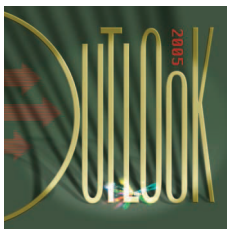
Self-Assembled Architectures and the Temporal Aspects of Computing



Self-assembly and fabrication-time programmability introduce a new perspective on computation that challenges hardware and software developers to reconsider the temporal aspects of computing. For some applications, combining fabrication-time and runtime computation can significantly improve performance.

Chris Dwyer
Alvin R. Lebeck
Daniel J. Sorin
Duke University

The conventional view of system design incorporates layers of abstractions including the software, architecture, logical implementation, and physical realization levels. System designers can use the interface between each of these levels to compartmentalize design tasks. For example, a software programmer does not need to know anything about the system's implementation or realization levels. Software written for a given architecture will operate correctly for multiple implementations of that architecture.

Despite the convenience of clean abstractions, technological trends are blurring the lines between design layers and creating new interactions between previously unrelated layers. One example is virtual machines such as VMware and Transmeta that implement the application-software-visible architecture (virtual architecture) in VM software, allowing more flexibility in the hardware/software interface beneath the VM layer. Designers can change the VM-software-visible architecture (physical architecture) in response to implementation or technology constraints without affecting application software that was written for the virtual architecture.

Technological trends are forcing a closer coupling of architecture to implementation and even realization. One such trend is the increasing delay for long on-chip wires relative to other circuit delays, which are decreasing. New architecture designs explicitly consider this low-level implementation issue. For example, the designs of both the instruction-level distributed processor (ILDP)[1] architecture and the recently proposed nonuniform cache architecture (NUCA)[2] minimize the number of long wires required in an implementation. Power is another issue that permeates all levels of system design, increasing the coupling between architecture, implementation, and realization.[3]

Even more interaction between the hardware levels exists in emerging nanocomputing systems. Numerous programmable logic array-like (PLA) computers have been proposed recently, including NanoFabrics[4] and array-based architectures.[5] The physical architectures of these systems are not separable from their crossbar-like physical realizations. Similarly, the nanoscale active network architecture (NANA)[6] is intimately tied to its physical realization. Because NANA is fabricated with DNA self-assembly, the interconnections of its computational nodes are random. This randomly interconnected computational substrate necessarily impacts the physical architecture.

Future technologies are likely to further increase the interactions between design layers. Programmable self-assembly is an emerging fabrication tech-

Published by the IEEE Computer Society

nology that must be considered in the higher layers of the computer system design.

The characteristics of self-assembled systems require architects to explicitly consider the fabrication process. Programmable self-assembly offers an opportunity to perform computation during the fabrication process itself. Before the advent of self-assembly, computation was performed either prefabrication (during design) or postfabrication (at compile time or runtime).

SELF-ASSEMBLED SYSTEMS

In the traditional approach to designing and fabricating computer systems, designers use a top-down scheme to specify exactly where every component should be placed, then the manufacturer fabricates the system according to these specifications. This intuitive top-down approach conforms with how we expect to design systems, and it is how all current commercial computer systems have been developed. However, the semiconductor industry has identified the difficulty of continuing to shrink the feature sizes for mass-produced electronic components,[7] and this impending roadblock has spurred research in bottom-up self-assembly approaches.

With bottom-up self-assembly, the designer specifies the components but cannot stipulate exactly where each one will be placed. Rather, the components must self-assemble to form a system. The simplest form of self-assembly is random; however, this approach is limited in what it can create. Since no order is imposed on the fabrication, the designer can introduce little complexity.

An alternative to random self-assembly is programmable self-assembly, in which the designer specifies how the components attach to one another, but not where any particular component will be placed. With programmable self-assembly, the designer has some control over the fabricated system and can thus create more sophisticated computers.

The predominant approach to nanoscale programmable self-assembly exploits DNA's ability to self-assemble. DNA's well-known double-helix structure is formed through its well-understood base-pairing rules—adenine (A) to thymine (T) and cytosine (C) to guanine (G). By specifying a particular sequence of bases on a single strand of DNA, we can exploit the base-pair rules as organizational instructions.

We can use unpaired, single-strand regions (ssDNA) extending from the ends of one nanoscale object to specifically bind to another object displaying the complementary ssDNA. Therefore, a region of ssDNA and its complementary ssDNA act as tags (Tag and Tag′) for orienting the two objects in three-space. These tags—sometimes called sticky ends due to their glue-like binding behavior—are a key feature of DNA self-assembly techniques. Moreover, appending DNA to nanoscale objects can act as "smart glue" for organizing those objects in space.[8] Thus, researchers can use DNA to form a "scaffolding" for nanoelectronic integration.[9-15]

If we encode a computational problem's inputs with appropriately chosen base-pair sequences, the DNA will combine to form the solution. Researchers have used DNA computation to solve several computational problems, including the directed Hamiltonian path problem.[16]

Programmable self-assembly offers an opportunity to perform computation during the fabrication process itself.

