Pursue robust indefinite scalability

Presented at
The 13^{th} Workshop on Hot Topics in Operating Systems (HOTOS-XIII)
May 2011, Napa, California, USA

David H. Ackley
Dept. of Computer Science
The University of New Mexico
Albuquerque, NM 87131

Daniel C. Cannon
Dept. of Computer Science
The University of New Mexico
Albuquerque, NM 87131

Abstract For research insights and development potential, we should explore computer architectures designed to scale indefinitely. Given physical limits, we argue an indefinitely scalable computer should or must (1) reveal to programmers its component spatial relationships, (2) forego unique addresses, and (3) operate asynchronously. Further, such a machine and its programming must be inherently robust against local failures and outages, and be operable during its own construction.

We propose the indefinitely scalable Movable Feast Machine, which defers many architectural decisions to an execution model that associates processing, memory, and communications functions with movable bit patterns rather than fixed locations. We illustrate basic and novel computational elements such as self-healing wire, simple cell membranes for modularity, and robust stochastic sorting by movable self-replicating programs.

1 Indefinite scalability and robustness

Although computation has scaled largely by increasing data widths and switching speeds, progress has stalled at present; multicores are currently ascendant, but cache coherence scales poorly. Achieving *indefinite scalability*—i.e., supporting open-ended computational growth without substantial re-engineering—will involve new distributed system architectures, and impact hardware, OS design, programming, and end-user value propositions. So although strategies such as multicore still have room to run, we view indefinite scalability as a useful beacon now, casting light into some still largely wild computational spaces beyond the serial machine.

A design for an indefinitely scalable machine amounts to a spatial tiling of hardware elements, with additional requirements as needed to preserve open-ended physical realizability. Three particularly consequential ones are:

1. *Pledge allegiance to the light cone*: Because hardware occupies space and light speed is finite, no fixed-cost global communications or arbitrary long range links are possible, ruling out hierarchical and other log time interconnects, at least above some granularity—and also implying asynchronous or dynamically-synchronizing operation.
2. *with computational relativity*: There must be only one or a few basic hardware functional types, interchangeable by kind, and without presumption of any globally unique identifying attributes[1]. Therefore, inter-element addressing depends, at least initially, upon relative spatial position.
3. *and robustness for all*: The system should provide non-stop operation, tolerating or repairing (or even exploiting) errors and disruptions, such as transients and misconfigurations, hot-swapping and new construction, etc. This applies beyond hardware to the entire computational stack—consistency and correctness cannot simply be assumed, lest a local error, however unlikely, derail the entire system.

Existing tiled hardware such as [9, 13] generally emphasizes finite (e.g., intra-chip) scalability, but may be useful components for indefinitely scalable designs; in any case software looms as a major challenge. Indeed, to imagine programming without perfect reliability and synchronization may seem simply crippling, but we have found it can also feel liberating and full of natural potential—more like how things actually get done in the world, even though stuff breaks, or people deliver late, or never, or deliver white and call it wheat. Society often uses systems that tolerate many such failures and perform useful work nonetheless, preferring minor inefficiencies to major catastrophes. From the vantage point of striving to add value despite the inevitable errors, it is traditional computing's demand for absolute perfection in input and processing—or else *"It's not my fault!"*—that seems petty and bureaucratic, like that bad apple with whom nobody wants to work. To keep on growing, from bottom to top, what computation needs is team spirit.

[1] A potentially contentious issue is whether each hardware element can be expected to offer an independent (pseudo) random stream.

1.1 Prefer expressiveness to parsimony. Implementations of *cellular automata* (CA) [3, 11]—one of the earliest extensively studied computational models—can satisfy many of our architectural criteria, so long as their physical form (as chips or wafers or boards, etc.) is indefinitely tileable. CA are obviously spatialized, communicate only locally, and processing nodes are completely fungible and relative-addressed. Although many CA models are globally synchronized, there are also numerous varieties of *asynchronous cellular automata* (ACA) [5, 8] that are indefinitely scalable as we use the term, and our model has much in common with them. Some grid rewriting systems such as [2] are also related.

