithm. It begins with the program type definition, `EdgeDetect`, which declares two name generators, `img_row` and `init`. In a system composed of multiple hardware devices, each name generator definition (the code representing the name generator) may reside on a different device, as partitioned at compile time. When instantiated, they execute concurrently.

The syntax of name generator declarations specify the name generator's *read* and *write* types. The read and write types specify the type structure of their channel interfaces when read from and written to respectively. In the example, the `img_row` name generator's read type includes an *error tolerance constraint*, `epsilon(2.0, 0.01)`. This specifies that the program can tolerate deviations in values communicated on the read interface, from their correct values, of magnitude up to 2.0, occurring with probability 0.01.

By convention, the name generator `init` is automatically executed by the runtime system of the device on which it is installed. In the `init` name generator, after a handful of variable declarations, a `for` loop (with the same syntax as in C) is used to create several instances of the `img_row` name generator, via the `name2chan` construct. The `name2chan` operator takes a name (string) and a type, and if there exists an entry in the runtime with an identical name *and* type, yields a *channel*. If the name represents a name generator implementation, a new instance of the name generator is created (i.e., with a private stack), on the device on which the code exists, and the channel will be a link to that instance. In the example, the last term in the `name2chan` expression is a timeout in seconds.

The implementation of the `img_row` name generator defines an array corresponding to a row of an image, and its remainder facilitates reading from and writing to this array. As a result of the loop on line 19 of `init`, there will be 64 rows of 64 pixels, each allocated on a (possibly different) device in the system. As a result of the error tolerance constraint, the compiler may transform the implementation of the program, based on information known about the bit upset distributions on the network connecting together hardware devices, as well as information known about the typical distribution of values taken on by variables of different types, to ensure that the constraint is met.

5. Implementation

We have implemented a preliminary version of a compiler that translates programs written in **M**, into C programs. Although we have developed the necessary analysis to enable them [14], we are still in the process of implementing the program-level transformations to satisfy error-tolerance constraints within the compiler infrastructure.

The compiler is implemented in a manner similar to Niklaus Wirth's Oberon compilers, using a hand-written recursive descent parser to build an intermediate representation (an *abstract syntax tree [AST]*). A subsequent stage walks this in-memory representation to generate ANSI C that can be fed to any C compiler. The compiler can also generate detailed annotated graphs, in postscript form, of its intermediate representation and internal data structures. This aids debugging and extension of its implementation immensely, and also facilitates understanding the structure of input programs.

The decision to implement the compiler using a hand-written parser instead of a Yacc-driven parser, influenced the form of the language's grammar, which had to be ensured to be LL-1. However, a side effect common to all LL-1 recursive-descent parsers is that they can generate significantly better (from the user's perspective) error reports during compilation. We are exploiting this fact to make the compilation framework as appealing to users of the language as possible.

6. Related Research

There has been increasing recent interest in programming models for taking advantage of increased (coarse-grained) hardware parallelism. Transaction-level models for dealing with hardware concurrency [5] are one such area, as are the use of *software transactions*, which advocate a technique for synchronization that is easier to use than locks [4].

Our programming model is influenced by ideas from models such as Hoare's CSP [6] (channels), Actors [1] (name generators are similar to *Actors*, and the use of names is similar to *Actor mail addresses*), the π -calculus [9] (names in our model are analogous names in the π -calculus), Linda [2], and timed CSP [11].

The use of stochastic entities in our work is a departure from traditional uses of probability in programming languages, where the component which is probabilistic is the *behavior* of a computation or a composition of concurrent processes [13]. In this work, it is the *values* on which computation is occurring that may have *stochastic error tolerance bounds* — our inclusion of error-tolerance constraints explicitly encoded in the type structure of variables is a new direction of research. These constraints are a key component of program transformation techniques we have developed to enable application-specific forward-error-correction at the level of program variables. Unlike software-based schemes for fault detection [10, 12], the goal of our analyses is to bound the *numeric magnitude* of the deviations introduced by logic upsets, with the bounds explicitly specified in programs.

7. Summary

This paper outlined a model of computation for failure-prone, resource-constrained concurrent hardware, and its implementation in a small programming language, **M**. The model enables partitioning without replication, of appli-

cations, across multiple devices with constrained memory resources. It enables programs to specify the value deviation tolerable in individual variables, as well as tolerable latencies and erasures on communications. These constraints, together with compile-time assumptions about the logic upset distributions existent in the deployment environment, are used as input to compile-time transformations to enable forward error correction. The language model and implementation include constructs to enable the re-direction of control-flow when the tolerance constraints are violated, due to, e.g., more severe logic upset phenomena than assumed at compile-time.

The ideas presented in this paper are part of a larger effort to build *computationally-animate materials*. To this end, we have also developed hardware emulation environments, hardware prototypes and analysis metrics for quantifying the efficacy of systems in which performance, power consumption, battery lifetime and reliability are all of concern.

References

- [1] G. Agha. An overview of actor languages. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, pages 58-67, New York, NY, USA, 1986. ACM Press.
- [2] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444-458, 1989.
- [3] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1-11, New York, NY, USA, 2003. ACM Press.
- [4] D. Grossman. Software transactions are to concurrency as garbage collection is to memory management. Technical Report 2006-04-01, UW-CSE Technical Report.
- [5] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289-300, New York, NY, USA, 1993. ACM Press.
- [6] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666-677, Aug. 1978.
- [7] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 149-162, New York, NY, USA, 2005. ACM Press.
- [8] T. Liu and M. Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 107-118, New York, NY, USA, 2003. ACM Press.
- [9] R. Milner. *Communicating and Mobile Systems: The π -calculus*. Cambridge University Press, 1999.
- [10] N. Oh, S. Mitra, and E. J. McCluskey. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Trans. Computers*, 51(2):180-199, 2002.
- [11] G. M. Reed and A. W. Roscoe. The timed failures-stability model for csp. *Theoretical Computer Science*, 211:85-127, 1999.
- [12] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 243-254, Washington, DC, USA, 2005. IEEE Computer Society.
- [13] K. Seidel. Probabilistic communicating processes. *Theor. Comput. Sci.*, 152(2):219-249, 1995.
- [14] P. Stanley-Marbell and D. Marculescu. Program-Level Error Tolerance Transformations using Per-Type Analysis of Value Distribution. In *submission, POPL '07: ACM SIGPLAN 2007 conference on Principles of Programming languages*, 2007.