TU/e technische universiteit eindhoven

A Programming Model and Language Implementation for Error-Tolerant Networks of Computation

Phillip Stanley-Marbell

ES Reports
ISSN 1574-9517

ESR-2008-03
24 January 2008

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

People who are really serious about software should make their own hardware. — Alan Kay

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

A Programming Model and Language Implementation for Error-Tolerant Networks of Computation

Phillip Stanley-Marbell
Technische Universiteit Eindhoven
Den Dolech 2, 5612 WB Eindhoven
The Netherlands

ABSTRACT

Many embedded applications involve processing data from noisy analog signals and displaying information for human observation. In such systems, trade-offs often exist between performance, energy usage, and the accuracy of data processing. To achieve the combination of *both* low idle-power consumption and high peak performance often required in embedded systems, there is an increasing trend towards the use of multiple processing elements instead of a single high performance processor. Combined with the presence of correctness-performance trade-offs, this trend provides interesting new programming language challenges.

This paper introduces a programming model, its runtime system, and a language implementation, targeted at systems containing multiple resource-constrained processors which may be able to trade off performance or power consumption, for correctness and timeliness of computation. The programming model makes it straightforward to achieve partitioning of program implementations across the code storage memories of multiple resource-constrained processors, and introduces the idea of program-level constraints that permit compile- and run-time trade-offs between performance and correctness to be exercised. The ideas are presented in the context of two different hardware platforms we developed that benefit from the facilities of the programming model.

1. INTRODUCTION

With continually declining semiconductor device costs, there is an ongoing shift towards the use of programmable elements as building blocks of systems, in much the same way as logic gates formed the basis of traditional digital systems. This trend has particularly been on the rise in embedded systems, where requirements such as short design cycles, low cost, performance adaptability, and reliability, have all made the use of multiple microcontroller designs a natural choice.

Besides integrated systems containing multiple processing elements, there have also been an increasing number of applications which employ multiple separate computing systems, each of which contains one or more processing elements. One such example is the area of *sensor networks*, in which multiple *sensor node platforms* coordinate to achieve some task, such as monitoring temperature and air quality in a building and raising an alarm in the case of a detected fire.

Quite often, embedded applications involve processing data from noisy analog signal sources such as sensors, or displaying the results of computations solely for human observation. In such systems, trade-offs often exist between performance, energy usage and the accuracy of data processing. For example, data representing pixel values destined for a graphical display, might incur errors in individual bits or complete bytes, with no adverse effect on system utility. On the other hand, in applications such as anti-lock braking systems, such data corruption is not tolerable. It therefore desirable for the manner and occasions on which correctness constraints may be relaxed, to be explicitly specifiable.

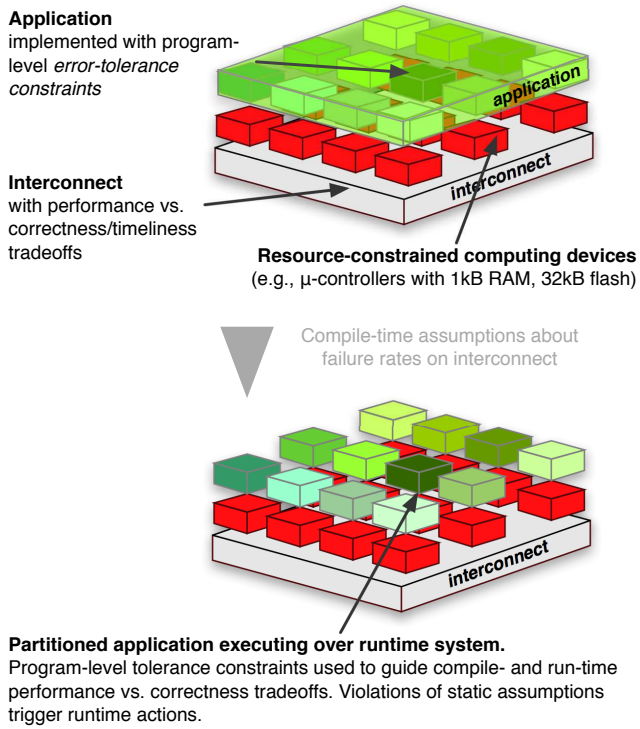


Figure 1: Illustration of the motivating hardware platforms of interest in this paper.

This paper introduces a programming model, its runtime system, and a language implementation, targeted at systems containing multiple resource-constrained processors employed in the execution of a single application, and which may be able to trade-off performance or power consumption for correctness and timeliness of computation. An illustrative organization of such platforms is depicted in Figure 1. The programming model introduces the idea of program-level *value deviation constraints*, inter-module *communication latency constraints*, and inter-module *communication failure tol-*

erance constraints; we refer to these constraints collectively as *language-level error-tolerance constraints*. The programming model and language implementation are intended to facilitate three primary goals:

1. *Partitioning without replication*, of applications, over a network of computing devices, in order to fit the partitions in devices with very limited memory resources.
2. *Language constructs for specifying value deviation, value loss, and latency-tolerance constraints*, and the change of program control flow when these constraints are violated.
3. *Program transformations that trade off performance for reliability or correctness*.

This paper addresses the first two of these goals. We present these ideas in the context of two different hardware platforms we developed, one being a low-power, performance-scalable, 24-microcontroller module, and the other a mobile computing platform that employs four embedded processors in its implementation.