THE TEMPORAL ASPECTS OF COMPUTING

Computation can be performed at three different stages in the computer fabrication process:

- *Prefabrication*. Computation can be performed during the design process that precedes fabrication of the computer. For example, an application-specific integrated circuit (ASIC) contains components that the developers have preconfigured to quickly perform specific computations such as fast Fourier transforms.
- *Postfabrication*. The most traditional time for performing computation is after the computer has been fabricated. General-purpose processors are a good example of this temporal aspect.
- *At-fabrication*. Programmable self-assembly of computer systems provides a new temporal opportunity for computation. The self-assembly process itself performs useful computation during fabrication of a computer system.

These three stages comprise a temporal spectrum of computation and create a new design point for computer systems.

Prefabrication computation

Prior to fabrication, the developers can plan ahead to specify how the computer system will perform specific computations in the future. A simple example is the use of a lookup table (ROM) that stores solutions to computations the computer is expected to perform.

More complex examples are ASICs and field-programmable gate arrays (FPGAs), which inherently compute portions of their application space before fabrication—that is, they incorporate appli-

At-Fabrication Computation Requirements

An example helps to visualize the volume required for at-fabrication computation, which is on the order of a few cubic meters.

Assume we want to perform additions for 1,000 years on a 10-GHz processor. Given 3.15×10^7 seconds in a year, this yields 31.5×10^{16} additions per year, which is 3.15×10^{20} additions in a millennium.

Now, we need to map this to some amount of work that we could perform with at-fabrication computation, so let's take the Oracle example in which we compute the sum of two n-bit numbers at fabrication for all possible combinations of n-bit values (2^{2n}). We use the total computed additions to determine what value n should be. The answer is the square root of (3.15×10^{20}), which is 1.77×10^{10}. Log base 2 of this is approximately 34, so it will take a millennium to compute all sums of all combinations of two 34-bit numbers.

For the material to accomplish this with our DNA at-fabrication approach, we assume that we can assemble a single addition operation for two specific 34-bit numbers in a DNA structure (lattice) that is 800 nm on each side and 4 nm thick. This assumption is based on experimental DNA self-assembly results[1] and our nanoscale design method.[2,3] The volume for each lattice is $(800 \times 10^{-9}\ \text{m} \times 800 \times 10^{-9}\ \text{m} \times 4 \times 10^{-9}\ \text{m}) \approx 2.56 \times 10^{-21}$ cubic meters. To construct all additions, we need one of these lattices for each combination of 34-bit values, so we need a total of 3.15×10^{20} lattices to perform the same number of additions. If we pack the lattice optimally, we end up with about 0.8064 cubic meters (approximately one cubic meter). With 1,000 liters per cubic meter, this ends up as 806 liters (~200 gallons) of material. Of course, this is a lower bound on the volume of raw material. We need more volume to allow for cooling, I/O, interconnect, power, and so on.

Determining any other computation would require normalizing the work—and DNA lattice area—to that of the add operation. This would either increase or decrease the volume based on what additional area is needed on each DNA lattice to implement the operation.

References

1. H. Yan et al., "DNA Templated Self-Assembly of Protein Arrays and Highly Conductive Nanowires," *Science*, vol. 301, 2003, pp. 1882-1884.
2. J.P. Patwardhan et al., "Circuit and System Architecture for DNA-Guided Self-Assembly of Nanoelectronics," *Foundations of Nanoscience: Self-Assembled Architectures and Devices*, 2004, pp. 344-358.
3. C. Dwyer et al., "Design Tools for a Self-Assembling Nanoscale Technology," *Nanotechnology*, vol. 15, 2004, pp. 1240-1245.

cation-specific optimizations. Defect-tolerant design—for example, systems like the Teramac that use redundancy[17]—is also a form of prefabrication computation with the goal of avoiding fabrication-time errors. Even general-purpose computers have some amount of hardware that is tailored to perform specific computations—for example, floating-point arithmetic.

Prefabrication computation is well suited to many types of computations, since designers often know ahead of time what computations a computer is likely to perform, or at least those computations it will perform frequently. However, prefabrication computation is resource-intensive both in terms of design time and area.

