The Computer Journal Advance Access published October 11, 2012
doi:10.1093/comjnl/bxs129

A Movable Architecture for Robust Spatial Computing

DAVID H. ACKLEY*, DANIEL C. CANNON AND LANCE R. WILLIAMS

Computer Science, University of New Mexico, Albuquerque, NM 87131, USA
**Corresponding author: ackley@cs.unm.edu*

For open-ended computational growth, we argue that: (i) instead of hardwiring and hiding component spatial relationships, computer architecture should soften and expose them; and (2) instead of relegating reliability to hardware, robustness must climb the computational stack toward the end users. We suggest that eventually all truly large-scale computers will be *robust spatial computers*—even if intended neither for spatial tasks nor harsh environments. This paper is an extended introduction for the spatial computing community to the *Movable Feast Machine* (MFM), a computing model in the spirit of an object-oriented asynchronous cellular automata. We motivate the approach and then present the model, touching on robustness mechanisms such as redundancy, compartmentalization and homeostasis. We provide simulation data from prototype movable elements such as self-healing wire for data transport and movable 'membrane' rings for spatial segregation, and illustrate how some larger computations like sorting or evaluating a lambda expression can be reconceived for robustness and movability within a spatial computing architecture.

Keywords: spatial computing; computer architecture; robustness; movable feast machine; globally asynchronous locally synchronous; functional programming

Received 21 December 2011; revised 26 June 2012
Handling editor: Jacob Beal

1. INTRODUCTION

1.1. Scaling beyond serial determinism

Over the last 70 years, ever more powerful computers have revolutionized the world, but their common architectural assumptions—of CPU and RAM, and deterministic program execution—are now all hindrances to continued computational growth.

Deterministic execution means that program outputs are completely and solely determined by the inputs. It allows programmers to ignore system errors and focus solely on program correctness, but also limits cost-effective scalability. An architecture promising determinism can employ error correction and fault tolerance, but must kill the entire program if any component failure—no matter how local or transient—escapes those mechanisms. Scaling up a deterministic machine thus increases both the chance of failure and the work at risk, and the expected loss grows with their product. For example, as the high-performance computing community contemplates the move to exascale computers, the cost of preserving determinism via 'fail stop' error processing is increasingly seen as untenable [1].

The solitary 'central' processor, similarly, focuses programmers on computation without communication or coordination—but CPU scalability depends heavily on now-dwindling increases in clock speed. Finally, the presumption of uniform cost 'random access' memory focuses programmers on disembodied logical function—but light speed and the inescapable geometry of physical space allow only a finite set of locations to fit within the unit time access of a finite processor.

For those reasons, we argue, any truly serious approach to scalable computing will be a *robust spatial computer*. Resilience, survivability, and graceful degradation will be required, both in the hardware and upwards through the computational stack. The assignment of function to spatial location—the literal *architecture* of computation—will be dynamic, adaptive and problem-dependent, and von Neumann-style serial determinism will hold only locally, if at all. Substantial computations will be spatially distributed, and they or their major components will be space-aware and aggressively fault tolerant.

This paper, expanded from [2], is an introduction for the spatial computing community to the *Movable Feast Machine* (MFM). In the rest of this section, we motivate our approach

using the notion of *indefinite scalability*, and place the work into broader computational contexts. Section 2 then presents model details and Section 3 addresses possible implementations and qualitative performance considerations. Then, to illustrate some of the motifs and techniques we have explored, Section 4 describes robust MFM approaches to basic tasks such as density regulation, point-to-point data transfer and spatial segregation, while Section 5 briefly describes two larger MFM computations, one for robust stochastic sorting and the other demonstrating serial expression evaluation within a distributed virtual machine. Finally, Section 6 concludes with a brief critique and a suggestion that robust spatial computing is a route to truly profound computational growth, and we as a community should begin fleshing out its body of knowledge in earnest.

1.2. Research context

We have presented [3] a research case for *indefinitely scalable* computer architectures which, by definition, support open-ended computational growth without re-engineering. Indefinite scalability rejects all internal limits—such as single-source clocking, or fixed-width global addresses for memory or processors—so machine size is limited only by external costs such as real estate, materials and construction, and power and cooling.

The physical realities of finite light speed and non-zero minimum device size mean that *indefinite scalability implies spatial computing*. Processors and memory must be distributed, and communication latencies can be no better than linear in the machine's physical extents. We argued that an indefinitely scalable machine is an up-to-three-dimensional spatial tiling of configurable elements that are initially interchangeable, communicate only locally via relative spatial addresses and execute asynchronously, at least above some granularity.

Indefinite scalability is a computational abstraction very different from serial determinism, but both styles admit a wide range of implementations. A conservative, coarse-grained approach to indefinite scalability could employ conventional computers with IP networking, avoiding the finiteness of IPv6 addresses via the aggressive use of *anycasting* [4, 5]—which offers indefinite spatial scaling in exchange for a finite set of interprocessor request types. For more exotic, longer-term strategies, emerging biochemical and nanoscale computing mechanisms (e.g. [6, 7]) might one day yield fine-grained robust spatial computers in which billions of devices are cost-effective, or ultimately even 'pourable computers' with devices by the mole.

The MFM employs an intermediate granularity, designed to offer flexible and expressive programmability for research explorations, while still possessing properties—such as regular structure, short spatial dependencies and error tolerance—to offer development and optimization opportunities for implementations in both existing and novel media.

1.3. Spatial computing

Spatial computing, described as 'computation distributed in space such that position and distance metrics matter to the computation' [8], is receiving increasing research attention (e.g. [9–14]); also, [15] is a good survey focusing on languages for spatial computing. Our particular point of entry—indefinite scalability—is distinct from, but largely compatible with, other motivations for spatial computing, falling perhaps most readily into the 'computationally intense' aspect of the Dagstuhl framework [8].

Although a spatial computer embedded within an external spatial task can offer choice low-hanging fruit, we focus more on architectural uses of space when computations are either not inherently spatial, or are spatialized differently from the machine itself. Of course, any machine design for indefinite scale must consider its physical embodiment, and its computations will typically employ spatial representations at multiple levels. Among the cross-cutting issues identified at Dagstuhl, fault tolerance and robustness are central to this enterprise, and are discussed below.

Compared to spatial computing languages framed in continuous or amorphous spaces such as Proto [16], or all-in 'vertical' models such as 'blob' machines [17], the MFM style of spatial computing is discrete, reified and designed bottom-up. We presume discrete sites in space and time, and employ a new form of asynchronous cellular automata (see section 1.4.3) that has a neighborhood far too large to allow construction of a state transition table. Instead of transition tables, we propose to use modified versions of familiar programming language constructs—such as sequential code, pointers, objects and classes—to specify state transitions. But despite those unusual elements, it is clear that the MFM approach is deeply compatible with other spatial computing approaches (e.g. [18]), and that is the motivation for [2] and this paper.

1.4. Architectural criteria

Taking indefinite scalability as the backbone requirement brings other principles to the forefront. Here we touch on three: robustness, movability and asynchronous design.

1.4.1. Robustness

Von Neumann [19] argued that 'future' computers would be more robust than his namesake approach, yet more than a half century later, here we are, still rooting around in CPU+RAM designs. We are now deploying millions of such machines *per week*, and connecting them to vast personal information and economic value—and in many ways, this is a disaster waiting to happen. The fragility that von Neumann discussed—the long chains of logic with no allowance for error—remains embedded in our approach to computing today. The very notion of a *central* processing unit—in which all instructions, regardless of origin, execute at the exact same physical location—is a key to our

current security nightmare in which single software flaws can, and regularly do, compromise entire machines.

The present work emerges from a belief that it is likely better if von Neumann's call for robustness came true sooner rather than later. Robustness can be a slippery concept, though, in part because the question 'Robust to what?' can ramify endlessly as we dream up ever more massive, or unlikely, or subtly malicious system perturbations. Drawing on biology in [20] the authors present useful robustness 'principles and parameters'—and in such terms, the examples in Sections 4 and 5 touch on spatial compartmentalization, redundancy, sloppiness and homeostasis.

As we view it, 'robustness' is strongly related to, but distinguishable from, such well-studied areas as error-correcting codes (e.g. [21, 22]), fault tolerance ([23–25], are surveys) and high-availability computing (e.g. [26, 27]). Such approaches recognize that system components sometimes cannot deliver reliable behavior to other components and levels, and provide techniques for using redundant resources to cover potential lapses—such as triple modular redundancy [28] to help mask hardware failures, or N-version programming [29] to help mask software faults. Although these approaches have much in common with our concerns, they typically focus only on maintaining correct program execution—while true robustness, as we see it, necessarily cares about managing the consequences of indisputably *incorrect programs* as well.

1.4.2. *Movability*

Computing machinery derives great economic leverage from its extreme programmability, deferring many decisions from before manufacture until after sale, thus amortizing a single design across many uses. But as a result, critical details may exist only inside one specific machine—and indeed, losses attributed to lack of backups are among the most common and vexing of problems for computer users.

For robust memory, data bits must be copied and moved elsewhere; for robust processing, computations must be copied and moved as well. In conventional computer systems, computational movability has been studied most intensively at the level of process migration ([30], is one survey) although other scales of movability have also been explored—up to entire virtual machines (VMs) (e.g. [31]), and down to individually mobile objects as in Emerald [32] or MagnetOS [33]. In comparison with even the latter efforts, the MFM's approach to movability is decidedly fine-grained and primitive, pushing mobility below programming language and operating system into architecture. Furthermore, we have found that embracing movability as a requirement up front can radically alter conventional design pressures and may provide paths toward architectures that are *inherently* robust to a wide range of failures. For example, beyond just moving a single entity, biological organisms gain dramatic robustness by copying and moving—by *reproducing*—themselves; we argue that purpose-designed computational processes can and should as well.

1.4.3. *Asynchronous design*

Separately clocked circuits require additional resources to communicate, tempting designers to unite them in a single clock domain, even though synchronization costs grow faster than the unified region's circuit complexity. But if synchronization could be guaranteed, then so could determinism, making a synchronous parallel machine, in some sense, the 'next best thing to serial'—and likewise only finitely scalable.

Consider *cellular automata* (CA), another computational model popularized by von Neumann [34], based on a suggestion by Ulam [35], which have been applied to phenomena as diverse as weather and self-reproduction. Even though CA are blatantly parallel and spatial, most are also synchronous, and given even rare timing failures, their computations can go drastically awry. There are, however, *asynchronous cellular automata* (ACA) (e.g. [36–40]), which have indefinite scalability potential. Some ACAs use stochastic transition rules, but in any case their site update order is usually non-deterministic.

An oft-rediscovered technique [41–43] can be used to emulate a synchronous CA on top of an ACA with reasonable efficiency—notwithstanding the claim originally in [2]. However, for indefinite scalability such a global approach is hopelessly fragile: a single stuck site would eventually block the entire universe.

Ultimately, global synchronization is unscalable—but fully asynchronous operation is aggravating and slow—motivating the idea of *globally asynchronous locally synchronous* (GALS) architectures (e.g. [44]), in which synchronized subregions interact within a larger asynchronous framework. We believe the design of such synchronized subregions and their interactions are akin to object-oriented design, and should be addressed via problem-specific programming rather than a one-size-fits-all architectural decision. The overhead of scalable asynchrony is simply the reality of large systems; it should not be avoided, but embraced and exploited.