1.1 Contributions

The specific contributions of this paper include:

- The concepts of *value deviation*, *value loss* and *latency tolerance* constraints in a programming language.
- A programming model which provides a natural means of expressing the language-level tolerance constraints in the context of applications executing over multiple resource-constrained programmable elements.
- A runtime system to support the programming model.
- A concrete language design implementing the programming model.
- A demonstration of concrete motivating hardware platforms that benefit from the programming model, runtime system, and programming language.
- Presentation of a minimal core language (a simply-typed lambda calculus with deviation-tolerant types) that captures the ideas of error-tolerances on typed program variables.

In the following section, we present an overview of two hardware platforms we developed as part of our effort to study the benefits of error-tolerance trade-offs. These platforms provide concrete examples of hardware in which it is beneficial to employ a large number of programmable elements, and in which it is possible to trade-off performance and power consumption for correctness. Section 3 introduces the programming model. The implementation of a language which implements the programming model, and which has constructs for value deviation, latency and erasure-tolerance constraints, is presented in Section 4, through an example. It is followed in Section 5 with an overview of the runtime system necessary to support the programming model and language implementation. Preliminary measurements demonstrating performance versus correctness tradeoffs in hardware are presented in Section 6, alongside a discussion of challenges such as type-checking in the presence of tolerance constraints on variables. Section 7 presents an overview of relevant related research, and the paper concludes in Section 8 with a summary and pointers to future research directions.

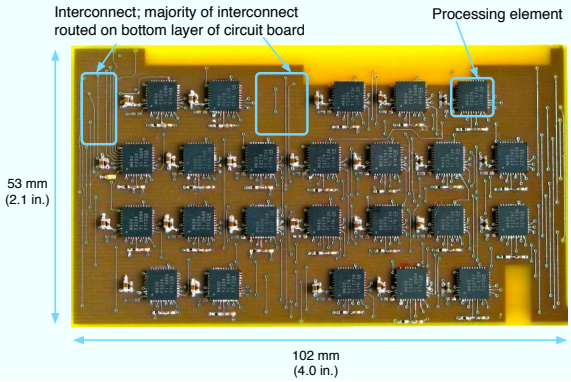


Figure 2: A 24-microcontroller system. Taking full advantage of its capabilities and trade off opportunities requires appropriate language and runtime system support.

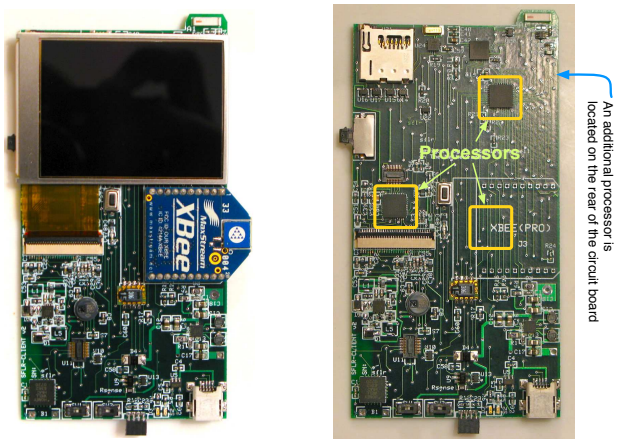
2. MOTIVATING HARDWARE

To motivate and provide a concrete basis for the discussions in the remainder of the paper, we briefly describe two hardware platforms we developed to highlight applications of the programming model and language implementation presented in this paper.

2.1 A scalable low-power multiprocessor

One challenge faced by many embedded systems, is the opposing constraints of ultra-low power dissipation when idle, and the availability of adequate program storage, runtime memory and computing resources when needed. These requirements are in conflict, since embedded processors with more sophisticated peripherals and larger program memories typically have higher idle power dissipation. This is due in part to the larger number of transistors needed to implement more functionality, as well as the large transistor cost of the static RAM (SRAM) typically employed in such on-chip memories. One solution to this problem is the use of systems comprising several microcontrollers, each of which can be powered down when unneeded. In this case, applications must now be partitioned for execution on such networks of processing elements. The data exchanged between the program partitions is carried over an interconnect network, and in some communication architectures, it is possible to trade-off communication data rates for the rate of bit-errors in the exchanged data.

Figure 2 illustrates such a platform, comprising 24 ultra-low-power microcontrollers interconnected in a communication network. The platform has power consumption over an order of magnitude smaller than a state-of-the-art low-power ARM microcontroller [2] with equivalent peak performance; it however requires appropriate programming support to take full advantage of its capabilities. The network topology employed, a Kautz network [10] topology chosen for its performance and redundancy properties, admits multiple techniques for forwarding data between non-adjacent nodes (routing). The choice of routing method enables a trade-off between communication performance and power



(a) Handheld platform employing four processors.

(b) With display removed, highlighting locations of three of four processors.

Figure 3: A handheld platform employing multiple processing elements.

consumption. The firmware in the microcontrollers implements the different routing schemes with low overhead, and may also adapt the bit-rate over a single hop. While high data rates may generally be perceived as desirable, electrical limitations lead to bit errors at very high data rates, yielding yet another trade-off between high speed and possible bit-level errors in communicated data. We are using this platform as a low-power but high-peak performance processor module for embedded systems.