CA research has often emphasized theoretical goals such as seeking minimal mechanisms for various computational primitives and properties such as universality; the CAM-6 and CAM-8 machines [11, 10], as custom hardware for cellular automata, are notable exceptions. Although the latter machine especially was extensively configurable, its fundamental update step still amounted to a globally synchronized 2^{16} entry table lookup; in that sense these machines implemented fairly conventional synchronous cellular automata with a semantic model roughly at the level of boolean algebra.

Our idea is to expose indefinitely scalable computational power to programmers using reinvented and restricted—but still recognizable—concepts of sequential code, integer arithmetic, pointers, and user-defined classes and objects. Within the space of indefinitely scalable designs, consequently, we prioritize programmability and software engineering concerns well ahead of either theoretical parsimony or maximally efficient or flexible tile hardware design.

Our strategy is to entice smart people to play with programming robust, indefinitely scalable computations by lowering barriers to entry, and as idioms, motifs, and best practices emerge from such explorations, we believe opportunities for wise optimization will arise.

1.2 Embrace biological parallelism. Natural biological systems have inspired many computational ideas, such as evolution for optimization [4] and using diversity for robustness [1]. One inherently biological strategy is *opportunistic reproduction for parallelism and robustness*: If resources are available, a bigger school of fish not only eats faster, but is also harder to wipe out. Unfortunately, to date this idea has entered computation most prominently via computer viruses and other malware—perhaps because if a person owns a serial processor and a single pipe, a program can gain cycles and bandwidth only if it reproduces onto somebody else's hardware.

We seek to rehabilitate and benefit from opportunistic reproduction, embodying it in systems with large potential parallelism, perhaps involving FPGA-ish devices or a FAWN-style [12] microcontroller array. Connected sets of lookup tables or other processing elements might cooperate, or compete, to copy their configurations into nearby available real estate. The vision is of *populations* of programs in a robust ecology that equilibrates under stable resources and workloads, but also both rides out local resource shortages and exploits new availabilities. Note that alongside opportunistic reproduction, *movability* is important for effective parallelism—e.g., it's easier to route I/O to an offspring that's moved away from its parent. Section 3.4 offers a demonstration.

2 The Movable Feast Machine

Figure 1 provides an architectural overview of a *Movable Feast Machine* (MFM), the indefinitely scalable computational model we are exploring. The MFM generalizes readily in several directions; here for concreteness we include parameter values of our current system in small font and set off by '{}'s, but note the demonstrations in Section 3 ran on a simulator; our first indefinitely scalable implementation, based on *Illuminato X Machina* [6] boards, is in progress.

Starting with hardware in the lower left of Figure 1, we envision an open-ended {2D} grid of *tiles*, each of which contains {1, 72MHz} processor(s), {32KB} volatile and {512KB} non-volatile memories, and {1Mbps} point-to-point links to each of its {4} nearest neighbor tiles. Each tile maintains an array of {48x48} *sites* in volatile memory; each site can hold one fixed-width {64 bit} *atom*. An atom, in turn, contains a {16 bit} header that specifies the interpretation of the other bits, as *element* type number, inter-atomic *bond* coordinates, and/or instance *data*.

In object-oriented terms, an atom is akin to a small, fixed-size object instance, linked by its type number to a class-like *element definition* (pseudocode only in Figure 1) containing an `update` method that defines the behavior of atoms of that element. A *bond* is a distance-limited and symmetric form of pointer, implemented by self-is-origin relative coordinate addressing, and offered in two {ℓ_1, 'city block'} distance-limited forms—*long bonds* {$\ell_1(long) \leq 4$} and *short bonds* {$\ell_1(short) \leq 2$}—to trade atomic bit costs against addressing range. Finally, a set of element definitions and initial conditions—a *physics* or 'periodic table'—is compiled into non-volatile tile memory, and propagated tile to tile via out of band signals with respect to the atomic processing level.

