An FPGA Implementation of a Distributed Virtual Machine

Lee A. Jensen and Lance R. Williams

Dept. of Computer Science, University of New Mexico, Albuquerque, NM 87131

Abstract. An expression in a functional programming language can be compiled into a massively redundant, spatially distributed, concurrent computation called a *distributed virtual machine (DVM)*. A DVM is comprised of bytecodes reified as actors undergoing diffusion on a two-dimensional grid communicating via messages containing encapsulated virtual machine states *(continuations)*. Because the semantics of expression evaluation are purely functional, DVMs can employ massive redundancy in the representation of the heap to help ensure that computations complete even when large areas of the physical host substrate have failed. Because they can be implemented as asynchronous circuits, DVMs also address the well known problem affecting traditional machine architectures implemented as integrated circuits, namely, clock networks consuming increasingly large fractions of area as device size increases. This paper describes the first hardware implementation of a DVM. This was accomplished by compiling a VHDL specification of a special purpose distributed memory multicomputer with a mesh interconnection network into a *globally asynchronous, locally synchronous (GALS)* circuit in an FPGA. Each independently clocked node combines a processor based on a virtual machine for compiled Scheme language programs, with just enough local memory to hold a single heap allocated object and a continuation.

1 Introduction

Research in *artificial life* often involves the construction of virtual worlds populated by artificial organisms reproducing and competing for resources. Whether the artificial organisms are programs encoded in assembly language [3, 23] or *cellular automata* [19, 29], concrete implementations make their resource use explicit, which is necessary for meaningful competition. In contrast, in *genetic programming,* programs are typically encoded in high-level languages, so that mutation and crossover can more efficiently explore the space of computations that solve a given problem [17, 25]. Although this permits more rapid evolution, the resource use of programs encoded in high-level languages can be difficult to accurately gauge. Ideally, the two approaches could be combined: self-replicating programs written in a high-level language could be compiled into concrete implementations in a virtual world where they would efficiently evolve into more complex forms by competing for resources.

As a step in this direction, one of us (the second author) recently described a novel artificial organism based on a self-hosting compiler for a small subset of Scheme [31]. The gap between abstract self-description (faster evolution) and concrete implementation (transparent use of resources) was spanned by making the artificial organism an object program that replicates by compiling its own source code. Both object program (phenome) and source program (genome) were reified as a *distributed virtual machine (DVM)*, a spatially distributed, concurrent computation that can be implemented as an array of communicating finite state machines, or *asynchronous cellular automata.*

Unfortunately, simulation of the replication process on a laptop computer required nearly 8 hours to finish. It goes without saying that without a huge speedup, the importance of self-replicating DVMs based on self-hosting compilers in evolutionary computation research will remain purely theoretical. The work described in the paper you are reading has, as its very practical goal, the design, implementation, and testing of a special purpose distributed memory multicomputer system able to host large numbers of self-replicating DVMs and speed up their execution by four orders of magnitude.

1.1 Emulation of SIMD by MIMD

At the present time, all of the world's fastest computers are *multicomputers* composed of a large number of general purpose processors with local memory (*nodes*) linked by a fast interconnection network. In Flynn's taxonomy [14], computers with this architecture are classified as distributed memory, multiple instruction, multiple data (MIMD) systems (see Figure 1).

Given the potential of multicomputers to run different programs on different nodes (the first 'M' in MIMD), it's remarkable that this rarely happens. Indeed, this capability is not used when solving instances of the class of problems to which they are most commonly applied, *i.e.,* so-called *embarrassingly parallel* problems for which it is possible to achieve a speedup of up to n times on n nodes [5]. Most commonly, multicomputers function as globally asynchronous, locally synchronous (GALS) emulations of very large, single instruction, multiple data (SIMD) systems.[1] Although using a multicomputer like this might not fully exploit its capabilities, it is nevertheless useful because synchronous implementations of SIMD systems do not scale; a global clock signal cannot be transmitted to increasing numbers of spatially distributed nodes without a corresponding increase in transmission latency.

1.2 Emulation of SISD by MIMD

Synchronous implementation also limits the scalability of more conventional single instruction, single data (SISD) systems. As the number of components in an

[1] This brings to mind the very interesting result concerning the ability of asynchronous cellular automata to emulate synchronous cellular automata with negligible slowdown[8].

