Robust Evaluation of Expressions by Distributed Virtual Machines

Lance R. Williams

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

Abstract. We show how expressions written in a functional programming language can be robustly evaluated on a modular asynchronous spatial computer by compiling them into a distributed virtual machine comprised of reified bytecodes undergoing diffusion and communicating via messages containing encapsulated virtual machine states. Because the semantics of the source language are purely functional, multiple instances of each reified bytecode and multiple execution threads can coexist without inconsistency in the same distributed heap.

1 A Pourable Computer

Let us consider a hypothetical molecular computer of the far future. Outwardly, it might look like a beaker filled with water. However, instead of transistors made of silicon, its active components would be billions of instances of hundreds of different molecular species, all in solution. Some species would represent instructions while others would represent data. Whether instructions or data, the interactions between the molecules in solution would be rapid, highly specific, and diffusion driven. The resulting computational process would be parallel, distributed, spatial, and asynchronous.

Compared to a conventional computer, a molecular computer of the kind described above would have several interesting properties. For example, if half of the contents of the beaker containing a running computation were poured into a second beaker we would expect the computation to continue uninterrupted in both. Similarly, if we were to continue this process, and were to pour half of the contents of the two beakers into two more beakers, we would expect the computation to continue uninterrupted in all four. Significantly, we would expect to be able to continue this process of dividing the computation until the volume of liquid in each beaker was so small that some beakers were missing instances of one or more of the molecular species necessary for the computation to continue. To summarize, we observe that, up to a point, dividing the computation into two changes neither its eventual result nor the time required for it to complete–it merely *decreases* the probability of its completion.

We become aware of a second and equally interesting property when we consider the effect of pouring the contents of two beakers (previously divided) back into one. We would expect that the computation in the first beaker would reintegrate with the computation in the second. Reactants and products from the

J. Durand-Lose and N. Jonoska (Eds.): UCNC 2012, LNCS 7445, pp. 222–233, 2012.

first beaker would combine indiscriminately with reactants and products from the second. Significantly, as was true when the computation was divided, the recombining of the computations changes neither its eventual result nor the time required for its completion–it merely *increases* the probability of its completion.

2 A Modular Asynchronous Spatial Computer

Now, it may be that a molecular computer like the one described above will never be practical. However, we would like to advance the radical proposition that computations with the same desirable properties can be achieved by (something which is in effect) a *simulation* of a molecular computer on a modular *asynchronous spatial computer* constructed from conventional electronic components. As an alternative to the von Neumann stored-program computer, asynchronous spatial computers have attracted considerable interest[1,4,5,12,13].

In this paper we focus on the *Movable Feast Machine*[1]. The MFM was chosen because its semantics are expressive and well defined, and (most importantly) because there is an actual hardware implementation based on open source Illuminato X Machina boards. An IXM board is small and square and has a connector on each of its four sides. Multiple boards can be connected to form large two-dimensional grids; each board draws its power from its neighbors. Significantly, IXM boards can be added to and removed from a grid without halting a computation. In the extreme, a computation running on a grid might finish on completely different hardware than it started on.

In the MFM, each IXM board is used to simulate a 48×48 array of *sites*. A site is either empty or occupied by an *atom*, which is subject to random motion or *diffusion*. An atom has 64 bits of state; sixteen of these bits are reserved and comprise the atom's header which leaves 48 bits available for general use. An atom can sense and change the state of other atoms in its *event window* (sites within L_1 distance four or less) which may straddle the boundary between adjacent IXM boards. The update process for atoms is random and asynchronous; there is no global clock. Atoms can form *bonds* with other atoms which restrict their relative motion so that they can remain in constant communication. *Long bonds* can join any pair of actors with overlapping event windows. *Short bonds* can join any pair of actors within L_1 distance two or less. In effect, bonds are short relative addresses which are automatically updated as the atoms they join undergo diffusion. See Figure 1.

