Forging a Future in Silicon

Hardware-Backed Dynamic Memory, Concurrency, and a Reinvented Stack Model

As our software demands intensify—driven by data-centric workloads, scalable web services, and sophisticated high-level languages—developers face persistent performance and memory bottlenecks. A key source of these challenges is the disconnect between the rigid assumptions of traditional hardware and the dynamic, concurrent nature of modern applications. Today's processors are still rooted in fixed-size operations and synchronous memory models, while modern software thrives on flexible data structures, arbitrary-size integers, and parallel execution strategies. The friction between these domains leads to unnecessary complexity and overhead in runtimes, compilers, and libraries.

This essay argues for a more profound architectural shift: integrating extensive hardware support for dynamic memory and concurrency directly into the processor and memory subsystems. We'll examine how this could transform the handling of arbitrary-precision ("bignum") arithmetic and dynamic arrays—two prime examples of features that require flexibility and often suffer from software-only solutions. We'll also consider how a hardware-assisted call/return model, using a stack-processor approach, could complement these advances. By treating dynamic data and concurrent execution as first-class citizens in silicon, we can streamline computation, reduce overhead, and create a more scalable platform for tomorrow's applications.

Elevating Dynamic Memory Management to Hardware

Today's processors assume that memory is a stable, static resource: pages are allocated, data is stored, and loads/stores follow predictable paths. In practice, modern applications use data structures that grow, shrink, and morph over time—think of dynamic arrays, lists, trees, and variable-sized buffers. Managing these structures efficiently demands constant memory (re)allocation and resizing. Currently, this occurs entirely in software, requiring complex allocators, copy operations, and careful synchronisation to avoid fragmentation or race conditions.

By adding hardware-level support, the processor could directly manage these dynamic operations. A specialised memory controller or instructions could handle array expansion, contraction, and compaction natively, much like today's memory management units support virtual memory. Instead of pausing threads or locking regions of memory, the hardware could transparently adjust memory layouts behind the scenes. Threads working concurrently on the same dynamic array would face fewer bottlenecks, as the hardware could arbitrate modifications swiftly and in parallel.

This approach would bolster higher-level abstractions. Consider map, filter, and reduce operations—common in functional programming, data analytics, and parallel computing. Today, applying these higher-order functions over large, growable arrays involves a delicate interplay of memory allocation, synchronisation, and iteration at the software level. With hardware handling dynamic memory, these operations could run as if on a fixed-size structure: the processor would "just know" how to expand or contract arrays while distributing work across multiple cores or pipelines, improving performance and scalability.

Bignums as Beneficiaries of Hardware Flexibility

Bignums, or arbitrary-precision integers, perfectly exemplify where hardware-assisted dynamism and concurrency can shine. Cryptography, scientific computing, and large-scale analytics depend on integers too large for standard 64-bit registers. Current solutions rely on software libraries that implement large integers as arrays of smaller words, looping through them to perform arithmetic. This is inherently slow, complex, and challenging to parallelise effectively.

A hardware-level solution might provide variable-length integer registers or arithmetic units that scale with the size of the numbers, performing multi-word additions and multiplications internally. Instead of orchestrating bignum arithmetic through intricate software loops, the processor could offer instructions that handle carry propagation, partial products, and even modular reductions natively. With concurrency in mind, multiple arithmetic units could split a large number into segments, processing them in parallel. This would bring bignum arithmetic closer to the speed and simplicity of fixed-size operations—vastly improving performance and reducing complexity.

Concurrent Garbage Collection in Hardware

Memory management doesn't stop at allocation and resizing. Managed languages rely on garbage collection (GC) to automatically free unused memory. Techniques like concurrent generational GC allow the collector to run alongside application threads, focusing on young objects frequently and old objects less so. Yet even concurrent GC requires complex software machinery, frequent write barriers, and can still cause unpredictable pauses.

By integrating GC support into hardware, we can streamline the process. Memory controllers and caches could tag objects with generation or reachability information. On-store operations could trigger automatic, hardware-level write barriers that record changes for the GC's benefit—no software instrumentation required. A dedicated GC coprocessor could continually trace object graphs, update references, and reclaim space without pausing the application. Such hardware assistance would smooth out performance spikes, reduce runtime overhead, and give managed languages a more stable performance profile.

