Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)

Indefinite Scalability for Living Computation

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

Abstract

In a question-and-answer format, this summary paper presents background material for the AAAI-16 Senior Member Presentation Track "Blue Sky Ideas" talk of the same name.

Q: So, what's the big idea here?

A: Traditional CPU and RAM computing, based on **hardware determinism**, is now struggling to scale up, as clock speed increases have stalled and multicore cache coherence grows increasingly expensive. Determinism is also a perpetual computer security nightmare, encouraging programmers to optimize efficiency, and thus fragility—while providing, by default, utter predictability to attackers.

The blue sky idea is: We should forgo deterministic execution and focus instead on *best-effort computing*—in hardware and software both—to develop *indefinitely scalable* computer designs. A machine from here to the horizon if we want it, built of locally-connected, interchangeable computing tiles; a machine so big parts of it will always be failing; a machine so big we'll be using it long before we finish building it.

To survive and prosper in such a system, software will be *living computation* in a far richer sense than today: It will collaborate and compete with other software, will take damage and heal, will reproduce for redundancy and parallelism, will migrate and spontaneously colonize new hardware, and so on. Programming and performing large-scale computations will be less like solving logic puzzles and more like agriculture or animal husbandry or ecosystem management.

Q: Wow! So this is all just wishes and fantasy, right?

A: Well, not entirely. It's still the earliest days, but for several years we've been working on indefinite scalability, mostly in simulation. Unless stated otherwise, the material in this summary paper is drawn from (Ackley and Cannon 2011; Ackley 2013a; 2013b; Ackley, Cannon, and Williams 2013; Ackley and Small 2014; Ackley and Ackley 2015). And though it can sound like science fiction, our proposed hardware architecture depends only on conventional electronics manufacturing and presumes no breakthroughs in materials science or unconventional computing media.

Q: So why, exactly, doesn't determinism scale?

A: Hardware determinism means that digital computer hardware *guarantees* to perform the rules of logic flawlessly, for as long as the program runs—and if the hardware can't do that, for any reason, it promises to "crash" the whole machine immediately. With that guarantee in hand, typical software utterly ignores the possibility of hardware errors—and that 'reliable hardware + efficient software' mindset is so deeply embedded, in academia and industry both, that it is rarely even mentioned, let alone questioned.

The scaling problem is that hardware's guarantee isn't quite 100%. Any real unit of hardware will have *some* small chance of undetected error. Now, if we are given a maximum computation size up front, we can design hardware units to make *that* size computation as reliable as we like. But if we then keep adding more and more of those hardware units to the system, and running it longer and longer, eventually the aggregate space-time computational volume will exceed the reciprocal of the hardware unit failure rate, and the machine *will* deliver undetected errors to the software level.

That's why, even in principle, *hardware determinism doesn't scale*, and in the supercomputing community, for example, it is already more than a merely theoretical concern (Cappello et al. 2009).

Q: But if determinism fails, doesn't that mean chaos?

A: Not necessarily at all. Computer science's focus on efficiency makes it easy to forget that digital hardware employs *massive redundancy* to accomplish its heroic acts of determinism. It uses whole wires to carry single bits, and deploys saturating amplifiers at every turn to reduce noise while the chance of error remains small.

When we renegotiate around best-effort hardware, software becomes obligated to look beyond efficiency and embrace *robustness*, by deploying redundancy effectively throughout the software stack. The resulting systemic error-resistance will enable scalability, and also fundamentally benefit security—not like a silver bullet miracle, but like the way sanitation and hygiene benefits the public health.

Q: Hmm. What do we need know most about this idea?

A: Here are the bottom lines: This proposal represents a huge but feasible change, and it is important for society and long overdue, and it needs your help.

It's built on three overlapping concepts—*indefinite scalability* for hardware and architecture, *robust-first computing* for software and programming, and *best-effort computing* for systems overall and models of computation.

In the rest of this paper, we will say a little more about indefinite scalability in computer architecture, and then consider the particular approach to it we have been exploring, which is called the *Movable Feast Machine*. We'll show a few simple examples to illustrate the ideas and the current level of research and development, and conclude with a call to action.