2.2 A mobile multi-microcontroller device

Another example of a hardware platform in which there are opportunities to trade off performance for correctness in the presence of multiple processing elements, is the mobile computing device shown in Figure 3. Of relevance to this paper, the platform employs four microcontrollers — one for system control functions, one for display processing, one implementing the wireless communication medium access control (MAC) protocol, and a fourth for compute-intensive tasks. In this platform, the processors are connected in a "star" topology, centered on the system controller. Communications to and from the display controller can be configured at different data rates, with the highest data rates increasing the chance of bit-errors. Communications with the radio interface also exhibit these trade-offs, and in addition, the system controller may buffer data destined for the wireless communication interface. This increases latency, but enables more effective use of the radio interface, which constitutes a large fraction of the system's power consumption when active. The platform is being employed to investigate the construction of *client* platforms for sensor networks which can be used in applications that require very long battery life (e.g., weeks) off a single charge.

2.3 Motivating observations

The foregoing hardware platforms are concrete examples of systems built out of a network of resource-constrained processing elements, for the express reasons of performance and energy-efficiency. The platforms also highlight the challenges that programming such hardware may pose, and possible trade-offs between performance and correctness that may exist in real systems. In particular, the following observations can be made:

- There are concrete practical advantages to building systems employing multiple low-power processing devices; appropriate programming models are however necessary to take advantage of such hardware platforms.
- Platforms may exhibit trade-offs between performance or energy-efficiency, and correctness. Examples of trade-offs are between communication performance and errors in communicated data, and between latency of communications and the energy cost of communications.

The remainder of this paper presents a programming model, runtime system, and language implementation designed to take advantage of opportunities such as those observed above.

3. THE PROGRAMMING MODEL

In this section, a programming model to enable the partitioning of applications across networks of resource-constrained devices, as well as the expression of application tolerance constraints to enable performance versus correctness tradeoffs, is presented. The programming model is independent of an actual implementation, and may be implemented as a library, or with primitives built into a programming language. In Section 4, we present an example application in a language implementing the programming model.

3.1 Model overview

The underlying idea in the programming model is the concept of *name generators*. A name generator is a self-contained collection of program statements, analogous to functions or procedures in Algol family languages. Each name generator is represented with a *name* in a runtime *name space* which is facilitated by the runtime system. Unlike functions and procedures which interact by explicit transfer of control flow in function and procedure calls, name generators interact by *communication* on these names.

Name generators, as their eponyms imply, may also *generate* new entries or names in the runtime name space. Names have associated basic or structured *types*, analogous to types in contemporary programming languages, with the addition that basic types therein may include type modifiers which add tolerance constraints; the nature of these tolerance constraints are discussed in detail in Section 3.7. Names are represented within name generators as *channels*. Channels are constructs on which blocking read (write) operations may be performed. The operations complete when a matching write (read) is performed on the same instance of the construct elsewhere in the program; their use in a programming model thus implies concurrency. Channels as a programming model and language construct are inherited from Hoare's CSP [8].

Channels may be made visible as names in the runtime name space and vice versa. A small set of operations can be performed on names (and the channels that represent them), and these operations form the basis for execution of name generators (equivalent to "calling" of functions) and communication between name generators. Figure 4 illustrates the

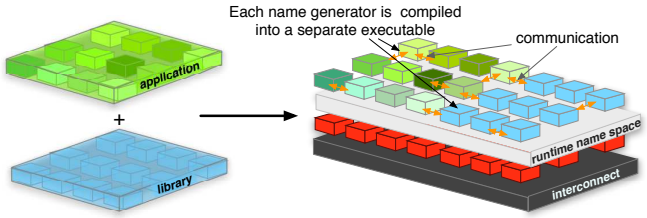


Figure 4: Illustration of the organization of applications into *name generators* and the mapping of these to a network of processing elements.

Table 1: The basic operators on names

Operator	Description
`name2chan`	Bind name in runtime system to a channel in program
`chan2name`	Make a channel in a program visible in runtime system
`nameread`	Read a name via the channel bound to it
`namewrite`	Write to name via the channel bound to it

organization of applications into name generators and the mapping of these to a network of processing elements. In what follows, the implementation of the runtime system will be treated abstractly, until Section 5, where we detail the data structures necessary for its realization.

We will refer to name generators that are not in execution as *latent name generators*; each such executable module is visible in the runtime system as a name. The behavior of executing applications consists of sequences of statement executions acting on machine state, and operations on names for interacting with other collections of code such as libraries, or performing system calls. The set of operations that may be performed on names is listed in Table 1, and the following sections elaborate on their semantics.

3.2 The operator `chan2name`

The operator `chan2name` takes a channel within a program and makes it visible within the runtime name space. The implementation of channels might be achieved as a data structure within an existing programming language, or, as we present in Section 4, as a primitive within the programming language. The type associated with an entry in the runtime name space reflects the structure of the corresponding channel from which it was created. In the implementation we present in Section 4, the type of the generated entry in the runtime name space is identical to that of the language-level channel.

3.3 The operator `name2chan`

The operator `name2chan` is the most fundamental operator in the programming model. It operates on a name, and yields as its result a channel. Names in the runtime system may correspond to channels in executing name generators, made visible in the runtime system via `chan2name`, and in that case `name2chan` achieves the connection of channels in two executing name generators.

As described previously, names may however also represent latent name generators, i.e., executable collections of program statements, analogous to functions or procedures.