3 Virtual Machine

A general purpose computer might accept input in the form of an expression in a programming language and then evaluate the expression, returning the result. A standard method of evaluating expressions is to compile them into programs in simpler languages, and then simulate the execution of those programs on a *virtual machine (VM)*. For the present, we ignore the problem of compilation

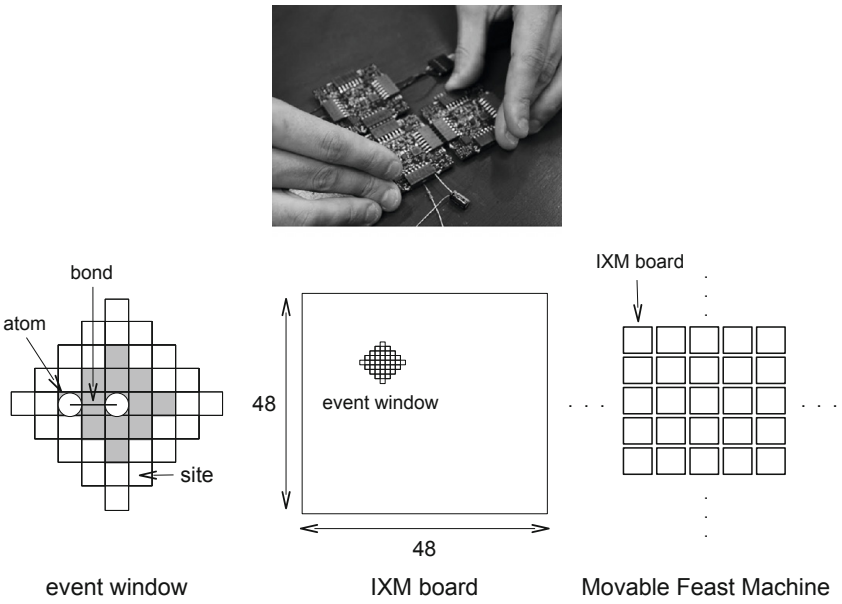

Fig. 1. In the MFM[1], each IXM board simulates a 48×48 array of *sites*. The MFM itself is a two dimensional array comprised of multiple IXM boards. An *atom* can sense and change the state of other atoms in its *event window* (sites within L_1 distance four). A *long bond* can join any pair of actors with overlapping event windows. A *short bond* can join any pair of atoms within L_1 distance two (the region shown in grey).

by assuming that it can be performed offline and that a compiled expression, a program comprised of bytecodes, will define the computation.

The process of evaluating expressions by compiling them into bytecodes which are executed on a VM was first described by Landin[11] for Lisp and was generalized for Scheme by Dybvig[8]. 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

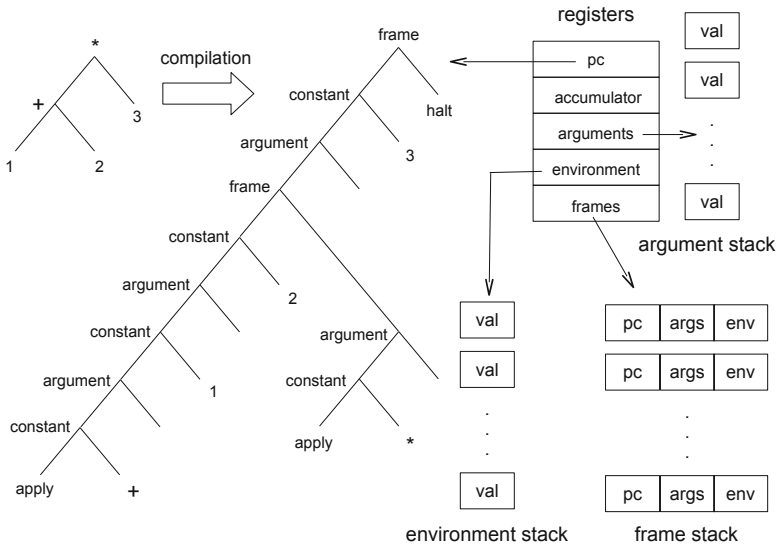

Fig. 2. Dybvig's *virtual machine* for evaluating compiled Scheme expressions showing its registers and associated heap-allocated data structures

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.

Lastly, we have extended Dybvig's VM with a bytecode which is identical to his *close* bytecode (used to create closures) 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.

4 A Reified Actor Model

Actors are universal primitives for constructing concurrent computations [10]. Although the actor model has been significantly elaborated over the years [2,3,6,9]. the basic theory is extremely simple. In essence, an actor is a lightweight process with a unique address which can send and receive messages to and from other actors. In response to receiving a message, and (depending on the message's contents) an actor can: 1) send a finite number of messages of its own; 2) create a finite number of new actors; and 3) change its internal state so that its future behavior is different. All of these things happen asynchronously.

In the MFM actors are *reified* as atoms undergoing diffusion. If more than 48 bits of state is needed (and it will be in the system described here) bonded pairs of atoms can be used instead. Message passing is accomplished when the sender of a message recognizes that the recipient of a message is within L_1 distance two and changes the recipient's state.