Asynchronous Logic Automata (ALA) [45], and their reconfigurable variant (RALA) [46], are another recent approach to asynchronous spatial computing. They offer low-level, self-timed logic gates with programmer-visible spatial layout as a primary concern. ALA presumes that the circuit's functional and spatial layout is cemented at manufacturing time, while RALA employs generic 'stem cell' elements that differentiate into specific functions after manufacture, under control of a pre-compiled bit stream describing the desired layout.

ALA and RALA are somewhat more conventional than the MFM in that robustness is not an explicit design concern, but conceptually they form a vector toward MFM-style computational movability. ALA offers immobile asynchronous logic elements, and then RALA offers rigid block movability of an entire computation during a privileged configuration phase before execution—while the MFM embraces movability as a first-class design element, fundamental to its definition and active during operation.

Other computational models, such as connectionist machines (e.g. [47–49]), may also offer indefinitely scalable architectures. Next to such an exotic approach the MFM can seem relatively conventional, with its highly localized, but still recognizable, form of serial programmability.

Broadly, our view is that von Neumann machines are fine as long as they are small and individually insignificant. Of course, the details matter—and that is our next topic.

2. THE MOVABLE FEAST MACHINE

Figure 1 summarizes the MFM, ranging from conceptual hardware 'tiles' at the bottom to pseudocode software 'elements' at the top. Although a truly complete description would be excessive for this introductory paper, we try to provide concepts and specifics sufficient to engage the reader's mechanical intuition.

Throughout this section, the phrase 'some chosen' is used to flag overtly parametric model properties; all the experimental results shown in this paper are based on one particular parameter set called P1, discussed further below.

2.1. Small programs change big neighborhoods

Viewed as a stochastic ACA, the MFM falls into a rather sparsely populated corner of ACA parameter space—with many possible states per site, and also many sites per neighborhood. The P1 parameters used in this paper, for example, employ 64 bits per site (see Section 2.4 for details), and a 41-site neighborhood (a 'Manhattan' or ℓ_1 distance of four in a 2D rectangular lattice)—implying about 10^{20} possible states per site, and a naïve state transition table with over 10^{800} rows. These quantitative parameter choices thus have qualitative consequences: additional design is needed to bypass that table's vast infeasibility.

Instead of a table lookup, the MFM executes sequential code—which we call an *event window update*—to perform a neighborhood state transition. Our design seeks to present a flexible and powerful, but still transparent and understandable, machine semantics to the state transition programmer, and to that end, we adapt familiar programming concepts—objects, classes, pointers—reinvented for indefinite scalability, and miniaturized to fit into an MFM site (for an object) and a neighborhood (the pointer addressing range). Programming a state transition feels somewhat familiar because an individual event executes with synchronous clocking and single-threaded semantics, revealing the MFM to be an odd but recognizable GALS architecture as discussed in Section 1.4.3.

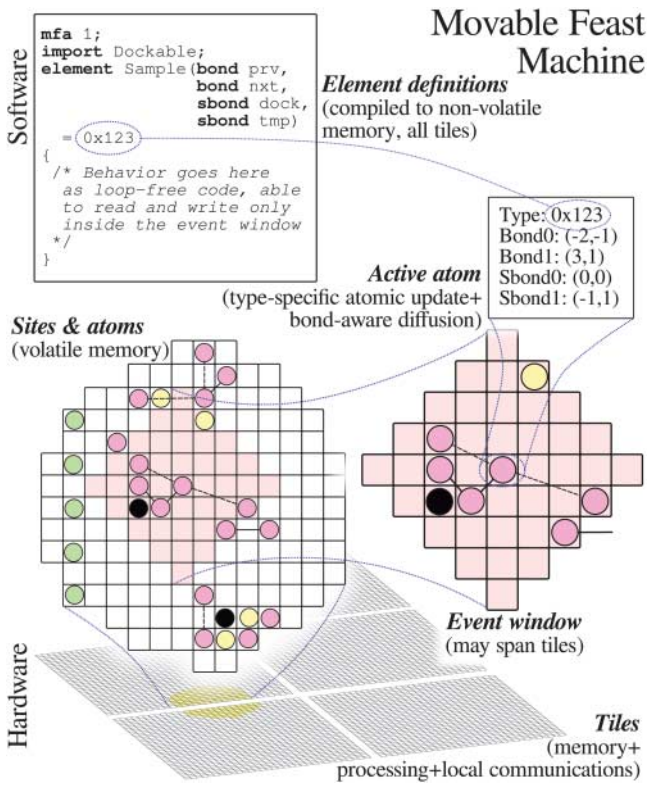


FIGURE 1. Architectural overview (see also Section 2).

2.2. Sites and spatial structure

MFM computation occurs at discrete *sites* arranged in some chosen lattice that is at least locally regular, and embedded in some chosen metric space called *machine space*. Some chosen *event window radius* is defined relative to the metric, and the set of sites at most one event window radius away from a given site is called its *neighborhood*. Although the computation is discrete in space-time, for physical realizability the MFM is grounded in effectively continuous spaces, distances and velocities. We presume that a site occupies finite but more than infinitesimal space, and we require machine space to map smoothly into $\mathbb{R}^3$, either by limiting machine space to 3D or less, which preserves indefinite scalability but excludes many topologies, or by using only a finite, if arbitrarily connected, machine space, such as a 2D grid with periodic boundary conditions, mapping smoothly into a torus in $\mathbb{R}^3$.

2.3. Event window processing

MFM computation proceeds asynchronously in parallel by executing *events*, each of which corresponds to one independent state transition, occurring in a compact volume of space-time called an *event window*. The spatial extent of an event window is the neighborhood of some selected site, and its temporal extent is the state transition's code execution time. For efficiency or implementability in any given media, there may be restrictions on the time, space, or other resources available to execute one event, but in the P1 parameter set, for research flexibility, there are no specific limits.

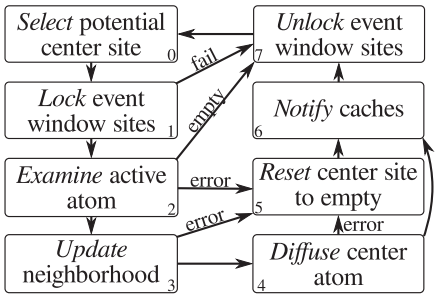


FIGURE 2. Event window life cycle (see Section 2.3).

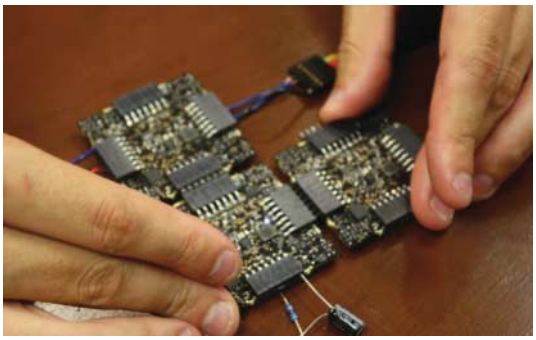


FIGURE 3. Hot-plugging *Illuminato X Machina* tiles [50], prototype indefinitely scalable hardware.

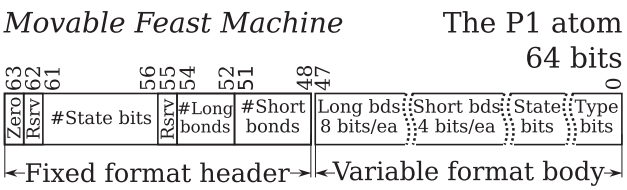


FIGURE 4. Atomic format used in this paper (see Section 2.4).

An event window life cycle is depicted in Fig. 2. A *center site* holding an *active atom* is selected, at random or by any starvation-free mechanism (step 0), such that its neighborhood is disjoint from the neighborhoods of any other currently selected center sites (step 1). The active atom is checked, and erased if any format problem is found (steps 2, 5); otherwise its *element type* is extracted and that type's *element description* is located (Section 2.4.1). If there is no such element, which can occur as a result of bit corruptions yielding a valid header but an unknown type, the active atom is likewise erased (steps 2, 5).

Next, the *update function* of the selected element description is invoked, which may alter the contents of the neighborhood arbitrarily (step 3). This freedom to modify the neighborhood differs from CA variants that only modify center sites—and adds complexities, such as in the cache management of step 6—but it also greatly eases programming tasks in general and maintaining bond consistency in particular (see Section 2.5).

After the update, the center site contents are optionally *diffused* (step 4)—possibly moving the active atom to an immediately adjacent empty site up, down, left or right, and updating all relevant bonds if so (Section 2.5 has additional information on this process). Then neighborhood modifications are communicated to all affected tiles (step 6), locks are released (step 7) and the event ends.

The charter of an indefinitely scalable MFM hardware implementation is to execute as many disjoint event windows as possible, parallel in space and consecutively in time, while providing reasonably, if not absolutely, reliable state maintenance and transitions. An initial MFM implementation on our first real, indefinitely scalable hardware (Fig. 3) is in progress, although it has limited communication channels; see Section 3 for some performance considerations on plausible but currently hypothetical hardware. For an MFM simulation—which includes any MFM that is *not* running on indefinitely scalable hardware—the actual execution time is usually misleading, and instead we adopt *average events per site* (AEPS) as the base unit of time; regardless of the MFM size, in 1 AEPS each site will, on average, be the center site of one event.

2.4. Atomic structure and semantics

Each MFM site contains some chosen number of modifiable bits; each possible combination of those bit values is called an *atom*, and the number of bits per site is called the *atomic width*. Some chosen *type function* abstracts an atom into a corresponding *type number*, which is associated with an *element*, which in turn specifies how to perform a state transition when that atom is active. In object-oriented terms, an atom is akin to a small, fixed-size object instance, linked by its type number to an element definition that acts like a class (pseudocode shown in Fig. 1), which supplies information such as bond counts (next section) as well as the `update` method that defines a state transition.

Figure 4 depicts the 'P1 atom', a 64-bit atom used in all the examples below. A 16-bit header specifies the interpretation of the remaining 48 bits, dividing them up among two types of 'bonds', general-purpose 'state bits' and zero or more 'type bits' that specify that atom's elemental type number. Managing and optimizing atomic 'bit budgets' is a common activity in MFM program design, and there is always pressure to 'poach' type bits for other purposes. But once zero type bits are assigned, the only possible element number is 0, and so only one such completely packed element definition can exist in a given MFM 'physics'.

2.4.1. *Element descriptions*

The elemental type number extracted from an atom is used as a key to locate the associated *element description*, which is a data structure providing the information needed to perform transitions. The element descriptions are identical across

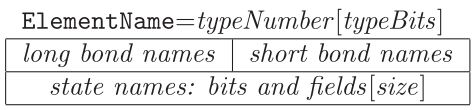


FIGURE 5. An 'element information box'. See also Section 2.4, and examples in Section 4.

processing tiles, and are read-only to atomic-level event window computations; techniques to alter them are discussed below.

Figure 5 shows a sample 'element information box', summarizing the main parameters of an element, containing: in the top row, the element name, type number and number of type bits; in the middle row, the names of any long and short bonds; and in the bottom row, the names of any state bits (unmarked) or state fields (with their bit counts).

Beyond the atomic layout information, a key component provided by each element is its *update function*, which is invoked (Fig. 2, step 3) to transform an event window when an active atom is of the given element. An element definition may also provide information affecting the diffusion process (Fig. 2, step 4).

