DesignCon 2012

Algorithmic Memory Brings an Order of Magnitude Performance Increase to Next Generation SoC Memories

Sundar Iyer, Memoir Systems
Sundaes@memoir-systems.com

Da Chuang, Memoir Systems
Dachuang@memoir-systems.com

Abstract

Historically, circuits and advances in lithography have been used in every generation as the approach to enhance memory performance. Unfortunately these approaches alone do not give enough performance improvement, and are not keeping up with applications that require higher memory performance. Therefore, memory performance remains a bottleneck with traditional approaches leading to the often quoted 'processor memory' gap. This paper describes a completely new technique - algorithmic memory, which uses the power of algorithms to make existing embedded memory faster. In particular, algorithms, which are implemented in standard RTL logic, are used with existing embedded memory to expose multiple memory interfaces. These interfaces individually provide true random access memory operations, and can be used independently and in parallel to dramatically increase the total number of memory operations per second.

Authors Biography

Sundar Iyer is co-founder and CTO at Memoir Systems, a start-up specializing in Semiconductor Intellectual Property (SIP) for algorithmic memories. Previously, Iyer was CTO and co-founder of Nemo ("Network Memory") Systems, acquired by Cisco Systems in '05. Iyer was a founding member at SwitchOn Networks (acquired by PMC-Sierra in '00), where he developed algorithms for associative memory and deep packet classification. In 2008, Iyer was awarded the MIT technology review (TR35) young innovator award for his work on network memory. He received his Ph.D. in Computer Science from Stanford University in 2008.

Da Chuang is co-founder and COO at Memoir Systems. Chuang was a founding member of Abrizio Systems (a terabit switch fabric company acquired by PMC-Sierra in 1999), and has past experience at both Nvidia and Adaptec. Prior to founding Memoir, Chuang co-led (along with co-author Sundar Iyer) the network memory group at Cisco Systems, where he helped architect and build multiple generations of high-performance memory sub-systems for Cisco's Enterprise and Data Center Ethernet products. Chuang received his Ph.D. in Electrical Engineering from Stanford University in 2004.

I. Rethinking Embedded Memory Design

The existence of the 'processor memory' performance gap is well known in industry [1]. Traditionally, this gap referred to the difference between the performance of processors and the external memories, which took hundreds of cycles or more to access. The obvious solution to closing this gap was to alleviate off-chip memory delay by integrating the processors with the memory, and other components, on the same chip - leading to the advent of System on Chip (SoC) designs. But does such a performance gap exist with embedded memory?

Today, a single-port embedded memory can perform one memory operation per clock cycle. Therefore embedded memory performance has traditionally been closely tied to memory clock speed and ultimately limited by it. Because embedded memory IP providers (responding to application needs for more on-chip memory) had to make design trade-offs early on that favored high density over high speed, memory clock speeds lag far behind processor clock speeds. Thus, there is also a performance gap between processors and embedded memory, which causes a processor to stall while it waits for memory accesses to complete. Due to faster processing speeds, pipelined architectures requiring accesses to certain shared central memories, and especially multi-core processing (where there can be simultaneous memory accesses from multiple cores), this gap is getting worse. [1] As a result, embedded memory performance has become the limiting factor in many applications. In this paper, we ask the following fundamental question:

1. Is it possible to increase memory performance without increasing clock speeds?

The performance limitations of embedded memories are largely a result of the way we have conceptualized the problem. In fact, it is possible to improve memory performance by an order of magnitude using currently available technology and standard processes. The problem is we have limited our thinking about embedded memories to a purely circuit and process oriented approach [2] [3] [4]. Thus, our focus has been on maximizing the number of transistors on a chip and cranking up the clock speed. This has been successful up to a point, but as transistors approach atomic dimension, we are running into fundamental physical barriers. For this reason, we need to rethink our approach to embedded memory design. For example, increases in processor performance have come not only because of advances in circuitry, but also because of architecture improvements, such as pipelined execution and exploitation of instruction-level parallelism. What if embedded memories could be designed to take advantage of architectural and parallel mechanisms similar to processor architectures?