Prefabrication computation involves more complex designs, and this added complexity leads to longer design times and more costly fabrication. For example, adding hardware to perform floating-point division is a form of prefabrication computation that complicates the system design and requires more chip area compared to having software synthesize floating-point division from more primitive hardware instructions. The obvious benefit is that the floating-point operations execute faster with the dedicated hardware than when using the software-synthesized operations.

Postfabrication computation

Postfabrication computation is the conventional execution of a sequence of instructions—either micro- or macrocode—at the moment of interest. Although we consider fabrication from the hardware perspective, it is possible to perform an analogous exploration of software. For example, we could consider software design to be prefabrication, compilation to be at-fabrication, and runtime execution to be postfabrication. From the hardware perspective, both compile time and runtime are postfabrication.

Forms of postfabrication computation include compilation, program transformations, static scheduling, and numerical precomputation. General-purpose processors such as the Intel Pentium 4 perform the bulk of their computation postfabrication. Even ASICs and FPGAs perform some postfabrication computation that complements the computation they perform prefabrication. For the purposes of this discussion, we consider general reconfigurable computing as having two components:

- a prefabrication step to create the mechanisms that enable reconfiguration, and
- a postfabrication phase that can include a single configuration then execution or, alternately, can configure, execute, reconfigure, and execute.

Postfabrication computation has the most flexibility, and it is the most general-purpose form of computation. Nevertheless, the system design can constrain postfabrication computation. For example, graphics cards include specialized program-

mable processors tailored to operate on pixels or vertices. Although they can sometimes be used for more general-purpose computation,[18] their performance on arbitrary programs is likely lower than on a general-purpose microprocessor.

One potential drawback of postfabrication computation is that execution time and power consumption are strongly dependent on intuition and knowledge of the computational problem (as opposed to the definition) and its implementation. For example, matrix multiplication has many different implementations that can yield dramatically different performance results based on cache memory behavior.

At-fabrication computation

Most current computers exploit pre- and postfabrication computation, but the advent of programmable self-assembly creates the opportunity to perform computation during the fabrication of a computer. This at-fabrication computation occurs by executing a well-defined rule set during self-assembly. The computation presolves a problem space—for example, a Hamiltonian path—in preparation for a simpler postfabrication computation. Current fabrication techniques lack these capabilities. Instead, developers must fully specify solutions to the problem space at design time (prefabrication) or compute it postfabrication.

DNA computing is one example of at-fabrication computation, with the base pairs determining the rule set. During fabrication (annealing), all possible paths are fabricated (for the Hamiltonian path DNA computation[16]), and a post-fabrication step searches for and extracts the correct solution.

Another example system that exploits at-fabrication computation is the proposed Oracle computer.[15] In this system, each self-assembled node is the solution to a predefined computational problem. For example, an Oracle for *n*-bit addition would self-assemble nodes that represent each of the possible *n*-bit additions the system can perform as well as their solutions. An Oracle computer performs virtually all of its computation at-fabrication. The only postfabrication computation it performs is a lookup to find the appropriate solution node. An Oracle is similar to an at-fabrication version of a prefabrication ROM lookup table. Different Oracles can be developed for specific problems—for example, a Hamiltonian path. In contrast to the Oracle approach, the decoupled array multiprocessor (DAMP)[15] relies more on postfabrication computation, using at-fabrication computation only to generate a large set of random numbers.

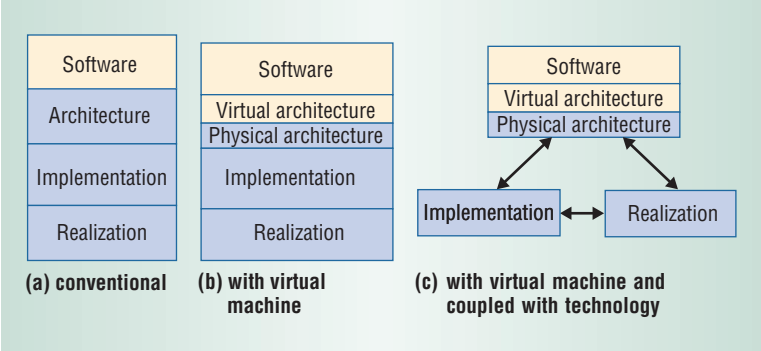


Figure 1. Layers of abstraction in system design: (a) conventional, (b) virtual machine, and (c) virtual machine coupled with technology. Shaded portions are hardware.

Programmable self-assembly enables at-fabrication computation on an unprecedented scale. With its scaling capabilities, this form of computation can reduce algorithmic complexity by orders of magnitude, thus reducing execution time and energy consumption. However, the tradeoff is that at-fabrication computation is very resource-intensive, at least with the current state of self-assembly technology. The "At-Fabrication Computation Requirements" sidebar provides additional information about the scale of this form of computation.