Q: What is indefinite scalability?

A: Indefinite scalability is a design principle saying that any admissible computer architecture must be **expandable to arbitrary size** without any fundamental re-engineering. We may be unable to grow our indefinitely scalable machine for *external* reasons—like we run out of money or real estate or power or cooling—but never because we hit some *internal* design limit, like reaching the limits of a fixed-width address or the light cone of a central clock.

Q: So, something like the internet?

A: No, as it is most commonly used, the Internet is only finitely scalable, because its (IPv4 or v6) address space is finite. The Internet *could be* indefinitely scalable, however, via aggressive use of "anycasting" (Partridge, Mendez, and Milliken 1993; Abley and Lindqvist 2006), which offers indefinite spatial scaling in exchange for a finite set of interprocessor request types.

Q: But how can over 10^{38} IPv6 addresses *not* be enough?

A: Because for indefinite scalability, "really big" amounts to "merely finite": To preserve the clarity and power of the idea, any appeals to practical sufficiency are irrelevant. Moreover, anecdotally at least, it seems that every internal limit in a design is a form of *technical debt* (Cunningham 1992; Allman 2012) that has consequences beyond the limit itself. A design built on globally unique node names drawn from a finite space, for example, incurs not only the risk of name exhaustion, but all the issues and risks of centralized naming and resolving, aliasing, spoofing, and so on. Indefinite scalability doesn't automatically solve or avoid any such issues, but it forces them all to the foreground; it helps keep us honest and thereby provides a more stable basis for evaluating and choosing between architectures.

Q: Well, if not the Internet, what *is* indefinitely scalable?

A: Good question! We'll introduce our primary example, the *Movable Feast Machine*, in a moment. A repurposed "anycast-Internet," as mentioned above, could be one example. And though cellular automata (CA) typically assume deterministic execution and thus render themselves only finitely scalable, *probabilistic* cellular automata (Grinstein, Jayaprakash, and He 1985; Agapie, Andreica, and Giuclea 2014, e.g.) do go beyond determinism—and some of them could plausibly be cast into indefinitely scalable form.

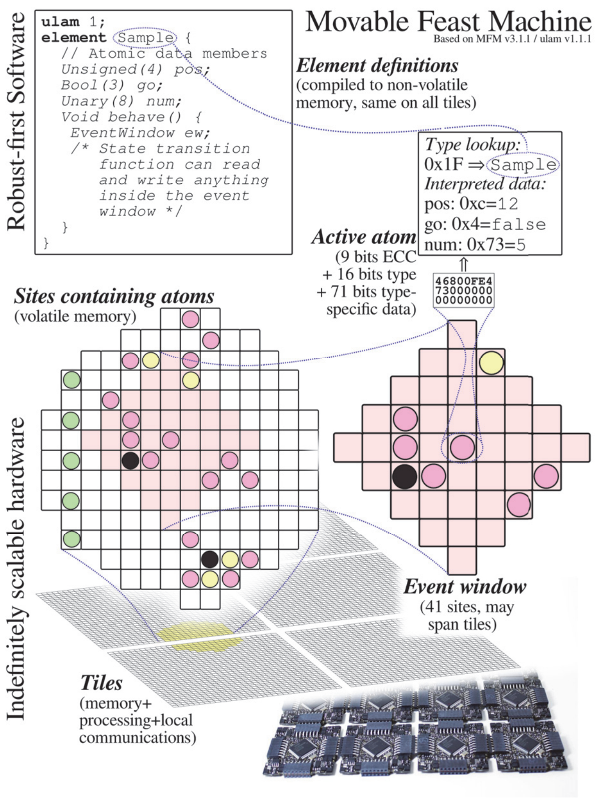
```
ulam 1;
element Sample {
  // Atomic data members
  Unsigned(4) pos;
  Bool(3) go;
  Unary(8) num;
  Void behave() {
   EventWindow ew;
   /* State transition
      function can read
      and write anything
      inside the event
      window */
  }
}
```
Robust-first Software
Movable Feast Machine
Based on MFM v3.1.1 / ulam v1.1.1
Element definitions
(compiled to non-volatile memory, same on all tiles)
Type lookup:
0x1F ⇒ Sample
Interpreted data:
pos: 0xc=12
go: 0x4=false
num: 0x73=5
Active atom
(9 bits ECC + 16 bits type + 71 bits type-specific data)
46800FE4
73000000
00000000
Sites containing atoms
(volatile memory)
Indefinitely scalable hardware
Event window
(41 sites, may span tiles)
Tiles
(memory+ processing+local communications)

Figure 1: The Movable Feast Machine (MFM) architectural overview, with 2009-era prototype tile hardware at bottom. (See text.)