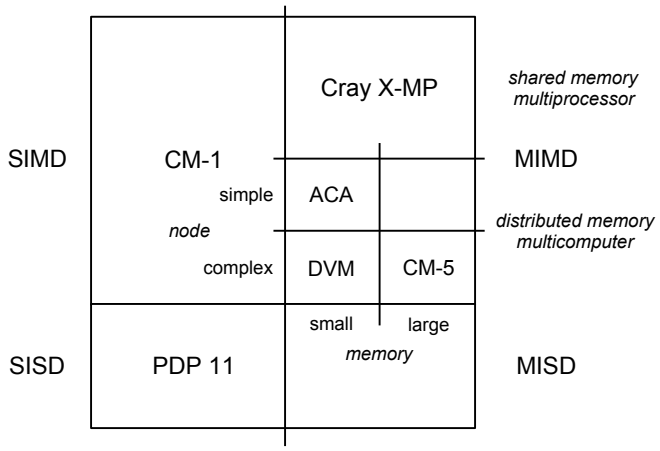

Fig. 1. Flynn's taxonomy [14] showing the relationship between SISD, SIMD and MIMD systems. The distributed virtual machine (DVM) implemented using VHDL and compiled to an FPGA is a distributed memory MIMD system where the nodes are processors based on Dybvig's virtual machine for Scheme[13] with a small amount of local memory (enough to hold a single heap allocated object and a continuation).

integrated circuit implementation of a SISD system increases, the fraction of the circuit devoted to the distribution of the clock signal increases correspondingly. This ultimately limits the number of components a fully synchronous circuit can contain [12].

We have seen that multicomputers can host very large SIMD computations, and in doing so, overcome the scalability limitations of fully synchronous implementations. It is worth asking whether a multicomputer can likewise host very large SISD computations, *i.e.,* computations requiring address spaces significantly larger than the address space of any single node of the network, and in doing so, overcome the scalability limitations of synchronous implementations of SISD systems.

This question has been answered in the affirmative in prior work reported in this conference on *distributed virtual machines (DVMs)* [30]. The key insight underlying DVMs is that expression evaluation can be implemented as a spatially distributed, asynchronous, message passing computation. The program heap (including the bytecodes representing the compiled program itself) is reified as a set of *actors* that can be distributed across the nodes of a distributed memory MIMD system. Each node combines a general purpose processor with a small amount of local memory. Actors can send messages containing encapsulated virtual machine states, *i.e., continuations*, to actors hosted on adjacent nodes. They can also allocate new heap objects (also reified as actors) on adjacent nodes (if the nodes are empty). So that any actor can (in principle) communicate with any other actor, and in order to make space for new heap objects, all actors are subject to constant random motion *(diffusion)* which moves them between adjacent nodes of the network.

It is ironic that in the emulation of a SIMD system by a multicomputer, that a large part of the system's distributed memory is inefficiently used representing

millions of identical copies of the same program (one copy per node), while in the emulation of a SISD system described above, a single copy of the program is efficiently distributed across all nodes. Sadly, due to the extreme slowness of the diffusion-based message passing, a DVM system like the above is unlikely to be built any time soon. Indeed, it is likely to be useful only when solving problems for which one is willing to wait a long time for the answer, yet also require a very large address space and cannot be decomposed into parallel subproblems.[2] Because this combination of factors is unlikely to occur in practice, it would seem that DVMs hosted on multicomputers are of purely theoretical interest. Happily, DVMs do have one advantage relative to conventional SISD systems, which is, they can use redundancy in the spatially distributed heap to solve problems more *robustly*.

1.3 Robust Evaluation of Expressions

Pure functional programming languages possess a property termed *referential transparency* that allows programs to be treated like expressions in mathematics [21]. In particular: 1) the value of an expression cannot depend on the order of evaluation of its subexpressions; and 2) functions must always return the same value when applied to the same arguments. Since side-effects would violate both properties, they are strictly forbidden. It follows that heap allocated objects in pure functional programming are *immutable, i.e.,* once created, they can never be changed.[3] The immutability of heap allocated objects has significant implications for DVMs since it means that multiple instances of each object (including bytecodes) can coexist in the same spatially distributed heap without inconsistency. Furthermore, multiple continuations representing parallel execution threads (each at a different point of progress) can also coexist in the same DVM. Because of referential transparency, objects created on one thread are completely interchangeable with objects created on other threads.

A DVM hosted on a modular substrate where each module represents a small fraction of the multicomputer nodes and interconnection network would possess some interesting features. First, hosted computations could survive the failure of a large fraction of the modules comprising the substrate. Second, modules could be added to the substrate either to replace modules that have failed or to extend it; hosted computations would proceed uninterrupted. Although this would not speed up a hosted computation, it would increase the likelihood that it will finish. Together, these two design features raise the possibility of computations with lifetimes longer than the hardware that hosts them.

---

[2] Deep Thought from *The Hitch Hiker's Guide to the Galaxy* comes to mind.