Extending the temporal spectrum

Most computers exploit or combine multiple points on the spectrum. Even general-purpose systems exploit arithmetic algorithms at design time to improve execution time, for example, with multiply unit, square root, ROM lookup table, and so on, and ASICs generally still have a postfabrication computation component.

Extending the range of the temporal spectrum to include at-fabrication computation introduces a new point at which computing can occur. It is now possible to couple at-fabrication and postfabrication computation as in the Oracle computer. Exploring the tradeoffs between these aspects of computation involves determining which computational tasks should be performed when. Architects will base these decisions on the relative cost—including design and fabrication time, performance, and power.

Similar to the recent shifts in architectural decisions that place greater significance on technology capabilities, future programmable self-assembled architectures will place greater significance on fabrication capabilities. The simple layered system design must change to preserve the advantages of abstractions while avoiding the introduction of rigid boundaries between the architecture, implementation, and realization/fabrication levels. Without these changes—illustrated in the progression from Figure 1a to Figure 1c—architectures may not be able to exploit the full capabilities of emerging fabrication techniques.

To exploit the ability to perform computation during the fabrication process, architects must partition computational tasks into three temporal

Table 1. Comparison of systems for solving the block edit problem.

System	Fabrication material	Postfabrication complexity
All postfabrication	None	$O[(n!)^2]$
At-fabrication block partitioning	$O(n!)$	$n! \times n^2 \rightarrow O(n!)$
At-fabrication block partitioning plus at-fabrication block reordering	$O[(n!)^2]$	$O(n^2)$

aspects. Shifts can take place from prefabrication to at-fabrication or from postfabrication to at-fabrication.

The shift from prefabrication to at-fabrication is likely to occur due to at-fabrication's advantages in terms of scaling. For example, the choice between performing a computation using a prefabricated read-only memory design or at-fabrication creation may favor the Oracle for larger problem sizes.

Programmable self-assembly's ability to efficiently preperform brute-force computations will motivate the shift from postfabrication to at-fabrication. For example, the architecture could incorporate an Oracle for a specific computation into a computer that is designed to solve that particular computation on a regular basis. The one-time cost of this at-fabrication computation would be amortized over the computer's postfabrication lifetime.

At-fabrication computation has an advantage over postfabrication computation for problems that require brute force because DNA self-assembly has the potential to perform on the order of 10^{12} to 10^{19} computations in parallel after accounting for yield and interconnect overheads.

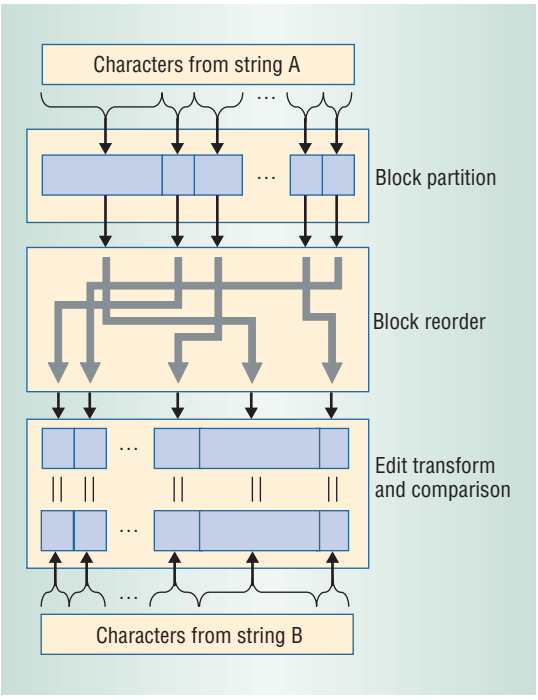


Figure 2. Overview of the brute-force solution to the block edit problem. This solution holds string B fixed while searching for a transformation of string A to string B.

AN EXAMPLE: SOLVING THE BLOCK EDIT PROBLEM

The block edit problem is found in several application domains including molecular biology and functional genomics, pen-based computing, speech recognition, and natural-language processing. Briefly, the block edit problem involves selecting blocks of characters from one string (A), reordering the blocks, and then comparing them to characters from a second string (B). The optimal solution minimizes the number of edits required to transform each block of characters in A to the corresponding characters in B after the edit process. Variations of the problem specify whether the blocks must cover all characters from either or both strings and whether the transformation can be made with disjoint use of the blocks or not.