In general, a design for an indefinitely scalable machine amounts to a spatial tiling of hardware units, with additional requirements as needed to preserve open-ended physical realizability. As an obvious example, fixed-latency global communications is disallowed by the finite speed of light, as are hierarchical or hypercube interconnects based on fixed-latency links. Less obviously, the presumption of a global reset or "boot time" is also disallowed.

Q: It's too hard! Is indefinite scalability really worth it?

A: It *is* hard, but the potential upside is computational systems of unlimited size, that are inherently tough, good at their jobs, and hard to attack. In the end, hardware determinism is a property of small systems. Best-effort computing is the future—it is what we should be optimizing, rather than determinism. We are hurting ourselves by delaying.

Q: What is the Movable Feast Machine?

A: The Movable Feast Machine is a tile-based indefinitely scalable architecture (see Figure 1). Each tile is a small von Neumann machine running an identical control program out of non-volatile local memory. The program treats local volatile memory as a patch of cellular automata grid, and

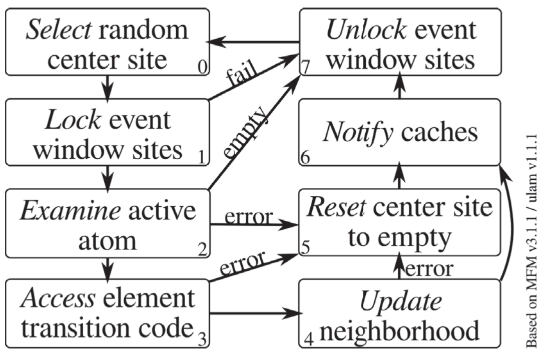
Select random center site 0
Lock event window sites 1
Examine active atom 2
Access element transition code 3
Unlock event window sites 7
Notify caches 6
Reset center site to empty 5
Update neighborhood 4
fail
empty
error
error
error
Based on MFM v3.1.1 / ulam v1.1.1

Figure 2: The MFM per-tile event loop. (See text.)

performs events on randomly-chosen local grid sites, while coordinating with adjacent tiles to seek consistent cache views of their relevant grid sites. There are no global clocks or synchronization barriers, and tiles race against each other to acquire intertile locks for events that may affect caches.

The overall effect is of an asynchronous, probabilistic cellular automata, but one in which an individual grid site's neighborhood—which we call an *event window*—is huge by CA standards, involving thousands of bits. As a result, a traditional CA state transition lookup table is completely infeasible; instead, a mostly-traditional serial deterministic function is called to perform the state transition, depending on the *type* bits in the event window's center site. Figure 2 summarizes the tile event loop.

Q: But if a transition is deterministic, what's the point?

A: That determinism only lasts as long as an event, which is short, and can affect only its event window, which is tiny by traditional programming standards. Although programming MFM element behavior functions 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 event, during which the event window acts like passive memory. The corresponding challenges are that event code cannot access persistent state outside the event window or assume anything about event window contents between invocations.

Q: What does state transition code look like?

A: Over the last year and a half, we have developed a programming language, called ***ulam***, specifically to express MFM state transitions. ***ulam*** has its own distinct flavors but is deliberately designed to seem reasonably familiar to programmers used to conventional object-oriented procedural languages. ***ulam*** compiles into C++, and from there via `gcc` to machine code for dynamic loading into the simulation, and we hope soon into live tile hardware as well. Figure 1 includes some legal but pointless ***ulam*** code, and Figure 3 presents a functional but complete lout of an element—a `ForkBomb` that attempts to fill space with copies of itself, with no regard for what might already be there: Hello World and then some.

```
ulam 1;

/** Fork bomb.
  \symbol FB
  \color #f00
  \symmetries all
*/
element ForkBomb {
  EventWindow ew;
  Void behave() { ew[1] = ew[0]; }
}
```

Figure 3: A complete ***ulam*** element. Copies itself from the event window center (`ew[0]`) to `ew[1]`, which in this case (due to the `\symmetries all` in the element metadata) is an adjacent site chosen at random on each event.