2.4.2. *Reprogrammable physics*

Collectively, a set of element descriptions—a *physics*—both enables and constrains the possible behaviors of an MFM. For robustness and consistency we expect element descriptions to be stored in non-volatile memory and be identical on all tiles, which could be accomplished permanently during manufacture—storing code and data in ROM or direct logic—and that would also be the most economical approach. At present, however, we are far from having so complete and useful a physics to justify such a rigid construction. Like the shift from hard-coded ALA to programmable RALA discussed in Section 1.4.3, for research flexibility we require reprogrammability of element descriptions—despite the challenges inherent in changing the laws of (MFM) physics.

The '*Santa Fe Board*' (SFB) software [51] that runs our prototype hardware [50] supports peer-to-peer tile reprogramming [52], using low-level intertile packets that are not confusable with MFM traffic. We expect that reprogramming at the level of MFM physics will use the same or similar mechanisms, and so we briefly describe the approach here. For peer-to-peer neighbor reprogramming, the SFB software provides three 'boot modes' for different trade-offs of convenience and security. A tile in 'green boot' mode will accept any user programming claiming to be newer than the board's existing user code. 'Blue boot' requires a matching 'program id' as well as a newer date, but note that there is no code signing or validation: primarily, blue boot enables touching tiles to run different user programs without overwriting each other. Finally, 'red boot' disables automatic peer-to-peer reprogramming entirely, demanding a physical button press on each tile, in a specific time window during power-up, to initiate its reprogramming.

A general security advantage of a robust spatial computer over a von Neumann machine is that many sites, rather than just one, must be suborned to take complete control—but if powerful global reprogrammability is implemented, for research or any reason, then access to that leverage can negate the distributed system advantage. The most robust MFM implementations would avoid global reprogrammability at the element description level entirely—forcing attacks to be physically localized in real space, like a button push or blob of solder, or to progress inside the grid like a ground war or a disease—taking and holding site after site, and thus tile after tile—but only in ways allowed by the laws of MFM physics.

2.5. Bonds and mobility

A namesake aspect of the MFM is that atoms *move*, for a variety of purposes, as implemented via copying and erasing site contents. A communications mechanism, for example, could employ atoms—interpreted as data or packets—moving relative to sites or other atoms acting like a channel or a wire (see Section 4.2 and [3]). Atomic mobility is also handy in self-reproducing systems, and so offspring might eventually move into space of their own [53].

When atoms can move, though, their current location cannot reliably be used to find them in the future, causing major headaches for distributed data structures, and spawning a variety of schemes ([54] is one survey; see also Section 1.4.2) for updating or forwarding pointers in the face of object migration.

In the MFM, an atomic *bond* can be used to join two atoms in a relationship that survives certain atomic motions. A bond connects precisely two atoms, but one atom can have multiple bonds, allowing the creation of a connected-component *molecule* involving perhaps many atoms. A molecule as a whole may be much larger than an event window, and so in general, an entire molecule cannot expect to update at one time. Bonds are designed so that the element description specifies a common bond layout for all atoms of that element, and the hardware always knows which bits of each atom represent bond information. Bonds are distance-limited and cannot be longer than the event window radius, and are represented by self-as-origin relative spatial coordinates. The P1 atom offers 'long bonds' and 'short bonds' (Fig. 4, also 'bond' and 'sbond' in Fig. 1) which trade address range (maximum ℓ_1 length 4 and 2, respectively) against atomic bit usage (8 and 4, respectively).

Importantly, each bond is symmetric—if atom A has a (long or short) bond to atom B, then B will have a same-type bond back to A. This redundant representation helps detect inconsistencies and ensures that the bond, like a physical chemical bond, can be detected, updated and broken from either side.

By default, atomic diffusion (step 4 of Fig. 2) only considers one site rectilinear moves, and it handles existing bonds automatically, refusing to break or overstretch them, while weakly preferring to minimize bond lengths, and it automatically updates bond coordinates as needed if the active atom does diffuse. In our simulations to date, an element description can also affect the default diffusion process by providing a set of per-direction *diffusion weights* to bias the outcome, or by replacing the default diffusion mechanism entirely; how much of that flexibility is worth direct hardware support remains to be seen.

3. PERFORMANCE ESTIMATION

The MFM is an abstract architecture designed for indefinite scalability, research flexibility and implementation potential in hardware both more and less conventional. It abstracts away the event window selection mechanism, requiring only that implementations avoid site starvation, and it is silent about the duration of any given event transition (Fig. 2, steps 3 and 4). Measuring computation time in AEPS, similarly, suppresses many details of any real machine, such as its number of sites, clock rate, and communications and locking overhead between adjacent processing elements.

Particularly at this early stage, we argue that such abstractions are appropriate for research in indefinitely scalable computations. If MFM-like models prove usable for robust computation at scale, we expect that subsequent science and engineering could yield indefinitely scalable implementations highly optimized for the capabilities and needs identified in the abstract. Still, some may worry that MFM overhead could be fundamentally prohibitive, and so for concreteness, in this section we describe one approach to caching and locking, and estimate its performance via simulation of a plausible though hypothetical hardware substrate.

We produce estimates of *event efficiency*, the percent of the total machine cycles actually devoted to event transitions as well as the *average event rate* (AER), the overall AEPS per second of the indefinitely scalable machine. The simulations suggest that if we can tolerate a factor of two in event rate variation across tile sites, then, for plausible event durations, we can expect an indefinitely scalable AER in the several hundreds to a thousand or more, without heroic engineering, using an affordable tile suitable for research.

3.1. A caching and locking model

As suggested by Fig. 1, the MFM posits *tiles* each containing a processor, internal memory for data and code, and external interfaces for intertile communications. Referring to Fig. 6, given some chosen *tile side* N, an $N \times N$ grid of sites is represented in the memory of each tile. Given some chosen *event window radius* R, the sites are divided into functional

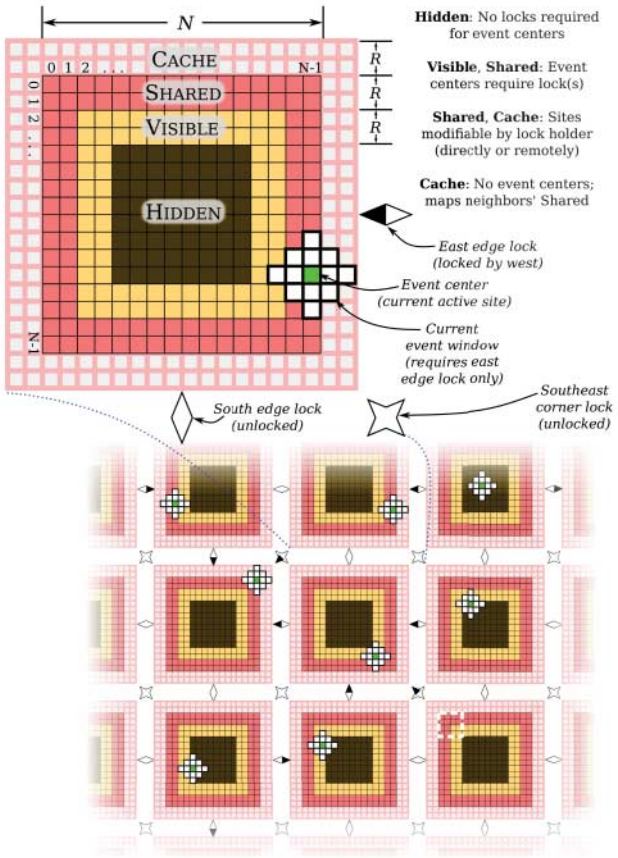


FIGURE 6. An indefinitely scalable caching and locking model for MFMs, illustrated with $R = 2$ (event window radius) and $N = 16$ (tile side); see also Section 3.1.

groups: the *shared* sites located within R of the grid edge, the *visible* sites located from $R + 1$ through $2R$ of the edge and the *hidden* sites located more than $2R$ sites inward. Also in tile memory is an R-wide *cache* surrounding those N^2 sites, containing an additional $4R(N + R)$ sites for local copies of the adjacent portions of neighboring tiles shared sites. Externally, the model possesses communications and locking interfaces, with a communications channel and a mutex on each edge between orthogonal neighbors, and a diagonal pair of channels and mutexes on the following corner.

In each tile, the center site selection and locking process (Fig. 2, steps 0 and 1) is a loop consisting of these steps:

(1) Select a candidate event center (x_c, y_c) at random in $(0 \cdots N - 1, 0 \cdots N - 1)$.
(2) Determine what locks (x_c, y_c) requires. For example, if x_c is east of hidden, require the east lock (Fig. 6, top); if in addition, y_c is north of hidden, require the north and northeast locks as well (Fig. 6, bottom, middle left tile; see other locking examples there as well).

(3) Attempt to acquire each required lock, without blocking. If all are obtained, then selection and locking is complete and event window processing continues (Fig. 2, step 2); otherwise release any obtained locks and go to step 1.

Since our primary purposes here are exposition and ballpark assessment, we sometimes favor simplicity over performance. We assume that *Notify* processing (Fig. 2, step 6) always transmits the entire event window, as uncompressed bits. Also, this method locks conservatively—for example, in Fig. 6 the lower right tile is barred from centering an event in the white-outlined region, even though any such event window actually would be disjoint.

Although no site starves, note also that since this mechanism reselects rather than waiting on any lock, the edges and corners see lower event rates due to communications delays and lock contention—in the simulations below, there is typically a factor of two between the least- and most-sampled sites. If needed, such hardware-induced rate variations can be reduced by decreasing performance and/or increasing cost and complexity—for example by warping site selection probabilities to compensate, using more and finer-grained locks, or increasing channel data rates relative to processor speed.

3.2. Hypothetical hardware substrate

Model performance depends on the chosen MFM parameters, the tile side and the durations of event transitions, as well as the speeds of processing, locking and communications. As yet we have no indefinitely scalable implementations, and so here we employ a hypothetical tile processor whose parameters are summarized in Fig. 7. Note that the average event duration may depend strongly on the interactions between the hardware details and the user-defined physics, and so we vary that parameter over three orders of magnitude to help explore the space. To keep the results manageable, we explore only the P1 parameters: 64 bits per site and $R = 4$; we simulate variable event transitions by choosing event durations uniformly at random in ±50% of the average.

Name	*Min*	*Max*	*Unit*
System clock	100		MHz
Instruction time	2		cycles
Intertile mutex lock try	20		cycles
Intertile mutex unlock	10		cycles
Intertile I/O speed	1	10	Mbps
Tile side (N)	20	40	sites
Average event duration	10^3	10^6	instructions

FIGURE 7. Parameters of a hypothetical MFM processing tile, for performance prediction. Non-empty *Max* indicates multiple parameter values tested.

Our chosen hardware specifications are deliberately at least an order of magnitude below state of the art in clock and I/O speeds. We envision a tile of moderate cost even manufactured in only research quantities—perhaps using a mid-range FPGA plus memory, or a microprocessor plus added interfaces. At present, however, all such details remain to be engineered.

3.3. Performance estimates

We built a discrete event simulator to count events and track where the clock cycles are used by the model. For each combination of parameters tested, we configured a 5 × 5 grid of tiles connected in a torus and simulated it for one second (100 M clocks per tile). Results are averages across the 25 tiles; the observed variance is low in all cases shown.

