Programming Crystalline Hardware

Phillip Stanley-Marbell, Diana Marculescu
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-3890
{pstanley, dianam}@ece.cmu.edu

ABSTRACT

Advances in VLSI technologies, and the emergence of new substrates for constructing computing devices, necessitate a rethinking of the manner in which software interacts with hardware. Traditional computing systems are typified by general purpose processors, which abstract their hardware capabilities through instruction sets. Instruction sets and the abstractions they encourage at the programming language level, were however created, and well suited for, situations in which available hardware was limited and had to be multiplexed in time. In particular, the organization of programs around tightly coupled modules, be they functions, procedures or similar abstractions, make it difficult to perform partitioning of applications both statically and at runtime.

This paper presents preliminary work on a programming language for crystalline architectures, based on the idea of structuring programs around communication. The communication between components in a program is made explicit by employing an abstraction of *names* on which communication occurs, rather than *functions* or *procedures*. Employing such an abstraction as the core of the language makes it possible to explicitly address issues such as information-theoretic analysis of the communication between components of a program, the safety of such communications, and fault-tolerance of applications which execute over a failure prone communication and computational substrate.

1. INTRODUCTION

Programming languages in use today evolved in the era where most machines had von Neumann architectures. Hardware was expensive, and to permit the use of restricted amounts of hardware to solve problems of reasonable size, problems were decomposed into elementary operations and sequences of these operations executed. The tradeoffs included requiring memory to store the sequences of operations (*instructions*) to be performed by hardware, and decoding these stored operations. Along with these hardware abstractions, imperative programming languages, which naturally correspond to the programming model of Turing [19], have been dominant over the last 4 decades. An equivalent programming model embodied by the work of Gödel [6] and Church [4, 5] forms the basis for applicative programming, in which the underlying primitive is that of functions. This model has likewise been dominant in various forms, both in high level programming languages and intermediate representations (such as SSA [16, 1]) for both imperative and functional programming languages.

The traditional von Neumann hardware architectures and the abstractions they provide to software, are however giving way to architectures which are crystalline/regular, looking more like cellular automata [21]. This change is being driven both by increasing device integration in VLSI technologies, coupled with increasing costs of wires, and by new technologies such as those fabricated by chemical self-assembly. As a result of high manufacture time defect rates, these architectures must be inherently regular in their design, such that a failed component can be easily supplanted by a surrogate fault-free one. The future hardware platforms share the common traits of:

- *Cheaper hardware real estate*. Hardware substrates manufactured by chemical self-assembly and other promising non-silicon technologies may provide orders of magnitude increased device integration. Having significantly more hardware resources means there is decreasing motivation to multiplex hardware in time to perform computation. As a result of cheaper hardware, there is now:

- *Increased deployment of microprocessor based systems*. Decreasing cost and physical dimensions makes it enticing to embed processing hardware in more devices of everyday use, or to employ large numbers of them. The devices must communicate, both internally and between devices to achieve maximal utility. Technology constraints however mean that there is:

- *Increasing cost of wires and communication.* The relative cost of wires in designs is increasing in traditional VLSI designs, and the trend may likely also be true for non-silicon technologies. Even though structures such as carbon nanotubes hold the promise of very fast wires, the difficulties involved in making Ohmic contacts between devices might still make many forms of inter-device communication expensive. It is thus desirable to expose the occurrence and costs of communication in hardware to the layers above, that it may be considered during optimizations.

Despite these trends, the *models of computation* currently prevalent in programming languages do not lend themselves well to hardware architectures that are cellular in nature. An underlying issue, is the close association of operations to be performed to *where* they should be performed. It is most apparent in imperative programming languages, but is not inherently absent in other paradigms [2]. Given an application composed of modular units (e.g. functions), it is difficult to partition it for execution over a crystalline substrate, statically or dynamically. This difficulty is largely due to the transfer of control between modules such as functions. The fundamental goal in calling functions is an *interaction*—passing arguments to another module of code, and possibly receiving results—not the transfer of control flow or vectoring of a program counter value to a location in memory.

