Homeostatic architectures for robust spatial computing

David H. Ackley
Computer Science
University of New Mexico
Albuquerque, New Mexico 87131
Email: ackley@cs.unm.edu

Lance R. Williams
Computer Science
University of New Mexico
Albuquerque, New Mexico 87131
Email: williams@cs.unm.edu

***Abstract*—For open-ended computational growth, we argue that: (1) Instead of hardwiring and hiding the spatial relationships of computing components, computer architecture should expose and soften them; and (2) Instead of minimizing reliability as just a hardware problem, robustness should 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 overtly spatialized tasks, nor harsh environment deployments. This paper is an 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, with which we are exploring robust, indefinitely scalable computations. We briefly motivate the approach and present the model, then illustrate some robustness mechanisms such as redundancy, sloppiness, and homeostasis, showing how a basic task like sorting can be reconceived for robustness within a homeostatic architecture.**

I. Introduction

COMPUTERS of the von Neumann machine design have revolutionized the world, but CPU and RAM, its twin conceptual pillars, are now the greatest hindrances to its continued computational growth. The solitary "central" processor focuses programmers on computation without communication—but CPU scalability depends heavily on now-dwindling clock speed increases. Similarly, uniform cost "random access" memory focuses programmers on logical function without spatial forms—but the inescapable geometry of physical space allows only a finite set of locations to fit within unit time access of a single finite-sized processor.

A. Indefinitely scalability and spatial computing

We have presented [1] a research case for *indefinitely scalable* computer architectures which, by definition, support open-ended computational growth without re-engineering. To do that, indefinite scalability rejects *all* internal scaling limits—such as single-source clocking, or fixed-width memory or network addresses—so machine size is limited only by external costs such as materials, construction, real estate, power, cooling, and maintenance. We argued that such an architecture amounts to a three (or less) dimensional spatial tiling of configurable elements that are initially interchangeable, communicate only locally using relative spatial addresses, and execute asynchronously, at least above some granularity.

Presented at the *Spatial Computing Workshop* at SASO 2011 in Ann Arbor, MI. This version revised Oct 10, 2011.

While such design criteria are admittedly controversial in the OS and distributed systems communities, they are mostly old hat here, because any indefinitely scalable machine will be a *spatial computer* in the sense of this workshop. Our approach is discrete, reified, and designed bottom-up, compared to spatial computing languages framed in continuous or amorphous spaces such as [2], or all-in "vertical" models such as [3]. And although external tasks aligned to a spatial computer can be choice low-hanging fruit, we focus more on architectural uses of space when computations are either not inherently spatial, or are spatialized differently than the machine itself. But still, indefinite scalability, and the Movable Feast Machine, seem deeply compatible with spatial computing approaches (e.g., [4]), and so we have prepared this paper.

B. The importance of being robust

Indefinite scalability also provides a clean motivation for explorations in robustness, self-adaptation and self-organization. An indefinitely scalable machine will likely be in use before it's "finished," so it must recruit resources as they become available, and be robust in the face of local outages, failures, changing uses and configurations, and its own construction.

Over sixty years ago, von Neumann [5] argued that "future" computers would need to be more robust than his namesake approach, yet in design space there we remain today, still rooting around near CPU+RAM—despite all its avowed fragility and nightmarish security properties—and we are deploying them by the millions *per week* and connecting them to vast personal information and economic leverage.

The present work emerges from a belief that it's likely better if von Neumann's call for robustness came true sooner rather than later, but robustness can be a slippery concept, 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, [6] presents useful robustness "principles and parameters"—and in such terms, the examples in Section III touch on spatial compartmentalization, redundancy, sloppiness, and homeostasis.