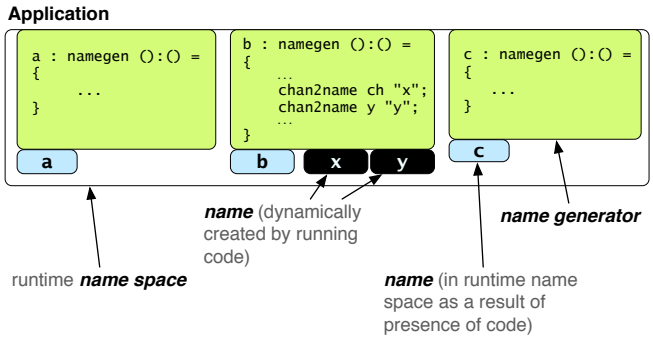


Figure 5: A simple name generator example.

A `name2chan` operation on such a latent name generator initiates its execution, and yields a channel that can be read from or written to, to communicate with the initiated name generator. The type associated with such a channel corresponds to the type or *signature* associated with the name generator. As described in more detail in Section 5, each such instantiation of a name generator is analogous to an activation frame of a function, and the runtime system maintains the necessary data structures to enable multiple or recursive instantiations of name generators.

3.4 The operators `nameread` and `namewrite`

The `nameread` operator performs a synchronous (blocking) read on an entry in the runtime name space, through a channel associated with the name; it yields a value whose type is that of the name being read. Similarly, the operator `namewrite` performs a blocking write to a channel associated with a given name in the runtime name space.

3.5 A simple name generator example

Figure 5 illustrates an application composed of three name generators, `a`, `b` and `c`. The syntax in the figure is not specific to the programming model, but hints at the language implementation that will be presented in Section 4. Each name generator loaded onto an execution platform is visible in the runtime system via a name, and thus on the runtime system on which the application of Figure 5 is loaded, the names `a`, `b` and `c` are visible. Information maintained by the runtime system (Section 5) enables a hardware substrate to support multiple logical collections of such name generators, i.e., multiple applications.

In Figure 5, name generator `b` exposes the channels `ch` and `y` as the names `x` and `y` in the runtime name space, using the `chan2name` operator. 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, and data written to one channel can be read from the other.

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 and 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`, `b` or `c` in Figure 5) is a valid channel that behaves just as though the identifier were ex-

Table 2: Basic types that may be associated with names.

Type Name	Description
`bool`	1-bit value
`nybble`	4-bit unsigned value
`byte`	8-bit unsigned value
`string`	Vector of 16-bit Unicode values
`int`	32-bit signed, two's complement format
`real`	64-bit double precision, IEEE-754 format
`fixed`	16-bit fixed point

Table 3: Structured types (type collections).

Type Collection	Description
`array`	Vector of items of a single type
`adt`	Aggregate data type
Tuple	Unnamed `adt`. An unnamed collection of items of possibly-different type
`set`	Unordered collection of data items
`list`	Recursive list

plicitly 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 is in this manner that "callers" and "callee's" are uniquely linked.

If this application were compiled for execution on one of the platforms illustrated in Section 2, each name generator might be loaded into the memory of a different processing element. The runtime system, which is system software present on all the processors in the platform, facilitates the generation and delivery of necessary communications between processors.

3.6 Types

Names in the runtime system have *types* associated with them. Like variables in a programming language, these types can be basic types or structured types. The set of basic types in the name generator model is listed in Table 2. The basic types may have *error-tolerance type modifiers* associated with them, and these are discussed separately in more detail in Section 3.7. The base types may also be used to form aggregate types through a set of type collections or structured types, shown in Table 3.

These types have relevance in two parts of a system implementing the name generator model — in the runtime system and in the programming language. Within the runtime system, types are represented as bit-vectors or strings. In our current implementation, canonical representations of the structure of aggregate types are obtained by a post-order walk of the parse tree for an aggregate type as it would appear in a language. These strings are represented in the runtime system as literal ASCII-encoded strings, and an alternate implementation may employ a more compact binary encoding thereof.

3.7 Error-tolerance constraints on name and channel types

As highlighted in Section 2, many hardware platforms, as well as the applications executing on them, may have the ability to trade off performance or energy-efficiency for some notion of *correctness*. In the name generator model, the precise nature of such correctness constraints are constraints on *value deviations* on the values taken on by data items sent on channels, *latencies on channel or name operations*, and *losses (erasures*[1]*) on channel operations*. By associating such constraints with entries in the runtime name space, it is possible to adapt the interaction between portions of a program, partitioned across a hardware substrate, to take into consideration the trade-offs that the program explicitly permits. The following provide a more precise definition of the nature of these tolerance constraints.

3.7.1 Channel value deviation constraints

A value deviation constraint on a channel `c`, is a list of expressions of the form

$$\texttt{epsilon}(m_1,\ A_1), \ldots, \texttt{epsilon}(m_n,\ A_n),$$

applied to arithmetic members of the channel type, which specifies the constraint that the *numeric deviation* incurred in values transmitted over the channel should exceed value m no more than a fraction A of the times the value is communicated. In other words, the *time average* fraction of reads from, or writes to the channel, yielding deviation greater than m, should be less than A. An alternative definition of this construct could have been in terms of what is known as the *ensemble-average*, denoting, in the general use of the term, the average across all occurrences of a given event. Such a definition is more difficult (and not always meaningful) to associate with a single value in a program — it is more reasonable to think of what a single value or variable does over time[2].

3.7.2 Channel latency tolerance constraints

In addition to a value deviation constraint, channels may have constraints on their tolerable latency. This exposes to programs the fact that the interconnect underlying an implementation of the programming model and runtime system is not an idealized system, and communications have an associated delay. A latency tolerance constraint on a channel `c`, is a list of expressions of the form

$$\texttt{tau}(m_1,\ A_1), \ldots, \texttt{tau}(m_n,\ A_n),$$

which specifies the constraint that the *latency in microseconds* incurred on a channel read or write operation on `c`, should exceed value m no more than a fraction A of the times the channel is accessed. Such channel latency tolerance constraints expose to the runtime system the fact that a given operation may be delayed for a given amount of time, without violating the semantics of the application.