For a tile side of 32, Fig. 8 shows event efficiency and the AER for a range of event durations and communications speeds. Unsurprisingly, event efficiency rises with event duration, while AER falls; faster communication helps both but the gains dwindle as events lengthen. Figure 9 illustrates the sampling bias discussed above, showing the AEPS/sec seen by different sites within a 32 × 32 tile. The corner sites require three locks and see about 630 events/second; the hidden sites require none and do about 1100.

To suggest the cost of eliminating that spatial variation, should that prove worthwhile, Fig. 10 plots the 'uniform AER' obtained by limiting all sites to the event rate seen by corner sites. With a 3 Mb data rate, $N = 32$ and 10 K instruction events, a uniform AER around a thousand looks achievable.

The simulation results confirm that in this implementation, MFM performance is highly sensitive to communications costs. The transmission of a full uncompressed window for each event is expensive but robust—requiring little agreement between tiles and tending to stabilize cache contents after transient errors or tile reconnections. If, in practice, most event durations turn out to be brief and make few event window changes, a more efficient cache update mechanism could be a valuable model improvement, despite the added complexity and fragility.

Evaluating such engineering decisions depends largely on whether we should consider 500 or a 1000 AEPS/second a good AER or a bad AER, uniform or otherwise. We can always want more performance, but ultimately the answer to that depends on what sorts of things we can do with events, which is the subject of the remainder of this paper.

4. BASIC MOVABLE ELEMENTS

Here we present some of our exploratory results from programming MFM elements and simulating their behaviors in various-sized grids. First, we examine the basic task of distributed density regulation; then we look at simple 'wire' mechanisms for point-to-point data transport, and finally we

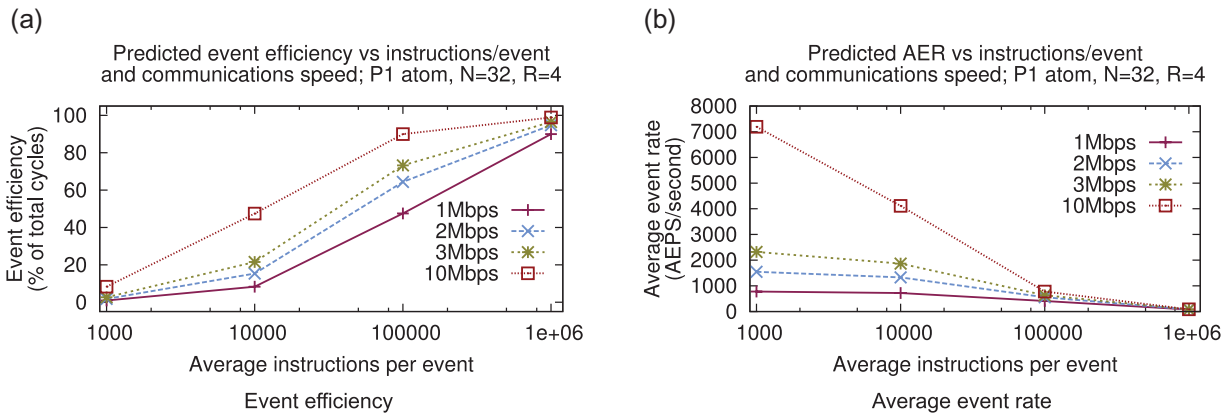


FIGURE 8. Tiled event processing.

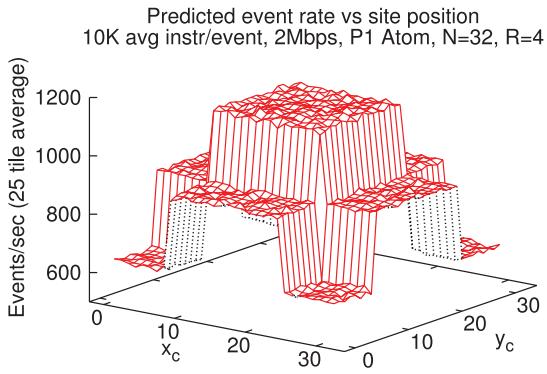


FIGURE 9. The event rate by site.

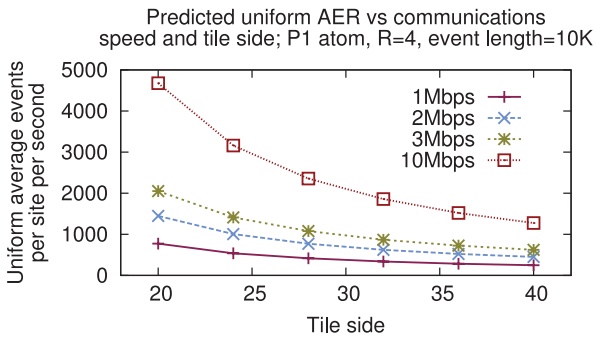


FIGURE 10. A uniform event rate.

modify the wire and close into a ring to form a simple 'membrane' that divides space.

As mentioned, all these examples are based on the P1 parameter set—a rectangular lattice in 2D with Manhattan distance, varying grid sizes, event window radius of 4, and atomic width of 64 with a 16-bit header and 48 programmable bits. Our purpose at this early stage of research is to explore possibilities and trade-offs, and so the different experiments and demonstrations focus on different aspects and subsets of robustness, functionality and movability, as discussed in each section.

4.1. Dynamic regulation

It is easy to overlook, but in the MFM probably the single biggest spatial issue is empty space management. In an asynchronous universe especially, empty sites are precious to facilitate the activities of nearby occupied sites—for example, for temporary use during reconfigurations, or to allocate for new atoms—as well as to enable the movements of travelers passing through. And since any given empty site is in the neighborhood of many other sites, without effective open space management a tragedy of the commons can easily ensue, producing traffic jams, gridlock and similar hazards.

Traditionally such problems are managed by careful design, and capacity simulations and analysis, but hardcore robustness offers a much sloppier idea: if empty sites get rare, just *make more*, by erasing some atoms—which might disrupt some ongoing computations, but so what? If the computations are robust, they will have spares or make repairs—and if they are not, we cannot rely on them anyway.

We explore controlling the *occupied site density* (OSD) with `DReg`, a diffusing 'Dynamic Regulator' element that forms no bonds and maintains no persistent state (Fig. 11). On each update, `DReg` inspects a random adjacent site. If the site is empty, `DReg` might create a general-purpose 'resource' atom (element `Res`), or rarely another `DReg`; if it is occupied, `DReg` might erase it, particularly if it is another `DReg`. See the pseudocode in Fig. 12.

Over time, a lone `DReg` will fill MFM space with a churning mix of `DReg`, `Res` and empty sites, with an OSD related to the ratio of the creation chance to the sum of the chances of creation and destruction. We commonly use a creation ratio of about $1/3$ (close to $dodds/(modds+dodds)$ in Fig. 12), but the specific probabilities also matter: smaller values yield a looser

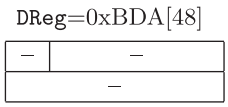


FIGURE 11. DReg: dynamic regulator (see also Fig. 5).

```
// Pseudocode
element DReg() = 0xBDA {
  constant modds = 40   // approx odds to make Res
  constant dodds = 20   // approx odds to destroy
  Loc at = randomLoc((1,0), (-1,0), (0,1), (0,-1))
  Type atType = window[at].type
  Type t = undefined

  if atType is Empty.type then
     if randomOneIn(1000) then t = DReg.type
     else if randomOneIn(modds) then t = Res.type
  else if atType is DReg.type then
     if randomOneIn(10) then t = Empty.type
  else if randomOneIn(dodds) then t = Empty.type

  if t is not undefined then
     unbond window[at]
     window[at] = new atom of t // may be empty
}
```

FIGURE 12. DReg update pseudocode. Also see text.

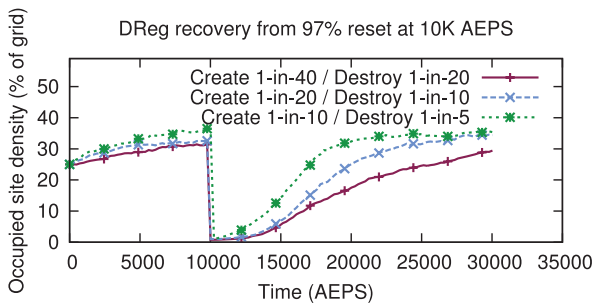
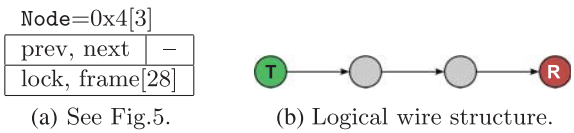


FIGURE 13. DReg density regulation after a transient.

regulation and slower transient response; larger probabilities are faster but more disruptive. Figure 13 shows OSD regulation for three different `DReg` parameter sets after a system is shocked by erasing 97% of the sites at 10 K AEPS.

OSD regulation is a basic housekeeping task, but if we add other elements that perform some useful computation, while competing for `Res` to reproduce themselves, we can produce a combined system in which `DReg` operations are 'space shared' with other tasks, as shown in Section 5.1. As we have gained experience, we have found dynamic regulation and `Res` production to be so fundamental and useful that sometimes, instead of `DReg`, we use 'direct dynamic regulation', incorporated into event window processing, in



(a) See Fig.5. (b) Logical wire structure.

FIGURE 14. A simple wire.

effect as an additional component of step 2 in Fig. 2, with separate parameters for atomic deletions and `Res` creation.

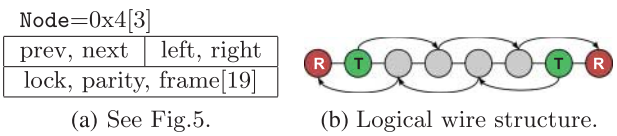
4.2. Movable data transport

While the MFM provides no explicit mechanism for sending messages between atoms, atoms may communicate with each other by reading and writing each other's state. However, this level of communication is possible only between two atoms that are separated by no more than one event window radius. While atoms can use bonds to maintain their proximity, locality is a precious resource, and modularity requires mechanisms for longer range communication.

We have previously [3] introduced the concept of using a bonded chain of atoms—a 'wire'—to facilitate non-local communication. Our most straightforward version, the *Simple Wire* depicted in Fig. 14, is essentially a spatialized doubly linked list. A Simple Wire chain consists of a long-bonded sequence of `Node` atoms forming a path between a `Transmitter` and a `Receiver`, with each `Node` atom's `next` and `prev` bonds linking downstream and upstream, respectively. (Recall that all MFM bonds are bidirectional and symmetric; the arrows in Fig. 14b, from tail to head, join atoms with matching `next` and `prev` bonds, respectively.) Such a wire chain can be used spatially or mechanically (e.g. as a navigational aid to locate a destination), but can also be used to transport a bit stream, using the leftover state of each node to store in-flight data chunks as they propagate down the wire.

A *Simple Wire* `Node` has 29 bits available for state. We draft one bit as a `lock` to indicate whether the contents of `frame` are valid, and deploy the rest as a 28-bit `frame` data field. When a `Transmitter` wishes to send a message, it waits for its downstream `Node`'s `lock` to clear, then writes the next 28 bits of data into that `frame` and sets its `lock`. Then, whenever a `Node` is unlocked and has an upstream locked `Node`, it copies the upstream `frame`, clears the upstream lock and locks itself; a `Receiver` disposes of the data however it wishes, and then unlocks the upstream frame.