Some variations of the problem such as noncovering and nondisjoint can be solved in polynomial time. Unfortunately, many of the interesting varieties of the block edit problem are NP-complete.[19]

An NP-complete block edit problem can be solved optimally for a fixed input size—larger than is feasible with present-day computers—using post- and at-fabrication time computation. Although approximation algorithms may exist for finding suboptimal solutions, we are interested in finding the optimal solution. Our systems mix the temporal aspects of computing to achieve a range of postfabrication time performance at the expense of increased manufacturing costs.

Table 1 summarizes three systems for solving the block edit problem, distinguishing between these systems based on how they divide the computational work between at-fabrication and postfabrication.

Purely postfabrication computation

A purely postfabrication optimal solution to the NP-complete block edit problem—cover and disjoint to cover and disjoint—naïvely requires $O[(N_A!)^2]$ running time, where N_A is the length of the input string. Figure 2 shows the brute-force approach.

Our method holds string B fixed while searching for a transformation of string A to string B. Since there could be as many as N_A single-character blocks, there are $O(N_A!)$ ways to partition string A. Each partition of blocks must be evaluated under all block arrangements, of which there are $O(N_A!)$ since in the worst case there could be N_A blocks.

The final step is to perform a per-block edit transform for each block partition and arrangement, which is $O(N_A \times N_B)$ running time with the use of

a linear program. Combining these terms shows that in the worst case the brute-force approach takes $O[(N_A!)^2 \times N_A \times N_B]$ or approximately $O[(N_A!)^2]$ to find an optimal block edit transformation from string A to string B.

The astounding complexity of this problem means that, for a simple 15-word sentence (replacing characters with words maintains the constraint that the transformations do not destroy the sentence's lexicon), it would take the better part of a millennium to find the optimal transformation to another sentence if each step takes only 10 femtoseconds (100 THz evaluation rate).

At-fabrication components of optimal block edit solutions

There are many alternatives for exploiting at-fabrication computation to solve the block edit problem. The only way to optimally solve this problem is to perform an exhaustive search over the block partition space and the block reordering space. Here we focus on at-fabrication computation of block partitions and block reordering. The tradeoff that architects must reason about is between the amount of material required at-fabrication and the amount of postfabrication computation.

At-fabrication block partitioning. Block partitioning is the process of dividing the input string (A) into all possible substrings, where each substring is called a block. At-fabrication block partitioning is supported by using tiles, where each tile is the realization of one specific block. A tile can realize the block by either fabricating the appropriate substring directly during self-assembly or by incorporating electronics that identify and store the appropriate substring when the string A is first input to the system. The "DNA Self-Assembly and Nanoelectronics" sidebar provides additional information about how tiles are formed using DNA.

Figure 3 illustrates a few tiles from the $O(N_A^2)$ set of tiles that partition the source string (A). Each tile represents a start and end position in A. The figure shows the tiles used to create all substrings that start with the first character in A up to the fourth character in A.

A circuit assembly forms a string partition by concatenating the tiles in the original string. To achieve this, the tiles specify the self-assembly rules such that a tile (T_1) will only bind to another tile

DNA Self-Assembly and Nanoelectronics

The particular set of nanotechnologies we are exploring uses carbon nanotube field-effect transistors (CNFETs) that are self-assembled using DNA as a scaffold. To fabricate a circuit, strands of DNA are attached to the carbon nanotubes[1,2] and used to connect the ends of the nanotube to the scaffold if and only if the base pairs on the strands are complementary. The circuit's function is determined by specifying the type of nanotube (semiconducting or metallic) and placement on the scaffold. Figure A shows an example circuit that implements a NAND gate.The dashed and solid lines are metallic nanotubes below and above the DNA scaffold, respectively. The CNFETs are formed within the grid-shaped DNA's (bulbous outline) "cavities." Metallization techniques can then be used to form electrical contact between components on the scaffold to complete the circuit.

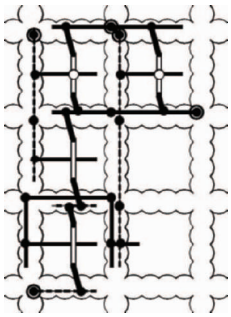


Figure A. NAND gate layout. The dashed and solid lines are metallic nanotubes below and above the DNA scaffold, respectively. Crossed semiconductor and metallic nanotubes form the CNFETs in the scaffold cavities.

Reference

1. C. Dwyer et al., "DNA Functionalized Single-Walled Carbon Nanotubes," *Nanotechnology*, vol. 13, 2002, pp. 601-604.
2. K.A. Williams et al., "Nanotechnology—Carbon Nanotubes with DNA Recognition," *Nature*, vol. 20, 2002, p. 761.