[1] A more detailed description of the reasons for the growing processing embedded memory gap is described in [9].

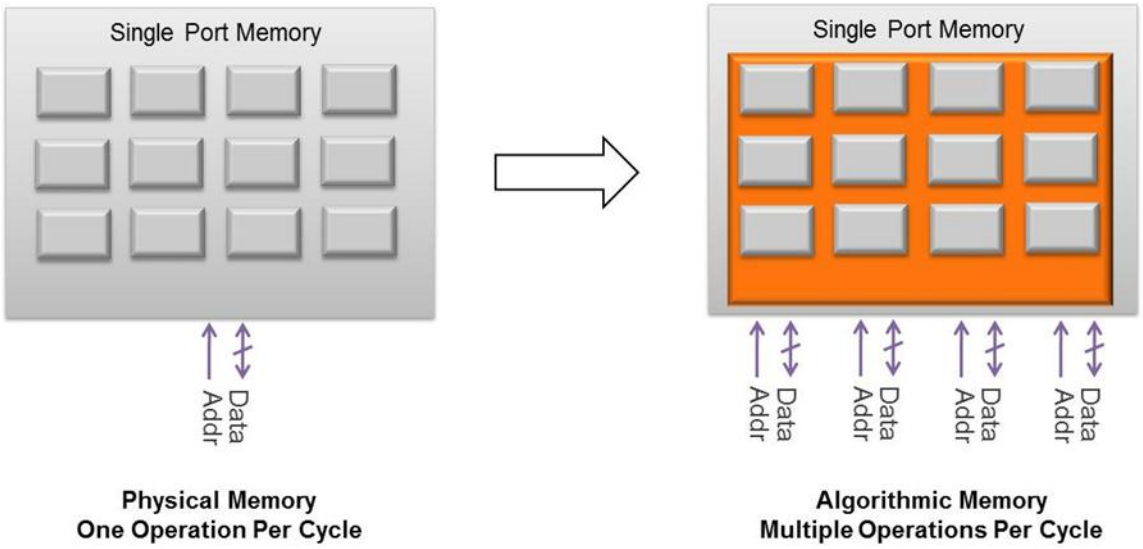

Figure 1: Algorithmic memories are created by adding logic to existing embedded memory macros, enabling higher memory performance.

II. Introducing Algorithmic Memory

A new approach called Algorithmic Memory™ technology does exactly that. Algorithmic memories are created by adding logic to existing embedded memory macros enabling them to operate much more efficiently. Within the memories, algorithms intelligently read, write, and manage data in parallel using a variety of techniques such as buffering, virtualization, pipelining, and data encoding. These techniques are woven together to create a new memory that internally processes memory operations an order of magnitude faster and with guaranteed performance. This increased performance capability is made available to the system through additional memory ports such that many more requests can be processed in parallel within a single clock cycle as shown in Figure 1. The concept of using multi-port memories as a means of multiplying memory performance mirrors the trend of using multicore processors to increase performance over uniprocessors. In both cases, it is parallel architecture rather than clock speed that drives performance gains.

Algorithmic memory technology is implemented as a soft RTL. The resulting solutions appear exactly as standard multi-port embedded memories. A system architect can specify the level of memory performance that is required from a customized algorithmic memory. As will be described later, an algorithmic memory can also significantly lower area and reduce memory power in certain instances. Using this approach requires no change to existing memory interfaces and ASIC design flows. The technology is both process node and foundry independent. In essence, algorithmic memories open the door to allow system architects to rapidly and reliably create customized memory solutions that can be optimized for specific applications.