The partitioning of applications for execution over crystalline hardware is of interest not only for hardware architectures which are crystalline at the micro scale, such as hardware constructed via chemical self-assembly or wire-limited VLSI designs. It is also of great interest in architectures which are crystalline at the macro scale, such as large arrays of processing elements such as microprocessors. In both cases, the communication between processing elements is a limiting factor that must be directly optimizable, and it is thus desirable for it to be explicitly represented.

In addition to the changes in store with respect to hardware organization, are changes in the reliability of hardware. Computing systems today rely on an abstraction of perfectly reliable hardware. This abstraction becomes increasingly difficult to maintain as the underlying hardware becomes inherently prone to runtime failures and manufacturing defects. The reliance on the fault-free abstraction is exacerbated by the increasing dependence in modern culture on computing systems. Not all computations however, require perfectly reliable hardware. The data path portions of many signal processing applications may be able to tolerate errors in computations, whereas the control flow in those same applications may not.

Organizing applications around the fundamental concept of computation as interaction, enables many of the aforementioned issues to be addressed in a coherent manner. Exposing the underlying communication in interactions between modules in an application makes partitioning them for regular / crystalline substrates easier. Once the basic unit of computation is communication, it may become possible to apply ideas developed in Information Theory for communication in the presence of errors, to the units of computation, which now rather than being the invocation of functions, are communication.

1.1 Example

The following is an implementation in the C language, of the *Sieve of Eratosthenes*, an algorithm for computing prime numbers:

```
void sieve(void)
{
        int     i, pi, nums[100];

        for (i = 0; i < 100; i++)
        {
                nums[i] = i+1;
        }

        for (i = 2; i <= 10; i++)
        {
                for (pi = 0; pi < 100; pi++)
                {
                        if (!(nums[pi] % i) && (nums[pi] != i))
                        {
                                nums[pi] = 0;
                        }
                }
        }

        for (i = 1; i < 100; i++)
        {
                if (nums[i] != 0)
                {
                        printf("%d ", nums[i]);
                }
        }

        return;
}
```

It proceeds by iteratively zeroing out entries in a list of numbers which are multiples of any number less than the square root of the largest number in the list. The list in this case is implemented as an array of 100 elements. This implementation is inherently monolithic, and it is difficult to conceive how it could be partitioned to take advantage of a hardware substrate that had a very large number of processing elements. It might be conjectured that this implementation is the natural approach encouraged by the nature and capabilities of the C programming language in which it is implemented.

An alternative implementation might be to structure the program as a set of *coroutines*, which act as "sieves" on a stream of numbers. The interaction between these routines will have to occur by the transfer of control flow between routines. Partitioning such an implementation for execution on a regular computing substrate is made difficult by the interaction between the component routines, which are manifest as transfers of control between the routines.

This paper describes ongoing efforts to design a programming environment based on the aforementioned ideas. Section 2 describes our proposal, as embodied in a prototype language implementation. Section 3 provides a comparison between our proposal and related efforts, by way of an example. Section 4 outlines additional challenges that need to be addressed, and new opportunities presented by our approach. Related research is discussed in section 5, and the paper concludes with a summary of the ideas presented and notes on the current status of our efforts in Section 6

2. COMMUNICATION ORIENTED PROGRAMMING

To explore the aforementioned ideas, we have begun an effort to design and implement an experimental programming language, tentatively named M, based partly on formalisms of the π-calculus [12]. M is a simple language, in which the primary operators are *communication* on names which have *type structure*. The language runtime is constructed as a *name space*—a hierarchy of names which are interfaces to system software support. The core ideas in M are:

- *Structuring of applications around communication on names*. Programs are organized into *modules*, which are invoked by communication on *names* which represent them, rather than transfer of control between modules.
- *Representation of the runtime system as a hierarchical name space with typed names*. The runtime system and facilities akin to standard libraries, are implemented as names which have types.
- *Ability to extract type information from names in the runtime system*. This is essential for both the partitioning and composition of applications—it enables the enforcement of a correct interface between communicating components.
- *A specified alphabet of communication operations for accessing and creating names in the runtime name space*. Employing a specified alphabet for accessing and creating entries in the runtime name space enables a new direction in optimization: *Encodings may be developed for the interactions on names*. It is possible, for example, to specify encodings for interaction on module names which enable, in essence, reliable interaction with modules in the presence of noise, which translates to being able to perform optimizations for, in a sense, "fault-tolerant function calls". Hardware that implements this encoding as its interface is indistinguishable from a software module, and so this may further ease the integration of hardware and software components.

2.1 Language overview

The goal in the design of M, is to provide a language structure that enables the description of concurrency, centered on the concept of all operations (computations) as communication, and enabling the automatic partitioning of programs. To this end, applications are structured as a collection of *modules*, which interact by communication over *names* in a *runtime name space*. The communication on names is performed using a small alphabet of operations, and the encoding of the communication on names provides a possible avenue for interesting optimizations. By making all interaction between program components (modules) explicit, it is possible to employ alternative encodings for the communication between modules, and thus, it is possible to begin reasoning about the information-theoretic aspects of module interactions.

2.2 Modules

Modules are the basic units of applications in M. A module is a collection of program statements. A module's *interface* defines what *name(s)* it makes visible in the runtime namespace. Modules in an application exist concurrently, and interact with each other over these names.

For example, the following is the definition of a module type, followed by the definition of the module implementation:

```
Sort : module
{
    --  Sort interface difinition: This example
    --  interface contains two items
    bsort : string;
    qsort : string;
}

Sort () =
{
    --  Code for the module Sort.
}
```

The above could be taken as a valid program that defines a single module, `Sort`. Its interface defines two names, `bsort` and `qsort`, which both have the type `string`. The above module interface definition leads to entries in the runtime name space, as described in the following section. The names defined in the module interface, may be bound to variables in a module implementation. Such variables that are bound to names in the runtime name space are called *channels*—reading them will block until the corresponding name in the runtime is written to and vice versa. They are therefore identical to channels in CSP [9] or languages such as Alef [23] or Limbo [17].

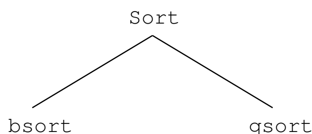

Figure 1: Runtime name space due to the definition of the `Sort` module above.

2.3 Runtime System

It is through the runtime system that individual modules may create communication paths to each other, access system software support such as facilities for performing I/O (file access, printing to standard output), etc. The entries in the runtime name space resulting from the definition of the module `Sort` in the previous section are illustrated in Figure 1.

The runtime system is organized as a hierarchy of *names*. For example, the hierarchy in Figure 1 has the name `Sort` at its top, and the names `bsort` and `qsort` at one level below. The structure of the tree in essence represents the *scope* of names in the runtime name space. Nodes in the tree represent modules, and leaves are names defined within some module interface.

It is through the runtime name space that components of applications which have been partitioned for execution on multiple devices interact. The runtime name space serves as the means by which different modules executing on different devices, may be connected seamlessly. This is facilitated in the following manner: Modules executing on a particular device interact with names in their local runtime name space. The runtime name space on a particular device may be connected to that of another, such that names from one runtime become visible on another, transparent to the executing modules. The connection of runtime name spaces occurs by making it accessible outside the runtime system, e.g., as an address/port accessible over a network, or as a shared memory window—the exact means by which this connection is achieved will be dependent on the particular hardware platform, and is transparent to applications.