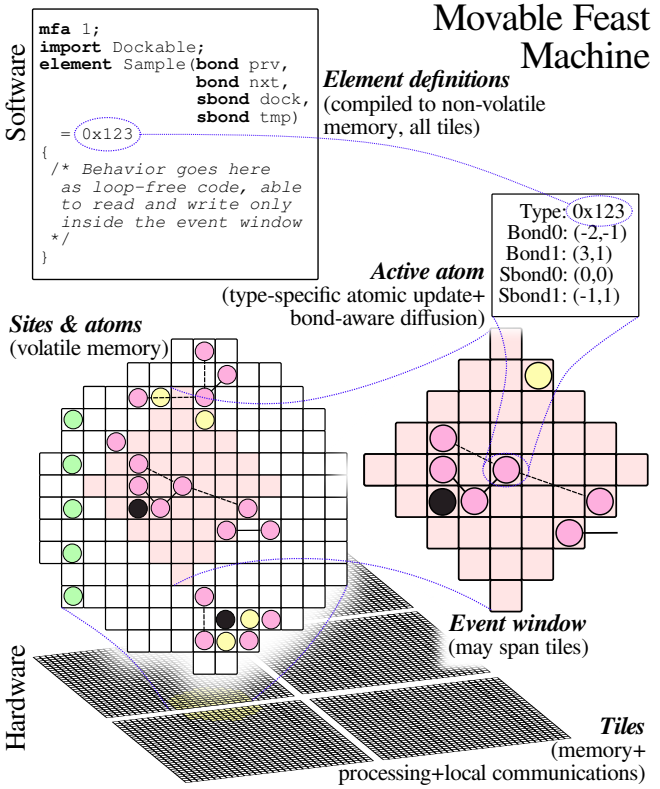


Fig. 1: Architectural overview. See Section II.

C. Asynchronous cellular automata

Also popularized by von Neumann [7], based on a suggestion by Ulam [8], *cellular automata* (CA) are widely-used parallel spatial computing models that have been applied to phenomena as diverse as weather and self-reproduction. As architectures, many CAs are limited by single-source clocking, but *asynchronous cellular automata* (ACA) (e.g., [9], [10], [11]; perhaps [3]) have indefinite scalability potential.

Some ACAs use stochastic transition rules, but in any case their site update order is usually non-deterministic. Although a technique such as [12] can be used to implement a synchronous CA on top of an ACA, for indefinite scalability such an approach is, by itself, hopelessly fragile—for example, a single stuck site will eventually lock the entire grid.

Scalable asynchrony is not so easily dispatched as that.

II. The Movable Feast Machine

Figure 1 summarizes the *Movable Feast Machine* (MFM). Though limited space precludes a truly complete description, we try to provide concepts and details sufficient to engage the reader's mechanical intuition. Throughout, the phrase 'some chosen' is used to flag overtly parametric model properties.

A. Small programs change big neighborhoods

Viewed as a stochastic ACA, the Movable Feast Machine falls into a rather sparsely populated corner of ACA parameter space—with many possible states per site, and also many sites per neighborhood. The MFM instances in this paper, for example, use 64 bits per site, and a 41 site neighborhood (Manhattan 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* to perform a state transition on a neighborhood, and our design seeks to present a flexible and powerful, but still transparent and understandable, machine semantics to the state transition programmer. To that end, we adapt familiar programming concepts—objects, classes, pointers—reinvented for indefinite scalability, and miniaturized to fit into a MFM site (for an object) and a neighborhood (the pointer addressing range). The programming of a state transition feels somewhat familiar because it executes with synchronous clocking and single threaded semantics, which also means the MFM is an odd but recognizable form of *globally asynchronous locally synchronous* (GALS) architecture [13].

B. 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*. Though 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 $\mathbb{R}^3$.

C. Event window processing

MFM computation proceeds asynchronously in parallel by executing *events*, each of which corresponds to one independent state transition, occuring 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.

An event window life cycle is depicted in Figure 2. A *center site* holding an *active atom* is selected, at random or

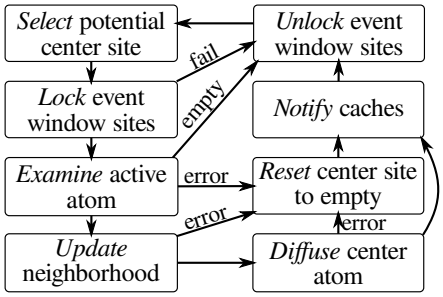


Fig. 2: Event window life cycle. See Section II-C.