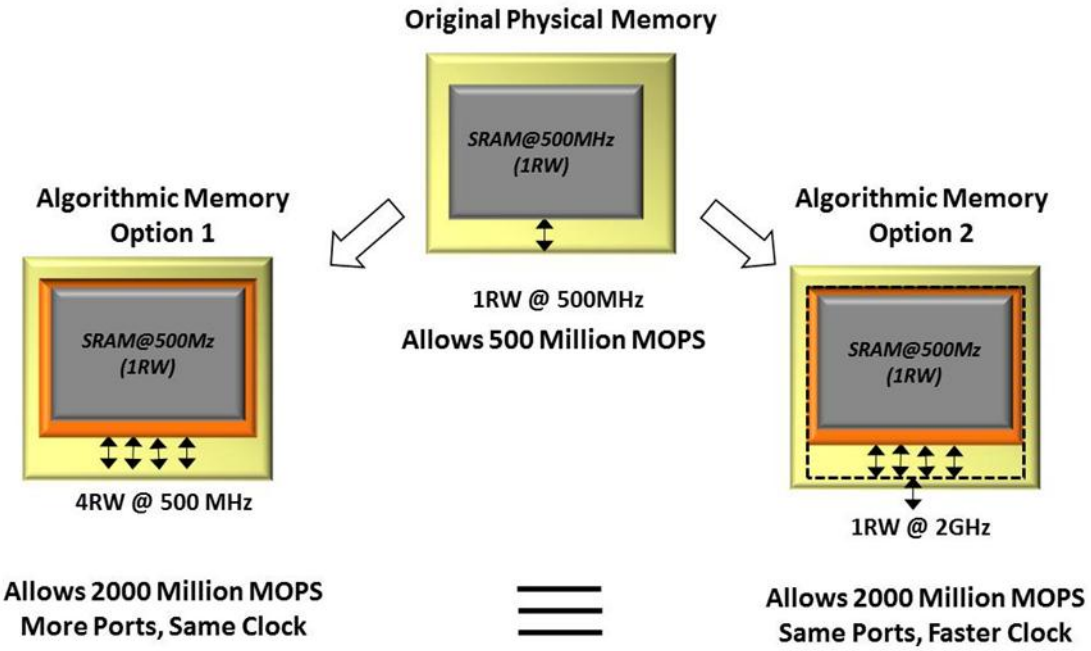

Figure 2: The higher performance of algorithmic memory can be realized by multiple interfaces or via a higher clock speed interface.

1. Interfacing to Algorithmic Memory

How can SoC architects avail themselves of the higher performance that an algorithmic memory can offer? There are essentially two ways in which a 4X memory acceleration can be achieved using algorithmic memory, as shown in the example in Figure 2. In the first option, an algorithmic memory appears as a multiport memory. In the example shown, an algorithmic memory presents four interfaces that together allow four accesses to the memory array in a single clock cycle, even though the embedded physical memory only allows one access per clock cycle. Thus in the above example, the original single-port memory with a clock speed of 500 MHz, would only be able to perform 500 Million memory operations per second (MOPS).[2] The algorithmic memory, on the other hand, will boost the performance of the physical memory to 4 interfaces X 500 Million MOPS = 2000 Million MOPS. This allows SoC to achieve higher memory performance, while operating at lower clock speeds. A second option wraps the algorithmic memory shown in option 1 in a faster clock domain. As shown in the example in Figure 2, the algorithmic memory now exposes only a single-ported interface running at 2 GHz, though internally it still continues to post four operations at a 500 MHz clock. This option is preferred if the processors interfacing to the algorithmic memory operate at a faster clock. Note that in both options no changes are made to the internal physical memory, which continue to operate at 500 MHz.

[2] The use of MOPS to measure memory performance is analogous to the use of IOPS (Input/Output Operations Per Second) to benchmark storage devices. It is a more inclusive measure of memory performance than memory bandwidth [14].

III. How does Algorithmic Memory Work?

To understand how this works, recall that every algorithmic memory consists of a number of memory macros, where each of the memory macros can be accessed in parallel. Each memory macro has its own physical address and data bus. Thus, four external accesses that address four different macros can take place in parallel in a single clock cycle. The matter gets more complex when all four accesses are trying to access the same memory macro. In such a case, the algorithm directs the other accesses to other macros within the memory.