2.1 Serve small bowls of serial. Computation in the MFM is based on *event windows*, each of which consists of a *center site* and its {$\ell_1(radius) = 4$} nearest neighbors, exclusively held for the event duration (Figure 2). Center sites are selected in a hardware-dependent but starvation-free manner, and an indefinite number of non-overlapping windows may occur in parallel. In an event, the appropriate element `update` method is invoked on the *active atom* in the center site. If the center site is

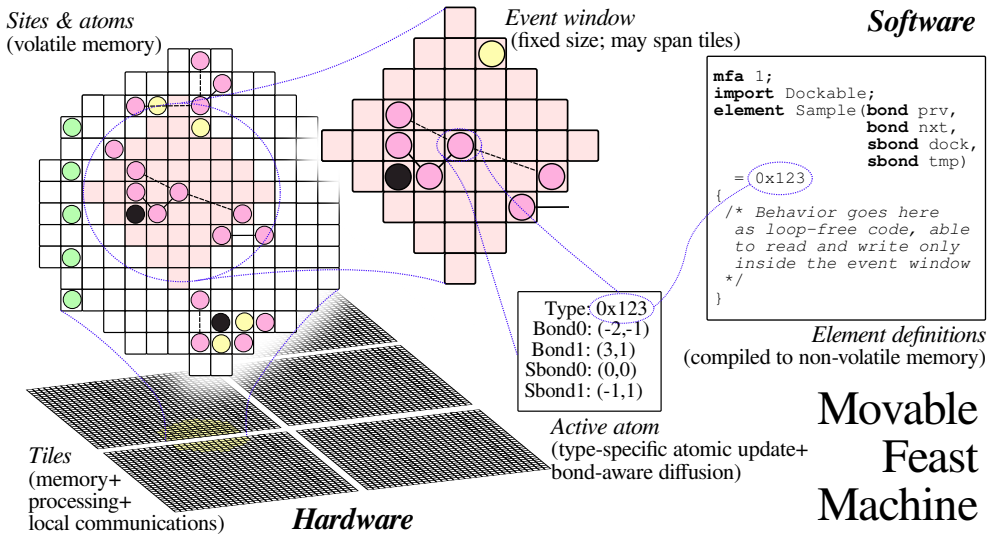

Figure 1: Architectural overview. See text for details.

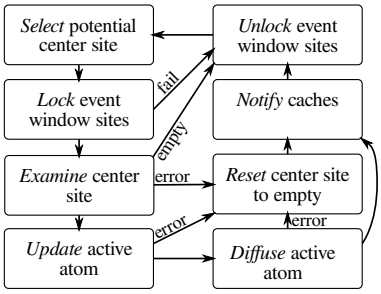

Figure 2: Event window generation. See text for details.

non-empty after the update, by default its atom is then *diffused*, which potentially moves it to a random {von Neumann, $\ell_1 = 1$} neighborhood site, if that is possible without violating any constraints such as overextending a bond.

Although programming MFM element update rules is rather less obvious than classical serial programming, it is also much more intuitive than composing typical CA rule tables, primarily because execution is serial and effectively single-threaded within an `update` method invocation, during which the event window acts like passive memory. The corresponding challenges are that an `update` invocation cannot access any persistent state outside the event window or assume anything about event window contents between invocations.

2.2 Hardware performance. MFM hardware fills space-time with as many events as it can, handling inter-tile communications for locking and caching sites to avoid overlapping event windows and maintain sequential site consistency, so all effects of a prior event are seen by involved sites before relocking. Perhaps the most obvious performance metric for an MFM implementation is 'events per site per second', or equivalently *events per site-second* (EPSS). Peak EPSS would typically occur when all sites are empty, and would likely be just as misleading but ubiquitous as peak MIPS; better would be average EPSS with respect to some benchmark computation. EPSS is intended to measure not absolute power, but *computational density*, as the number of sites grows without bound. So, strictly speaking, any MFM implementation that is limited to a bounded number of sites—such as a serial simulator, for example—fails to have an EPSS. For present purposes, to draw a line in the sand, we will be happy if our first hardware implementation can achieve an average EPSS of 1 on `DReg` physics (Section 3.1, below).