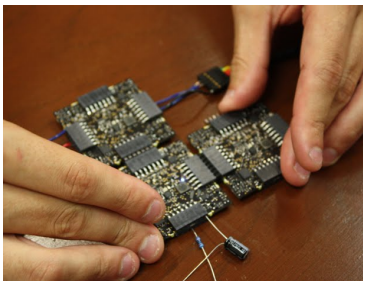


Fig. 3: Hot-plugging *Illuminato X Machina* tiles [14], prototype indefinitely scalable hardware.

by any starvation-free mechanism, such that its neighborhood is disjoint from the neighborhoods of any other currently selected center sites. The active atom is checked, and erased if any format problem is found; otherwise its *element type* is extracted and that type's code sequence, if any, is executed, *updating* the neighborhood arbitrarily. This freedom to modify the neighborhood differs from CA variants that only modify center sites—and adds complexities, such as the cache notifications shown in Figure 2—but also greatly eases tasks like maintaining bond consistency (see Section II-E). After the update, the center site contents is optionally *diffused* if possible, then neighborhood modifications are communicated to all affected tiles, locks are released, 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 indefinitely scalable hardware (Figure 3) is in progress. For a MFM simulation—which includes any MFM that is *not* running on indefinitely scalable hardware—actual execution time is usually misleading, and instead we adopt *average events per site* (AEPS) as the base unit of time; regardless of MFM size, in 1 AEPS each site will, on average, be the center site of one event.

D. Atomic structure

Each MFM site contains some chosen number of modifiable state 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 Figure 1), which supplies information such as bond counts (next section) as well as the `update` method that defines a state transition.

In the examples below, the type function is implemented by designating the first 16 bits of each atom as a "header" that specifies the interpretation of the remaining bits, with part of that interpretation yielding the location and size of the element type number within that atom.

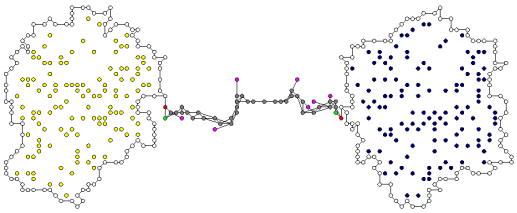


Fig. 4: Bonded atoms formed into membranes and wires.

E. Bonds and mobility

A namesake aspect of the Movable Feast Machine is that atoms *move*, for a variety of purposes, as implemented via copying and erasing, or swapping, 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 Figure 4 and [1]). Atomic mobility is also handy in self-reproducing systems, so offspring might eventually move into space of their own [15].

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 (e.g., [16] is one survey) 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. Bonds are designed so 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. Each bond is symmetric—if atom A has a bond to atom B then B will have a bond back to A or else an inconsistency has occurred. Finally, atomic diffusion automatically updates bond values, and will not break or overstretch them, and weakly prefers to keep bond lengths short.

Our current design also offers "short bonds" ('sbond' in Figure 1) which sacrifice addressing range to save atomic bits. Although here we focus on computations involving elements without bonds, Figure 4 offers a taste of the sloppy-structured mobility that MFM atomic bonds naturally support—with 'message' atoms being transported at supradiffusive rates between larger—movable but less agile—'cellular' structures.

F. Temporal structuring; laws of physics

Indefinite scalability also impacts the temporal structure of computations (see Section III-B, also [17]): Notions like "load time" or "program start"—or even "power on"—lack global meaning given 'hot configurable' hardware. For interactive computations our prototype indefinitely scalable hardware supports simple I/O between a tile and its embedding space, and for ongoing management and configuration it provides intertile communication channels beyond just the 'neighborhood updates' of Figure 2.

Finally, we touch on the element descriptions—the *physics* or 'periodic table'—that determine atomic behavior in a MFM.

For both efficiency and robustness, the element transition rules are stored in non-volatile memory, and they are expected to be identical on all tiles. But we are far from having so complete and useful a physics for it to be unalterably manufactured into MFM tiles, so downloadable physics, for research at least, is essential—despite the considerable conceptual challenges attending modifiable 'laws of physics'. Updated physics propagate across the machine—hopefully rarely—via those additional intertile communications channels.