Reimagining the Call/Return Mechanism with a Stack Processor

While dynamic memory and concurrency take center stage in this vision, the synergy can be extended by rethinking the call/return stack mechanism. Today's calling conventions require saving and restoring registers on the stack to preserve state across function calls. This process is both a source of overhead and a subtle interplay between compiler logic, runtime routines, and hardware assumptions.

If we factor the prime functionality of call/return from the need to explicitly manage register state, we can embrace a hardware-managed data stack—essentially a stack-processor model. In such an architecture, "registers" become entries on a hardware-managed data stack, growing and shrinking as functions are called and returned. The overhead of saving and restoring registers disappears into the hardware itself, allowing faster, more fluid call sequences.

This approach dovetails perfectly with the dynamic memory and concurrency model. The hardware-managed data stack can be treated as another dynamic resource, potentially scaling in size and even being split or reallocated transparently. Threads or coroutines could each have their own hardware-managed stacks, reducing context-switch overhead. GC could more easily trace stack references if the hardware provides clear boundaries and metadata. Bignum arithmetic and array transformations—often relying on temporary values and intermediate states—could push and pop these values onto the stack without going through a fixed register file and complicated spill/fill sequences.

A Unified Vision: Simplicity, Concurrency, and Scale

By merging hardware-assisted dynamic memory, concurrency support, bignum acceleration, and stack-based execution, we create a unified architecture attuned to today's software reality. Rather than using fixed-size registers and memory pages as fundamental assumptions, the hardware itself acknowledges that data sizes, usage patterns, and lifetimes vary dramatically. It provides the tools to handle this dynamism gracefully.

Language runtimes become simpler because the hardware handles much of the low-level plumbing. Compilers need not contort themselves to generate elaborate bignum arithmetic sequences or memory allocation idioms. Garbage collectors no longer bring everything to a halt. Function calls, once an expensive dance of saving and restoring registers, become fluid and efficient.

In parallel computing contexts, where multiple threads traverse large, evolving data structures or operate on massive numbers, hardware-level dynamic memory and concurrency features reduce contention and overhead. By scaling resources and distributing work natively at the silicon level, the system can achieve better throughput and lower latency. The data stack model and hardware GC each reinforce this scalability, smoothing inter-thread communication and memory reclamation.

Looking Ahead: From Rigid to Fluid Architectures

This vision is not without challenges. Adding dynamic memory instructions, complex GC logic, or a stack-based execution model to silicon consumes area and design time. The architecture must be sufficiently flexible and general to accommodate evolving runtime strategies and language semantics. Developers and hardware vendors must reach consensus on how these features are exposed and standardised.

Yet the trend in computing is clear: our workloads demand ever more flexibility, concurrency, and higher-level abstractions. Just as floating-point units and vector instructions once addressed pressing software needs, tomorrow's processors must consider dynamic memory management, large-scale integer arithmetic, concurrency, and function calls as first-class concerns. By engineering architectures that take these demands seriously—integrating memory resizing, GC support, parallel bignum arithmetic, and stack-based calling directly in hardware—we create a more stable, high-performance foundation for the next generation of applications.

In the end, this represents a fundamental shift in how we think about hardware and software. Instead of forcing complexity into compilers and runtime frameworks, we absorb it into the processor and memory subsystems, where it can be handled more efficiently. The result is a computing environment where advanced language features no longer feel like a costly indulgence, but a natural, hardware-supported aspect of everyday programming. In short, we move from rigid architectural assumptions to a more fluid, flexible, and scalable future—one built in silicon and ready for the dynamic, concurrent demands of modern computation.