[3] Despite this apparent limitation, functional programming languages are extremely expressive and modern compilers exploit referential transparency to perform powerful code optimizations.

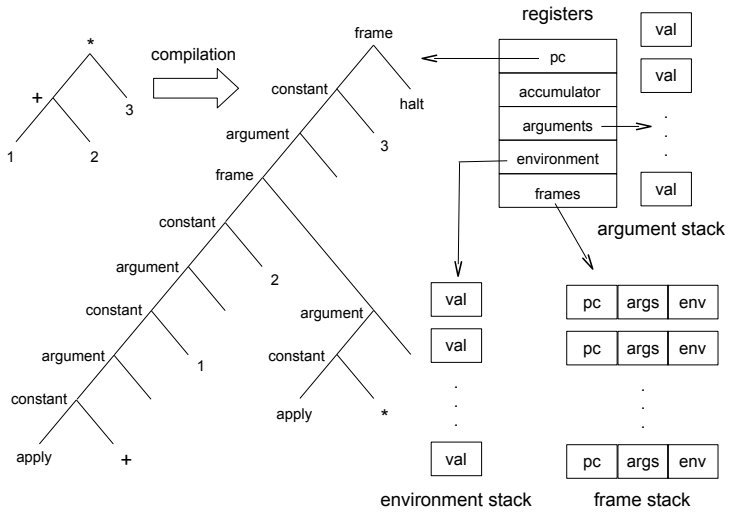

Fig. 2. Dybvig's *virtual machine* for evaluating compiled Scheme expressions showing its registers and associated heap-allocated data structures.

2 Virtual Machine

The process of evaluating expressions by compiling them into bytecodes which are executed on a VM was first described by Landin [18] for Lisp and was generalized for Scheme by Dybvig [13]. Because it plays an important role in our work, it is worth examining Dybvig's model for Scheme evaluation in some detail.

Expressions in Scheme can be numbers, booleans, primitive functions, closures, symbols, and pairs. A closure is an expression with free variables together with a reference to the lexical environment; these two items suffice to describe a function in Scheme. Symbols can serve as names for other expressions and pairs are the basic building blocks of lists. As such, they are used to represent both Scheme source code and list-based data structures. All other types are self-evaluating, that is, they are simply constants.

Evaluating an expression which is not a constant or a symbol requires saving the current evaluation context onto a stack, then recursively evaluating subexpressions and pushing the resulting values onto a second stack. The second stack is then reduced by applying either a primitive function or a closure to the values it contains. Afterwards, the first stack is popped, restoring the prior evaluation context. Expressions in Scheme are compiled into trees of bytecodes which perform these operations when the bytecodes are interpreted. For book keeping during this process, Dybvig's VM requires five registers (see Figure 2).

With the exception of the *accumulator*, which can point to an expression of any type, and the *program counter*, which points to a position in the tree of bytecodes, each of the registers in the VM points to a heap allocated data structure comprised of pairs; the *environment* register points to a stack representing the values of symbols in enclosing lexical scopes, the *arguments* register points to the stack of values which a function (or closure) is applied to, and the *frames* register points to a stack of suspended evaluation contexts.

Evaluation occurs as the contents of these registers are transformed by the interpretation of the bytecodes. For example, the *constant* bytecode loads the accumulator with a constant, while the *refer* bytecode loads it with a value from the environment stack. Other bytecodes push the frame and argument stacks (and allocate the pairs which comprise them). For example, the *frame* bytecode pushes an evaluation context onto the frame stack while the *argument* bytecode pushes the accumulator (which holds the value of an evaluated subexpression) onto the argument stack. Still other bytecodes pop these stacks. For example, the *apply* bytecode restores an evaluation context after applying a primitive function (or a closure) to the values found in the argument stack, leaving the result in the accumulator.

The most important of the remaining bytecodes in Dybvig's VM is *close* which constructs a *closure* and places a pointer to it in the accumulator. We have extended Dybvig's VM with a bytecode which is identical to his *close* bytecode except that the first value in the enclosed lexical environment of a closure created by our bytecode is a self-pointer. This device makes it possible to define recursive functions without the need for a mutable global environment. In this way, we preserve referential transparency without incurring the overhead associated with the use of the applicative order Y-combinator.