III. Robust spatial computations in the MFM

All these examples are based on a rectangular lattice in 2D with Manhattan distance, varying grid sizes, event window radius=4, and atomic width=64.

A. Density and homeostasis

It is easy to overlook at first, 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'll have spares or make repairs—and if they aren't, we can't rely on them anyway.

We explore controlling *occupied site density* (OSD) with `DReg`, a diffusing "Dynamic Regulator" element that each update checks a random nearby site. If the site is occupied, `DReg` might erase it, particularly if it's another `DReg`. If it's empty, `DReg` might create a general-purpose 'resource' atom (element `Res`), or rarely another `DReg`. 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 $1/3$ creation ratio, but the specific probabilities also matter: Smaller values yield looser regulation and slower transient response; larger probabilities are faster but more disruptive. Figure 5 shows OSD regulation for three different `DReg` parameter sets after a system is shocked by erasing 97% of the sites at 10K 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 the next section.

B. Robust spatial sort

To help perfuse robustness into the computational stack, we seek ways to intertwine it with functionality—and we are more

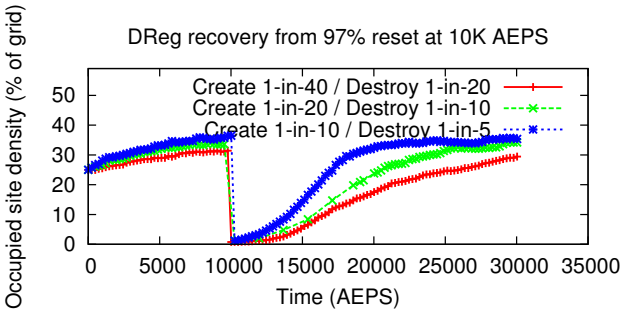
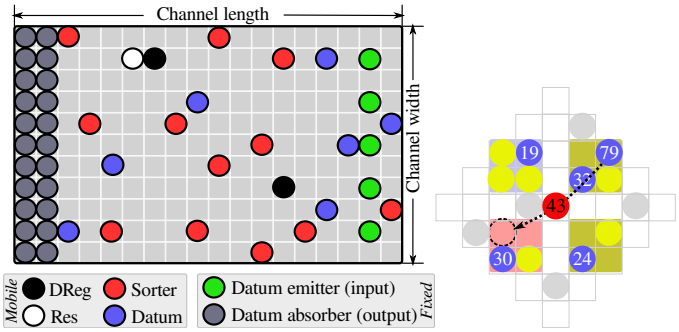


Fig. 5: DReg density regulation after a transient.



(a) Elements and global structure (b) `Sorter` behavior

Fig. 6: Elements of the Demon Horde Sort

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 Figure 6a, given the task of sorting or prioritizing an endless stream of `Datum` atoms which are injected at random intervals by "emitters" near the right side of the grid. Each `Datum` contains a 32 bit number (and an eight 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.

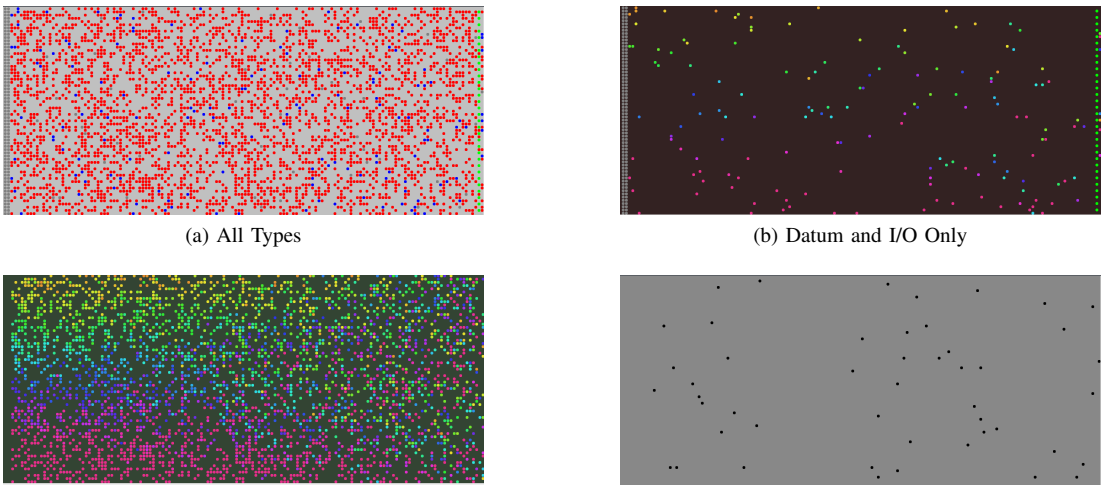
Here we focus on an *equal interval* goal where each output "bucket" (the absorbers on a single row) should receive the values of an equal portion of the underlying data range.[1] We measure performance by the *average positional error*—the average distance between the bucket that absorbs a `Datum` and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