These actual addresses are a form of virtual addressing that is kept track of in scratchpad memory so that they are correlated with the intended address. Since reads and writes can come in rapid succession and in all kinds of combinations, the logic in the algorithmic memory core has to be able to manage all the patterns of hot spots and multiple accesses to the same macro intelligently. When there is time, the algorithm can move them and rearrange them. However, the logic must also handle a worst-case-for-life scenario and intelligently rearrange things so that the operations continue to be posted.

Memory read operations are a little more complex. For example, in the case of a two-read access, if the application wants data from the same memory macro, they cannot be accessed in parallel and trying to access them sequentially would cost performance (latency). Therefore, all the data that is stored in the physical memory is encoded using a variety of schemes to allow the algorithm to infer the read data using data from other macros.

1. Proving Predictable Memory Performance for Algorithmic Memory

Algorithmic memories are able to increase performance in a completely predictable manner with no exceptions whatsoever, including resolving all traditional row, address, and bank conflicts that may arise due to simultaneous accesses from the multiple interfaces. The performance guarantee of an algorithmic memory is first mathematically proven using adversarial analysis models [5]. The adversarial model used is the *strong adversary* model[3], i.e. an adversary who has complete knowledge of the inner working of the algorithms, can predict what the algorithm does cycle by cycle, is aware of any randomization effects of the algorithm, and has complete control over the memory access pattern. The advantage of using the strong adversary model is that if performance is proven against a strong adversary, then the performance is guaranteed irrespective of the sequence of simultaneous address accesses and data patterns.

2. Guaranteeing Memory Performance for Algorithmic Memory

Each implementation of algorithmic memory is formally verified using formal model checking which enables independent exhaustive proofs for the implementation of algorithmic memory. As a particular example, algorithmic memories were verified using the formal property checking tool Magellan [6]. Magellan formally verifies user-specified

[3]This is referred to in Academia as the *adaptive offline adversary*.

properties for a given design. Properties are specified using System Verilog assertions, and Magellan's formal property checkers try to mathematically prove whether the behavior of a particular design conforms to the given assertions. If the assertions are static properties (i.e. invariants) of the design and if Magellan can validate that these properties are universally true or false (as specified by the user) — irrespective of the input parameters, then this leads us to a path for a formal proof that a particular design works. Note that if the tool converges, it either returns a `Proven` or `Falsified` outcome for each instance of the algorithmic memory. The `Proven` outcome means that the design has the correct behavior for any input satisfying the assertions. The `Falsified` outcome returns a scenario in which the design violates the given property.

3. Algorithmic Memory Tradeoffs

A typical algorithmic memory that is optimized for performance may use up to 15% additional area for 2X increase in performance. The area tradeoff varies depending on the size of the memory instances, and is discussed in detail in Section IV. It is important to note the additional logic does not add clock cycle latency. A portion of the algorithmic memory management logic adds a few muxes worth of sub-clock delay. As an example, this is typically around 150 picoseconds in 28nm process node. This delay is squeezed within the same clock for most implementations and typically does not result in any clock cycle latency. Insofar as one is prepared to tradeoff some area, memories can be made significantly faster and up to 10X increase in performance is possible. In practice, the majority of applications benefit from up to 4X in memory performance.

4. Example: Increasing Performance in a Multicore Cache

Consider the following customer request for algorithmic memory in a multicore SoC with 16 processor cores, where it is used for the L2 instruction cache. Both memory and processors are clocked at 800MHz. The L2 instruction cache consists of 2MB of SRAM (32K deep × 64 bits wide) in 28nm technology node. For certain applications, each of the 16 32-bit processor cores can fetch instructions as frequently as every other clock cycle, since fetches are 64 bit or two instructions per read. The maximum memory demand is 6400 Million MOPS (16 processors × 800MHz ÷ 2 cycles/fetch). A single-port memory could provide only 800 Million MOPS and would therefore severely limit system performance since the processors would have to wait for instruction fetches to complete. The SoC manufacturer would like eight read ports (8R) and an addition port for occasional write-backs (1W) from main memory to update the cache. A 9 port algorithmic memory (8R1W) could provide the required 6400 Million MOPS (8 ports × 800 MHz) and while supporting the occasional writes to cache from main memory. The conventional embedded memory would have supported only 800 Million MOPS and required 0.5 mm^2, while the new algorithmic memory would boost the conventional memory to 6400 Million MOPS with 1.4 mm^2 area. The SoC manufacturer would consider this a good trade-off.