An application is structured as a set of modules, and the modules interact through the runtime name space. The actual processing devices on which the component modules of an application execute, is irrelevant, as long as the runtime name spaces of the individual devices are connected together. For example, an application which consists of many modules including the `Sort` module above, may be partitioned such that different modules that comprise it execute on different devices in a hardware substrate. The constituent modules will interact with the `Sort` module by attempting to access the name `Sort` in the runtime, and the names `bsort` and `qsort` in the hierarchy below it.

What is then required to enable the execution of the application, whose constituent modules are on different devices in an interconnect network, is to provide a means by which communications on names are delivered to the appropriate devices. In its simplest form, each unique name in the runtime name space space may be conceived as a unique channel in a communication medium, and communication on a name is in essence communication on a specific channel. The number of names that such a runtime name space will be able to support will then be limited by the bandwidth of the communication interconnect between devices.

2.4 Names

The operators defined on names are `name2type`, `name2thread`, `name2scope`, `name2chan`, `nameread` and `namewrite`, corresponding to operations for type extraction, creation of new module instances from names, listing the names defined within a module, making a name accessible as a channel, reading a name and writing a name, respectively. There is an additional operator, `chan2name` which is defined on channels, and makes them available as names in the runtime name space. The encoding of this alphabet of operations on names is independent of the actual operations, and may be pursued as an optimization point.

2.4.1 `name2type`

The `name2type` operator, when applied to a name in the runtime name space returns the type of the name. The `name2type` operator is in some ways similar to the `tagof` operator in the Limbo language, for determining the variant type of a Limbo *pick ADT*. The result of `name2type` is however more expressive : it details the entire structure of a type in terms of the language primitive types, whereas `tagof` simply results in a unique numerical value for each variant, useful only for comparing two types for equality. The result of the `tagof` does not in itself express the structure of the type it is applied to.

It is feasible to declare variables in an application, based on a type extracted from a name in the runtime name space. For example, in the following, a channel variable is declared to be of the type of the system formatted I/O facility, `"/sys/printf"` in the runtime name space. A string subsequently sent on the channel bound to that name is printed on the console:

```
print : name2type "/sys/printf";

print <- "Hello\n";
```

2.4.2 `name2thread`

Applying the `name2thread` operator to a name which is a node (*not* a leaf) in the runtime name space will create a new

```
Sieve () =
{
	i       : int <- 2;
	status  : string;
	outchan : chan of int;

	chan2name outchan "streamsource";
	status <- name2thread "Worker" (2);

	for ( ; status != nil ; )
	{
		outchan <- i;
		i <- i + 1;
	}
}

Worker (ourprime : integer) =
{
	print          : name2type "/system/print";
	outchan        : chan of int;
	newprime, n    : int;
	inchan         : name2chan "streamsource";

	chan2name outchan "streamsource";
	print <- ("%d", ourprime);
	for ( ; !((newprime <- inchan) % ourprime) ; )
	{
	}

	status <- name2thread "Worker" (newprime);
	for ( ; status != nil ; )
	{
		if ((n <- inchan) % ourprime)
		{
			outchan <- n;
		}
	}
}
```

Figure 2: An implementation of the *Sieve of Eratosthenes* algorithm for generating a stream of prime numbers, in M.

instance of the module, resulting in the creation of a new node in the runtime name space. Naturally, if the name to which the operator is applied is a leaf in the runtime name space, i.e., it is a name defined within some module interface and not itself a module interface, the operation will fail.

2.4.3 `name2scope`

The `name2scope` operator returns a (possibly empty) list of names within the scope of the name to which it is applied.

2.4.4 `name2chan`

The `name2chan` operator is used to bind channels in modules to names in the runtime name space.

2.4.5 `nameread`

The `nameread` operator corresponds to the operation that occurs when a channel variable that is connected to a name in the runtime system is read from, via the channel communication, `<-`.

2.4.6 `namewrite`

The `namewrite` operator corresponds to the operation that occurs when a channel variable that is connected to a name in the runtime system is written to.

2.4.7 `chan2name`

The `chan2name` operator makes a channel in a module visible in the runtime system as a leaf in the runtime name space.