5 Distributed Virtual Machine

We propose to use a set of actors reified as bonded pairs of atoms in the MFM as a *distributed heap*. 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. We call the set of *bytecode actors* representing a compiled program, a *distributed virtual machine (DVM)*. 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. One might say that if the central premise behind the von Neumann computer is "program as data," then the central premise behind the DVM is "program as computer." 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.

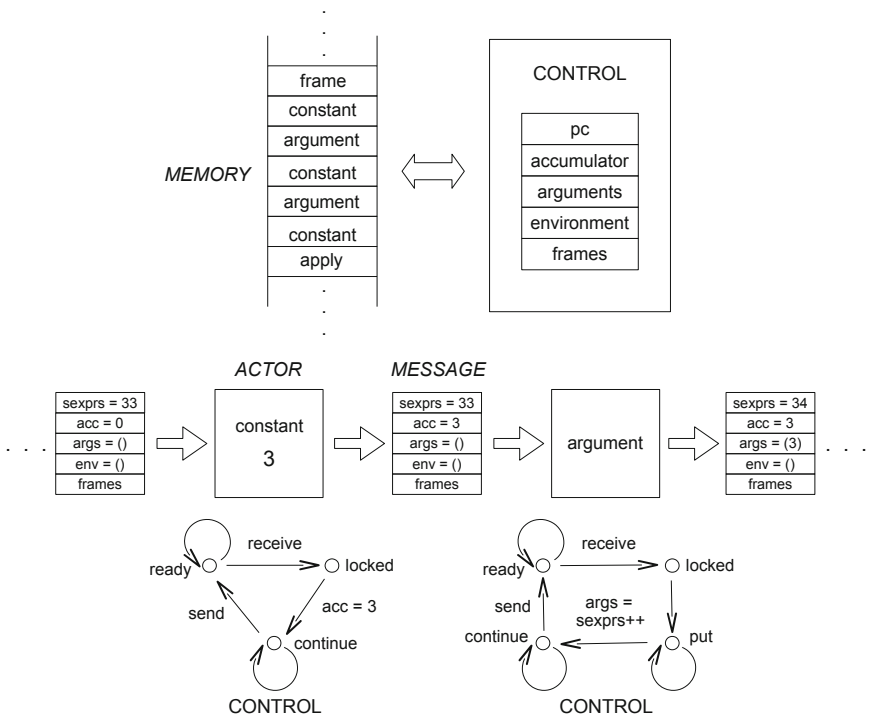

Fig. 3. Conventional *virtual machine* (top) and *distributed virtual machine* (bottom). In the DVM, the registers are encapsulated in a message called a *continuation* which is passed between bytecodes reified as actors. The *sexprs* register holds the next free address on the execution thread. No program 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.

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 within L_1 distance two, 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 (possibly for many MFM updates) until it sees two adjacent empty sites in its event window. After creating the bonded pair of atoms representing the new pair actor on the empty sites, it increments the register representing the last allocated heap address (for this execution thread) and enters the *continue* state.

6 Redundancy

By restricting ourselves to pure functional programs, forgoing mutation and side-effects, we accrue important benefits which will be critical in achieving our goal of increased robustness. The most important of these is that two heap-objects with the same address are absolutely interchangeable and this is true irrespective of which actors created them and when they were last accessed. Significantly, this means that it is possible for multiple instances of each heap-object and multiple execution threads to coexist without inconsistency in the same distributed heap.

We can demonstrate the theoretical robustness of bytecode redundancy obtainable in a DVM, compared to the more obvious approach of simply cloning VMs, using an argument similar to von Neumann's [14] discussion of machines cross-checking each of their operations. Suppose a program needs to execute $n = 256$ bytecodes to finish, and the probability that a bytecode will fail is $p = 0.01$, and we want at least a 0.99 chance of success. If there are m cloned VMs, the probability that at least one of them will succeed is $1-(1-(1-p)^n)^m$. Consequently, m must be at least 59 to achieve a 0.99 chance of success. On the other hand, in a DVM with r copies of each bytecode, the probability of success is $(1-p^r)^n$, and a mere $r = 3$ suffices to achieve 0.99 percent chance of success. If we compare communication costs, the result is similar; although the cloned VMs require only $O(nm)$ continuation messages total, and the distributed virtual machine requires a quadratic $O(nr^2)$, given that $m = 59$ versus $r = 3$ with the above parameters, the cloned VMs must send 15104 continuation messages, while the DVM requires only 2298.