Gordon Charlton, Dec 2024
with assistance from Chat-GPT

Appendix: Recommended Reading

Computer Architecture & Dynamic Data Structures

- Hennessy, John L., and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.
- Sohi, Gurindar S. ‘‘Instruction-Level Parallelism: The Road Ahead.’’ *Proceedings of the IEEE*, vol. 89, no. 11, 2001.
- Kozyrakis, Christos et al. ‘‘Breaking the Abstraction Barrier: How Hardware Reshapes High-Level Programming.’’ (Explores the interplay between hardware design and high-level programming abstractions.)
- Wolfe, Michael. *High-Performance Compilers for Parallel Computing*. (Explores compiler optimisations for dynamic and parallel data structures.)
- Vitek, Jan et al. ‘‘Dynamic Memory Management in Embedded Systems.’’ (Focuses on integrating dynamic memory strategies into constrained environments.)

Garbage Collection and Hardware Support

- Azul Systems Pauseless GC overview: Click, Cliff, et al. ‘‘The Pauseless GC Algorithm.’’ *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE’05)*, 2005.
- Hudson, Richard L., and J. Eliot B. Moss. ‘‘Incremental Collection of Mature Objects.’’ *IWMM ’92: Proceedings of the International Workshop on Memory Management*, 1992.
- Jones, Richard, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall, 2011.
- Mutlu, Onur et al. ‘‘Hardware Acceleration for Garbage Collection.’’ (Explores hardware-level strategies for efficient GC in managed languages.)
- Click, Cliff. ‘‘Why Hardware Garbage Collection?’’ (Discusses the feasibility and benefits of integrating GC mechanisms into silicon.)

Bignum Arithmetic and Parallelisation

- Granlund, Torbjörn, and Peter L. Montgomery. ‘‘Division by Invariant Integers Using Multiplication.’’ *SIGPLAN Notices, vol. 29, no. 6, 1994*. (Relates to efficient integer arithmetic techniques.)
- Bernstein, Daniel J. ‘‘Multidigit Multiplication for Mathematicians.’’ (Details algorithmic approaches that could inspire hardware acceleration.)
- Brodersen, Bob et al. ‘‘The Case for Application-Specific Instruction Set Processors.’’ (Discusses tailored hardware solutions for specific computational needs, such as bignums.)
- Herlihy, Maurice, and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. (Background on parallelisation concepts crucial for efficient bignum processing.)

Stack Machines and Alternative Architectures

- Koopman, Philip. *Stack Computers: The New Wave*. Ellis Horwood, 1989. (A deep exploration of stack-based architectures.)
- Henderson, Peter, and James H. Morris. *"A Lazy Evaluator." Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages (POPL '76)*, 1976. (Background on functional evaluations that benefit from stack models.)
- Tomasulo, et al. "A Stack Machine Architecture for Dynamic Language Support." (Proposes optimisations for dynamic language execution using stack processors.)
- Shen, John P., and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. (Discusses stack models and their interplay with modern processor designs.)

Concurrent and Parallel Models

- Herlihy, Maurice, and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. (Background on concurrency principles that can guide hardware-level concurrency features.)
- Olukotun, Kunle, et al. "The Case for a Single-Chip Multiprocessor." *ASPLOS '96: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996. (Foundational thinking in parallelism that may inform hardware concurrency support.)
- Kozyrakis, Christos et al. "Hardware Acceleration of Parallel and Dynamic Workloads." (Insights into how hardware can alleviate concurrency bottlenecks.)
- Patterson, David, and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann. (Covers foundational concepts relevant to concurrency at the hardware-software boundary.)

Lisp Machines and Symbolic Processors

- Moon, David A. "Architecture of the Symbolics 3600." *Proceedings of the 12th Annual International Symposium on Computer Architecture (ISCA '85)*, 1985.
- Gabriel, Richard P., and Jon L. White. "CLOS in Context: The Shape of the Design Space." *OOPSLA '91*. (Provides insight into dynamic languages and their memory needs, historically addressed by Lisp machines.)
- Stallman, Richard, and Guy L. Steele. "Forwarding Garbage Collection on the Lisp Machine." (A study in hardware-assisted garbage collection in symbolic processors.)
- Jacob, Bruce et al. *Memory Systems: Cache, DRAM, and Beyond*. (Includes discussions of memory innovations that inform GC and dynamic memory handling in hardware.)