3. EXAMPLE

The example in Figure 2 illustrates an implementation of the *Sieve of Eratosthenes*, an algorithm for the generation of prime numbers, in M. The implementation follows the natural solution of a series of "sieves" acting on a stream of integers. There are two primary components, module `Sieve` and module `Worker`. At runtime each of these is visible in the runtime name space, as `"Sieve"` and `"Worker"` respectively.

An instance of `Sieve` generates a stream of integers on a channel, `outchan`, which is made visible in the runtime name space as the name `"streamsource"`, via the `chan2name` operator. The `Sieve` module creates a new instance from a name in the runtime name space, `"Worker"`, of the module `Worker`. Each instance of `Worker` further creates more instances, at lower levels in the runtime name space hierarchy, to act as further sieve stages. Each instance of `Worker` reads from the name `"streamsource"` from the level above in the hierarchy, and creates a new name `"streamsource"` which will be read by the level below.

At runtime, each `name2thread` might lead to the creation of an instance of the module on one device in a hardware substrate, versus another, based for example, on a runtime defect map or other constraints. It is irrelevant that the implementation of the module `Worker` is defined alongside that of `Sieve`—they only interact through names in the runtime name space. As long as the communication interconnect between devices supports the communication on names, modules which are part of an application can interact seamlessly. Once the runtime name spaces of different devices are connected together, it is irrelevant on which particular device `Worker` executes.

In contrast to the previously described implementation in C, the structuring of the implementation around communication is more straightforward in M, as it would be in any language that provides language-level communication primitives such as channels. Partitioning is made possible by the decoupling of interfaces into the facilities of the runtime name space, and is made safe by the enforcement of types on entries in the runtime name space.

For comparison, an equivalent implementation in the *Pict* language [14], taken from the official Pict distribution is shown in Figure 3. Pict is a language that aims to directly implement ideas of the π-calculus. The π-calculus is a formalism for representing concurrency and has as its basic operation the *interaction* between agents.

4. CHALLENGES, OPPORTUNITIES AND OPEN ISSUES

In addition to the direct goal of enabling partitioning of applications for crystalline hardware substrates, making the communication components of programs explicit enables new directions for exploration, which are of increased importance in non-silicon computing systems.

4.1 Fault-Tolerance

In hardware substrates with high manufacture-time defect rates, as well as high runtime failure rates, mechanisms for enabling fault-free operation in the presence of failures are essential. For example, hardware architectures constructed by chemical self-assembly must employ regular structures [7], in order to be able to employ fault-free devices as surrogates for defective devices.

If such an architecture acts as a general purpose processing device, software must execute over the array of failure-prone elements, with communication occurring over failure-prone interconnections. Exposing communication at the language level is therefore a step towards enabling fault-tolerant computation. Since the basic operation in M programs is communication, they lend themselves to, e.g., ideas from Information Theory, such as coding to enable reliable communication over a noisy medium. For example, the encoding of the alphabet for communication on names may be an exciting avenue for further research — op-

```
{-
 - The sieve of Eratosthenes
 -}

now (reset checks)

def interval (min:Int max:Int):(List Int) =
  if (>> min max) then nil else (cons min (interval (inc min) max))

def sieve (max:Int):(List Int) = (
 def again (l:(List Int)):(List Int) =
   if (null l) then
     nil
   else
    (val n = (car l)

     if (>> (* n n) max) then l
     else (cons n (again (list.filter #Int l \(x) = (<> (mod x n) 0)))))

 (again (interval 2 max))
)

def prPrime (idx:Int x:Int):[] =
  if (== (mod idx 10) 9) then ((int.pr x); (nl))
  else ((int.pr x); (pr " "))

(list.itApply (sieve 4000) prPrime);
```

Figure 3: An implementation of the sieve of Eratosthenes in Pict, taken from the Pict distribution.

timizations may involve trading off encodings which employ a larger number of bits per operation (e.g. for an operation such as `nameread`) versus greater resilience of the operation to failures in the interconnect network between devices, which acts as the support for the runtime name space. The same encoding might be employed for all the operators defined in this paper, or different encodings might be employed for different operations, etc.