3.7.3 Channel erasure tolerance constraints

It is possible that the underlying interconnect supporting the exchange of data between name generators may fail to deliver a datum exchanged between two name generators, or might *wish* to not deliver such data. The reasons for this might be, e.g., to conserve energy resources, or to satisfy the timing constraint of *another* name generator. An erasure tolerance constraint on a channel `c`, is a list of expressions of the

[1]The term erasure is borrowed from communication theory, where it refers to ostensibly missing data in a data stream.

[2]In *ergodic* systems, in which each state may be visited infinitely often, the time- and ensemble-averages are by definition the same.

form

$$\texttt{alpha}(m_1,\ A_1), \ldots, \texttt{alpha}(m_n,\ A_n),$$

which specify the constraint that the *number of failed or discarded transactions* occurring on a channel communication on `c`, should exceed value m failures per second, no more than a fraction A of the time.

3.8 Consequences of the model

Since all interactions between portions of an application are through the abstraction of names, the components making up an application can easily be placed on different processing elements, and the connection between application portions is achieved by the runtime system. Naturally, the component name generators of an application might be mapped to the same processing element if it has sufficient memory resources. While analogies might be drawn between this partitioning and parallelization of programs, it is important to note that in this case, the goal might not be to achieve greater performance (even though that might be a side effect), but rather to fit an application on a collection of resource-constrained processing elements. For example, in the hardware platform presented in Section 2.1, each of the 24 processing elements has only 32 KB of flash code storage and 1 KB of RAM, and the implementation of applications with larger code and memory footprints is facilitated by mapping their component name generators to different processing elements.

The tolerance of applications to three kinds of errors — deviation in values communicated between program modules, latencies of communications and missing data items in communications, are made visible through type information associated with entries in the runtime system supporting the programming model. In the following section, we illustrate a concrete language implementing the name generator programming model, through an example.

4. M: A NAME GENERATOR LANGUAGE

To illustrate the ideas presented thus far, we present a small example program (Figure 6) in a language, **M**, which implements the name generator programming model. The example 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. 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-tolerance constraints.

Syntactically, programs in **M** 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, and 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 is as follows: any component

```
1  EdgeDetect : progtype
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 init =
11 {
12    x, y        : int;
13    image       : array [64] of chan of
14                  (bool, int, byte epsilon(2.0, 0.01));
15
16    #    Instantiate name generators to hold dynamic
17    #    data structures across devices on network
18    for (i := 0; i < 64; i++) {
19       image[i] = name2chan img_row "img_row" 4E-6;
20       out_image[i] = name2chan img_row "img_row" 4E-6;
21    }
22
23    #    Obtain channel to a hardware image sensor
24    #    device which implements a name generator in HW
25    sensor := name2chan S.imgsensor "mem@0xA0FF";
26
27    #    Read in image from sensor
28    for (x = 0; x < 64; x++) {
29       for (y = 0; y < 64; y++) {
30          sensor <-= (x, y);
31          image[x] <-= (WRITE, y, <-sensor);
32       }
33    }
34
35    #   Now, loop over image and perform convolution
36    for (x = 0; x < 64; x++) {
37       for (y = 0; y < 64; y++) {
38          matchseq {
39             y == 0 || y == 64   => sum = 0;
40             x == 0 || x == 64   => sum = 0;
41          }
42
43          #   Core of loop elided for clarity.
44
45          #   Write result pixels to output image
46          out_image[x] <-= (WRITE, y, 255 - sum);
47       }
48    }
49 }
50
51 img_row =
52 {
53    row := array [64] of byte;
54
55    for (;;) match {
56       <-img_row   => {
57          (op, idx, val) := <-img_row;
58          match {
59                op == WRITE   => row[idx] = val;
60                op == READ    => img_row <-= row[idx];
61          }
62       }
63    }
64 }
```

Figure 6: Illustrative example of an application implemented in the M language.

of a program can be made visible in the runtime name space through constructs provided in the programming model; as a result, arbitrary *cuts* can be made in the data-flow graph, *projecting* live variables and channels 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 in Figure 6 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 rep-

Name Table for name generators

name	type	PC

Activation Record (AR) Table for name generators

ID	name generator AR

Channel Table for channels

channel ID	name gen. ID	name gen. channel index	rendezvous	timer	xform	remote address	remote name gen. ID	remote channel index

Figure 7: Structures underlying the implementation of the runtime system.

resenting 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 the channels resulting from their instantiation, when read from, and written to, respectively. In the example, the `img_row` name generator's read type includes a value deviation 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 an average one out of every hundred communications on the channel.

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 the C programming language) 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 executing 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 18 of `init`, there will be 64 rows of 64 pixels, each allocated on a (possibly different) device in the system.

5. THE RUNTIME SYSTEM

A collection of data structures supports the underlying operations being performed by executing programs. At the heart of the runtime system implementation is a set of three tables maintained on each processing element in a hardware platform: the *name table*, *activation record table* and the *channel table*, illustrated in Figure 7.