We assume an unknown, perhaps non-stationary, data distribution, 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 III-A)

[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 this is a simplified version compared to [1].



(a) All Types

(b) Datum and I/O Only

(c) Sorters Only

(d) DReg and Res Only

Fig. 7: Filtered views of the same state of a 150x65 flow sorter, sorting right to left, at $t \approx$ 100K AEPS. `Datums` in *(b)* and `Sorters` in *(c)* rendered with colors representing their value (of data or threshold, respectively, with *orange* smallest to *pink* largest), otherwise colors represent element types. No `Res` are present in *(d)* because of rapid `Sorter` transmutations.

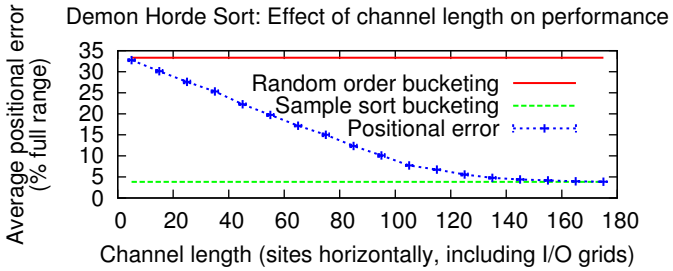


Fig. 8: Sorting performance

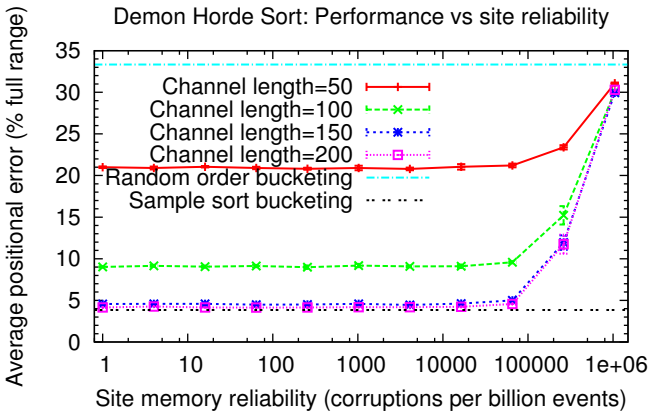


Fig. 9: Robustness to hardware failures

and adds a `Sorter` element, illustrated in Figure 6b. In this version, `Sorter` has two primary functions. First, whenever it sees a `Res` it transmutes it into another `Sorter`, so the `Sorter` population level is indirectly controlled by `DReg`. Second, `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 Figure 6b, the `Sorter`'s threshold will soon be 79.

In this standalone demonstration the emitters and absorbers suppress their MFM diffusion and don't otherwise move; 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 7 illustrates a demon horde that has been running for some 100K AEPS in a 65 bucket flow sorter, showing all the elements at once (7a), and various subset views. In Figure 7c, which displays `Sorters` colored by their thresholds, the typical equilibrium structure of a demon horde is visible: 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 Figure 8. 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 9 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 atomic bits. We can see that while channel length helps performance, it doesn't help robustness against this system perturbation—but the system is strikingly robust anyway, tolerating upwards of one multibit corruption per 100K events with essentially no visible performance degradation, regardless of channel length. Above one error per 10K events the system reliably falls apart—and the pathology appears to run a reliable course: The bit flips trigger the error pathways in Figure 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 `Sorter`s, which leaves the emitted `Datum`s 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 `Datum`s 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 tradeoffs and sweet spots in MFM design space.