4.2 The Hardware-Software Interface Contract

An *Instruction Set Architecture (ISA)* is the contract between the hardware and the programmer. The integer functional units on a modern microprocessor are available to a programmer through integer instructions, likewise floating point units through floating point instructions. In the presence of failure prone "operations", various methods could be used to provide asymptotically small probability of failure in an aggregate device, though techniques employing redundancy and multiplexing [20]. Such techniques however, provide asymptotically small (though not zero) failure rates (in terms of the current notion of what reliable hardware is), and do so with potentially large hardware overhead.

For some applications however, the possibility of error in operations is acceptable, such as in the *datapath* operations of some signal processing applications. Given failure prone hardware therefore, it is natural to imagine including as part of the hardware-software interface contract, the notion of "probabilistic" instructions. For example, an `ADD` operation becomes an `ADD_p_0.9`, an `ADD` which operates correctly with probability 0.9. Part of the task of a compiler then becomes to generate code appropriately such that the right type (i.e. both the *operation* and the *probability*) of operation are appropriately generated. Being able to determine this however, requires that the compiler has some semantic information about what is being performed at the high level, i.e., whether a piece of code is an FFT or a password verification routine. It is therefore desirable to incorporate such high level notions in programs.

Given the possibility of defect variation over time however, it will be difficult to guarantee such instruction behaviors for a particular piece of hardware. The location independence afforded by the use of a runtime name space for accessing all resources might provide a means of guaranteeing such operation probabilities from a collection of hardware devices.

4.3 Security

The partitioning of applications poses new challenges with respect to information security and data integrity. In traditional computing systems, where applications are executed on a single processor or tightly coupled multiprocessor, very little of the application's internal communication is susceptible to attack by an external entity. With applications partitioned to execute over large numbers of computing elements, concerns about the privacy and integrity of data exchanged across components of an application becomes more acute. Making the communication between components explicit, in the form of interaction over names in the runtime name space makes it easier to approach the task of devising measures to enable safe execution of applications in a possibly malicious environment. For example, the encoding of the operations on names could be used as one avenue for addressing data integrity and privacy.

4.4 Issues

Many outstanding issues remain to be addressed, some of which are already clear, and others which become apparent as we proceed with the design, implementation and evolution of the prototype language M. One such issue is the manner in which the runtime name space should be organized. It is currently conceptualized as a hierarchy which corresponds to the scope structure of modules, however it is not clear whether this is the most appropriate organization.

5. RELATED RESEARCH

Related efforts can be categorized broadly into three groups:

1. Languages based on, or derived from Hoare's CSP[9]. Such languages include Occam [11], Newsqueak [15], Alef [23] and Limbo [17]. Although all the aforementioned languages include language level channels, facilities for creating threads either statically or dynamically and provide direct support for many of the ideas in CSP, they also suffer from one of CSP's admitted drawbacks: there is no true modularity in terms of channels, and to communicate on a channel

```
init()
{
        i := 2;
        sourcechan := chan of int;
        spawn sieve(i, sourcechan);
        while ()
        {
                sourcechan <-= i++;
        }
}
sieve(ourprime : int, inchan : chan of int)
{
        n : int;
        print("%d ", ourprime);
        newchan := chan of int;

        while (!((n = <-inchan) % ourprime))
        {
        }
        spawn sieve(n, newchan);
        while ()
        if ((n = <-inchan) % ourprime)
        {
                newchan <-= n;
        }
}
```

Figure 4: An implementation of the sieve of Eratosthenes in the Limbo programming language

a process must already hold a reference to it. In a case where a language employed only processes and communication on channels as a means of breaking programs down into components, the channel reference requirement would make it difficult to implement facilities such as libraries—how can you communicate with a library over a channel unless you already have a reference to it ?