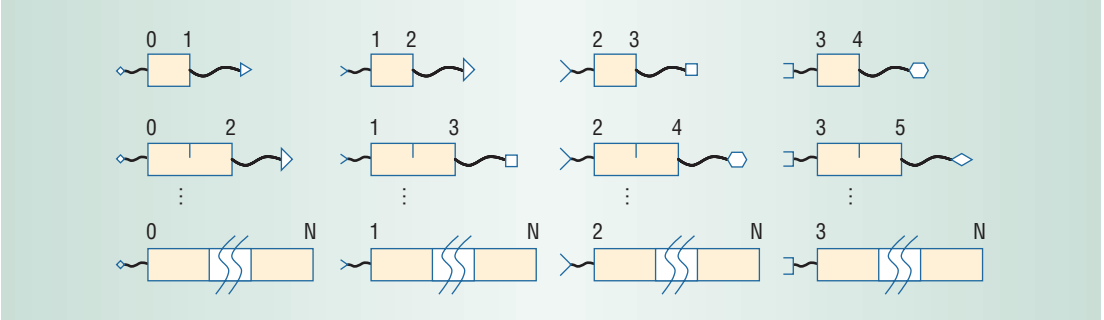


Figure 3. Sample tiles for creating substrings. The shapes on the tile ends represent the rules that define how tiles can be concatenated to form a string.

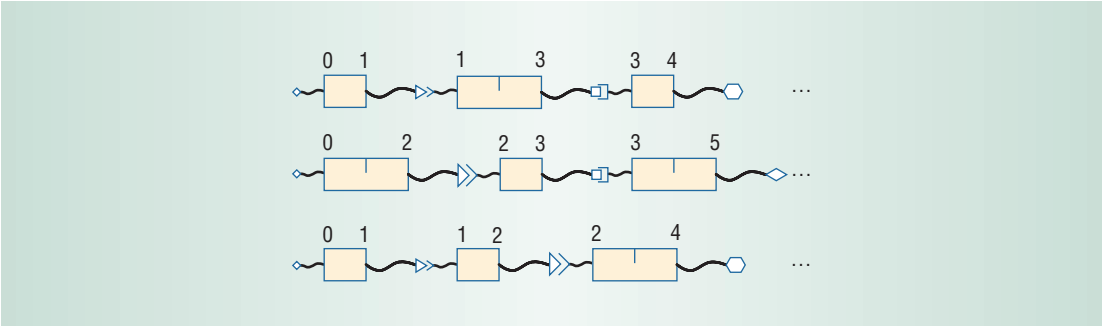


Figure 4. Block partition computations. The blocks from Figure 3 self-assemble randomly to form partitions. Each partition represents one of the $O(N_A^2)$ possible configurations.

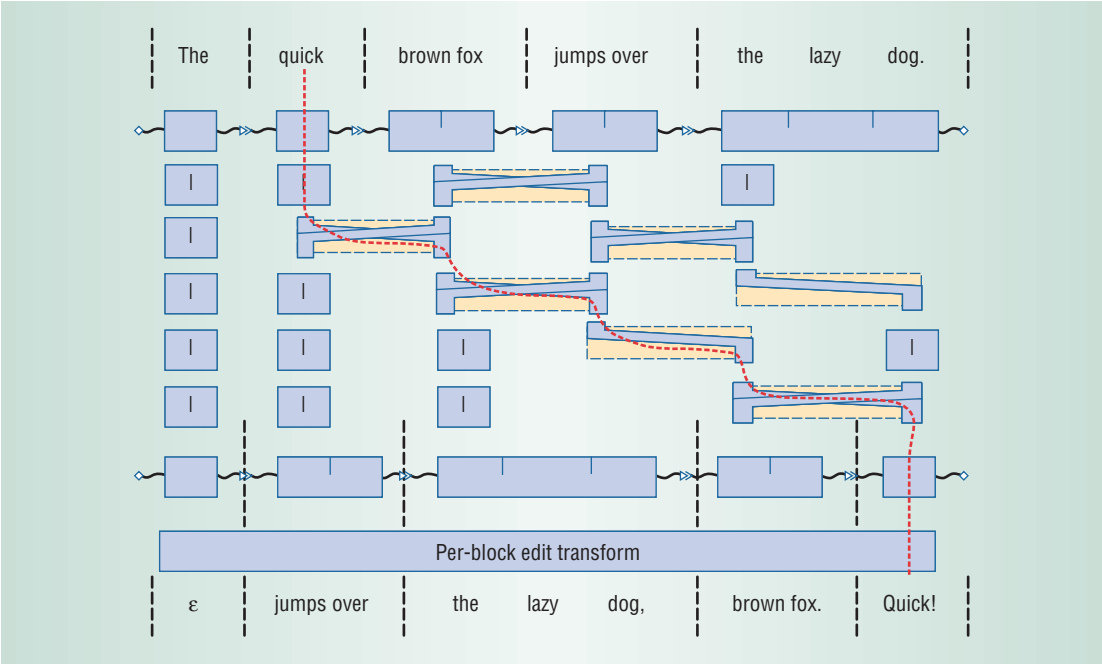


Figure 5. Block reordering. In the example, the word "quick" is reordered in each stage to arrive at its final position.