IV. 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. But 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.

And from the direction of traditional programmable computers, one obvious criticism is this all 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's 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. And 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 for what it's worth 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 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 studied, characterized, graded, floorplanned, and blueprinted; and farmed and developed, and subdivided and rented and sold for computational advantage—and in the process, computer architecture will turn into, well, *architecture*. It is high time.

References

[1] D. H. Ackley and D. C. Cannon, "Pursue robust indefinite scalability," in *Proc. HotOS XIII*, Napa Valley, California, USA, May 2011.

[2] J. Bachrach, J. Beal, and J. McLurkin, "Composable continuous-space programs for robotic swarms," *Neural Computing and Applications*, vol. 19, no. 6, pp. 825–847, 2010.

[3] F. Gruau and C. Eisenbeis, "Programming self developing blob machines for spatial computing." in *Computing Media and Languages for Space-Oriented Computation*, ser. Dagstuhl Seminar Proceedings, S. J. Brams, K. Pruhs, and G. J. Woeginger, Eds., vol. 07261. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[4] J. Beal, O. Michel, and U. P. Schultz, "Spatial computing: Distributed systems that take advantage of our geometric world," *TAAS*, vol. 6, no. 2, p. 11, 2011.

[5] J. von Neumann, "The general and logical theory of automata," in *Cerebral Mechanisms in Behaviour*, L. A. Jeffress, Ed. Wiley, 1951.

[6] E. Micheli-Tzanakou and D. C. Krakauer, "Robustness in biological systems: A provisional taxonomy," in *Complex Systems Science in Biomedicine*, ser. Topics in Biomedical Engineering. International Book Series, T. S. Deisboeck and J. Y. Kresh, Eds. Springer US, 2006, pp. 183–205.

[7] J. von Neumann and A. W. Burks, Eds., *Theory of Self-Reproducing Automata*. Urbana, IL, USA: University of Illinois Press, 1966.

[8] S. Ulam, "Statistical mechanics of cellular automata, 1952," *Proceedings of the International Congress on Mathematics*, vol. 2, pp. 264–275, 1950.

[9] O. Bouré, N. Fatès, and V. Chevrier, "Robustness of cellular automata in the light of asynchronous information transmission," in *UC*, ser. Lecture Notes in Computer Science, C. S. Calude, J. Kari, I. Petre, and G. Rozenberg, Eds., vol. 6714. Springer, 2011, pp. 52–63.

[10] D. Cornforth, D. G. Green, and D. Newth, "Ordered asynchronous processes in multi-agent systems," *Physica D: Nonlinear Phenomena*, vol. 204, no. 1-2, pp. 70 – 82, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/B6TVK-4G77HGN-3/2/ab40980df5fc7e111078b74f37fe611b

[11] J. Lee, S. Adachi, F. Peper, and K. Morita, "Embedding universal delay-insensitive circuits in asynchronous cellular spaces," *Fundam. Inf.*, vol. 58, pp. 295–320, May 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=1006455.1006462

[12] C. L. Nehaniv, "Asynchronous automata networks can emulate any synchronous automata network," *IJAC*, vol. 14, no. 5-6, pp. 719–739, 2004.

[13] D. Chapiro, "Globally asynchronous locally synchronous systems," Ph.D. dissertation, Stanford University, October 1984, STAN-CS-84-1026.

[14] Liquidware.com, "Illuminato X Machina," http://illuminatolabs.com, 2011.

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

[16] E. Pitoura and G. Samaras, "Locating objects in mobile computing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, pp. 571–592, 2001.

[17] E. Schulte and D. Ackley, "Physical evolutionary computation," University of New Mexico, Albuquerque, NM, USA, Tech. Rep. TR-CS-2011-01, 2011, http://cs.unm.edu/∼treport/tr/11-04/paper-2011-01.pdf.