3 Prototypes and results

Here we have space for a few tastes of the mechanisms and techniques we are exploring; the interested reader is invited to the MFM website [7] to find more depth, videos, and other examples ranging from a movable NAND gate to a simple stack machine interpreter.

3.1 Robust homeostasis. Long-lived systems that adapt or are maintained risk losing their robustness over time, either accidentally or for short-term efficiency gains, during those periods when failures *don't* happen. To fight that, a system can be structured to *challenge itself* with 'artifical failures' that exercise its robustness features. We have explored this approach using a simple 'Dynamic Regulator' (`DReg`) element, which usually just diffuses, but probabilistically may delete an atom in a nearby site, or create a new 'resource' atom (element `Res`), or another `DReg`, in an nearby empty site. Given time, a single `DReg` fills MFM space with a churning mixture of `DReg`, `Res`, and empty sites in roughly fixed proportions depending on the probabilities.

Structures sharing space with DReg must thus recover from arbitrary atom deletions, by repair and/or reproduction and death. Although uncontrolled reproduction would amount to cancer, if components are designed to reproduce only by transmuting existing Res atoms, then this DReg homeostatic mechanism will regulate their density as well. Section 3.4 illustrates this technique.

3.2 Self-healing wire. In the MFM, direct communication is possible only between atoms separated by no more than one event window radius. While two atoms can remain close using a bond, locality is a precious resource, and compositionality is facilitated by mechanisms for longer range interactions.

Figure 3 illustrates Self-Healing Wire (SHW), a construct based on four elements—Wire, Transmitter, Receiver, and Message—that provides an extended-range, bidirectional communication channel with redundancy and repair for improved robustness. If a Wire bond or atom fails, adjacent atoms attempt to reconstruct it from the information encoded by the remaining bonds.

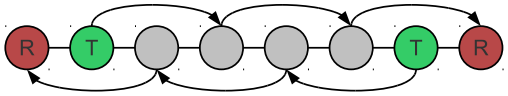

Figure 3: Logical bond structure of self-healing wire: Wire atoms *(unlabeled)* linked by a short bond backbone, with redundant long bonds for data transport (arrows indicate data flows; all bonds are symmetric), transmit (T) and receive (R) terminals. See also Figure 5.

Each Message atom holds an arbitrary 32 bit payload. To send a Message, it is long-bonded to a Transmitter atom's 'docking' bond. Then, when diffusion brings the Message within long bond range of the next forward Wire atom, and that has an empty dock, the Message rebonds to it, and continues in that manner until it arrives at a Receiver which handles it in a problem-dependent way (see Figure 5 and videos at [7]).

Figure 4 illustrates the robustness gain provided by the SHW design, compared to a single-linked chain, and to a doubly-linked chain that does not heal.

3.3 Cell membranes. In biology, the cell wall—a 'selectively permeable spatial divider'—is crucial for modularity. Making a movable 'cell membrane' in the MFM is a bit of a challenge because, in general, the same empty sites that give membrane atoms enough room to move at all can be used by other atoms to move through the membrane. We are handling this via a notion of *bond exclusion*, flagged by a 'BX' bit in the atomic header. While update methods may do what they wish, the diffusion step in Figure 2 restricts the destination sites (in a regrettably complex way) when BX atoms are encountered.

