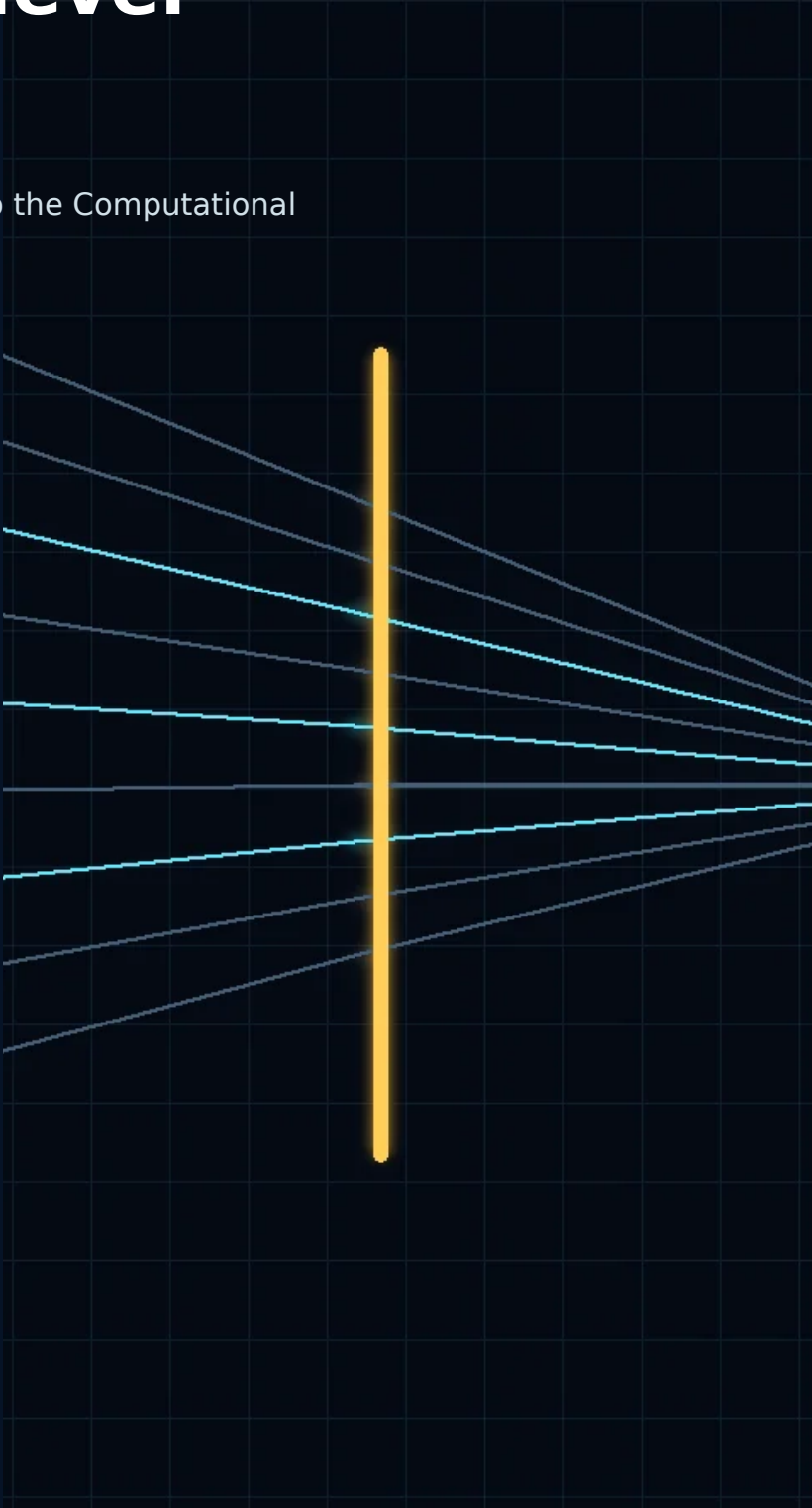


The Work We Never Needed to Do

From Voodoo2, UltraHLE, and PowerVR to the Computational Shortcuts Behind XERXES SI



And sometimes thousands of tiny decisions like these can be stacked together until a machine behaves as if it possesses far more raw computational power than its electrical consumption, memory footprint, processor age, or thermal output would suggest.

That last category is particularly important to me.

It describes a significant part of how I have approached **Xerxes SI**.

There is no single trick responsible for its efficiency.

There are many.

Some are large architectural decisions. Others are tiny shortcuts.

By *shortcut*, I do not mean cutting corners.

I mean finding a **shorter correct path**.

If something is already known, do not reconstruct it.

If only three possibilities remain, do not evaluate thirty.

If a deterministic operation can answer the question, do not invoke a probabilistic general-purpose system.

If one module has already established an invariant, the next module should inherit that knowledge rather than rediscover it.

If a branch is impossible, kill it before it becomes expensive.

If a high-level operation can be translated directly, do not reproduce every low-level mechanism that originally produced it.