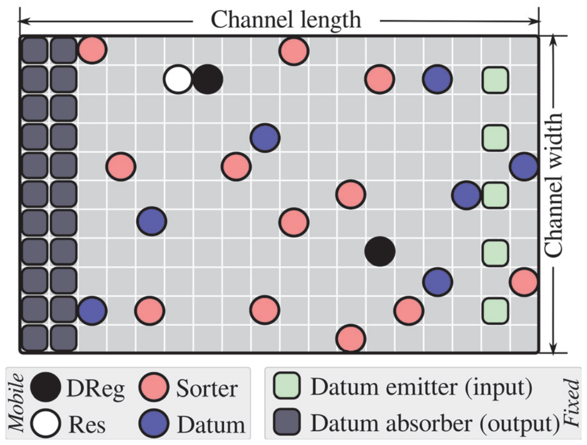

Figure 4: A *Demon Horde Sort* stochastic sorter. (See text.)

Q: That's cute, but how do more complex things work?

A: In a general sense, building larger structures in the Movable Feast involves three aspects:

1. *Functional*: Programming more complex element behavior functions, and involving greater numbers of interacting element types,
2. *Spatial*: Deciding where to locate the involved atoms relative to each other, and if and how they should move, to make possible local interactions that are useful within the larger structures and purposes, and
3. *Temporal*: Using spatial and functional mechanisms to implement staged processes that unfold in time, such as growth phases. Also, adjusting rate constants so some subprocesses run much slower or faster than others. This can enable programmers to use constant approximations (if slower) or equilibrium approximations (if faster) to simplify reasoning about otherwise complex dynamics.

Q: Well, maybe I asked for that. How about an example?

A: Figure 4 is a schematic representation of one of the earliest machines we investigated, called the *Demon Horde Sort*