3 Distributed Virtual Machine

The actors comprising the distributed heap can represent any of the datatypes permissible in Scheme including numbers, booleans, primitive functions, closures, and pairs. Significantly, they can also represent the bytecodes of a compiled Scheme program. Like other heap-objects, a bytecode actor will respond to a *get* message by returning its value, but unlike actors representing other heap-objects, it can also send and receive encapsulated virtual machine states, or *continuations*. Upon receipt of a continuation, a bytecode actor transforms it in a manner specific to its type, then passes it on to the next bytecode in the program, and so on, until the continuation reaches a *halt* bytecode at which point the *accumulator* field of the continuation contains the result of evaluating the expression. In contrast to a conventional VM, where all control is centralized, control in a DVM is distributed among the bytecodes which comprise it; instead of fetching bytecodes to one location where they update centralized virtual machine state, we encapsulate that state and pass it from one bytecode actor to the next (see Figure 3).

Recall that applying a function requires the construction of a stack of evaluated subexpressions. In the simplest case, these subexpressions are constants, and the stack is constructed by executing the constant and argument bytecodes in alternation. We will use this two bytecode sequence to illustrate the operation of a DVM in more detail.

An actor of type constant bytecode in the *locked* state loads its accumulator with the address of its constant valued operand and enters the *continue* state. When a bytecode actor in the *continue* state sees its child in the bytecode tree

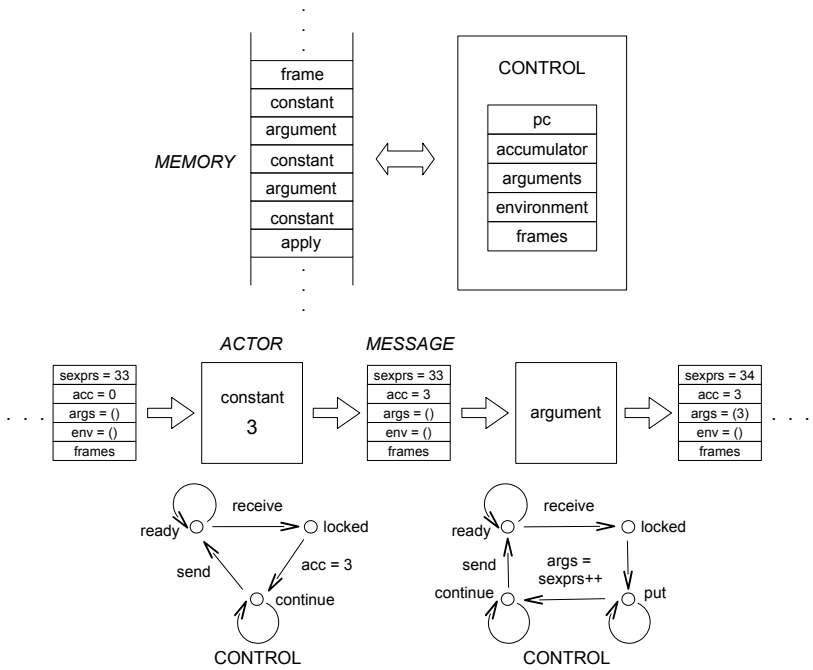

Fig. 3. Conventional *virtual machine* (top) and *distributed virtual machine* (bottom). In the DVM, the registers are encapsulated in a message called a *continuation* which is passed between bytecodes reified as actors. The *sexprs* register in the continuation holds the next free address on the execution thread. No program counter is needed since each bytecode actor knows the address of its children in the bytecode tree. Each actor is a finite state machine which transforms the continuation in manner specific to its type then passes it to the next bytecode in the program. Control is distributed not centralized.

in its neighborhood, it overwrites the child actor's registers with the contents of its own, sets the child actor's state to *locked*, and returns to the *ready* state.

The behavior of an actor of type argument bytecode in the *locked* state is more complicated. It must push its accumulator onto the argument stack, which is comprised of heap-allocated pairs. Since this requires allocating a new pair, it remains in the *put* state until it sees an adjacent empty site in its neighborhood. After creating the new pair actor on the adjacent empty site, it increments the register representing the last allocated heap address (for this execution thread) and enters the *continue* state.

For the most part, we have faithfully implemented the heap-based compiler for Scheme described by Dybvig [13] and have also respected the semantics of his VM in the implementation of the transformations performed on continuations by the bytecode actors which comprise our DVMs.

4 Four Implementation Models

In this section we describe four possible approaches to implementing DVMs, culminating in the approach which is the focus of this paper, a globally asyn-

chronous, locally synchronous circuit implemented using a field programmable gate array (FPGA).

4.1 Shared Memory Multiprocessor

Erlang [6] is a functional programming language based on the *actor model* of concurrent computation [4, 7, 11]. Because communicating processes *(actors)* do not share state, all communication is by message passing. Actors can send messages to others if they possess their identifiers.

Given its support for the actor model, it would be straightforward to implement a DVM in Erlang; bytecodes and other heap allocated objects would be represented by actors and unique identifiers would be associated with heap addresses. A native code compiler would then compile the Erlang source code into one (or more) object programs which would then run on a uniprocessor (or a shared memory multiprocessor) system.