The above languages however do not really have to deal with this issue, since they are not pure CSP-derived languages, and unlike CSP, contain function call abstractions. This however, is the matter of concern in this work. The program extract in Figure 4, in the Limbo language, is an implementation of the sieve of Eratosthenes. The versions in Newsqueak, Occam and Alef are almost identical, modulo minor syntactic differences. In the figure, the processes `init` and `sieve` share a reference to the channel `sourcechan`. If it were desired to replace the executing `sieve` thread with another instance or, for example, with a hardware implementation, there would be no way to obtain a reference to the channel *a posteriori*.

2. The second group of languages are those built around the ideas of streaming architectures, and examples in this group include Brook [10] and StreaMIT [8]. These languages hold as an underlying *concept* the construction of programs as *filters* what act on *streams*. They still employ the same program organizations of traditional imperative non-concurrent programming languages, and implement the ideas of streaming and filters, *over* and traditional language structure.

3. The third class of languages are those based on the π–calculus, such as Pierce's *Pict* language. Pict is in essence an implementation of the attributes of the π–calculus in a functional language. Its implementation of the features of the π–calculus are in terms of function calls, and Pict is not designed with the ability to automatically partition programs in mind. For example, the Pict program in Figure 3 implements the sieve of Eratosthenes.

The ideas of the runtime name space may be likened to that of *tuple spaces* in *Linda* [3], in the sense that language level operators can be used to create entries, read and write entries in a runtime space. Unlike tuple spaces, entries in the runtime system in M have type structure, and are organized in a hierarchical structure. The idea of automatic program partitioning has been pursued under the ideas of *closure conversion* [22, 18, 13].

6. SUMMARY AND STATUS

This paper presented our preliminary investigations into a programming platform for failure-prone regular architectures. Our proposal, embodied by our prototype implementation, M, is based on the formalism of the π-calculus, and employs as the basic operation in programs the communication on names which have type structure. Programs are organized into *modules* which interact with each other by communicating on names in a hierarchical runtime name space.

Organizing applications into modules which interact through names in a runtime name space removes the restrictions imposed by function calls on the partitioning of programs. All ties between components of a program are now through entries in the runtime name space. It however introduces problems of its own, such as interfacing with the runtime system and facilities such as standard libraries. Organizing the runtime as a name space, and structuring the language around basic operators for binding to names and creating new entries in the runtime name space mitigates these issues. It has further benefits—by defining a simple alphabet for accessing entries in the runtime, hardware may directly create entries in the runtime name space, and such entries are indistinguishable from those created by software modules. The runtimes of different applications, executing on different hardware components can also be connected using this simple set of operations. Interaction between modules, possibly executing on different devices, and hardware, poses a potential problem to scaling applications. The use of names with type structure and operations for extracting types from names in the runtime, makes it feasible to safely connect modules from different devices, with hardware and with each other.

We have completed a preliminary design of a language incorporating the ideas proposed in this document, and are in the process of a preliminary implementation of a compiler for the language. The current design contains the ideas presented in this paper; an EBNF grammar is presented in the appendix. Our initial implementation is being performed on the Inferno operating system, since it affords us a major benefit—resources in Inferno are typically represented through a filesystem interface, making it possible to make many system resources available to M programs as names, obeying the familiar semantics of reads and writes.

APPENDIX

An EBNF grammar for the language is listed in Figure 5.

1. REFERENCES

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In ACM, editor, *POPL '88. Proceedings of the conference on Principles of programming languages, January 13–15, 1988, San Diego, CA*, pages 1–11, New York, NY, USA, 1988. ACM Press.

[2] J. Backus. Can Programming be Liberated from the von Neumann Style? *Communications of the ACM*, 21(8):613–641, 1978.