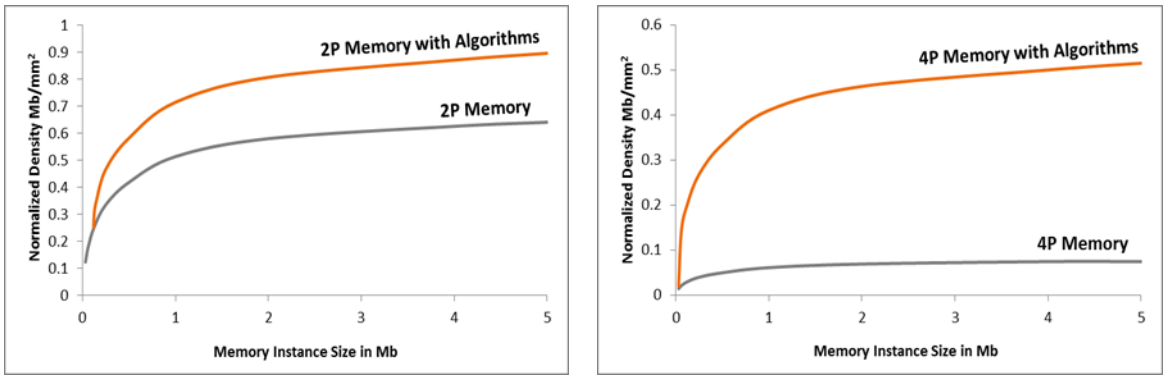

Figure 3: Algorithmic memory technology increases the density (lowers area) of physical memories. This also reduces the leakage power consumption. Graphs are normalized with single-port memory density = 1Mb/mm^2.

IV. Lowering Memory Area and Power

In some cases, Algorithmic Memory technology can also be used to lower memory area and power consumption without sacrificing performance. There is a significant area and power penalty when a higher performance memory is built using circuits alone. In contrast, an algorithmic memory is built from a lower performance memory (which typically has lower area and power), with additional memory and logic. This algorithmic memory achieves the same MOPS as a high performance memory built using circuits alone, but can have lower area and power. In general, the area overhead for an algorithmic memory depends on many factors including the physical memories available, the increase in the read MOPS and/or write MOPS required, and the capacity of the instantiated memory.

The additional memory and logic that algorithmic memories use to increase memory performance becomes proportionally smaller as memory capacity increases. Thus, as capacity increases, algorithmic memories use space more efficiently and thus offer better density (Mb/mm^2) than circuit-only memories. For instance, in one analysis using commercially available memory IP, comparison of two-port (2P) memories showed algorithmic memories yielded up to 50% greater density and thus required about 50% less area (Figure 3a). When four-port (4P) memories were compared, algorithmic memories used about 50-80% less area for the same capacity (Figure 3b). Because area and energy consumption are closely related, there is a corresponding savings in leakage power as well, which results in absolute power savings. Note that in both scenarios, if the capacity of the memory instances is below a cut-off point (usually in the order of kilobits), the physical memory may be more area (and power) optimal than the corresponding algorithmic memory.

1. Example: Saving Area and Power in a Networking SoC

To put this in the context of a real application, a SoC architect was redesigning a network switch to support 500 Million arriving packets/sec. The design required a 64 Mb memory (256K deep × 256 bits wide), and support 1000 Million MOPS. This required an area of 16.5 mm^2 and 4.1 Watts in power in 32nm process node. Using algorithmic

memory, a high density eDRAM running at 500 MHz was accelerated by 2X performance to give 1000 Million MOPS. The resulting solution had a total area of 9.6 mm^2, and a total power of 590 mWatts. This resulted in ~7mm^2 of area savings and 3.5 Watts in power savings on the die.