Sadly, the DVM implementation described above would have no advantages relative to a conventional SISD computer. Notably, it would not permit the simulation of SISD computations with address spaces larger than the memory of the shared memory multiprocessor. Furthermore, its lack of redundancy would give it no additional robustness.

4.2 Distributed Memory Multicomputer

This leads to a second possible DVM implementation. Erlang can (in principle) be compiled to set of programs distributed across the nodes of a multicomputer [33]. If the number of processors permitted, the addresses of heap allocated objects could be mapped to actors in one-to-one fashion, and actors (in turn) to nodes in many-to-one fashion using a static allocation strategy. The fact that the mapping is static would allow efficient routing of messages between communicating processes. Unlike the multiprocessor implementation sketched above, a multicomputer implementation would indeed be able to simulate a SISD computation with an address space larger than the memory contained in any single node. Furthermore, if the number of processors permitted redundancy in the representation of the distributed heap (the one-to-one address to actor mapping replaced by a one-to-many mapping), then the implementation would also be robust to node failure. However, the property which makes routing of messages relatively efficient, *i.e.,* static allocation, is incompatible with the design principle of *indefinite scalability*.

4.3 Movable Feast Machine

In recent work, Ackley et al. [1] introduced the idea of a distributed memory multicomputer system with an address space of *a priori* unknown size. Such an *indefinitely scalable* computer consists of independently clocked modules which tile space and only communicate with neighboring modules. Because information

can propagate no faster than the speed of light, and because processing elements have finite size, processors and memory in an indefinitely scalable computer must be spatially distributed.

The multicomputer implementation described in the last section is not indefinitely scalable since the specifics of any static allocation strategy permitting efficient message routing would necessarily depend on the number of nodes in the network. This suggests a third possible DVM implementation, based on *reified actors*. Unlike actors in the classical actor model, which inhabit an absolute address space indexed by unique global identifiers, reified actors occupy locations on a 2D grid, and can only communicate with other actors in their neighborhoods [30]. This restriction, together with the fact that expression evaluation can potentially require a message to be sent from any object to any other object in the address space, necessitates the constant random motion of actors representing heap allocated objects on the grid.

In more recent work, Ackley and Ackley [2] describe a concurrent programming language for implementing reified actor models. In theory, *ulam* serves as a high-level interface to a low-level substrate consisting of an array of asynchronous cellular automata (ACA). In practice, it is a compiled language that targets an indefinitely scalable modular computer called the *Movable Feast Machine (MFM)*.

Like the multicomputer implementation, an MFM implementation of a DVM would be able to simulate a SISD computation with an address space larger than the memory contained in any single node. It would also be robust to failure of MFM modules. However, unlike the multicomputer implementation, it would (in fact) be indefinitely scalable, since modules could (in principle) be added to the machine and a running DVM computation could make effective use of them by increasing the redundancy of its heap representation.

Each module of the MFM contains a single processor with enough memory to simulate a small contiguous region of the (potentially) infinite 2D grid which forms the domain of a spatial computation. Although the size of this region is variable, in typical applications, modules simulate regions comprised of 48×48 sites, or 2304 sites per processor. A more direct and potentially much more efficient DVM implementation would allocate one processor per site, and these processors would implement the instruction set of the Dybvig virtual machine in hardware (as opposed to interpreting bytecodes in software).[4] These final refinements lead to a fourth possible implementation, the one we actually pursued.

4.4 Field Programmable Gate Array

Although they differ in significant respects, the three implementation models described thus far have one thing in common, namely, they all represent bytecodes and other heap allocated objects as communicating processes (actors). In the multiprocessor and multicomputer implementations, the actors existed in a

[4] The first integrated circuit implementation of a processor customized for efficient execution of compiled Lisp programs was described by Steele and Sussman [16].

non-physical, abstract identifier space. In the MFM implementation, the actors were reified by assigning them positions on a 2D grid and relying on diffusion for message passing. The fourth implementation is also actor-based, but the actors represent processors in a mesh-connected network, not heap allocated objects.

A *field programmable gate array (FPGA)* consists of an array of programmable logic blocks together with a configurable interconnection network [15]. By means of programming in the field, *i.e.,* after manufacture, FPGAs are capable of implementing a huge combinatorial space of application specific integrated circuits. VHDL is a concurrent programming language designed by the Dept. of Defense in the 1980s as a hardware description language for very high speed integrated circuits [22]. Used judiciously, a concurrent program written in VHDL can be automatically compiled to an FPGA implementation. The compilation *(synthesis)* process assigns VHDL constructs to individual logic blocks in specific locations in the device and configures the interconnection network to implement the specified functionality.