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

[4] A. Church. A set of postulates for the foundation of logic part I. *Annals of Mathematics*, 33:346–366, 1932.

```
program             ::=     {module}
module              ::=     ident "(" decllist ")" "=" scopedstmtlist
decllist            ::=     [typedeclaration {"," typedeclaration}]
tuple               ::=     "(" exprlist ")"
identlist           ::=     ident {"," ident}
scopedstmtlist      ::=     "{" stmtlist "}"
stmtlist            ::=     [statement {";" statement}]
statement           ::=     nullstatement | forstatement | ifstatement
                            | typedeclaration | assignstmt
typedeclaration     ::=     identlist ":" (type | typexpr)
                            identlist ":" (type | typexpr) "<-" expression
assignstmt          ::=     ident "<-" expression
nullstatement       ::=     ";"
ifstatement         ::=     "if" "(" expression ")" scopedstmtlist
                            | "if" "(" expression ")" scopedstmtlist "else" scopestmtlist
forstatement        ::=     "for" "(" [statement] ";" [expression] ";" [statement] ")"
                            scopestmtlist
factor              ::=     "(" expression ")" | integerconst | stringconst
                            | ident | "nil" | tuple
term                ::=     factor {hprecbinop factor}
monadicexpr         ::=     monadicop term
typexpr             ::=     "name2type" expression
newmodexpr          ::=     "name2thread" tuple
expression          ::=     term (booleanop | lprecbinop) term | monadicexpr | newmodexpr
ident               ::=     letter {letter | digit}
integerconst        ::=     {digit}
stringconst         ::=     """ [{unicodech}] """
exprlist            ::=     expr {"," expr}


booleanop           ::=     ">" | "<" | "=" | "&&" | "||" | ">=" | "<=" | "!=" | "<-" | "=="
hprecbinop          ::=     "*" | "/" | "%" | "^"
lprecbinop          ::=     "+" | "-" | "|"
monadicop           ::=     "+" | "-" | "~" | "!" | "name2chan" | "chan2name"
                            | "name2type" | "nameread" | "namewrite" | "name2scope"
                            | "name2thread"
type                ::=     "integer" | "string"
letter              ::=     "A" | ... | "Z" | "a" | ... | "z"
digit               ::=     "0" | ... | "9"
unicodech           ::=     Any Unicode character
```

Figure 5: The M language grammar in EBNF

[5] A. Church. A set of postulates for the foundation of logic part II. *Annals of Mathematics*, 34:839–864, 1933.

[6] K. Gödel. *Über die Vollständigkeit des Logikkalküls*. PhD thesis, University of Vienna, 1930.

[7] S. C. Goldstein and M. Budiu. Nanofabrics: Spatial computing using molecular electronics. In *28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, 2001. ACM SIGARCH / IEEE.

[8] M. I. Gordon, M. K. W. Thies, 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*, pages 291–303, 2002.

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

[10] M. Horowitz, P. Hanrahan, B.Mark, I. Buck, B. Dally, B. Serebrin, U. Kapasi, and L. Hammond. Brook: A Streaming Programming Language. Technical report, 2001. `http://graphics.stanford.edu/streamlang/`.

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

[12] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100:1–77, 1992.

[13] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 271–283, New York, NY, USA, 1996. ACM Press.

[14] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

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

[16] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In ACM, editor, *POPL '88. Proceedings of the conference on Principles of programming languages, January 13–15, 1988, San Diego, CA*, pages 12–27, New York, NY, USA, 1988. ACM Press.

[17] P. Stanley-Marbell. *Inferno Programming with Limbo*. John Wiley & Sons, Chichester, 2003.

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

[19] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, Series 2(42):230–265, 1936-1937.

[20] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, pages 43–98, 1956.

[21] J. von Neumann. *Theory of Self-Reproducing Automata*.

University of Illinois Press, Urbana, 1966. Edited and completed by A. W. Burks.

[22] M. Wand and P. Steckler. Selective and lightweight closure conversion. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: Portland, Oregon, January 17–21, 1994*, pages 435–445, New York, NY, USA, 1994. ACM Press.

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