V. Unlocking System Performance

The point of increasing memory performance is ultimately to improve system performance. To that end, the first question to ask is where is the bottleneck? In some cases, the bottleneck is obvious. In other cases though it is only after the designer starts figuring out the micro-architecture and analyzing all pipeline stages, inputs, outputs and the logic in between that the bottlenecks become evident.

1. Example: Accelerating Memory for Wire Speed Learning

In networking applications, memory performance tends to be the weak link. One reason for this is simply that network wire speeds are increasing much faster than memory performance. Couple this with the fact that operations within a switch are memory intensive and often require several operations per packet. Consider that a 48-port10-Gigabit Ethernet network router which can receive 15 Million packets per second per port for a total of 720 Million packets per second. Since data is received in both directions per port, this is an aggregate of 1440 Million packets per second. A router must perform a Media Access Control (MAC) address look up for every client on the network.[4] Now consider what happens each time the router is power cycled and must relearn all of the MAC addresses. A given subnet might support tens of thousands of clients, and if all the MAC address look ups had to be handled in software it could take several minutes to rediscover all the clients. If this function were performed within the router at wire speeds the recovery would be instantaneous. Each lookup per packet requires two memory reads, plus a write-back to update the database. This means a total of three memory accesses must be performed for each packet received (3 × 1440M packets/sec), requiring a total of 4,320 Million MOPS. If the SoC system clock operated at 720 MHz, then a 720 MHz single-port memory can only perform 720 Million MOPS, and system architects will be challenged to meet this demand using conventional memory technology. On the other hand, algorithmic memory offers an elegant solution.

2. Why are Current Solutions Inadequate?

Today, system designers use a wide array of ingenious system level mechanisms such as hierarchical caches, pipelined memory architectures, memory striding, and static memory allocation to avoid memory bottlenecks [1]. Application specific solutions can give high memory performance [7] [8] but are however limited in scope. Similarly, software based solutions such as loop interchange [9] [10] and modulo interleaving [11], require careful arrangement and partitioning of data in memories, and need re-design for each new memory instantiation. Statistical solutions, which involve using memory banks efficiently, are also commonly used in Industry. For example, a statistical solution might intelligently multiplex and demultiplex accesses to different banks. Since each bank has

[4] A MAC address is a unique 48-bit identifier that must be learned by the router and entered into its database.

its own I/O bus this can improve performance as long as sequential requests are to different banks. However, if two consecutive requests are made to the same bank, then the second one will be delayed until the first one completes. If a third access is to a different bank, then in order to preserve order and maintain memory coherency, it must be put into a FIFO until the previous two requests are completed. The problem with statistical approaches is that the performance is unpredictable and varies according to the pattern of data accesses. Often, to achieve performance goals a designer will make compromises such as replication of memory hardware or increased design complexity. However, system level approaches are not always applicable and do not always provide the necessary sustained performance.

An ideal solution for the wire speed learning problem above would be a multiport memory with four read (4R) ports and a two special purpose write-back (2W) ports, that can read and write-back the same memory location in a single clock cycle. Each port would need to sustain 720 Million MOPS, for a total of 4320 Million MOPS, which would require about a 6X acceleration of a 720 MHz embedded memory core. Using algorithmic memory technology such a memory could be readily created.

3. Customizing Multi-port Memories for High Performance SoCs

Multi-port memories are an excellent design choice for L2 caches, L3 caches, tag tables, TLBs, in multiprocessor systems. They are also useful for data path buffers, and control path memories such as Netflow, State tables, lookup tables, statistics counters, and packet linked-lists for network processors. They also find use in a number of specialized memories in shared memory buffers, instruction fetch memories, dual ported DSP caches, as well as in high performance graphics and video processors. They are an ideal match for multi-core systems where multiple cores can access cache coherency memories simultaneously. Yet multi-port memories and the type of specialized memory solution above would be extremely difficult, if not impossible, to create with circuit-only techniques. Many SoC architects would like to use multi-port memories, but are hesitant to do so because of the complexities, cost and long time periods associated with the silicon manufacturing, testing and validation processes. Algorithmic memories eliminate these problems and give SoC architects new tools to unlock system performance.