This mechanism successfully transfers bits from `Transmitter` to `Receiver`, but it is fragile. If a single atom is erased, due to corruption or `DReg` or any other reason, the entire wire fails irretrievably; even short of that catastrophe, corrupted bits can cause bitstream data errors or the loss or duplication of entire frames. To address the fragility of the *Simple Wire*, we have devised a more robust variant that we call *Self-Healing Wire* (SHW), illustrated in Fig. 15. SHW



(a) See Fig.5. (b) Logical wire structure.

FIGURE 15. A self-healing wire.

is an extended-range, bidirectional communication channel with redundancy and repair mechanisms that consists of two wires, laid out in opposite directions and stitched together by short bonds. Each `Node` of an SHW has a `left` and `right` bond to its immediate neighbors and a `prev` and `next` bond to the neighbors of its neighbors. As with *Simple Wire*, `prev` and `next` bonds indicate the direction in which messages travel; however, messages may now travel in two directions along the wire. Furthermore, if a SHW `Node` bond or atom fails, adjacent atoms will attempt to reconstruct it using the information encoded by the remaining bonds. Of course, there is no free lunch, and implementing SHW repairability costs two short bonds and one state bit for `Node` parity. That leaves only 19 bits per `frame`, and so the maximum data rate is reduced—and if a second `Node` failure occurs nearby before a first failure has been repaired, the SHW chain will fail anyway.

To improve the robustness of data, we implemented a TMDS [55] encoding layer above the bitstream, even though many features of TMDS—such as its DC-balanced signaling—are unnecessary for the purpose at hand. TMDS provides packet framing and clock recovery, converting a raw bitstream into a more useful self-synchronizing byte stream, but our use of it does significantly reduce the efficiency of data transmission. Without TMDS, sending 1 KB (8000 bits) over a 100-atom *Simple Wire* yielded a raw data rate of 7.03 bits/AEPS ($\sigma = 0.07$), averaged over 200 error-free trials. Under the same conditions, the SHW yielded 4.80 bits/AEPS ($\sigma = 0.04$). When TMDS encoding overhead plus one-for-one synchronization packets are included, the *Simple Wire* effective data rate drops to 2.808 bits/AEPS ($\sigma = 0.03$) and SHW's drops to 1.915 bits/AEPS ($\sigma = 0.01$). Less frequent synchronization packets would improve the data rate but decrease robustness; optimal parameters depend on the specific computation and failure modes.

We have previously [3] shown that SHW significantly outperforms *Simple Wire* when measuring the survival time (i.e. time until the wire becomes disconnected) at any non-zero fault density. However, this provides little insight into the real performance trade-offs of each wire implementation. To investigate further, we simulated the transmission of a TMDS-encoded 1 KB message along a 100-atom *Simple Wire* and SHW between a transmitter and a receiver, each anchored at either end of the wire. The error model is atom deletion via the 'direct dynamic regulation' (Section 4.1), with `Res` production set to zero and varied rates of site erasure.

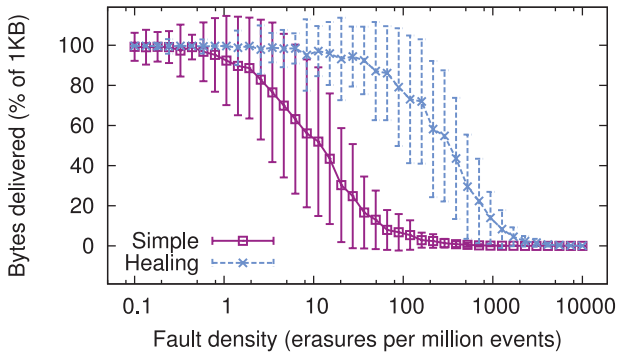


FIGURE 16. The average fraction of a 1 KB message successfully delivered for each wire implementation.

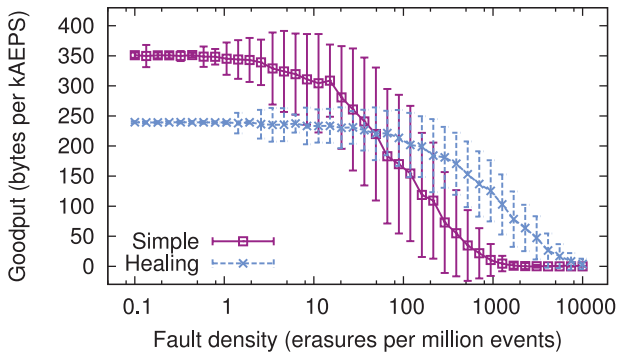


FIGURE 17. The average goodput for each wire implementation.

The resulting data reflects the expected efficiency-vs.-resilience trade-off, both in terms of the fraction of bytes lost (Fig. 16) and the rate at which correct bytes were delivered (Fig. 17) during the transmission of a 1 KB message at various fault densities. *Simple Wire* achieved a far greater goodput at low fault densities than the SHW; however, as the error rate increased, the performance of the Simple Wire degraded far more rapidly than the SHW. Variances are high in all cases because of the timing variability and catastrophic impact of a wire breakage.

4.3. Movable membranes

An absolutely fundamental aspect of spatial computing, and robustness generally, is the ability to compartmentalize space. This is sometimes easy to achieve—for example during the manufacturing of non-movable architectures—but to produce movable and yet isolated compartments is more challenging. We are exploring *movable membranes* to implement a selectively permeable spatial divider, which derive their inspiration—though little else—from the cell membranes of biological systems.

Of course, given the two-dimensional worlds of the P1 parameter set, these 'membranes' are really more like rings.

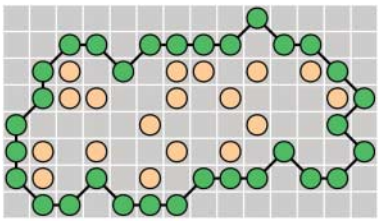


FIGURE 18. A movable membrane whose contents cannot diffuse out. Also see text.

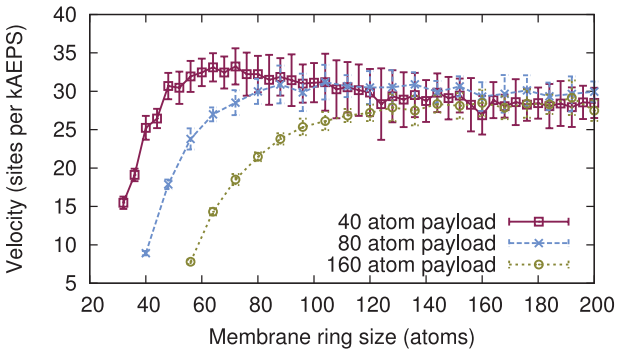


FIGURE 19. Velocity of direction-biased AVMM rings vs. membrane size for various payloads. Also see text.

To date, though not in a single simulation, we have prototyped movable membranes that: grow to arbitrary sizes, repair damage caused by `DReg` and bit corruptions, communicate with other membranes via SHW and transport unbonded contents through space in a fixed direction. Here we demonstrate only the last, after a brief general discussion.

An MFM design for a movable membrane is constrained by two opposing requirements: the membrane should be impermeable to the passage of non-membrane atoms undergoing normal diffusion, but at the same time the membrane's constituent atoms must have sufficient free space to allow the membrane itself to move. A *Simple Wire* chain fashioned into a ring, for example, is typically very movable but also very porous to the passage of atoms through it. Conversely, a box of the adjacent, immobile atoms is impermeable to diffusion but also unable to move.

We have previously employed in [3] a fairly elaborate approach to constructing semipermeable membranes, but more recently have found that we can achieve similar effects with considerably less complexity, by combining the limited set of directions considered by diffusion (Section 2.3) with an additional constraint on membrane bond lengths. Given a short-bonded ring of atoms, under normal circumstances each ring atom may be separated from its neighbors by up to a short bond length ($\ell_1 \leq 2$). If we instead require that membrane atoms limit their separation from neighbors to an ℓ_∞ distance of 1—that is, they must both lie within each other's Moore neighborhood—then we still have some room to move but can prevent the escape of diffusing particles, as illustrated in Fig. 18.

While in the absence of errors this simple membrane design is both movable and impermeable, it suffers from the same fragility as the *Simple Wire* (Section 4.2), but we can also improve it the same way. A *Self-Healing Membrane* (SHM) consists of atoms with two short bonds to their immediate neighbors in the ring and two long bonds to the neighbors of their neighbors, with a parity bit to assist reconstruction. At present, however, we have just the beginnings of experiments with the SHM. We also have preliminary work on membrane growth mechanisms based on transmuting and incorporating environmental `Res`, but that is likewise immature.

Here, we show one additional membrane technique that we have explored in some depth. In addition to creating distinct inside-vs.-outside spatial environments, membranes that are *movable* can act like ratchets or guides, shepherding their contents in some specific direction, even when the contained atoms move solely by diffusion. An *Absolute Vectored Movable Membrane* (AVMM) atom is like a normal membrane node except that it also overrides the diffusion scoring mechanism, scaling the diffusion scores based on a 4-bit 'preferred direction' field. With that change, plus some mechanism—either initial conditions or something more dynamic—to achieve a preferred direction consensus around the ring, significant 'motive force' can be generated.

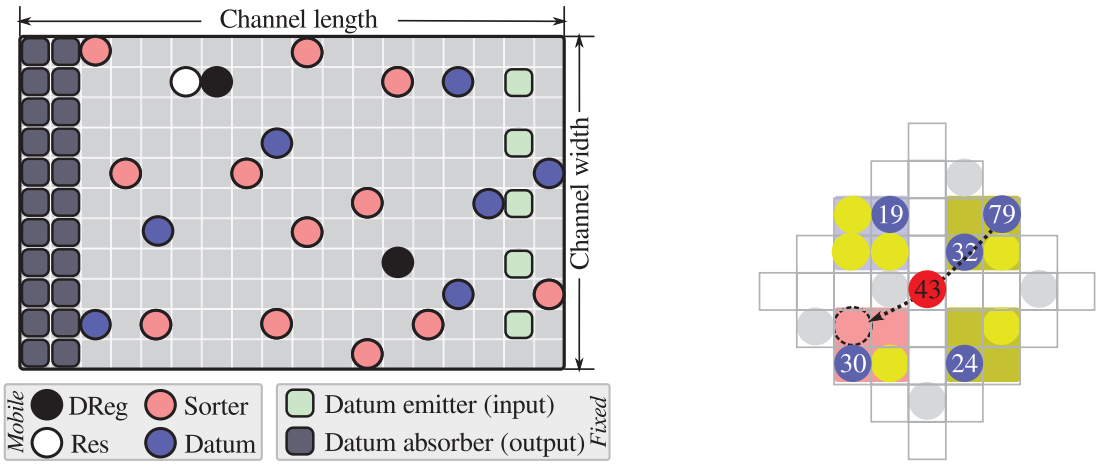
Figure 19 illustrates typical results, showing the velocities attainable by AVMM rings of varying sizes containing varying numbers of unbonded payload atoms.

5. INTEGRATED COMPUTATIONS

The previous section illustrated a few strategies and components of robust spatial computing in the MFM. Here we offer brief looks at two MFM approaches to more complex computational tasks.

5.1. Robust spatial sort

To help perfuse robustness into the computational stack, we seek ways to intertwine it with functionality—and we are more than ready to reframe notions of functionality to that end. Here, for example, to help break our obsession with correctness and efficiency, we explore a sorting task that is impossible to solve perfectly. We imagine a rectangular 'flow sorting channel', depicted in Fig. 20a, given the task of sorting or prioritizing an endless stream of `Datum` atoms that are injected at random intervals by 'emitters' near the right side of the grid. Each `Datum` contains a 32-bit number (and an 8-bit checksum), and is to be transported to the left and also sorted vertically, so that small values rise and large ones sink. Once a `Datum` is close to the left edge, it will be consumed by a nearby 'absorber' and output from the sorting channel.