If a problem becomes simpler in another representation, change the representation.

If an answer can be reached after five operations, there is no virtue in performing five thousand.

These shortcuts accumulate.

And when they are arranged properly, they do more than add.

They compound.

That is the historical and technical argument of this article.

What if the most important performance optimization is not doing work faster, but discovering that the work never needed to be done?

1. S3 ViRGE: A Feature List Is Not an Architecture

The S3 ViRGE is remembered with a nickname that was cruel because it was sometimes deserved:

“3D decelerator.”

The joke hides a more useful lesson.

S3 already knew how to build competent 2D graphics hardware. The problem was not that the company had never built a display accelerator.

The problem was that adding a collection of 3D functions did not automatically produce a balanced 3D system.

A chip might nominally support:

- texturing;
- filtering;
- perspective correction;
- depth operations;
- blending;
- polygon rendering.

What matters is what happens when real software asks it to perform those operations together.

That is where specification sheets stop and architecture begins.

A system is not the collection of things it can nominally do. A system is the cost of using those capabilities together.

This remains true today.

Adding more modules to an intelligent system does not automatically make it better.

Memory.

Retrieval.

Planning.

Reasoning.

External models.

Vector databases.

Search.

Agents.

Tools.

Counterfactuals.

All of these can become useful.

They can also become overhead.

If every subsystem wakes up for every request, duplicates state, generates another representation, and reevaluates questions that have already been narrowed elsewhere, sophistication becomes inefficiency.

The real architectural question is:

What is the shortest reliable path between the state we have and the answer we need?

2. Savage and S3TC: Sometimes the Winning Move Is to Change the Representation

The Savage family contained ideas far more interesting than its eventual market position suggests.

One was **S3 Texture Compression**.

Textures cost memory.

Moving textures costs bandwidth.

Bandwidth costs energy and time.

The obvious solution is:

More memory. More bandwidth.

S3 pursued another path:

Represent the useful information more efficiently.

That matters because reducing the amount of information that must be moved can be better than increasing the speed at which a larger amount is moved.

This principle became important enough that the underlying block-compression approach survived S3's position in the graphics market.

The product can disappear while the useful representation survives.

There is a very direct parallel to modern computation.

Suppose a system represents the same state in twelve different ways and passes all twelve through the pipeline.

The brute-force solution is faster hardware.

The architectural solution may be:

Why are there twelve representations?

Perhaps three are sufficient.

Perhaps one can be generated lazily.

Perhaps another is unnecessary if a previous module has already classified the state.

Efficiency begins before execution.

3. MeTaL and Savage 2000: Optimization Without Correctness Is Not Optimization

S3's **MeTaL** API demonstrated another principle: software and hardware can sometimes work extremely well when they share a specialized semantic path.

But specialized paths introduce a requirement that cannot be compromised:

correctness.

Those who used various Savage cards remember that performance was not always the only problem.

Sometimes the rendering was simply wrong.

Surfaces disappeared.

Transparency behaved incorrectly.

Geometry could become malformed.

Textures could be absent.

The Savage 2000 and its ambitious transform-and-lighting story made the problem even clearer.

A hardware feature that exists but cannot reliably produce the correct result is not an acceleration architecture.

This matters enormously in the work we are doing.

There is a tempting misunderstanding about computational shortcuts.

A shortcut does **not** mean:

Skip something and hope.

A valid shortcut means:

Establish that the longer path is unnecessary.

That distinction is everything.

A Xerxes SI optimization that eliminates 90 percent of potential work but occasionally eliminates the correct answer would be unacceptable.

The correct equation is not:

less computation

It is:

less computation + preserved semantics + verification + fallback

That is how shortcuts become architecture rather than hacks.

4. Matrox G400: More Apparent Complexity Without More Literal Complexity

The Matrox G400 produced one of the memorable visual moments of that generation.

Environment-Mapped Bump Mapping.

Water seemed to ripple.

Reflections bent across surfaces.

Materials appeared to have fine structure that was not actually being modeled with enormous geometric complexity.

The important insight is not merely that the effect looked good.

The important insight is:

The renderer changed the representation of the problem.

If the question is:

What should this irregular reflective surface look like?

the expensive solution could be:

Model every irregularity geometrically.

The elegant solution is:

Represent the visual consequence of those irregularities at a cheaper level.

That is a shortcut.

A correct one.

The computation does not become faster because multiplication became magical.

The computation becomes cheaper because the system stopped insisting that microscopic visual structure had to be represented as microscopic physical geometry.

This pattern appears everywhere in computing.

5. BitBoys: Good Ideas Can Outlive Products That Never Arrive

BitBoys became famous for graphics chips that always seemed approximately one product cycle away from changing the world.

Some of its proposals looked extraordinary.

Embedded memory.

Huge effective bandwidth.

Advanced filtering.

Antialiasing.

Architectural ideas well ahead of much of the shipping desktop market.