(T_2) if T_1's start matches the end ordinal of T_2. This is represented in Figure 3 by the shapes on the tile ends.

Figure 4 illustrates some example partitions using the tiles from Figure 3. The first partition breaks string A into length 1, 2, 1, … blocks, the second into length 2, 1, 2, … blocks, and the third into length 1, 1, 2, … blocks. These partitions are randomly distributed over many such tile assemblies (formed at-fabrication time) to probabilistically cover the partition space. In this way, tile assembly computes partitions before any postfabrication computation, using $O[(N_A)!]$ material.

Given the partition assemblies this computation creates, we can either perform the remainder of the algorithm in postfabrication computation or implement more of the algorithm at-fabrication. For the first option, the postfabrication complexity remains $O(N_A! \times N_A \times N_B)$ or approximately $O(N_A!)$. Shifting more of the computation to at-fabrication further reduces runtime complexity.

At-fabrication block reordering. Figure 5 illustrates a block edit problem using at-fabrication block reordering. The five intermediate stages in Figure 5 perform all possible block swaps through self-assembly to probabilistically cover all possible block orders. The figure shows one block reordering from the complete set of all possible reorderings that the circuit assembly will construct. At each stage, a block can remain in its current position, swap positions with an adjacent block, or shift over one position. These operations are encoded in tiles, and reordering occurs through tile assembly.

The highlighted path in Figure 5 shows how the word "quick" is reordered in each stage to arrive

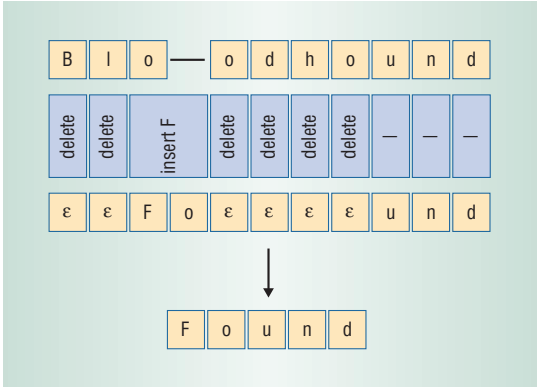


Figure 6. Self-assembled character edit sequence. In the example, a sequence of deletions and insertions transforms the source block Bloodhound to the target block Found.

at its final position. In this way, the tiles can preselect a partition and block order to trade fabrication material for the $O[(N_A!)^2]$ runtime complexity of the brute-force block edit solution. For this approach, the fabrication material required is $O[(N_A!)^2]$ and the postfabrication computation complexity is $O(N_A \times N_B)$.

Alternative points in the temporal design space. Two additional points in the design space help to show the range of the temporal computing spectrum: at-fabrication character edits and at-fabrication construction of all transformations and string instances.

The first alternative is an incremental approach that moves the next step in the algorithm (character edits) from postfabrication to at-fabrication. Even though well-known solutions exist to solve this problem in $O(N_A^2)$ running time, a simple solution exists at-fabrication. Figure 6 illustrates how a random sequence of insert and delete operations can transform the source block to the target block. A sequence of deletions and insertions transforms the source block Bloodhound to the target block Found. The insert and delete operations self-assemble similar to the block partition elements through a set of binding rules.

An even more extreme form of at-fabrication computation takes the method to the limit by fabricating not only block partition, ordering, and edit assemblies but also every possible instance of the problem. For example, given a small enough length of strings, self-assembly can form all possible strings that are incorporated as inputs to the at-fabrication computations. This method is inefficient because the postfabrication task of communicating the two strings that define the problem instance to be solved can take as long as the alternative postfabrication solutions. This implies that a design should balance the ratio of postfabrication computation complexity to at-fabrication material costs.

The primary distinction between the three systems for solving the block edit problem is how they divide the computational work between at-fabrication and postfabrication. Self-assembly provides new opportunities and design points for architects across the temporal spectrum of computing. Inherent in this is the tradeoff between the material required for at-fabrication and postfabrication runtime complexity.