5.1 The name table

The name table contains an entry for each name generator installed on a device, along with an entry representing the type structure of the name generator. The `name` entries are strings representing the name generator, qualified by the program type (`progtype` type in the **M** language implementation) of which they are part. At runtime, new entries are added to the name table whenever an executing name generator performs a `chan2name` operation, as well as whenever new code is installed on a device. By convention, a name generator with the identifier `init` immediately begins executing once loaded. Loading an application implementation with the same `progtype` as an extant one, into the runtime system (e.g., on a different device) is equivalent to overwriting code memory of a running application in a traditional program.

When a name generator performs a `name2chan` operation, the local name table is first consulted. If no matching name and type is found locally, the name tables of all devices in the network are consulted[3]. If such an operation is successful, i.e., the name and type match an entry in the local name table, a new instance of the name generator (based on the program counter (PC) entry) begins executing, with its own private stack. Such an instance is termed a name generator *activation*. The state for currently instantiated name generators is maintained in the *activation record table*.

5.2 The activation record table

The activation record table maintains the state corresponding to each name generator instantiation (as created by a `name2chan`, or an `init` name generator). The `ID` field uniquely identifies an instantiated name generator, and is used to identify name generators for all other operations. For example, all channels are associated with a particular name generator instance, and the instance's identifier is used to track this correspondence. An instantiation of a name generator may create new entries in the runtime name space, associated with variables or channels in the name generator; these are only visible to the name generator that caused their instantiation (and to themselves). Such dynamically created entries are tracked in the *channel table*.

5.3 The channel table

The channel table contains entries for all channels associated with names in the runtime name space. A name generator that performs a send or receive operation on a channel sleeps on a *rendezvous structure* in the channel table. When the channel communication operation completes (e.g., message successfully transmitted over network and an acknowledgment received), the sleeping name generator is woken. The `xform` field contains a matrix (logically a part of the channel's type) representing the transformations that must be used to encode and decode the data exchanged between devices, and is derived from the channel error-tolerance constraints. The messages on the network which are generated as a result of operations on channels are described in the next section.

5.4 Name communication protocol

A small alphabet of messages may be exchanged between devices as a result of language-level constructs related to channels. The list of messages in this alphabet is provided in Table 4. The following details the effect of the receipt of messages in Table 4, on the runtime system data structures, and on execution at the recipient node.

[3]Logically, the query is a broadcast, but an implementation may perform any number of optimizations to make this lookup more efficient.

Table 4: Name communication protocol.

Message	Description	Associated M Language Construct	Parameters
`Tname2chan`	Bind name to channel; if name is a name gen. create instance	`name2chan`	name, type
`Rname2chan`	Response: channel index or nil		
`Tnameread`	Channel receive	Channel receive expression (`<-c`)	channel ID
`Rnameread`	Response: type structured data		
`Tnamewrite`	Channel send	Channel send expression (`c <-=`)	data, channel ID
`Rnamewrite`	Acknowledgment		

5.4.1 `Tname2chan`

Execution of a `name2chan` expression in a name generator will initiate the generation of a `Tname2chan` message on the network. A device which contains a matching entry (on both name and type) in its name table responds with a `Rname2chan` message. It is possible that no such device might exist, in which case the language-level expression will evaluate to a null value after the language-level-specified timeout (e.g., as in lines 19 and 20 in Figure 6).

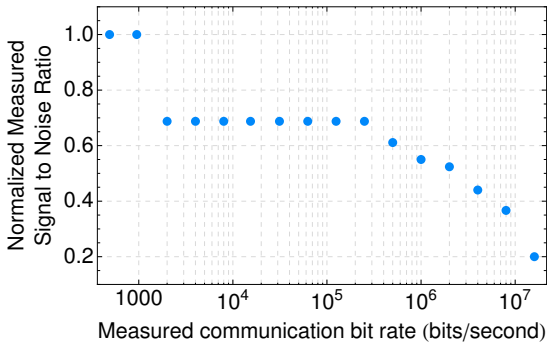
If the supplied name, in the name space, represents a latent name generator, a new activation of the remote name generator is created, and corresponding entries are created for the send and receive interface channel tuple in the remote device's channel table. The index of the allocated entry in the remote name table is returned to the initiating device in a `Rname2chan` message. A new entry is created in the local channel table, and this entry is used to store the received identifier (in its `remote name generator ID` field). The entry also stores the address of the device on which the remote name generator exists, in the `remote address` field. Subsequent operations on the channel associated with this `name2chan` operation will occur with the specific instantiation of the remote name generator.

5.4.2 `Tnamewrite`, `Tnameread`

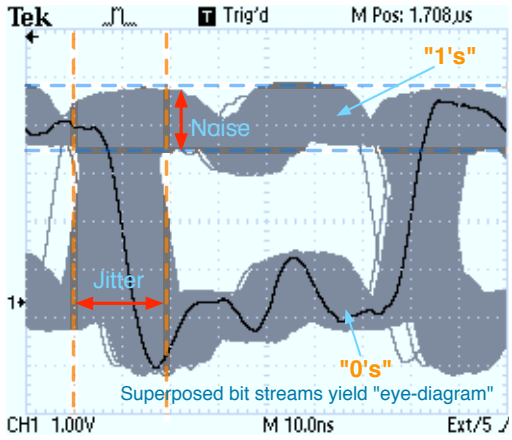
A channel send operation causes a `Tnamewrite` message to be generated on the network. The message target is determined by a lookup in the local channel table for (1) the destination network address (in the `remote address` field), (2) the destination name generator identifier, and (3) the destination name generator channel index (in the `remote channel index` field). The latter identifies a channel in a specific instance of the remote name generator that the values should be delivered to. The `timer` field of an entry in the local channel table is updated with a timestamp, which is used to determine timeouts. An equivalent set of operations occurs for `Tnameread` and `Rnameread` messages.