With a bonded ring of appropriate BX-enabled atoms, we create a membrane that is impermeable to diffusion

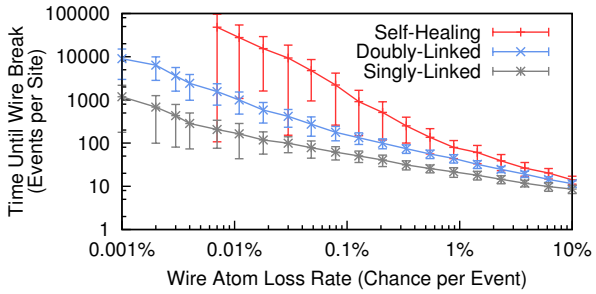

Figure 4: Robustness of SHW compared to singly-linked and doubly-linked, but non-healing, wire. (SHW exceeded testing time limits at lower failure rates.)

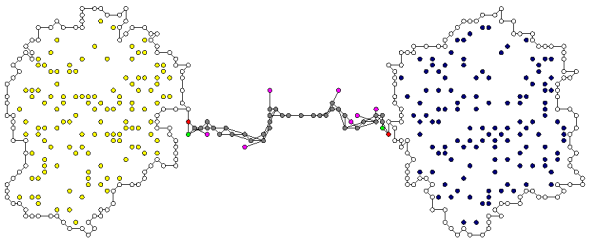

Figure 5: Two cells exchanging messages via SHW.

and can function as a container for a computation. In Figure 5, additional physics on the membrane atoms and on SHW Receiver allow cell membranes to bond to SHW ends for tasks like point-to-point communications, multistage computations, cell routers, and so forth.

3.4 Robust sorting via programmed replication. A final example, which we call *demon horde sort*, demonstrates multiple robustness mechanisms—including opportunistic reproduction (Section 1.2) and dynamic regulation (Section 3.1)—while sorting an endless stream of Datum atoms, each carrying a 32 bit number. A bounded region of 128x64 sites is used, with a self-regenerating column of Input atoms on the right edge, which emit random-valued Datums, and an analogous column of Outputs that consume them, on the left. A 'horde' of Sorter atoms fills the region, each moving Datums right-to-left into empty sites—and also up or down based on the comparison of each Datum's value to the value of the previous Datum moved by that Sorter.

DReg is also loose on the grid, making Res and destroying atoms (even sometimes a Datum, costing a data point), so the horde must replenish itself. To do that in a somewhat general way, we developed a movable program + interpreter mechanism. We pack a sequence of four-bit opcodes into 'base chain' (BC) atoms linked together into an oriented strand. Each BC atom carries four opcodes, and sports an extra 'docking' bond usable by a simple processing element we call 'base chain interpreter' (BCI). When a BCI encounters an open BC molecule head end, it docks on and begins working its

way down (and up) the BC chain executing its opcodes, side-effecting registers stored in BCI instance data, as well as atoms in the event window. Finally, we made a general-purpose chain-copying element that we call 'base chain polymerase' (BCP), which also uses the BC docking bond to build a duplicate of any BC chain, atom by atom from its head, given sufficient Res and time.

Overall, the machine's initial configuration contains some DReg and BCI and seed atoms for the I/O columns, but no actual Sorters. Included instead are several preconstructed five-atom BC molecules each encoding the program 0x20ab5522bc52bc150, which instructs a BCI to transmute Res to make two Sorters, a BCP, and a BCI; and then detach. BCI executions of this program, on top of the DReg physics, creates and maintains the demon horde. Several in-progress copies of this program are visible in Figure 6, which has been 'unfolded' by hand to help illustrate some of the richness of the resulting dynamics; [7] offers video.

Once the machine warms up, the demon horde sorts very well, albeit imperfectly, and it is remarkably resilient, recovering from insults like flipping bits in many atoms at once, or flat-out resetting two-thirds of the grid sites. We tried to stress it with increased Datum flow, but it surprised us by working *better*—like a 'sorting muscle', the horde is sloppy in idleness but improves with increasing work, until finally it can't clear sites around the Inputs fast enough, so data losses start to climb.