Although VHDL can (like Erlang) be used as a general purpose concurrent programming language, if it was merely used to implement a simulation of a DVM where bytecodes and heap allocated objects were represented as communicating processes (like the three other implementations), then there would be no reason to believe that the resulting concurrent program would be *synthesizeable, i.e.,* could be compiled to an FPGA implementation [9]. Furthermore, even if the program were synthesizeable, then there would be no reason to believe that its synthesized elements would operate with enough parallelism to produce a speedup relative to a sequential implementation; a concurrent program at a different level of abstraction is required to guarantee both of these properties. To ensure both synthesizeablity and effective parallelism, the communicating VHDL processes must represent the nodes of a distributed memory multicomputer hosting a DVM, not the heap allocated objects comprising the DVM itself.[5]

5 Technical Details

In our VHDL specification, the processes modeling multicomputer nodes are driven by independent local clocks implemented as ring oscillators [24]. A ring oscillator typically consists of an odd number of NOT gates connected in series with the last gate connected to the first gate in a feedback loop; see Figure 4 (top). The odd number of gates insures that the output of the last gate is inverted compared to the input of the first gate. When power is applied, the circuit begins to oscillate spontaneously at a period of approximately twice the sum of the individual gate delays. The frequency of the oscillator can be decreased or increased by adding (or removing) an odd number of gates to (or from) the ring. Unfortunately, the use of ring oscillators in FPGA design is problematic since most design tools aggressively try to prevent these so-called *combinatorial*

[5] Others have used FPGAs to implement distributed memory multicomputers as arrays of soft processors [27, 28].

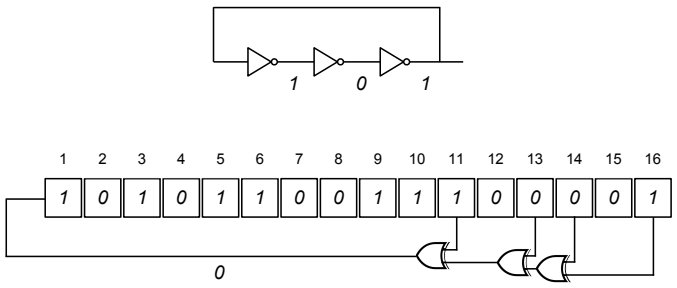

Fig. 4. Ring oscillator used to provide independent timing signals (top) and 16-bit Fibonacci linear feedback shift register (LFSR) used for pseudorandom number generation (bottom) at each node.

loops and aggressively optimize away what seem to be superfluous gates. These optimizations can be overcome using directives that allow for combinatorial loops and marking gates to be excluded from removal during optimization. We generate a ring oscillator at each node with a random length between 9 and 31 gates resulting in a clock frequency in the range 30 MHz - 100 MHz.

The globally asynchronous, locally synchronous circuit requires a source of randomness to implement the diffusion process that enables message passing. After exploring pseudorandom number generation using cellular automata, and true random number generation using ring oscillators, we settled on *linear feedback shift registers (LFSR),* a simple and commonly used method of generating pseudorandom numbers in hardware [20]. More specifically, we used a 16-bit Fibonacci LFSR in our implementation; see Figure 4 (bottom). Each node contains a process implementing the LFSR that is clocked by a local ring oscillator. At each clock tick, the register shifts right 1 bit and the bit positions called *taps* are combined by XOR and fed back to the leftmost (input) bit. The output is the rightmost bit of the LFSR. A maximum-length period $(2^n - 1)$ is produced if the polynomial defined by the taps has an even number of terms and the tap indices are co-prime. In our implementation the seed and taps for each of the 16-bit LFSRs are randomly assigned by the code generator.

Communication and transfer of data between two nodes with independent clocks requires that the two nodes agree both that: 1) the transfer is going to occur; and 2) that the transfer has finished. If this agreement does not occur, multiple processes might simultaneously attempt to read or write data to a single node, resulting in an inconsistent device state. To avoid this problem, our design uses a *four phase handshake protocol* to ensure that data transfer between adjacent nodes is synchronized[10].

6 Experimental Results

We have implemented a DVM with an 8 bit address space on a Xilinx XC7A100 CSG324-2 FPGA [32]. The FPGA chip is manufactured using 28 nm technology and contains 101,440 logic blocks. The FPGA is hosted on a Trenz Electronics

development board with a 100 MHz clock that communicates with the Xilinx Vivado Design Suite running on a Windows PC via a JTAG to USB adapter. We have been able to use this FPGA to implement DVMs with up to 40 nodes. To demonstrate the speedup due to parallelism in the implementation, we have conducted an experiment using the expression

```
(pred (+ 2 3))
```

where *pred* is the function that subtracts one. This expression compiles to 12 bytecodes. During evaluation, 5 additional actors representing heap allocated objects (2 numbers and 3 pairs) are created. It follows that there is enough room on a 4×5 grid to host the actors comprising the distributed heap at its maximum size of 17.