The expected desktop products never materialized in the way enthusiasts hoped.

But that does not make every underlying idea meaningless.

The useful distinction is between:

1. inventing an architecture;
2. manufacturing the architecture;
3. shipping a reliable commercial system.

All three matter.

But failure at one stage does not necessarily invalidate the others.

Environment-Mapped Bump Mapping itself escaped BitBoys and appeared commercially through Matrox.

Later, BitBoys moved toward mobile graphics.

That is worth remembering because the mobile world eventually became the place where computational efficiency was not merely elegant.

It was mandatory.

6. 3dfx and Glide: Specialization as a Shortcut

The original Voodoo architecture was wonderfully strange.

It did not even try to be the complete graphics subsystem.

The normal display card handled ordinary video.

The Voodoo specialized in 3D.

A VGA cable physically connected them.

It was almost embarrassingly literal specialization.

And it worked.

Glide created a software path closely connected to the hardware.

The system did not need to behave like every imaginable future graphics device.

It needed to execute the operations games of that era required extremely well.

That is another form of shortcut:

Do not pay for generality that the current problem does not require.

There is always a danger.

Specialization can become a dead end.

That happened when broader standards such as Direct3D and OpenGL became increasingly important.

So the lesson is not that everything should become proprietary and specialized.

It is:

Specialize internally where doing so produces efficiency, while maintaining an escape path to broader standards.

That principle strongly influences how I think about Xerxes SI.

A deterministic internal skill can be highly specialized.

A reasoning module can be highly specialized.

A memory operation can be highly specialized.

None of that requires the system to lose the ability to interact with general-purpose models or external APIs when necessary.

7. Voodoo2 SLI: Parallelism Does Not Eliminate the Need for Architecture

Two Voodoo2 boards connected through Scan-Line Interleave were one of the most visible demonstrations of parallel processing a consumer could own.

Two boards.

Ribbon cable.

Alternating scan lines.

The division of work was physically visible.

But SLI also demonstrated something we often forget:

Parallelism does not make unnecessary work necessary.

If a task contains 1,000 operations and 900 of them can be removed, the first question should not be:

How do I divide all 1,000 operations between two processors?

It should be:

Why are we doing the 900?

Then parallelize the remaining 100 if parallelism still helps.

This distinction matters today because modern hardware makes it extraordinarily easy to respond to inefficiency with additional concurrency.

More cores.

More threads.

More GPUs.

More nodes.

More accelerators.

That can be useful.

But removing work first can amplify the value of every processor that remains.

8. NVIDIA: The Horsepower Path Wins

NVIDIA's progression through TNT, TNT2, GeForce 256, GeForce2, and later generations demonstrated the enormous commercial and technical power of scaling general graphics hardware.

More throughput.

More integration.

Hardware transform and lighting.

Increasing programmability.

Increasing memory bandwidth.

The strategy worked.

I experienced that transition directly.

I had a **Diamond Viper V770 Ultra**, based on the TNT2 Ultra, and returned it for a **Creative 3D Blaster** from the newer GeForce generation.

Eventually the GeForce became the center of my graphics system.

But for a while the machine still contained the earlier specialists.

Two Voodoo2 cards remained because of Glide and UltraHLE.

The Hollywood Plus remained for MPEG-2.

The system looked almost absurd by modern standards.

But every board existed because, for some class of computation, it represented the shortcut.

As general-purpose hardware absorbed these capabilities, the specialized boards disappeared.

This gave the industry an extraordinarily successful default response:

more computation → more horsepower

There is nothing inherently wrong with that.

The danger is forgetting the alternative:

more computation → ask whether the computation is necessary

9. Creative Unified: Translation Is a Shortcut

Creative Unified attempted to bridge Glide software onto other graphics hardware.

The specific implementation and its history were messy.

But the underlying objective was important.

Software knows how to ask for operation A.

New hardware natively understands operation B.

The naive choices are:

- preserve the old hardware forever;
- break the software.

Translation gives us a third:

$A \rightarrow \text{equivalent } B$

The machine avoids reproducing a complete obsolete subsystem.

It translates the meaning.

This is exactly the type of shortcut that becomes profound in emulation.

10. UltraHLE: One of the Greatest Computational Shortcuts of Its Era

UltraHLE remains one of the most important examples in this entire discussion.

The Nintendo 64 was not ancient when UltraHLE appeared.

It was contemporary hardware.

Traditional thinking suggested that accurately emulating a contemporary console required vastly more computational power than the original machine.

Then UltraHLE appeared and effectively said:

What if we do not emulate everything?

Instead of reproducing every low-level mechanism, it intercepted higher-level behavior and translated useful functions into operations the PC and its Glide hardware could execute efficiently.

That was a radical shortcut.

Not:

Skip correctness.

But:

Preserve the relevant behavior at a higher level.

The difference is enormous.

This is one of the architectural questions I repeatedly ask about Xerxes SI:

Are we reproducing a process because the result requires that process—or simply because that is how somebody previously implemented it?