7 Experimental System

The programming language used in our experimental system is a purely functional subset of Scheme. Because it is purely functional, *define*, which associates values with names in a global environment using mutation, and *letrec*, which also uses mutation, have been excluded. Also, for simplicity, closures are restricted to one argument. Consequently, user defined functions with more than one argument must be written in a curried style. This simplifies the representation of the lexical environment which is used at runtime by making all variable references integer offsets into a flat environment stack[7]. Finally, we introduce a new special-form, *lambda+*, which creates a closure which contains a self-pointer, and which can be used to create locally defined recursive functions at runtime without *define* or *letrec*. In all other ways, we have faithfully implemented the heap-based compiler for Scheme described by Dybvig[8] 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. To accomplish this, each bytecode actor must possess sufficient state to represent 9 heap addresses:

- 1 address representing the current size of the heap
- 4 addresses for the VM registers encapsulated in the continuation

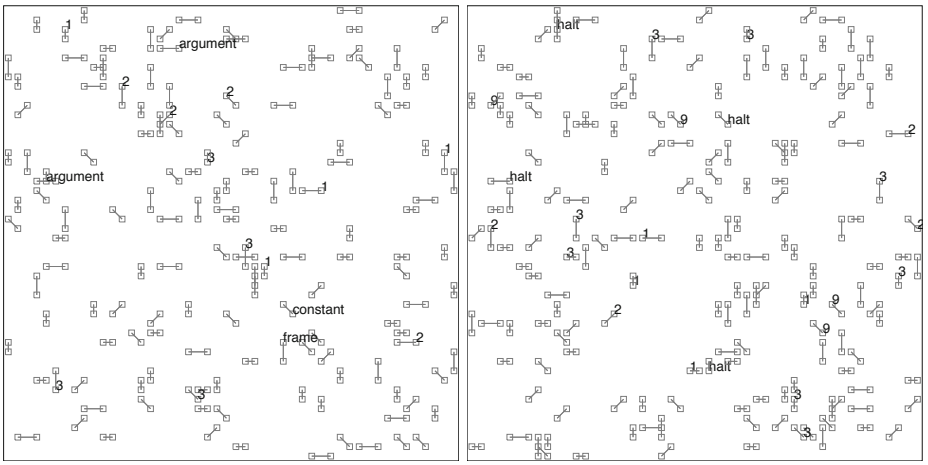

Fig. 4. Screenshots of a DVM evaluating the Scheme expression (* (+ 1 2) 3). The simulated MFM is of size 48×48, *i.e.*, one IXM board. Actors are reified as bonded pairs of atoms and the redundancy level is four. Only bytecodes which possess a continuation and numbers are labeled. In the right screenshot, all four continuations have reached halt bytecodes; the four number nines are the result of the computation.

- 1 address for a self-pointer
- 2 addresses for child-pointers in the bytecode tree
- 1 address for message recipient.

In addition, 5 bits are required to represent heap-object type and 5 bits are required to represent execution state for the most complex of the byetcodes. Because atoms in the MFM can contain at most 48 bits of state, we reified a bytecode actor as a pair of atoms joined by a short bond. See Figure 4. This mechanism gives actors up to 88 bits of state each (since 4 bits in each atom are required to maintain the short bond), permitting an 8 bit address space. Although not very large, an 8 bit address space permits the evaluation of relatively complex expressions like ((lambda+ f x (if (= x 1) 1 (+ x (f (- x 1))))) 9) which returns the sum of the integers between 1 and 9. This expression compiles into 67 unique heap-objects consisting of a mixture of bytecodes, numbers, closures, and primitive functions. With a redundancy level of 16, these are reified as 2176 bonded pairs of atoms. Evaluating this expression requires a heap size of 232–well within the 8 bit maximum.

The goal of the first experiment was to determine the effect of redundancy on the time required to evaluate an expression. We evaluated ((lambda+ f x (if (= x 1) 1 (+ x (f (- x 1))))) N) for N in the range 1 to 9 and for redundancy levels of 1, 2, 4, 8 and 16. Ten trials were run for each condition. The dimensions of the simulated MFM were fixed at 128 × 128. We observe that at all redundancy levels, the average time required for the first execution thread to reach the halt bytecode increases linearly with N. See Figure 5. Furthermore, this time is inversely proportional to the redundancy level, which strongly suggests that