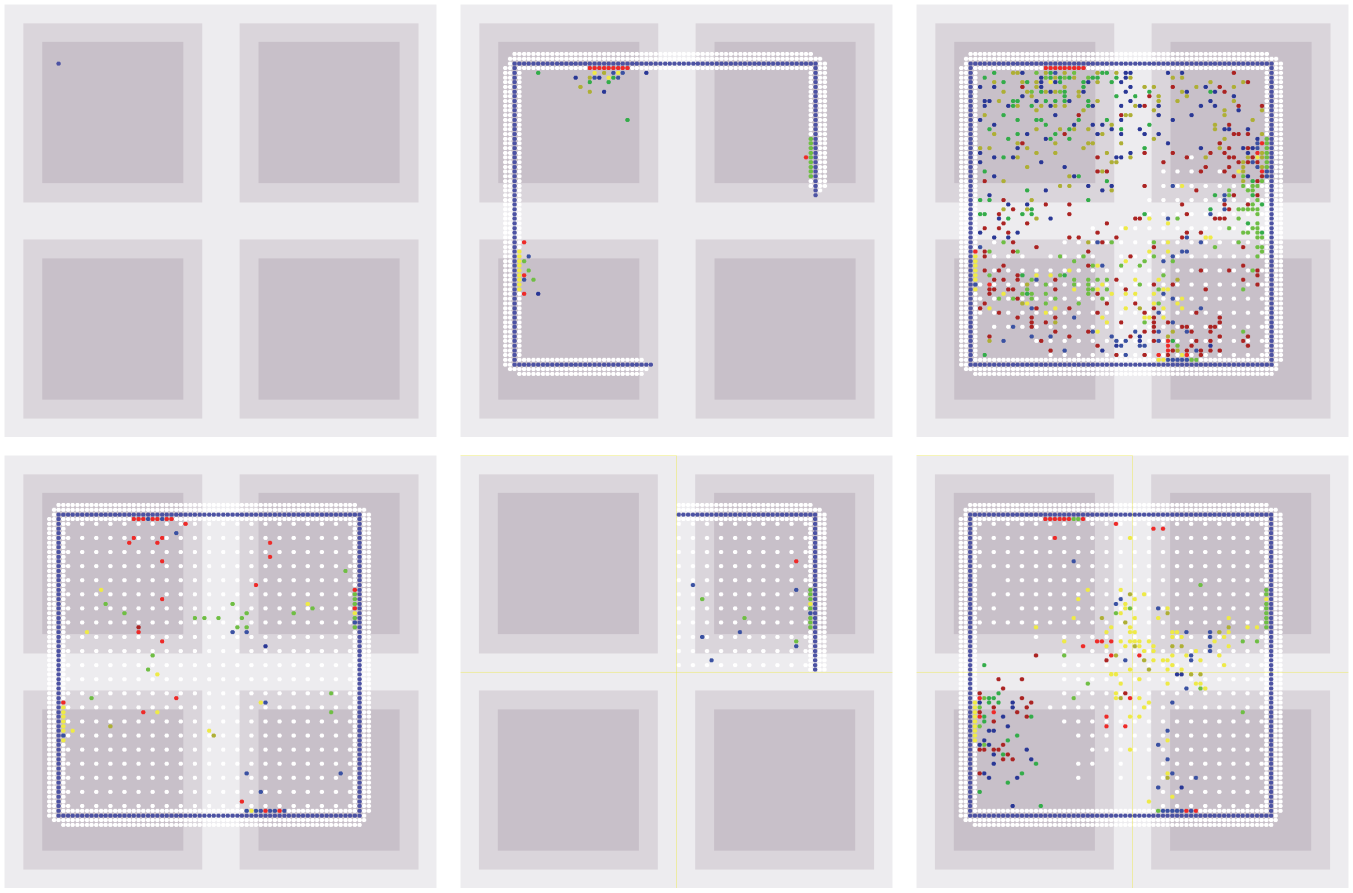

Figure 5: Six events in the life of a self-assembling four-port data switch, spreading over four simulated MFM tiles. *Top left:* Initial state. *Top center:* After 100 average events per site (AEPS). *Top right:* After 1000 AEPS. *Bottom left:* After 2000 AEPS. *Bottom center:* After three tiles are reset at 2500 AEPS. *Bottom right:* After 3000 AEPS. See text.

(DHS). Data atoms (*blue*), each holding a random 32 bit number, are placed on the grid by the emitters (*green*) at the right, and are carried right-to-left and moved up and down by the Sorters (*red*), and eventually are extracted by absorbers on the left (*dark grey*). The trick is, each Sorter remembers the value of the last Data it moved, and when it considers moving the next Data atom right-to-left, it also tries to move it up or down based on the comparison of the current and previous Data values.

The DHS illustrates all of the compositional mechanisms just mentioned. By copying the Data value into the Sorter's internal storage, Data-Data comparisons can be performed even when the Data items aren't close enough to interact directly: The existence and behavior of the Sorters enable local interactions that advance the goals of the larger structure. Similarly, by making basic spatial and geometric assumptions—that input is East, output is West, small is North and large is South—the Sorters can autonomously take actions that will help in the computation as a whole.

And finally, the "DReg" and "Res" in Figure 4 are part of a "Dynamic Regulator" feedback system that maintains the Sorter population at a reasonable density. Though the DReg mechanism is interesting in its own right, here it just serves as an example of the temporal layering of dynamics: The DReg operations are so slow compared to the Sorters that the DReg can basically be ignored except when considering the long-term dynamics of the system.

Q: OK, but, how can the DHS possibly sort correctly?

A: It doesn't. Its inherent resolution is determined by the channel width, and its sorting quality generally improves with increasing length/width aspect ratio. More basically, though, in this problem formulation, correctness really isn't an option. Given that the emitters produce Data atoms intermittently and unpredictably, it's not even well-defined what the "correct" maximum value really is at any given moment. Welcome to best-effort.

Q: I start to see.. Have you built other machines?