If a conventional intelligent system reaches a deterministic result through a huge statistical process, and Xerxes SI can establish the same correct result through structured state, a rule, a known relationship, a skill, or another cheaper path, then recreating the expensive process would be wasteful.

UltraHLE was a proof that the level at which a problem is represented can completely change its apparent computational requirements.

11. Rosetta 2 and Cocoa: Preserve Meaning at the Right Boundary

Rosetta 2 carries a related philosophy.

Apple did not need an Intel processor inside every Apple silicon machine in order to preserve Intel software.

The system could translate.

Again:

Preserve the required semantics, not the obsolete machinery.

Cocoa demonstrates a related idea from another layer.

Software designed around stable high-level APIs can survive major architectural transitions because the implementation below the API can change.

The broader rule is:

Choose the boundary at which meaning remains stable, and allow everything below that boundary to evolve.

That is a powerful shortcut because it prevents every software layer from needing to understand every lower-level implementation detail.

12. Java, ART, JITs, and Python: Wait Until You Know More

Java and modern managed runtimes add another important shortcut:

Do not make every optimization decision before you know how the program actually behaves.

A JIT compiler can observe:

- frequently executed paths;
- common types;
- rare branches;
- stable values;
- hot functions.

Then it can optimize the work that matters.

Android ART similarly combines multiple compilation strategies.

Python ecosystems do something related through JIT systems and highly optimized native numerical libraries.

The lesson is not that interpretation is faster than native machine code.

It is:

Runtime knowledge can expose shortcuts unavailable at compile time.

This leads directly to something I think may become important for Xerxes SI:

A runtime capable not merely of choosing instructions, but of choosing **representations**.

A representation JIT.

More on that shortly.

13. Hollywood Plus: Specialization Dies and Returns

The Hollywood Plus MPEG-2 decoder represented a time when DVD decoding was expensive enough to justify dedicated silicon on a PCI board.

Eventually general CPU and GPU hardware became powerful enough that the extra board was unnecessary.

Specialization disappeared.

Then mobile computing arrived.

Suddenly energy mattered.

Heat mattered.

Battery life mattered.

And dedicated encode/decode blocks returned inside integrated systems.

Why?

Because:

Being capable of performing an operation is not the same as being efficient at performing it.

A large CPU can decode video.

A dedicated video block can often do it while switching far fewer transistors.

That difference becomes electrical power.

Then heat.

Then cooling.

Then battery life.

This principle matters directly to our future hardware thinking.

14. PowerVR, Dreamcast, and Kyro: The Computation That Never Happens

PowerVR is perhaps the strongest historical analogy to what I want to test next.

A traditional renderer may shade surfaces that later turn out to be invisible.

PowerVR's tile-based deferred rendering asks:

Why perform expensive pixel work before establishing that the pixel will survive?

That is computational elegance.

The scene is organized into tiles.

Visibility is resolved.

Hidden work can be rejected.

Expensive processing occurs only for what survives.

The Sega Dreamcast benefited from this philosophy through PowerVR2 graphics.

Kyro and Kyro II later brought the architecture to PC graphics.

The raw specification sheet could therefore be misleading.

One GPU might advertise vastly more theoretical fill rate.

But how much of that fill rate is spent producing pixels nobody ever sees?

This gives us two very different performance metrics:

operations per second

versus:

useful results per operation

The second metric interests me much more.

15. The Elegant Path Loses—Until Physics Brings It Back

Desktop graphics increasingly chose horsepower.

More pipelines.

More memory bandwidth.

More shader units.

More watts.

The strategy succeeded spectacularly.

But PowerVR-style thinking did not disappear.

Mobile computing resurrected it.

A desktop computer can hide enormous inefficiency inside a large power supply and cooling system.

A phone cannot.

Every external memory transaction costs energy.

Every unnecessary calculation becomes heat.

Every wasted pixel drains battery life.

Necessity resurrected architectural efficiency.

That should make us cautious whenever someone dismisses an elegant architecture because brute force currently makes it unnecessary.

The relevant constraint may simply not have arrived yet.

16. Modern Software and the Luxury of Waste

We now possess hardware powerful enough that sloppy computation often remains invisible.

Gigabytes of RAM.

Many CPU cores.

GPU acceleration.

Fast storage.

Cloud scaling.

All of these are extraordinary achievements.

But they allow architectures to accumulate waste.

One subsystem recomputes state.

Another stores a duplicate.

Another serializes it.

Another retrieves information that could already have been excluded.

Another invokes a huge model to decide something that a small deterministic function could have resolved.

None of the individual mistakes destroys the system.

They simply become:

- more RAM;
- more processors;
- more servers;
- more power;
- more cooling;
- more cost.

Eventually enormous hardware is being used to solve a problem whose intrinsic complexity may be much smaller.

That is the kind of waste I want Xerxes SI to resist at the architectural level.