Programmable self-assembly provides new options for solving difficult computer architecture design problems. In particular, architects can use self-assembly to explore the engineering tradeoffs involved in combining at-fabrication and postfabrication computation. Although at-fabrication computation is resource-intensive, architects and system designers can use it to precompute solutions to problems that are too computationally intensive for postfabrication solutions. A spectrum of designs for solving the block edit problem illustrates the numerous possible design points that programmable self-assembly enables and the potential for novel architectures that exploit this capability. ■

References

1. H.-S. Kim and J.E. Smith, "An Instruction Set and Microarchitecture for Instruction Level Distributed Processing," *Proc. 29th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 2002, pp. 71-81.
2. C. Kim, D. Burger, and S.W. Keckler, "An Adaptive, Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, 2002, pp. 211-222.
3. T. Mudge, "Power: A First-Class Design Constraint," *Computer*, Apr. 2001, pp. 52-57.
4. S.C. Goldstein and M. Budiu, "NanoFabrics: Spatial Computing Using Molecular Electronics," *Proc. 28th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 2001, pp. 178-191.
5. A. DeHon, "Array-Based Architecture for FET-Based, Nanoscale Eletronics," *IEEE Trans. Nanotechnology*, vol. 2, 2003, pp. 23-32.
6. J.P. Patwardhan et al., "Circuit and System Architecture for DNA-Guided Self-Assembly of Nanoelectronics," *Foundations of Nanoscience: Self-Assembled Architectures and Devices*, 2004, pp. 344-358.
7. Int'l Sematech, *International Technology Roadmap for Semiconductors, 2003 Edition*; http://public.itrs.net/files/2003ITRS/home2003.htm.
8. N. Seeman, "DNA Engineering and Its Application to Nanotechnology," *Trends in Biotech*, vol. 17, 1999, pp. 437-443.
9. B.H. Robinson and N. Seeman, "The Design of a Biochip: A Self-Assembling Molecular-Scale Memory Device," *Protein Eng.*, vol. 1, no. 4, 1987, pp. 295-300.
10. N. Seeman, "Nucleic Acid Junctions and Lattices," *J. Theoretical Biology*, vol. 99, 1982, pp. 237-247.
11. E. Braun et al., "DNA-Templated Assembly and Electrode Attachment of a Conducting Silver Wire," *Nature*, vol. 391, 1998, pp. 775-778.
12. K. Keren et al., "DNA-Templated Carbon Nanotube Field-Effect Transistor," *Science*, vol. 302, 2003, pp. 1380-1382.

13. K. Keren et al., "Sequence-Specific Molecular Lithography on Single DNA Molecules," *Science*, vol. 297, 2002, p. 72.
14. C. Dwyer et al., "The Design of DNA Self-Assembled Computing Circuitry," *IEEE Trans. VLSI*, vol. 12, 2004, pp. 1214-1220.
15. C. Dwyer et al., "DNA Self-Assembled Parallel Computer Architectures," *Nanotechnology*, vol. 15, 2004, pp. 1688-1694.
16. L. Adleman, "Molecular Computation of Solutions to Combinatorial Problems," *Science*, vol. 266, 1994, pp. 1021-1024.
17. J.R. Heath et al., "A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology," *Science*, vol. 280, 1998, pp. 1716-1721.
18. C. Thompson, S. Hahn, and M. Oskin, "Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis," *Proc. 35th Ann. Int'l Symp. Microarchitecture*, IEEE CS Press, 2002, pp. 306-317.
19. D. Lopresti and A. Tomkins, "Block Edit Models for Approximate String Matching," *Theoretical Computer Science*, vol. 181, no. 1, 1997, pp. 159-179.

Chris Dwyer *is an assistant professor in the Department of Electrical and Computer Engineering at Duke University. His research interests include nanoscale computer system design, self-assembled computer architecture, and DNA self-assembly. Dwyer received a PhD in computer science from the University of North Carolina at Chapel Hill. He is a member of the IEEE, the ACM, and the AAAS. Contact him at dwyer@ece.duke.edu.*

Alvin R. Lebeck *is an associate professor in the Department of Computer Science at Duke University. His research interests are nanoscale computer architectures, energy-efficient computing, and multiprocessors. Lebeck received a PhD in computer science from the University of Wisconsin–Madison. He is a member of the ACM and a senior member of the IEEE. Contact him at alvy@cs.duke.edu.*

Daniel J. Sorin *is an assistant professor in the Department of Electrical and Computer Engineering at Duke University. His research interests include dependable computer architecture and system design. He received a PhD in electrical and computer engineering from the University of Wisconsin–Madison. He is a member of the IEEE, the IEEE Computer Society, and the ACM. Contact him at sorin@ece.duke.edu.*