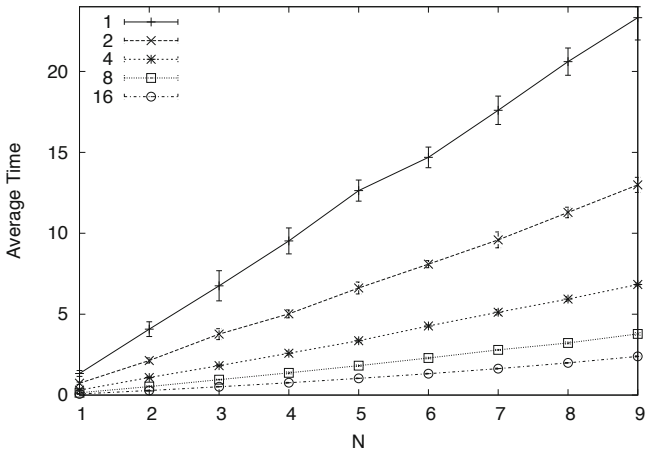

Fig. 5. Average time (measured in millions of updates per actor) required to evaluate ((lambda+ f x (if (= x 1) 1 (+ x (f (- x 1))))) N) as a function of N and for different levels of redundancy. Error bars show plus or minus one standard deviation.

message latency is independent of the number of distinct addresses (heap size) and depends only on the number of copies of each address (redundancy). The analogy with molecular computation is compelling since (for an MFM of constant size) redundancy corresponds to reactant concentration. However, unlike actual molecules which have negligible volumes, bonded pairs of atoms in the MFM have non-negligible areas. Consequently the product of redundancy and heap size cannot exceed some fraction of the area of the MFM before the area occupied by actors is so large that it impedes diffusion. Hence, message latency can only be decreased by increasing redundancy to this point.

The goal of the second experiment was to explore the robustness of DVMs with different levels of redundancy to a constant background rate of actor failure. We assume that when an actor fails, the bonded pair of atoms representing it is removed from the MFM and that there is no other form of corruption. The expression evaluated was (* (+ 1 2) 3) which compiled to 29 unique heap-objects. Redundancy ranged from 1 to 16 and failure rate ranged from 1 to 128 failures per millions actor updates. In order to keep the actor concentration (and hence message latency) constant for all levels of redundancy r, the dimensions of the simulated MFM were set to $N \times N$ where $N = 16 \times 2^{\frac{\log r}{2}}$.

One hundred trials were run for each experimental condition. A trial was classified as a success when any execution thread reached the halt bytecode. A trial was classified as a failure when either no actors remained or when time equaled 5×10^4 updates per actor (on average). The results of this experiment are shown in Figure 6. Error bars represent 95% confidence intervals. The most striking thing about these results is that they show that beyond a failure rate of 64 per million updates, additional redundancy has no effect on robustness.

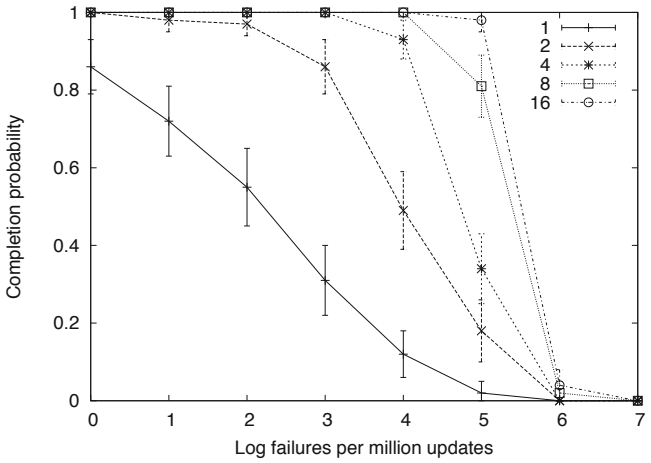

Fig. 6. Robustness of DVM evaluating (* (+ 1 2) 3) with different levels of redundancy as a function of actor failure rate (log failures per million updates). Error bars show 95% confidence interval.

Although this result might initially be puzzling, it makes perfect sense when one considers that removing actors from the simulation without also decreasing the area of the MFM decreases reactant concentration and therefore increases message latency. This (in turn) slows the rate of the computation, making it even more vulnerable to actor failure, which further decreases reactant concentration, and so on. The effect on completion probability is pronounced.

The goal of the last experiment was to explore the robustness of DVMs with different levels of redundancy to failures of different fractions of the whole MFM. A fractional failure of size x consists of the removal of all sites (and bonded pairs of atoms occupying those sites) outside a square region of area $n \times n$ positioned in the lower left corner of a simulated MFM where $n = \sqrt{(1-x)}N$.