(a) Elements and global structure

(b) Sorter behavior

FIGURE 20. Elements of the demon horde sort.

Here we focus on an *equal interval* goal where each output 'bucket' (the absorbers on a single row) should receive the values of an equal portion of the underlying data range.[1] We measure performance by the *average positional error*—the average distance between the bucket that absorbs a `Datum` and its correct equal interval bucket, as a percentage of the number of buckets.

We assume an unknown, perhaps non-stationary, data distribution, and so perfect bucketing is simply off the table. For a baseline comparison, we use this 'Sample Sort' heuristic: Given N buckets, repeatedly buffer up N `Datums`, sort them and then output one sorted `Datum` to each bucket in order.

We call our robust spatial sorting strategy the 'Demon Horde Sort'.[2] It builds on `DReg` and `Res` (Section 4.1) and adds a `Sorter` element, illustrated in Fig. 20b. In this version, `Sorter` has two primary functions. First, whenever it sees a `Res`, it transmutes it into another `Sorter`, and so the `Sorter` population level is indirectly controlled by `DReg`. Secondly, `Sorter` transports `Datums` from right to left when possible, and also up or down based on the comparison of the `Datum`'s value with a 32-bit *threshold* stored in the `Sorter`. When a `Datum` 'crosses' the `Sorter` during a move, the `Sorter` copies the `Datum`'s value to its threshold—in Fig. 20b, the `Sorter`'s threshold will soon be 79.

In this standalone demonstration the emitters and absorbers suppress their MFM diffusion and remain stationary; in addition to their I/O functions, they 'buddy check' their same-element neighbors and recreate them if they are missing but the site is available—e.g. after an erasure by `DReg`. The initial condition consists of some `DRegs` and `Sorters` scattered in the channel, and appropriately placed emitter and absorber 'seeds' from which the I/O grids establish themselves. Figure 21 depicts a demon horde that has been running for some 100 K AEPS in a 65-bucket flow sorter, illustrating its typical equilibrium structure: near the emitters, on the right, the thresholds are choppy as diverse data values pass through, but after some distance the thresholds become largely laminar, making (and remaking) increasingly fine distinctions as `Datums` close in on the absorbers.

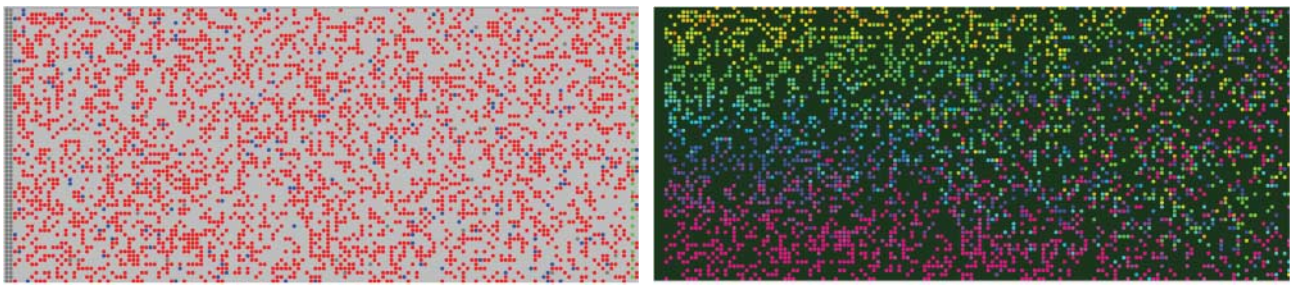
A pleasant aspect of writing modular, 'low commitment' behaviors, like `Sorter`'s locally sensible notion of sorting, is the extra freedom it can provide later in the design. For example, varying the channel length allows trading off hardware and latency against sorting performance, as illustrated in Fig. 22. At a channel length of 5 (in which case the emitter and absorber neighborhoods overlap) performance is random; by the time the channel length reaches 150 or so, performance begins to saturate at about the 'sample sort' heuristic level.

The demon horde sort's performance may be just adequate, by that measure, but its robustness seems quite impressive. Figure 23 shows results of one experiment in which we randomly corrupted site memory with simulated bit errors at a range of probabilities. Each error occurrence selects a random site and then flips from one to eight of its 64 atomic bits. We can see that while channel length helps performance, it does not help robustness against this system perturbation—but the system is strikingly robust anyway, tolerating upward of 10 multibit corruptions per million events with essentially no visible performance degradation, regardless of channel length.

Above about 50 errors/Mevent the system reliably falls apart—and the pathology appears to run a reliable course: the

[1] More general is an *equal frequency* goal, which asks buckets to receive `Datums` equally often on average; here the two goals coincide because the test distributions are uniform random.

[2] Note that this is a simplified version compared with that in [3].



(a) All Types (b) Sorters only

FIGURE 21. Filtered views of a 150 × 65 flow sorter, sorting right to left, at $t \approx 100$ K AEPS. Colors (or gray levels) in (**a**) represent element types as in Fig. 20a; (**b**) shows just `Sorter` thresholds from smallest (*orange*) through intermediate (*blue*) to largest values (*pink*).

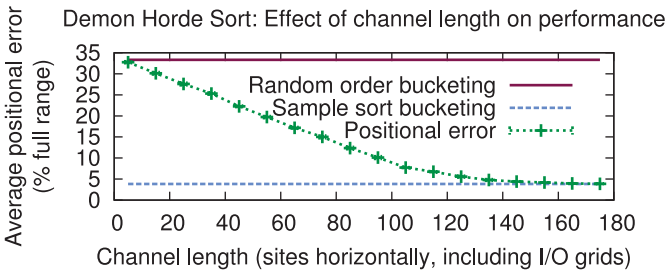


FIGURE 22. Sorting performance.

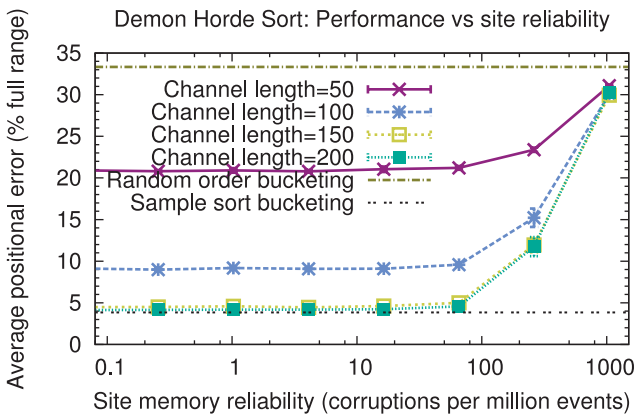


FIGURE 23. Sort performance on unreliable hardware.

bit flips trigger the error pathways in Fig. 2, which wipes out the `DReg` population the fastest because they are rare and the slowest to reproduce, and their demise accelerates the extinction of the `Sorters`, which leaves the emitted `Datums` moving only by diffusion, and they mostly fail to reach any absorber (let alone the right one) before they too are detectably corrupted and erased—and for scoring purposes, we count such lost `Datums` as if they had arrived at a random bucket.

If we dispensed with `DReg` and `Res`, and added direct `Sorter` behaviors for managing their own population level, the system would likely be somewhat more robust to this specific perturbation, at the expense of more custom physics and giving up some compositionality, such as the possibility of simultaneous regulation of multiple elements via competition for `Res`. We have only begun identifying such trade-offs and sweet spots in MFM design space.

5.2. Distributed virtual machines

The examples in Section 4 are admittedly low-level, and the Demon Horde Sort turned sorting from an algorithm into a stochastic process, so what can we say for 'good old-fashioned' serial determinism? In principle, if robustness and indefinite scalability are ignored, then a Turing machine or a von Neumann machine can be implemented, painfully, in an MFM, and traditional computational effects thus produced—but of course we argue that indefinite scalability and robustness are both critical for future computational growth.

A movable feast computation is indefinitely scalable if it can always make effective use of additional hardware. Although the ability to solve larger problems is one effective use of additional hardware, there are others, such as the ability to solve a fixed-size problem more quickly, or—as our focus in this section—with an increased chance of success. Here, we describe a method for MFM evaluation of compiled expressions of a functional programming language. Figure 24 provides an overview of the entire process.

One standard method [56, 57] to evaluate an expression in a functional programming language is, first, to compile it into a *bytecode program* (Fig. 24, left). Traditionally, such a bytecode program is then interpreted by a *Virtual Machine* (VM), a program that simulates a von Neumann computer. The VM maintains a set of registers containing numbers which can also serve as addresses of memory locations that contain both instructions and data. A special *program counter* register holds the address of the current instruction, which when executed transforms the contents of the registers and memory in some way, then the program counter is updated to address the next

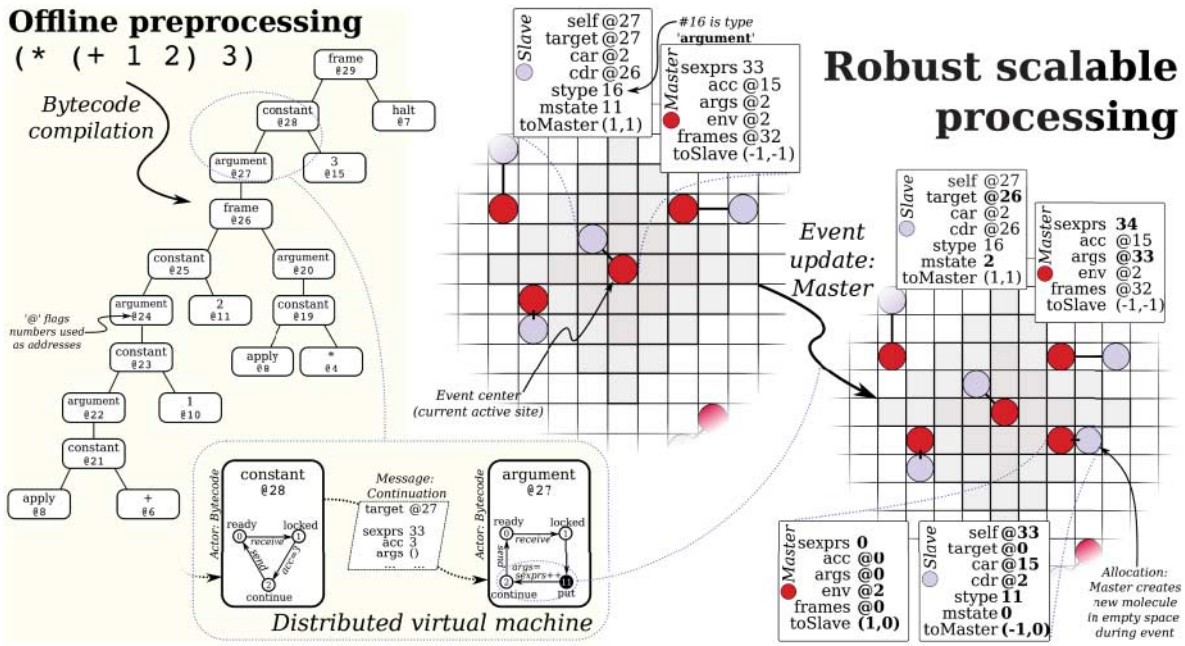


FIGURE 24. Robust distributed evaluation of a functional language expression. *Left to right*: Compilation of a Scheme expression to a bytecode tree; part of its implementation as a distributed virtual machine (DVM), using actors passing continuations; one of those actors reified as an MFM molecule, allocating a new pair during an MFM event. Also see text.

instruction and the computation proceeds step-by-step until a halt bytecode is executed, at which point some part of the VM state represents the result.