Density is grid size divided by redundancy. There is a complex relationship between density and expected evaluation time. Expected evaluation time is a function of both expected message passing latency and expected object allocation time. Expected message passing latency decreases with increasing density because senders of messages must wait less time before encountering the recipients of their messages. However, expected object allocation time increases because actors allocating objects must wait longer for empty sites to appear in their neighborhoods.[6] It follows that for a given expression and desired level of robustness, there is a density that minimizes expected evaluation time.

The experiment was run with three different conditions: 4×5 $(\times 1)$, $8 \times 5 (\times 1)$ and 8×5 $(\times 1)$ where $m \times n$ $(\times k)$ indicates a grid of size $m \times n$ initialized with k copies of each bytecode actor. These conditions were chosen because the second and third have twice the number of nodes as the first, while the first and third have equal actor density. Equal density removes the confounding factors of different message passing latencies and different object allocation times. It consequently permits measurement of parallel speedup.

A code generator written in Java generates the VHDL code at the desired grid size and randomly populates the grid with the bytecode actors representing the compiled expression at the desired level of redundancy. The VHDL code is then synthesized by the design tool, which outputs a bitstream that is used to program the FPGA. We also insert the Integrated Logic Analyzer (ILA) core into the bitstream so we can capture data from the running device for our experimental results.

The implementation contains an additional process driven by the 100 MHz development board clock that increments a 32 bit counter on each clock pulse. This counter is used to get accurate timing at 10 ns intervals per counter increment. When a *halt* bytecode receives a continuation, the counter is stopped and the ILA is triggered to capture data. The counter value is the time required by the DVM to evaluate the compiled expression. Three different conditions were tested and each condition was run ten times. Evaluation times are shown in Table 1.

[6] Think of the so-called "8-puzzle" and its sliding plastic tiles.

Table 1. Evaluation time in microseconds (μs)

	4×5 ($\times 1$)	8×5 ($\times 1$)	8×5 ($\times 2$)
mean	1321.65	1953.48	1585.64
standard deviation	471.46	603.38	391.80

The 8×5 ($\times 1$) condition is *slower* than the 4×5 ($\times 1$) condition because the lower actor density increases message passing latency. Actors must diffuse twice as long on average before bumping into the recipients of their messages. However, it is not twice as *slow*, and this is because of the *decreased* expected object allocation time of the 8×5 ($\times 1$) condition. A heap containing 17 objects barely fits on the 4×5 grid but there is plenty of room on the 8×5 grid.

Consistent with the fact that expected message passing latency decreases with increasing density, we observe that the 8×5 ($\times 2$) condition is *faster* than the 8×5 ($\times 1$) condition. However, it is not twice as *fast*, and this is because of the *increased* expected object allocation time of the 8×5 ($\times 2$) condition. A heap containing 34 objects barely fits on the 8×5 grid but a heap containing 17 objects fits quite easily.

Finally, the evaluation time for the 8×5 ($\times 2$) condition is only slightly longer than for the equal density 4×5 ($\times 1$) condition. This demonstrates that the FPGA implementation is an actual parallel circuit, solving a problem of twice the size in (essentially) the same amount of time. We believe that the evaluation time for the 8×5 ($\times 2$) condition is longer because the implementation of the DVM on the 8×5 grid very nearly fills the entire FPGA, resulting in less efficient component placement by the synthesis algorithm. We hypothesize that if the experiment were repeated using an FPGA with extra capacity, then the ratio of the times required to solve the different sized problems in the case of equal densities would be closer to one.

7 Conclusion

Recent work showed how an expression in a functional programming language can be compiled into a massively redundant asynchronous spatial computation called a distributed virtual machine (DVM). Because the semantics of expression evaluation are purely functional, DVMs can employ massive redundancy in the representation of the heap to help ensure that computations complete even when large areas of the physical host substrate have failed [30]. Because they can be implemented as asynchronous circuits, DVMs also address the well known problem affecting traditional machine architectures implemented as integrated circuits, namely, clock networks consuming increasingly large fractions of area as device size increases.