In each trial, a single fractional failure (ranging in size from 0.1 to 0.9) was simulated at the (condition independent) time when 7500 updates per actor had been completed (on average). The expression evaluated and the termination criteria were the same as those in the second experiment. One hundred trials were run for each experimental condition. Inspection of the results (shown in Figure 7) reveals that higher levels of redundancy result in higher probabilities of successful completion over the full range of fractional machine failure sizes.

Significantly, even with a fractional failure size of 70%, the simulated MFM with redundancy level of 16 still successfully finishes 90% of the time. We conjecture that this trend would continue indefinitely so that tolerance to fractional board failures of any degree less than 100% could be achieved by a sufficiently large MFM.

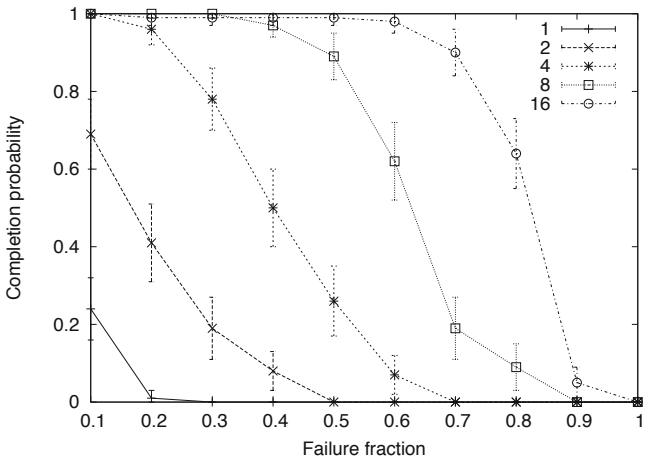

Fig. 7. Robustness of DVM evaluatting (* (+ 1 2) 3) with different levels of redundancy as a function of machine failure fraction. Error bars show 95% confidence interval.

8 Conclusion

We have shown how expressions written in a functional programming language can be evaluated on a modular asynchronous spatial computer. This was accomplished by compiling the expressions into a distributed virtual machine comprised of reified bytecodes undergoing diffusion and communicating via messages containing encapsulated virtual machine states.

Because the semantics of the source language are purely functional, multiple instances of each reified bytecode and multiple execution threads can coexist without inconsistency in the same distributed heap. Significantly, it was shown that evaluation efficiency and robustness both increased with increased redundancy. It was further shown that the evaluation process is robust to two types of hardware failure...but less so to the second, namely, failures which result in a decrease in the spatial density (concentration) of actors representing heap-allocated objects. However, it was shown to be extremely robust to the elimination of entire regions of space (since this doesn't affect concentration), and this may be the more realistic failure model in an asynchronous spatial computer comprised of discrete modules.

Acknowledgements. Thanks to Dave Ackley for sharing his vision of a post von Neumann future.

References

1. Ackley, D.H., Cannon, D.C.: Pursue robust indefinite scalability. In: Proc. HotOS XIII (May 2011)
2. Agha, G.: Actors: A model of concurrent computation in distributed systems (1986)

3. Baker, H.: Actor Systems for Real-Time Computation. PhD thesis (January 1978)
4. Beal, J., Michel, O., Schultz, U.P.: Spatial computing: Distributed systems that take advantage of our geometric world. TAAS 6(2), 11 (2011)
5. Chapiro, D.M.: Globally Asynchronous Locally Synchronous Systems. PhD thesis (1984)
6. Clinger, W.: Foundations of Actor Semantics. PhD thesis (1981)
7. De Bruijn, N.G.: Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae 34, 381–392 (1972)
8. Kent Dybvig, R.: Three Implementation Models for Scheme. PhD thesis (1987)
9. Greif, I., Hewitt, C.: Actor semantics of PLANNER-73. In: Principles of Programming Languages (January 1975)
10. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: IJCAI, pp. 235–245 (1973)
11. Landin, P.J.: The mechanical evaluation of expressions. The Computer Journal 6(4), 308–320 (1964)
12. Muttersbach, J., Villiger, T., Fichtner, W.: Practical design of globally-asynchronous locally-synchronous systems. In: ASYNC, pp. 52–59 (2000)
13. Sipper, M.: The emergence of cellular computing. IEEE Computer 32(7), 18–26 (1999)
14. von Neumann, J.: The general and logical theory of automata. In: Jeffress, L.A. (ed.) Cerebral Mechanisms in Behaviour. Wiley (1951)