Instead of such a traditional, centralized VM, here we develop a *distributed virtual machine* (DVM) implemented using actors. An *actor* [58–62] is a universal primitive of concurrent computation, a lightweight process with a unique address which can exchange messages with other actors. On receiving a message, an actor can send messages, create new actors and update its internal state.

We use actors to represent simple versions of Scheme data types including pair, number, primitive function, closure and boolean—and especially, a *bytecode actor*, representing a bytecode of a compiled Scheme program. To avoid the central registers of a von Neumann machine, we encapsulate the VM state in a message called a *continuation* that can be sent from actor to actor. Overall, a compiled set of bytecode actors, using message passing to exchange continuations, forms a DVM (see a fragment of a DVM in Fig. 24, lower middle). When an active bytecode actor sees that a continuation has arrived, it transforms it in a manner specific to its kind, then sends it on to the next bytecode in the program, as determined by the compilation process. Eventually the continuation reaches a *halt* bytecode and its *accumulator* contains the evaluation result.

Finally, we reify each DVM actor as a two-atom molecule undergoing diffusion in the MFM (Fig. 25, and Fig. 24, right). With this approach, an 'address' represents a value to be matched, rather than a memory location fixed in space. In this approach, messages are not reified as atoms or molecules; instead, a message transmission occurs implicitly, during a single event, when an active message sender recognizes the addressed recipient in its neighborhood, and changes the recipient's state, and its own, accordingly.

The event window update depicted in (Fig. 24, right) shows an actor creation event, when the 'argument' actor with address 27 creates a new pair referring to the constant 3—part of setting up the function call that will perform the multiplication specified in the original expression.

By using only pure functional programming with no side-effects, we guarantee that any heap-objects that are compiled to, or created at, any given address are absolutely interchangeable for purposes of that program, no matter which actors created them or when they were last accessed. As a result, multiple active continuations can coexist without inconsistency in one distributed heap, and we can improve robustness by including redundant sets of heap-objects—as we explore momentarily.

The programming language used is a purely functional subset of Scheme [63]. In addition, we restrict user-defined functions to one argument—so higher arity functions must be curried—allowing us to implement all variable references by indexing into a flat evaluation stack [64]. We also introduce a new Y-combinator-like special form *lambda+* to create locally defined recursive functions at runtime.

In all other ways, we faithfully implemented the heap-based compiler and VM for Scheme described by Dybvig [57], which created tremendous pressure on the atomic bit budget. Each

`Master`=0x4[4]

–	toSlave
sexprs[8], acc[8], args[8], env[8], frames[8]	

(a) Reified actor: Master. See Fig. 5.

`Slave`=0x3[2]

–	toMaster
self[8], target[8], car[8], cdr[8], stype[5], mstate[5]	

(b) Reified actor: Slave. See Fig. 5.

FIGURE 25. Actor component elements.

bytecode actor needs to represent nine heap addresses: five for the VM registers encapsulated in the continuation, two for the child pointers in the bytecode tree ('car' and 'cdr'), one for the message recipient and one for a self-pointer—and the number of bits allocated per address determines the maximum heap size available jointly for compiled program and all runtime data. In addition, five more bits are needed for the heap-object type, and four more bits to represent the execution state of the most complex bytecode.

Given all that, the 48 P1 bits proved just too tight, so we reified each actor as a small molecule: a two atom master-slave pair joined by a short bond (Fig. 25). For simplicity we put all the 'behavior' in the Master element update; the Slave element update does nothing. These actor molecules offer 88 programmable bits and support 8-bit addresses which, though small, do permit the evaluation of a relatively complex expression like

```
((lambda+ f x
     (if (= x 1) 1 (+ x (f (- x 1))))) 9)
```

which sums the integers from 1 to 9, and compiles into 67 unique heap-objects. As a first experiment, we randomly distributed 16 copies of that compiled set in a 128 × 128 fault-free simulation, and found that it typically ran for about 3 million AEPS before any of the 16 continuations successfully halted, reaching a heap size of 232 in the process.

With confirmation that the DVM was computing correctly in an error-free environment, we began to probe redundant DVM robustness. In a second experiment, we added random bonded pair deletions as the only possible fault, in which an entire failing actor molecule is erased cleanly, with no other form of corruption. We evaluated `(* (+ 1 2) 3)` as illustrated in Fig. 24, which produced a compiled set of 29 unique heap-objects—the 19 visible in the bytecode tree plus various other built-in constants and primitive functions. Another 13 heap-objects are created while the program runs. The fault rate ranged from 1 to 256 molecule deletions per million events; the redundancy r ranged from 1 to 16. Since the average message delivery time depends on actor density, we scaled the MFM size to keep the starting density constant across redundancy; the size of the simulated MFM was $N \times N$ where $N = 16 \times 2^{\log r/2}$.

Figure 26 shows the results, with the y axis giving the fraction of 100 trials per condition that succeeded—meaning that any continuation executed the *halt* bytecode before all actors vanished or the simulation time limit of 50 kAEPS was

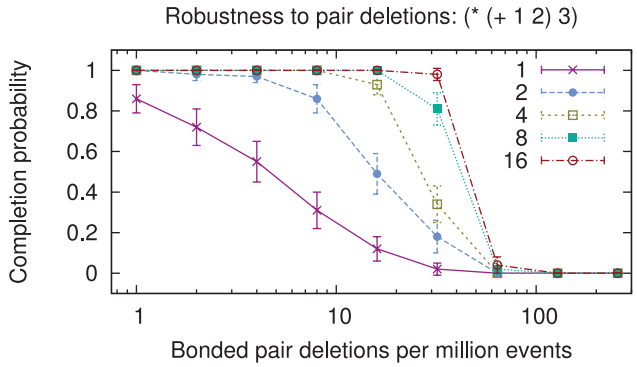


FIGURE 26. The performance of DVMs with different levels of redundancy as a function of the actor failure rate.

reached. A perhaps unexpected result is that beyond about 50 or 100 molecule deletions per million events, additional redundancy seems to have no effect on robustness. However, deleting actors (without also removing sites) decreases actor density, which increases message latency, which slows the overall computation, decreasing its chance of ultimate success. Here, DVM computations are less like a traditional VM and more like chemical reactions in which, all else being equal, it is the reactant *concentrations*, rather than their absolute number, that govern the reaction rate.

Given the results of the second experiment, it was natural to explore DVM robustness from another direction, and consider DVM success rates given catastrophic permanent removal of a fraction of the entire machine, rather than scattered individual actor deletions. As we explored it, 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 7500 AEPS; the expression evaluated, termination criteria and trials per condition were the same as in the second experiment. Figure 27 reveals that higher levels of redundancy result in higher probabilities of successful completion over the full range of fractional board failure sizes. For example, even with a fractional failure of 70%, the 16-way redundant DVM successfully completed 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.

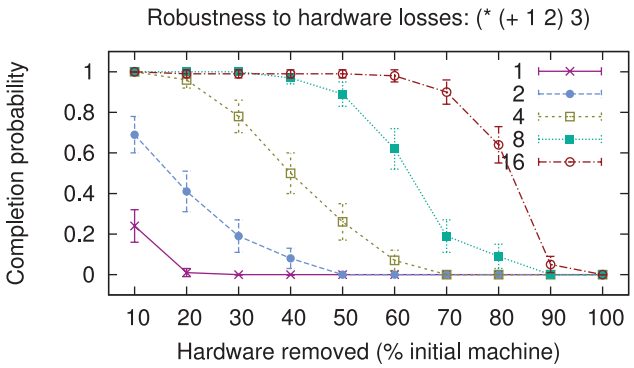


FIGURE 27. The performance of DVMs with different levels of redundancy as a function of machine failure fraction.

6. CRITIQUE AND CONCLUSION

From the perspective of cellular automata, it is certainly true that the MFM is a more complex and articulated design, and its state transition programming is much harder to explain than the pure simplicity of a state table. Of course, for designing computations at scale, the state table's apparent simplicity is an illusion, not dealing with complexity, but simply pushing it into larger assemblages, like trying to implement a conventional computer using only NAND gates.

Furthermore, from the direction of traditional programmable computers, one obvious criticism is that all this makes the programmer's job that much harder, asking for a spatial, as well as a functional, implementation of the same behavior. Historically, systems making such extra demands have met only limited success, and it is possible such a fate awaits the MFM, but there are also reasons for hope. On the one hand, previous systems aspired only to finite scalability, limiting the rewards offered for the extra work required. On the other hand, lately it seems that the programmer's job is getting harder anyway—specifically because the networked world is finally and increasingly losing tolerance for the *lack* of robustness and security that is the dark flip side of the von Neumann machine's zero-dimensional convenience.

Where it is applicable—for example, in relatively small and safe contexts like those of its early years—the CPU+RAM model of computation is simple, powerful and a joy to use, intoxicating in its master-of-the-universe positioning of the programmer—and indeed there is something deeply right about it as a model of a conscious *mind*. But, it is just as deeply wrong as a physical implementation of a *brain*—and it is essentially sociopathic as a model for a team member.

As computing inevitably and now quickly scales beyond the purview of a single actor, we suggest that the additional complexities attending robust spatial computing are only the price of admission to this new and larger arena. Computation cannot continue to ignore space for that much longer, but physical computational spaces can, and will, be floorplanned and blueprinted, farmed and developed, sold and rented—and in the process, computer architecture will turn into, well, *architecture*. It is high time.

ACKNOWLEDGEMENTS

The authors thank the anonymous referees for their helpful comments.

FUNDING

Funding to pay the Open Access publication charges for this article was provided by D.H.A.

REFERENCES

[1] Cappello, F., Geist, A., Gropp, B., Kal, L.V., Kramer, B. and Snir, M. (2009) Toward exascale resilience. *IJHPCA*, **23**, 374–388.

[2] Ackley, D.H. and Williams, L.R. (2011) Homeostatic Architecture for Robust Spatial Computing. *Spatial Computing Workshop at IEEE Self-Adaptive Self-Organizing Systems*, Ann Arbor, Michigan, USA, October. IEEE.

[3] Ackley, D.H. and Cannon, D.C. (2011) Pursue Robust Indefinite Scalability. *Proc. HotOS XIII*, Napa Valley, CA, USA, May. USENIX Association.

[4] Partridge, C., Mendez, T. and Milliken, W. (1993) Host Anycasting Service. RFC 1546 (Informational).

[5] Abley, J. and Lindqvist, K. (2006) Operation of Anycast Services. RFC 4786 (Best Current Practice).

[6] Pei, R., Taylor, S.K., Stefanovic, D., Rudchenko, S., Mitchell, T.E. and Stojanovic, M.N. (2006) Behavior of polycatalytic assemblies in a substrate-displaying matrix. *J. Am. Chem. Soc.*, **128**, 12693–12699.

[7] Franco, E., Friedrichs, E., Kim, J., Jungmann, R., Murray, R., Winfree, E. and Simmel, F.C. (2011) Timing Molecular Motion and Production with a Synthetic Transcriptional Clock. *Proc. Natl. Acad. Sci.*, **108**, E784–E793.