Although the use of self-replicating DVMs [31] in evolutionary computation research can potentially combine the advantages of the artificial life and genetic programming approaches, this cannot happen without a DVM implementation

in hardware that is orders of magnitude faster than current software simulations. In this paper, we have described the first hardware implementation of a DVM. This was accomplished by synthesizing a globally asynchronous, locally synchronous circuit in an FPGA from a VHDL specification of a special purpose distributed memory multicomputer with a mesh interconnection network. The nodes of the multicomputer combine a processor based on Dybvig's virtual machine for executing compiled Scheme programs [13] with just enough local memory to hold a single heap allocated object and a continuation. Each node contains its own clock and pseudorandom number generator and synchronization between adjacent nodes is implemented using a four phase handshake protocol. A working implementation consisting of 40 nodes arranged in a 5×8 grid was used to evaluate a compiled Scheme expression. Significantly, the measured evaluation times were consistent with a parallel implementation. Use of FPGA devices with greater numbers of logic blocks will allow the implementation and testing of DVMs with larger grid sizes, capable of evaluating more complex expressions and with increased levels of redundancy.

References

1. David H. Ackley, Daniel C. Cannon, and Lance R. Williams. A movable architecture for robust spatial computing. *The Computer Journal*, 56(12):1450–1468, 2013.
2. D.H. Ackley and E.S. Ackley. The ulam programming language for artificial life. *Artificial Life*, 22:431–450, 2016.
3. Chris Adami, C. Titus Brown, and W.K. Kellogg. Evolutionary learning in the 2D artificial life system "Avida". In *Artificial Life IV*, pages 377–381. MIT Press, 1994.
4. Gul Agha. An overview of actor languages. *ACM SIGPLAN Notices*, 21(10):58–67, 1986.
5. Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, pages 483–485, 1967.
6. Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
7. Henry Baker. *Actor Systems for Real-Time Computation*. PhD thesis, MIT, January 1978.
8. Piotr Berman and Janos Simon. Investigations of fault-tolerant networks of computers. In *STOC*, pages 66–77, 1988.
9. Eduardo Bezerra and Djones Vinicius Lettnin. *Synthesizable VHDL Design for FPGAs*. Springer, 2013.
10. Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.
11. Will Clinger. *Foundations of Actor Semantics*. PhD thesis, MIT, 1981.
12. Peter J. Denning and Ted G. Lewis. Exponential laws of computing growth. *Communications of the ACM*, 60(1):54–65, January 2017.
13. R. Kent Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina, 1987.

14. M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computer*, C-21(9):948–960, 1972.
15. Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
16. Guy L. Steele Jr. and Gerald J. Sussman. Design of a LISP-based microprocessor. *Commun. ACM*, 23(11):628–645, 1980.
17. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
18. P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
19. Christopher G Langton. Self-reproduction in cellular automata. *Physica D: Nonlinear Phenomena*, 10(1):135–144, 1984.
20. T.G. Lewis and W.H. Payne. Generalized feedback shift register pseudorandom number algorithm. *Journal of the ACM*, 20(3):456–468, 1973.
21. John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, New York, NY, USA, 2002.
22. Volnei A. Pedroni. *Circuit Design with VHDL*. MIT Press, Cambridge, MA, USA, 2004.
23. Thomas S. Ray. An evolutionary approach to synthetic biology, Zen and the art of creating life. *Artificial Life*, 1:179–209, 1994.
24. M. Singh, S.M. Ranjan, and Z. Ali. A study of different oscillator structures. *International Journal of Innovative Research in Science, Engineering and Technology*, 3(5), 2014.
25. L. Spector and A. Robinson. Genetic programming and auto-constructive evolution with the Push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, 2002.
26. Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.
27. W. Vanderbauwhede and K. Benkrid. *High-Performance Computing Using FPGAs*. Springer, 2013.
28. I. Vassanyi. Implementing processor arrays on FPGAs. In *International Workshop on Field Programmable Logic and Applications*, pages 446–450, Tallinn, Estonia, 1998.
29. John von Neumann. Theory of self-replicating automata. *Urbana: University of Illinois Press*, 1966.
30. Lance R. Williams. Robust evaluation of expressions by distributed virtual machines. In *Unconventional Computation and Natural Computation*, pages 222–233, Orleans, France, 2012.
31. Lance R. Williams. Self-replicating distributed virtual machines. In *14th Intl. Conf. on the Synthesis and Simulation of Living Systems (ALIFE '14)*, New York, NY, 2014.
32. Xilinx. *7 Series FPGAs Data Sheet: Overview*, August 2017.
33. Beshar Zuhdy, Peter Fritzson, and Kent Engström. Implementation of the real-time functional language Erlang on a massively parallel platform, with applications to telecommunications services. In *High-Performance Computing and Networking*, Milan, Italy, 1995.