6. DISCUSSION

The trade-offs that may be exercised as a result of error-tolerance constraints in name generator programs, will depend on the nature of the applications (e.g., how many constraints they impose, how loose those constraints are, and so on). In this section, we evaluate the performance of basic channel communication primitives in the programming model on the hardware platforms presented in Section 2, to provide insight to the possible energy and performance benefits of the tolerance constraints in the name generator programming model.



(a) Normalized SNR for data rates from 490 b/s to 16 Mb/s



(b) Example eye-diagram hardware measurement for communication at 16 Mb/s.

Figure 8: Measured normalized *signal to noise ratio, (SNR)* **(a predictor of bit-error rate) as a function of communication bit rate, for a representative link from the 24-processor hardware platform of Figure 2.1 ((a), top). An example of the** *eye-diagram* **hardware measurement from which the SNR values were computed is shown in (b).**

Figure 8 illustrates the tradeoffs that exist in the platform from Section 2.1, between the communication speed and the signal to noise ratio, an indicator of the likelihood of bit errors. The data in the figure was obtained by transmitting a data stream over a representative link in the interconnect of the 24-processor platform, and characterizing the separation between high ("1") and low ("0") logic values using an eye-diagram. An eye-diagram measurement (Figure 8(b)) captures the superposition of all logic levels in a transmitted data stream, and is often used to characterize the noise-immunity of a communication channel. From Figure 8(a), we see that there is a clear trade-off between data rate and likelihood of bit errors.

Despite our promising experiences with the name generator model and its implementations, there are open questions that remain to be answered. One such question is the flexibility afforded by the current method for specifying error-

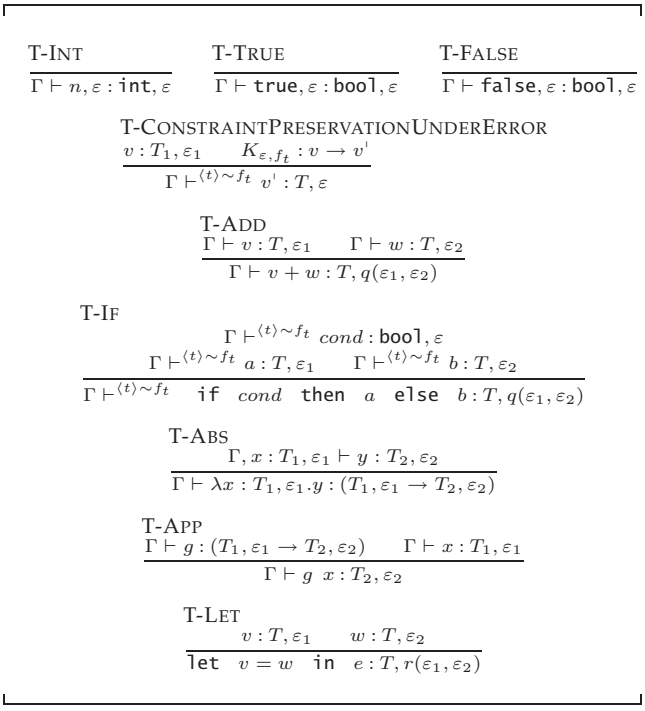


Figure 9: Type inference rules for a simply-typed lambda calculus with deviation-tolerances in type annotation.

tolerance constraints. The most flexible form in which constraints specifying tolerable value deviation could be provided, would be as a *tail distribution* on value deviation. For example, it might be desirable that the probability of value deviation in a variable being greater than x should vary as $\frac{1}{x}$. The notational complexity of representing such value deviation-tolerance constraints would be significant — since it is logical for the deviation-tolerance constraint to be placed in the type annotation, the type would then need to contain an expression in a variable (x in the above example). This approach is therefore avoided. Experience with the language and deviation-tolerance constructs may necessitate revisiting this restriction.

As an illustration of the idea of error-tolerance constraints in a program's type annotation, consider a small core language, a typed lambda calculus, λ_ε, in which the type ascriptions have deviation-tolerance constraints. The type inference rules for λ_ε are shown in Figure 9. The first three inference rules are straightforward. For example, the value `true` with error-tolerance constraint ε has type `bool`, with type error-tolerance constraint ε. The fourth type inference rule, T-CONSTRAINTPRESERVATIONUNDERERROR is the key component that captures the notion of error-tolerance transformations on programs. The concept it embodies is that, if v has type T and error-tolerance constraint ε, and K_{ε,f_t} is a transformation that takes as parameters the error-tolerance constraint ε and a set of assumptions about the hardware f_t, and transforms v to give v', then under the occurrence of a error conditions $\langle t \rangle$ which fall within the assumptions f_t, v' obeys the type and error-tolerance constraint ascriptions of v.

7. RELATED RESEARCH

There have been several proposals for domain-specific languages targeting a variety of issues relating to resource-constrained embedded systems. The nesC language [5], for example provides a programming model and language primitives that are a good match for event-driven systems, such as the TinyOS operating system in which it is employed. While nesC provides what one might refer to as *node-level programming*, other recent proposals such as SpatialViews [15], Kairos [7], Pleiades [11] and Regiment [14] target *network-level macroprogramming*, treating a collection of embedded systems as a single programmable substrate. In contrast to these existing programming models and language implementations, the ideas presented in this paper are targeted at general-purpose embedded system platforms containing multiple resource-constrained processors. The name generator programming model that we introduce is focused on enabling the straightforward partitioning of single applications for such multi-processor systems, and to enable programs to expose their tolerance to runtime faults of various types, which might be manifest after they have been so partitioned.