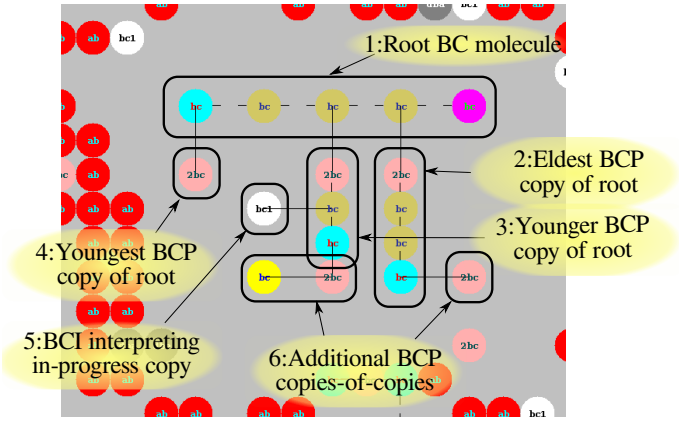

Figure 6: Annotated closeup showing multiple reproduction threads—rearranged for clarity in a cleared patch—space-sharing with Sorters and other elements.

4 Critique and call to action

Two obvious criticisms of our approach are: (1) We've thrown the baby out with the bathwater, and (2) our hardware costs will be too high and our event rate too low to compute cost-effectively. Granted, the MFM is a long journey from computation as usual, but we have a landmark—biological and ecological approaches to robustness and scale—in sight to guide us, and after all, with small steps the only way off a hilltop is down.

And yes also, this does seem like a profligate use of hardware—for example we allow great latitude in what an update does, suggesting essentially a full-fledged MCU might be needed to do it. The event window size and locking are also costly. But we believe performance concerns are premature, and that the first goal is to find an interesting 'periodic table of elements' from which to compose useful molecules, cells, and larger systems. If we can do that, there's every reason to expect smart hardware folk will be able to crush costs and explode the event rate, and indefinite scalability will have our backs.

And finally, to challenges on expressiveness—arguing the MFM will be too painful to program because our atom or event window is too small, or bonds too few, or lengths too short, or system dimensionality too low, we say: You might well be right. Let's find out.

Acknowledgments

Lance R. Williams collaborated with Ackley on the Movable Feast Machine concepts and design; but for prior exigencies he would have been an author on this paper. Also, the anonymous HOTOS reviewers provided open-minded comments that greatly improved the paper.

References

[1] Forrest, S., Somayaji, A., and Ackley, D. H. Building diverse computer systems. In *Proc. HotOS VI* (Washington, DC, 1997), pp. 67–72.

[2] Furnas, G. Video: Bitpict implements itself. http://hdl.handle.net/2027.42/83491, 2011.

[3] Ganguly, N., Sikdar, B. K., Deutsch, A., Canright, G., and Chaudhuri, P. P. A survey on cellular automata. Tech. rep., Centre for High Performance Computing, Dresden Univ. of Tech., Dec. 2003.

[4] Holland, J. H. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.

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

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

[7] Movable Feast Machine research team. The movable feast machine landing page. http://movablefeastmachine.org/.

[8] Peper, F., Isokawa, T., Kouda, N., and Matsui, N. Self-timed cellular automata and their computational ability. *Future Generation Computer Systems 18*, 7 (2002), 893 – 904.

[9] Tilera Corporation. http://www.tilera.com, 2011.

[10] Toffoli, T. Programmable matter methods. *Future Generation Computer Systems 16* (1998), 201.

[11] Toffoli, T., and Margolus, N. *Cellular Automata Machines: A New Environment for Modeling (Scientific Computation)*. The MIT Press, 1987.

[12] Vasudevan, V., Franklin, J., Andersen, D., Phanishayee, A., Tan, L., Kaminsky, M., and Moraru, I. FAWNdamentally power-efficient clusters. In *Proc. HotOS XII* (Monte Verita, Switzerland, 2009).

[13] XMOS Corporation. http://www.xmos.com, 2011.