A: Certainly. As one last example, just briefly, Figure 5 shows six snapshots in the self-assembly of a toy four-port data switch we have recently been exploring.

Starting from a single 'switch wall' atom, the switch builds a box (*blue outline*), insulates it with passive Walls (*white*) embedded with four I/O ports (*red, yellow, green, and blue line segments*), and builds a routing grid (*evenly-spaced white dots*). The routing grid atoms observe nearby ports and then gossip among themselves to form gradients to guide the data cells (*scattered dots, colored to show their destination port*). Each data cell carries a 32 bit payload and an eight bit sequence number, although this switch does not attempt to perform packet reassembly.

After an average of one thousand events per site (1000 AEPS), the switch has completely self-assembled but still has a backlog of data cells; by 2000 AEPS the switch is operating smoothly. Later, at 2500 AEPS, we severely damage it by resetting three of the four underlying tiles, but it immediately begins to reassemble itself and, by 3000 AEPS, it is almost healed.

Q: Very cool! But, how does all this affect AI?

A: Traditional computing is about constructing things once and then trusting them to remain intact indefinitely. That's why it's, at once, so marvelously efficient and so deathly fragile. Robust-first computing, on the other hand, is about continuous self-construction and maintenance, so that information structures are automatically refreshed as needed, or at some rate, or both.

Sixty-five years ago, von Neumann (1951) predicted that hardware determinism would soon be supplanted, but with design lock-in and network effects it remains virtually unchallenged today. In AI's long-running cross-pollination, let us call it, between the neats and the scruffies, that dominance has given the neats sole possession of the home court advantage—deterministic execution—without anybody really calling them on it. But now the costs of playing on that court are rising too high.

It's time for the neats to try playing an away game.

Q: Last question! I hate to ask, but who funds this work?

A: Eventually, a substantial research and development effort, performed and supported by many people and organizations, will be needed to develop the science and engineering of best-effort computing. Perhaps you will be part of that. For these early stages, support has come from a brave and visionary few. The work summarized here was supported in part by a Google Faculty Research Award, and in part by grant VSUNM201401 from VanDyke Software.

References

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

Ackley, D. H., and Ackley, E. S. 2015. Artificial life programming in the robust-first attractor. In *Proc. of the European Conference on Artificial Life (ECAL)*.

Ackley, D. H., and Cannon, D. C. 2011. Pursue robust indefinite scalability. In *Proc. HotOS XIII*. Napa Valley, California, USA: USENIX Association.

Ackley, D. H., and Small, T. R. 2014. Indefinitely Scalable Computing = Artificial Life Engineering. In *Proceedings of The Fourteenth International Conference on the Synthesis and Simulation of Living Systems (ALIFE 14) 2014*, 606–613. MIT Press.

Ackley, D. H.; Cannon, D. C.; and Williams, L. R. 2013. A movable architecture for robust spatial computing. *The Computer Journal* 56(12):1450–1468.

Ackley, D. H. 2013a. Bespoke physics for living technology. *Artificial Life* 19(3_4):347–364.

Ackley, D. H. 2013b. Beyond efficiency. *Commun. ACM* 56(10):38–40. Author preprint: http://nm8.us/1.

Agapie, A.; Andreica, A.; and Giuclea, M. 2014. Probabilistic cellular automata. *Journal of Computational Biology* 21(9):699–708.

Allman, E. 2012. Managing technical debt. *Queue* 10(3):10:10–10:17.

Cappello, F.; Geist, A.; Gropp, B.; Kal, L. V.; Kramer, B.; and Snir, M. 2009. Toward exascale resilience. *IJHPCA* 23(4):374–388.

Cunningham, W. 1992. The WyCash Portfolio Management System. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, 29–30. New York, NY, USA: ACM.

Grinstein, G.; Jayaprakash, C.; and He, Y. 1985. Statistical mechanics of probabilistic cellular automata. *Phys. Rev. Lett.* 55:2527–2530.

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

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. 15–19. Also appears as pages 302–306 in A.H. Taub, editor, *John von Neumann Collected Works: Volume V – Design of Computers, Theory of Automata and Numerical Analysis*, Pergamon Press, 1963.