17. Wozniak, Breakout, and the Meaning of a Shortcut

Steve Wozniak's Breakout design is a perfect story for this discussion.

He reportedly reduced the prototype to approximately 45 chips.

That is an extraordinary example of seeing relationships other engineers did not see.

Reuse the same circuitry.

Exploit timing.

Make one component perform more than one role.

Find the shortcut.

But Atari later redesigned the production hardware with significantly more integrated circuits because the extremely minimized design was difficult to manufacture and maintain.

This is enormously important.

It tells us what a shortcut **is not**.

A shortcut is not:

Remove something merely because fewer parts sound impressive.

A valid shortcut is:

Find a shorter path that preserves everything the system actually needs.

The target is:

minimum necessary structure

not:

minimum possible structure

That is exactly how I think about Xerxes SI.

Its efficiencies do not come from indiscriminately removing capabilities.

They come from repeatedly asking whether there is a shorter correct route.

18. Xerxes SI: Many Small Shortcuts Become an Architecture

This is the point where the history becomes directly relevant to Xerxes SI.

Xerxes SI does not achieve its efficiencies through one miraculous discovery.

The architecture contains **many shortcuts that I designed to work with one another**.

Again, the word *shortcut* matters.

These are not corners being cut.

They are situations where a longer computational path can be avoided because another module has already established enough information to use a shorter one.

A simplified example:

```
Input
↓
State already known?
├ yes → reuse it
└ no
  ↓
Can deterministic logic resolve it?
├ yes → resolve locally
└ no
  ↓
Can constraints eliminate possibilities?
├ yes → reduce state
└ no
  ↓
Does an existing skill apply?
├ yes → execute specialized path
└ no
  ↓
Is counterfactual expansion necessary?
├ no → suppress branches
```

```
└ yes → generate bounded alternatives
  ↓
Does the surviving ambiguity require a model?
└ no → finish
└ yes → escalate selectively
```

The important thing is not any one box.

The important thing is that **every box receives less work because of the boxes before it.**

That is why stacked shortcuts are so interesting.

One shortcut saves a little.

The next shortcut operates on what remains.

A third removes more of the remainder.

A fourth recognizes that another expensive stage never needs to wake.

A fifth reuses previously calculated state.

A sixth selects a deterministic specialized path.

A seventh avoids moving information outside the local computational region.

Eventually the system has not merely accelerated an expensive pipeline.

It has prevented much of the expensive pipeline from existing.

That is a fundamentally different design philosophy.

19. Why the Savings Can Compound

Let:

$$W_0$$

represent all candidate work.

Suppose one shortcut removes fraction:

$$r_1$$

of that work.

The remainder is:

$$W_1 = W_0(1 - r_1)$$

Then the second shortcut acts only on the remainder:

$$W_2 = W_1(1 - r_2)$$

Continuing:

$$W_k = W_0 \prod_{i=1}^k (1 - r_i)$$

If the rejection fraction is approximately constant:

$$r_i = r$$

then:

$$W_k = W_0(1 - r)^k$$

That is exponential decay of surviving candidate work with respect to the number of filtering stages.

For a simple illustration, suppose each inexpensive stage eliminates half of what reaches it.

After one:

50 percent remains.

After two:

25 percent.

After three:

12.5 percent.

After four:

6.25 percent.

After ten:

approximately 0.098 percent.

That is why the architecture is interesting.

The shortcut modules do not merely add their savings.

Under suitable conditions they **multiply one another's effect**.

Of course, real systems contain overhead.

The filters cost something.

Some filters overlap.

Some requests cannot be pruned.

Some work must always occur.

Therefore I do not claim that every Xerxes SI workload automatically receives exponential reduction in wall-clock time or electrical consumption.

That would require measurement.

But the mechanism is mathematically real:

Sequential reduction of the remaining candidate space can produce exponential decay in surviving work.

That gives us a concrete benchmarkable hypothesis.

20. Computational Occlusion

PowerVR gives us an excellent term by analogy:

computational occlusion.

In graphics:

hidden pixel → do not shade

In general computation:

state that cannot affect the answer → do not calculate

Suppose we have candidate states:

$$S = \{s_1, s_2, \dots, s_n\}$$

and an expensive operation:

$$f(s)$$

A normal system might evaluate f for each state and reject unwanted answers afterward.

Now introduce a cheap exclusion function:

$$g(s)$$

If:

$$g(s) = \text{impossible}$$

then:

$$f(s)$$

is never executed.

The optimization is worthwhile when:

$$C(g) + P(\text{survive})C(f) < C(f)$$

That equation should govern every shortcut.

No ideology.

No magic.

Did the cheap operation eliminate more cost than it introduced?

If yes, keep it.

If no, remove it.

21. Multidimensional Euclidean Computation

The next research step is more ambitious.

What if a set of calculations can be converted into geometry?

Represent a computational state as:

$$x = (x_1, x_2, \dots, x_N) \in \mathbb{R}^N$$

The dimensions need not represent physical space.