The observation that different portions of programs, or of hardware, may require differing amounts of fault-protection, has previously been applied to hardware systems, and recently, to phases of programs [18]. Our treatment in this paper of *per-variable deviation-tolerance constraints* is the first to expose such constraints within the programming language. There have recently been attempts to formalize the effects of soft-errors on the behavior of programs [21]. In [21], the model addressed is one in which the goal is to attempt to nullify the effect of soft-errors (faults), by redundant computation — this is a different idea from our goal of *bounding the value deviation* caused by faults.

One early description of a language structure to describe concurrency is Hoare's Communicating Sequential Processes (CSP) [8]. Components of a CSP program, rather than interacting by transfer of control flow, interact by communication over shared references called *channels*. As Hoare points out, the shared references in CSP are fixed, and there is no way to create new shared references at runtime. Languages based on, or influenced by CSP, such as Occam [12], Newsqueak [16], Alef [22] and Limbo [23], although including language-level channels, still require that to communicate on a channel, a process must already hold a reference to it.

The programming model that was introduced in Section 4 is influenced by ideas from models such as Hoare's CSP [8] (channels), Actors [1] (*name generators* are similar to actors, and the use of names is similar to Actor *mail addresses*), the π-calculus [13] (names are like names in the π-calculus), Linda [3], and timed CSP [17]. Like in CSP and Actors, the interaction between name generators is not by transfer of control flow, but rather by communication. By explicitly exposing the interaction between components of programs as communication, we can apply correctness-performance trade-off analyses, not just to the values of variables in programs, but also to their *interactions*. Like in CSP, but unlike in dataflow machines and in Actors, the communication between the name generators introduced in Section 4, is synchronous, rendezvous. Unlike Actors, name generators are sequential processes, in much the same manner as processes in CSP [8]; concurrency arises from the composition of these

sequential processes.

There have been previous attempts at introducing constructs in programming languages to enable fine-grain parallelization as in Jade [19] and in theoretical studies under the ideas of *closure conversion* [20]. Languages built around the ideas of streaming architectures, such as Brook [9] and StreaMIT [6] hold as an underlying *concept* the construction of programs as *filters* what act on *streams*. Yet other language proposals such as Sing# from the Singularity system [4] provide language-level abstractions for fast messaging. Unlike in this work, they are concerned with obtaining high performance from reliable hardware substrates.

8. SUMMARY AND FUTURE RESEARCH DIRECTIONS

The programming model, language implementation and runtime system presented in this paper, are, to our knowledge, the first to expose the idea of *tolerance to errors* at the programming language level. By employing the programming model abstraction of *name generators*, we facilitate the partitioning of applications across networks of processing elements, in which the communications between processors may be prone to failures, or in which applications may be able to trade off correctness of execution for application performance, or performance for energy efficiency.

We are developing applications for the hardware platforms described in Section 2, using an implementation of the language **M**, illustrated in Section 4, which implements the name generator programming model. Our immediate activities are focused on more detailed empirical evaluations of these applications, to provide quantitative evidence of the benefits of the performance, energy and correctness tradeoffs enabled by the programming model.

9. 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] Atmel, Inc. Datasheet, AT91 ARM Thumb-Based Microcontrollers. 2006.

[3] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[4] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.

[5] 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.

[6] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, New York, NY, USA, 2002. ACM Press.

[7] R. Gummadi, N. Kothari, R. Govindan, and T. Millstein. Kairos: a macro-programming system for wireless sensor networks. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2, New York, NY, USA, 2005. ACM.

[8] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[9] M. Horowitz, P. Hanrahan, B.Mark, I. Buck, B. Dally, B. Serebrin, U. Kapasi, and L. Hammond. Brook: A Streaming Programming Language. 2001.

[10] W. H. Kautz. Bounds on directed (d,k) graphs. *Theory of cellular logic networks and machines*, AFCRL-68-0668 Final report:20–28, 1968.

[11] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 200–210, New York, NY, USA, 2007. ACM.

[12] D. May. Occam. In *IFIP Conference on System Implementation Languages: Experience and Assessment*, Canterbury, Sept. 1984.

[13] R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[14] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498, New York, NY, USA, 2007. ACM.

[15] Y. Ni, U. Kremer, A. Stere, and L. Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 249–260, New York, NY, USA, 2005. ACM.

[16] R. Pike. The implementation of Newsqueak. *Software — Practice and Experience*, 20(7):649–659, July 1990.

[17] G. M. Reed and A. W. Roscoe. The timed failures-stability model for csp. *Theoretical Computer Science*, 211:85–127, 1999.

[18] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. *ACM Trans. Archit. Code Optim.*, 2(4):366–396, 2005.

[19] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[20] P. A. Steckler and M. Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, Jan. 1997.

[21] D. Walker, L. Mackey, J. Ligatti, G. Reis, and D. August. Static typing for a faulty lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA, September 2006. ACM Press.

[22] P. Winterbottom. Alef Language Reference Manual. In *Plan 9 Programmer's Manual*, Murray Hill, NJ, 1992. AT&T Bell Laboratories.

[23] P. Winterbottom, S. Dorward, and R. Pike. The limbo programming language. In *Proceedings of Compcon 97*, 1997.