[8] DeHon, A., Giavitto, J.-L. and Gruau, F. (2007) 06361 Executive Report—Computing Media Languages for Space-Oriented Computation. In DeHon, A., Giavitto, J.-L., and Gruau, F. (eds), *Computing Media and Languages for Space-Oriented Computation*, Dagstuhl, Germany, Dagstuhl Seminar Proceedings, Vol. 06361. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[9] Borcea, C., Intanagonwiwat, C., Kang, P., Kremer, U. and Iftode, L. (2004) Spatial Programming Using Smart Messages: Design and Implementation. *In International Conf. on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan, pp. 690–699.

[10] Zambonelli, F. and Mamei, M. (2005) Spatial Computing: An Emerging Paradigm for Autonomic Computing and Communication. In Smirnov, M. (ed.), *Autonomic Communication*, Lecture Notes in Computer Science, Vol. 3457, pp. 227–228. Springer, Berlin/Heidelberg.

[11] Beal, J. and Bachrach, J. (2007) Programming manifolds. In DeHon, A., Giavitto, J.-L. and Gruau, F. (eds.), *Computing*

Media and Languages for Space-Oriented Computation, Dagstuhl, Germany, Dagstuhl Seminar Proceedings, Vol. 06361. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[12] Gruau, F., Eisenbeis, C. and Maignan, L. (2008) The foundation of self-developing blob machines for spatial computing. *Physica D: Nonlinear Phenomena*, **237**, 1282–1301. Novel Computing Paradigms: Quo Vadis?

[13] Yamins, D. and Nagpal, R. (2008) Automated Global-to-Local Programming in 1-d Spatial Multi-Agent Systems. *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, Vol. 2, Richland, SC AAMAS'08, pp. 615–622. International Foundation for Autonomous Agents and Multiagent Systems.

[14] Mamei, M. and Zambonelli, F. (2009) Programming pervasive and mobile computing applications: the TOTA approach. *ACM Trans. Softw. Eng. Methodol.*, **18**, 1–56.

[15] Beal, J., Dulman, S., Usbeck, K., Viroli, M. and Correll, N. (2012) Organizing the aggregate: languages for spatial computing. *CoRR*, **abs/1202.5509**.

[16] Bachrach, J., Beal, J. and McLurkin, J. (2010) Composable continuous-space programs for robotic swarms. *Neural Comput. Appl.*, **19**, 825–847.

[17] Gruau, F. and Eisenbeis, C. (2007) Programming Self Developing Blob Machines for Spatial Computing. In Brams, S.J., Pruhs, K. and Woeginger, G.J. (eds), *Computing Media and Languages for Space-Oriented Computation*, Dagstuhl Seminar Proceedings, Vol. 07261. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[18] Beal, J., Michel, O. and Schultz, U.P. (2011) Spatial computing: distributed systems that take advantage of our geometric world. *TAAS*, **6**, 11.

[19] von Neumann, J. (1951) The General and Logical Theory of Automata. In Jeffress, L.A. (ed.), *Cerebral Mechanisms in Behaviour: The Hixon Symposium (1948)*. Wiley.

[20] Micheli-Tzanakou, E. and Krakauer, D.C. (2006) Robustness in Biological Systems: A Provisional Taxonomy. In Deisboeck, T.S. and Kresh, J.Y. (eds), *Complex Systems Science in Biomedicine*, Topics in Biomedical Engineering. International Book Series, pp. 183–205. Springer US.

[21] Hamming, R.W. (1950) Error detecting and error correcting codes. *Bell Syst. Tech. J.*, **29**, 147–160.

[22] Peterson, W.W. and Weldon, E.J. (1972) *Error-Correcting Codes*. The MIT Press.

[23] Treaster, M. (2005) A survey of fault-tolerance and fault-recovery techniques in parallel systems. *ACM Comput. Res. Repository (CoRR)*, **501002**, 1–11.

[24] Cheatham, J.A., Emmert, J.M. and Baumgart, S. (2006) A survey of fault tolerant methodologies for fpgas. *ACM Trans. Des. Autom. Electron. Syst.*, **11**, 501–533.

[25] Xiong, N., Yang, Y., Cao, M., He, J. and Shu, L. (2009) A Survey on Fault-Tolerance in Distributed Network Systems. *2009 Int. Conf. on Computational Science and Engineering*, Vancouver, Canada, Vol. 2, 1065–1070. IEEE Computer Society.

[26] Gray, J. and Siewiorek, D. (1991) High-availability computer systems. *Computer*, **24**, 39–48.

[27] Ayari, N., Barbaron, D., Lefevre, L. and Primet, P. (2008) Fault tolerance for highly available internet services: concepts, approaches, and issues. *IEEE Commun. Surveys Tutorials*, **10**, 34–46.

[28] Lyons, R.E. and Vanderkulk, W. (1962) The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, **6**, 200–209.

[29] Avizienis, A. (1985) The N-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, **11**, 1491–1501.

[30] Milojíčić, D.S., Douglis, F., Paindaveine, Y., Wheeler, R. and Zhou, S. (2000) Process migration. *ACM Comput. Surv.*, **32**, 241–299.

[31] Nelson, M., Lim, B.-H. and Hutchins, G. (2005) Fast Transparent Migration for Virtual Machines. *Proc. Annual Conf. on USENIX Annual Technical Conf.*, Berkeley, CA, USA ATEC'05, pp. 25–25. USENIX Association.

[32] Black, A.P., Hutchinson, N.C., Jul, E. and Levy, H.M. (2007) The Development of the Emerald Programming Language. *Proc. 3rd ACM SIGPLAN Conf. on History of Programming Languages*, New York, NY, USA, HOPL III, pp. 11–1–11–51. ACM.

[33] Liu, H., Roeder, T., Walsh, K., Barr, R. and Sirer, E.G. (2005) Design and Implementation of a Single System Image Operating System for ad hoc Networks. *Proc. 3rd Int. Conf. on Mobile Systems, Applications, and Services*, New York, NY, USA, MobiSys'05, pp. 149–162. ACM.

[34] von Neumann, J. and Burks, A.W. (eds.) (1966) *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, IL, USA.

[35] Ulam, S. (1950) Statistical mechanics of cellular automata, 1952. *Proc. Int. Congr. Math.*, **2**, 264–275.

[36] Sipper, M., Tomassini, M. and Capcarrere, M.S. (1997) Evolving Asynchronous and Scalable Non-Uniform Cellular Automata. *Proc. Int. Conf. on Artificial Neural Networks and Genetic Algorithms (ICANNGA97)*, Norwich, England, pp. 67–71. Springer.

[37] Tomassini, M. and Venzi, M. (2001) Evolving robust asynchronous cellular automata for the density task. *Complex Syst.*, **13**, 185–204.

[38] Lee, J., Adachi, S., Peper, F. and Morita, K. (2003) Embedding universal delay-insensitive circuits in asynchronous cellular spaces. *Fundam. Inf.*, **58**, 295–320.

[39] Cornforth, D., Green, D.G. and Newth, D. (2005) Ordered asynchronous processes in multi-agent systems. *Physica D: Nonlinear Phenom.*, **204**, 70–82.

[40] Bouré, O., Fatès, N. and Chevrier, V. (2011) Robustness of Cellular Automata in the Light of Asynchronous Information Transmission. In Calude, C.S., Kari, J., Petre, I. and Rozenberg, G. (eds), *Proc. of the 10th International Conference on Unconventional Computation*, Turku, Finland, Lecture Notes in Computer Science, Vol. 6714, pp. 52–63. Springer.

[41] Nakamura, K. (1974) Asynchronous cellular automata and their computational ability. *Syst. Comput. Controls*, **5**, 58–66.

[42] Toffoli, T. (1978) Integration of the Phase-Difference Relations in Asynchronous Sequential Networks. *ICALP'78*, Udine, Italy, pp. 457–463.

[43] Nehaniv, C.L. (2004) Asynchronous automata networks can emulate any synchronous automata network. *IJAC*, **14**, 719–739.

[44] Chapiro, D. (1984) Globally asynchronous locally synchronous systems. PhD Thesis, Stanford University. STAN-CS-84-1026.

[45] Gershenfeld, N. (2011) Aligning the representation and reality of computation with asynchronous logic automata. *Computing*, **93**, 91–102.

[46] Gershenfeld, N., Dalrymple, D., Chen, K., Knaian, A., Green, F., Demaine, E.D., Greenwald, S. and Schmidt-Nielsen, P. (2010) Reconfigurable Asynchronous Logic Automata: RALA. *Proc. 37th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, New York, NY, USA, POPL'10, pp. 1–6. ACM.

[47] Rumelhart, D.E., McClelland, J.L. and PDP Research Group (eds) (1986) *Parallel Distributed Processing. Volume 1: Foundations*. MIT Press, Cambridge, MA.

[48] Ackley, D.H. (1987) *A Connectionist Machine for Genetic Hillclimbing*. Kluwer, Dordrecht, Boston.

[49] Ananthanarayanan, R., Esser, S.K., Simon, H.D. and Modha, D.S. (2009) The Cat is Out of the Bag: Cortical Simulations with 109 neurons, 1013 synapses. *Proc. Conf. on High Performance Computing Networking, Storage and Analysis*, New York, NY, USA, SC'09, pp. 63:1–63:12. ACM.

[50] Liquidware.com (2011) Illuminato X Machina. http://illuminatolabs.com (accessed September 2, 2012).

[51] Ackley, D.H. (2008) The SFB reference manual. http://livingcomputation.com/s/doc/ (accessed September 2, 2012).

[52] Ackley, D.H. (2009). Illuminato X Machina network arrays (video). http://www.youtube.com/watch?v=ZBFoFYhC9B4.

[53] Williams, L.R. (2011) Artificial cells as reified quines. *European Conf. on Artificial Life (ECAL'11)*, Paris, France, August.

[54] Pitoura, E. and Samaras, G. (2001) Locating objects in mobile computing. *IEEE Trans. Knowl. Data Eng.*, **13**, 571–592.

[55] Digital Display Working Group (1999). Digital visual interface: DVI. http://www.ddwg.org/lib/dvi_10.pdf (accessed September 2, 2012).

[56] Landin, P.J. (1964) The mechanical evaluation of expressions. *Comput. J.*, **6**, 308–320.

[57] Dybvig, R.K. (1987) Three Implementation Models for Scheme. PhD Thesis, University of North Carolina, Chapel Hill, NC.

[58] Hewitt, C., Bishop, P. and Steiger, R. (1973) A universal modular actor formalism for artificial intelligence. *IJCAI*.

[59] Greif, I. and Hewitt, C. (1975) Actor Semantics of PLANNER-73. *Principles of Programming Languages*, January.

[60] Baker, H. (1978) Actor Systems for Real-Time Computation. PhD Thesis, Massachusetts Insitute of Technology, Cambridge, MA. http://publications.csail.mit.edu/lcs/specpub.php?id=765.

[61] Clinger, W. (1981) Foundations of Actor Semantics. PhD Thesis.

[62] Agha, G. (1986) Actors: A model of concurrent computation in distributed systems, Massachusetts Insitute of Technology, Cambridge, MA. http://hdl.handle.net/1721.1/6935.

[63] Kelsey, R., Clinger, W. and Rees, J. (1998) Revised^5 report on the algorithmic language Scheme. *Higher-Order Symbol. Comput.*, **11**.

[64] Bruijn, N.G.D. (1972) Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math.*, **34**, 381–392.