They can represent:

- conditions;
- dependencies;
- constraints;
- probabilities;
- admissibility;
- temporal relationships;
- resources;
- prior state;
- semantic relationships.

Now a rule such as:

$$a_1x_1 + a_2x_2 + \dots + a_Nx_N > b$$

becomes:

Which side of an N-dimensional hyperplane contains the state?

Multiple rules can become:

$$y = Ax + b$$

That can allow many scalar comparisons to become one structured vector or matrix operation.

But the much more important possibility is exclusion.

22. Multidimensional Exclusion Zones

Suppose valid answers can exist only inside:

$$F \subset \mathbb{R}^N$$

Now define regions:

$$E_1, E_2, \dots, E_m$$

such that:

$$E_i \cap F = \emptyset$$

If the state enters one of these exclusion zones:

$$x \in E_i$$

then:

$$x \notin F$$

The downstream computation does not need to occur.

Now make the exclusion hierarchical.

Perhaps a 3D test eliminates 60 percent of candidates.

A 6D test eliminates 70 percent of those remaining.

A 32D test eliminates another 80 percent.

Then an expensive exact solver receives only the tiny unresolved remainder.

This is computational tile-based deferred rendering.

Not literally graphics.

The same principle:

Establish relevance before paying for expensive work.

23. Why the Dimension Can Change

Three dimensions are convenient for us.

They are not necessarily optimal for computation.

A problem may naturally be easier in:

$4D$

$6D$

$32D$

or:

$1000D$

Sometimes increasing dimension makes a nonlinear relationship easier to separate.

A difficult boundary in 3D might become a hyperplane in 32D.

Then, once the decision is made, the system can collapse the representation again.

For example:

```
128D state
↓
32D structured relationship
↓
6D exclusion
↓
3D local calculation
↓
scalar answer
```

Another problem could move upward:

```
4D input
↓
64D lifted representation
↓
simple separation
↓
8D surviving state
↓
answer
```

Dimension becomes another optimization choice.

24. The Representation JIT

This leads to an idea I find particularly interesting.

A normal JIT compiler chooses better machine code after observing runtime behavior.

What if a future system could choose a better **mathematical representation**?

Not simply:

instruction A
→ faster instruction B

but:

scalar problem
→ 6D geometric exclusion

or:

nonlinear 8D problem
→ 64D linearized representation

or:

1000D sparse state
→ 12D active local basis

or:

large symbolic tree
→ 1024-bit hypervector

The runtime would measure whether the transformation paid for itself.

If it did:

reuse it.

If not:

fall back.

This would effectively be a:

Representation JIT

The compiler would optimize not only the instructions used to execute the problem.

It could optimize the language in which the problem is temporarily expressed.

25. One Thousand Dimensions Can Be Cheaper Than One Hundred

Dimension itself does not tell us the cost.

A 100-dimensional dense floating-point representation can be expensive.

A 1,024-dimensional binary representation might use:

- XOR;
- permutation;
- bitwise operations;
- population count.

Modern processors can perform enormous amounts of such work extremely cheaply.

Similarly, a nominally thousand-dimensional sparse representation may have only twelve active components.

The useful cost function therefore looks more like:

$$C = f(N, \text{density}, \text{datatype}, \text{operation}, \text{hardware}, \text{memory movement})$$

The point is not to worship high dimensions.

The point is to choose the representation that creates the shortest correct computational path.

26. Memory Movement May Be the Larger Opportunity

Calculation itself is only part of the problem.

Information movement consumes enormous energy.

RAM to cache.

Cache to processor.

CPU to GPU.

GPU back to CPU.

Service to service.

Model to model.

Node to node.

Every unnecessary representation creates more movement.

Every unnecessary movement becomes latency and heat.

This is another place where Xerxes SI's stacked shortcut philosophy matters.

If one module can keep useful state local and pass only the relevant reduction to the next module, the system can avoid both computation and transport.

This is exactly why tile-local memory is so important in tile-based graphics.

The same insight may matter enormously for general computation.

27. The Same Techniques Can Be Applied to Chip Design

The most exciting extension may be hardware.

If software can determine that an operation is irrelevant, silicon should ideally avoid switching the circuitry that would have executed it.

Dynamic CMOS power is commonly approximated as:

$$P_{\text{dynamic}} \propto \alpha CV^2 f$$

where α represents switching activity.

Reduce unnecessary switching and dynamic power falls.

This gives us the hardware version of a shortcut:

operation proven irrelevant $\rightarrow$ execution hardware never activates

Imagine a processor organized around:

- hierarchical relevance;
- dimensional masks;

- region testing;
- sparse activation;
- local state;
- selective precision;
- deterministic specialized units;
- early exclusion;
- model escalation only when necessary.

Instead of sending every problem through the largest execution mechanism available, the chip could continually ask:

What is the smallest active structure required to resolve this state?

That is the same architectural question Xerxes SI asks at the software level.

Now place it into silicon.

The potential compounding becomes extraordinary.

28. Stacking the Shortcuts in Silicon

Imagine a workload with normalized computational demand:

$$W_0$$

A reuse mechanism removes repeated work.

A routing mechanism removes irrelevant branches.

A geometric mechanism excludes impossible states.

A sparse engine activates only relevant lanes.

A precision engine uses only the accuracy currently necessary.

A local-memory architecture avoids unnecessary data movement.

A dedicated deterministic block prevents a larger general unit from waking.

The effects compound.

Conceptually:

$$W_{\text{survive}} = W_0(1 - r_1)(1 - r_2)(1 - r_3)(1 - r_4) \cdots$$

Now combine that with reductions in switching activity.

Then reductions in memory movement.

Then reductions in cooling requirements.

Then potentially greater sustained clock efficiency within the same thermal envelope.

At that point, we are no longer simply asking:

How many operations per second can the chip perform?

We are asking:

How many operations can the chip prove it does not need to perform?

That may be an equally important measure of future computational power.

29. The Exponential Opportunity

This is where the idea becomes genuinely provocative.

Imagine two processors.

Processor A is capable of performing ten times as many raw operations.

Processor B performs only one tenth as many operations per second.

But Processor B's architecture eliminates 99 percent of the candidate work before expensive execution.

Processor B can win.

Not because its arithmetic units are faster.

Because most of Processor A's extraordinary arithmetic performance was spent on work that Processor B discovered did not matter.

Now stack multiple independent efficiencies.

Better state reuse.

Better routing.

Better exclusion.

Better representation.

Better locality.

Better specialization.

Better gating.

The effective computational advantage can become much larger than any one optimization suggests.

This is the opportunity.

Not claiming an infinite machine.

Not violating computational complexity.

Not pretending all problems can be pruned.

But recognizing that **useful computational power is the combination of what a machine can calculate and what it is intelligent enough not to calculate.**

30. The Opposite of “Just Use a Bigger Model”

This distinction may become especially important in artificial intelligence.

The dominant response to difficult AI problems has often been:

harder problem → larger model → more GPU → more memory → more power

That strategy has achieved astonishing things.

I am interested in the complementary question:

How much of the request can be solved before the large model becomes necessary?

Can memory answer it?

Can a deterministic skill answer it?

Can a previous result be reused?

Can constraints eliminate possibilities?

Can a small specialized module do it?

Can geometry establish an exclusion zone?

Can a compact local representation resolve the ambiguity?

Only then:

Do we need the expensive engine?

That is the philosophy behind many of the shortcuts I have designed into Xerxes SI.

The large tool is not forbidden.

It simply should not be the first tool merely because it is available.

31. Efficiency Is the Stack

This distinction is worth stating plainly.

When somebody looks at Xerxes SI and asks:

How can a system do this with so little memory or processing power?

the answer is not one algorithm.

It is not one neuron.

It is not one optimization.

It is not one clever compression mechanism.

The answer is the **stack**.

Many architectural shortcuts cooperate.

One prevents unnecessary memory reconstruction.

Another limits which pathway can activate.

Another preserves useful state.

Another recognizes when a deterministic solution exists.

Another prevents uncontrolled branching.

Another narrows an unresolved state before handing it onward.

Another avoids using a general-purpose model.

Another prevents the same problem from being solved twice.

Another keeps information local.

Another rejects irrelevant possibilities.

Another uses a different representation.

Each may look modest alone.

Together they change the economics of the entire system.

Xerxes SI's efficiency is not one shortcut. It is an architecture built from shortcuts that reinforce one another.

That is the important point.

And it is why the next stage of research—multidimensional exclusion—is so interesting.

It may become another layer in that stack.

32. The Future Experiment

We should test this rather than merely admire the theory.

Build several implementations of identical workloads.

Conventional Scalar

Optimized ordinary code.

SIMD

A strong conventional vector implementation.

3D Euclidean Exclusion

Use geometric regions to eliminate candidates.

6D and 16D

Test whether additional dimensions expose relationships that make exclusion cheaper.

32D and 64D

Batch related conditions using vectors and matrices.

256D to 1,024D

Test sparse and hyperdimensional representations.

Adaptive Representation

Let the runtime select dimension and representation according to measured cost.

Then measure:

- latency;
- operations executed;
- candidates eliminated;
- bytes moved;
- cache traffic;
- accelerator utilization;
- power;
- heat;
- transformation overhead;
- numerical error;
- false rejection;
- fallback frequency.

Do not construct a weak conventional baseline.

If normal code wins, normal code wins.

If geometry loses, remove it.

If the representation takes more time to construct than it saves, discard it.

That is how an architecture earns the right to survive.

33. The Historical Pattern Becomes Clear

Look again.

ViRGE: features are not enough.

Savage/S3TC: representation can remove bandwidth.

MeTaL: specialization can work, but only when correctness survives.

G400/EMBM: do not model complexity literally when a cheaper representation produces the required result.

BitBoys: ideas can survive companies and products.

Glide: specialization can shorten the software-to-hardware path.

Voodoo2 SLI: parallelize useful work, not waste.

GeForce: brute-force general computation can win spectacularly.

Creative Unified: translate instead of preserving obsolete machinery.

UltraHLE: preserve behavior at a higher level instead of emulating every mechanism.

Rosetta 2: translation can preserve an entire software ecosystem.

Cocoa: stable abstractions allow implementation to change beneath them.

Java and ART: wait until runtime provides more information before choosing the execution form.

Python JITs: the language you write is not necessarily the machine that executes.

Hollywood Plus: specialization disappears when general hardware becomes fast enough and reappears when power becomes important.

PowerVR, Dreamcast, and Kyro: do not shade what will never be seen.

Mobile GPUs: physics eventually punishes waste.

Wozniak's Breakout: find the shortcut, but never eliminate necessary structure.

And now:

Xerxes SI: stack many correct shortcuts so every stage receives less unnecessary work than the one before it.

These are not isolated stories.

They share one engineering instinct:

Understand the problem well enough to know which parts of the apparent computation are not actually part of the necessary computation.

34. What I Want Xerxes SI to Ask

The graphics industry asked:

How many pixels can we shade?

PowerVR asked:

Which pixels should never be shaded?

The emulation community asked:

How quickly can we reproduce the original machine?

UltraHLE asked:

Which parts of the original machine do we not need to reproduce?

Compiler engineers asked:

How should this program be compiled?

JIT engineers asked:

Can we wait until we know how the program actually behaves?

Modern AI asks:

How large a model can we run?

I want Xerxes SI to ask:

How much of the problem can disappear before the large computation begins?

Then:

What has already been learned?

Then:

What is impossible?

Then:

What can be solved deterministically?

Then:

What representation makes the remaining problem cheapest?

Then:

What dimension makes its boundaries simplest?

Then:

What is the smallest computational mechanism that can safely resolve the remainder?

And finally:

Can the same logic be designed into silicon so unnecessary computation never becomes switching activity in the first place?

That is the direction.

35. The Work We Never Needed to Do

Computing has become extraordinarily good at answering:

How fast can we perform this operation?

We should become equally good at asking:

Why are we performing it?

If it must be calculated, calculate it brilliantly.

If it has already been calculated, reuse it.

If a deterministic path exists, take it.

If a branch is impossible, eliminate it.

If many operations can collapse into one representation, collapse them.

If translation preserves the meaning, do not reproduce obsolete machinery.

If an exclusion zone proves a candidate irrelevant, do not evaluate it.

If a higher-dimensional representation makes the relationship simple, lift the state.

If a lower-dimensional projection preserves everything required, collapse it.

If uncertainty remains, increase precision.

If a large model is genuinely required, use it.

If it is not required, never wake it.

If the processor knows a result cannot matter, do not toggle the gates that would have calculated it.

That is what a shortcut means to me.

Not cutting corners.

Not sacrificing correctness.

Not hiding complexity.

A shortcut is the discovery that a shorter valid path exists.

Xerxes SI contains many such shortcuts.

The efficiency is created by their cooperation.

A shortcut eliminates work.

The next shortcut receives less work.

Another reduces the remainder.

Another avoids a memory transfer.

Another suppresses a branch.

Another reuses state.

Another selects a specialized pathway.

Another prevents an expensive model invocation.

And if future multidimensional exclusion can remove entire regions of computational possibility before evaluation, that becomes one more layer in the stack.

Eventually the result may be something much more important than a faster algorithm.

It may be a system whose architecture continually asks:

What is the least computation necessary to know this?

And if the same philosophy can be applied directly to processor design, the consequences extend beyond speed.

Less active computation.

Less memory movement.

Less switching.

Less electrical demand.

Less heat.

Less cooling.

More sustained useful computation inside the same physical envelope.

That is where the idea becomes truly interesting.

The future of computational performance may not belong only to whoever can build the machine capable of performing the most operations.

It may also belong to whoever builds the machine that is best at determining:

which operations never needed to happen

That is the work we never needed to do.

And finding it may be one of the most powerful shortcuts left.

— The Founder

Research Note

The historical examples in this essay illustrate documented architectural approaches. The proposed Xerxes SI extensions involving stacked computational shortcuts, multidimensional Euclidean exclusion, adaptive dimensional representations, representation-level JIT optimization, and geometry-aware relevance hardware are forward research directions.

Sequential filtering can mathematically produce multiplicative—and under approximately constant rejection ratios, exponential—decay in surviving candidate work. Actual reductions in latency, energy consumption, thermal output, or required hardware depend on filter overhead, workload structure, memory movement, fixed system costs, and implementation quality and therefore require controlled benchmarking.

The purpose of the proposed experiments is to establish exactly where the shortcuts outperform conventional optimized computation, where they do not, and whether enough individually modest efficiencies can be composed into a materially different computational architecture.