VI. Eliminating Application Pain Points

Every application has a different pain point. Do reads need to be faster, or writes, or both? Is power consumption an issue? Is die area a concern? How can we find the optimal balance of speed, area, and power? Understanding whether a bottleneck is the result of reads, writes, updates or any combination of reads and writes is the basis for determining what kind of memory is required to solve the problem and this is where algorithmic memory technology comes into play.

Algorithmic memory allows us to create customized memories that are highly optimized for specific applications. The more clearly we define the performance requirement for a specific application the better we can make the right tradeoffs in terms of speed, area, and power as shown in Figure 4. For example, if an application is mainly doing reads to a data structure, and that becomes a bottleneck, then perhaps the best solution would be a

four read port memory with just one write port (4R1W). In another case, an architect may decide two read ports and two write ports are needed. This could be satisfied with a four port algorithmic memory with two read and two write ports (2R2W), assuming the requirement is for equal amounts of read and write acceleration.

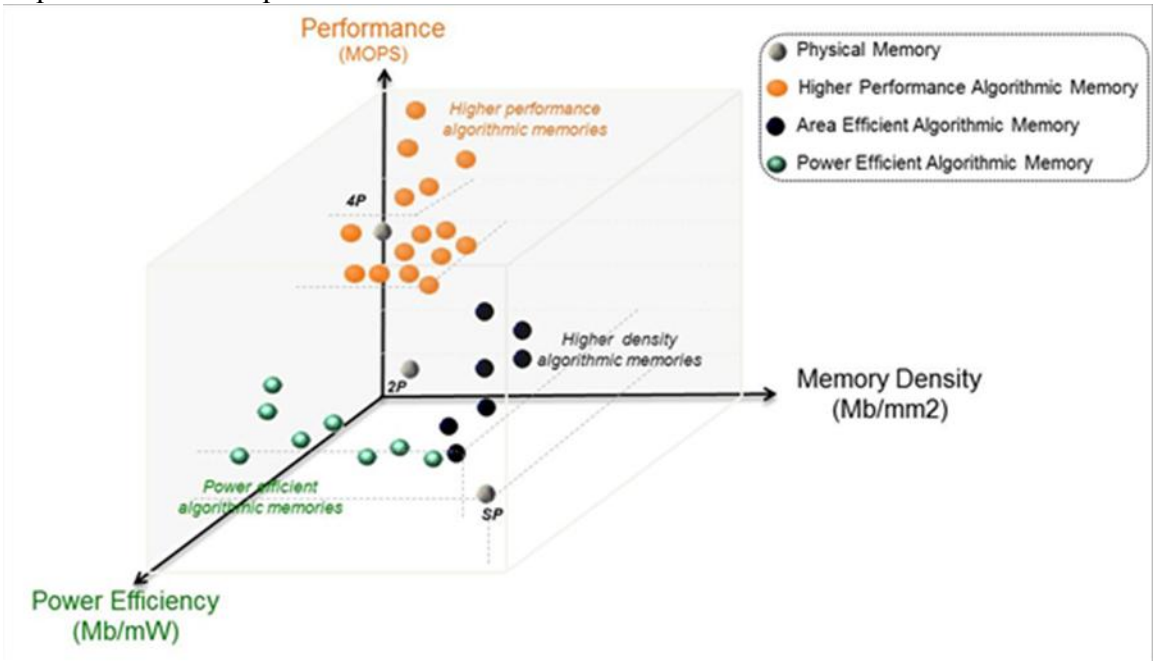

Figure 4: Algorithmic memory technology allows system designers to treat memory performance as a configurable entity with its own set of tradeoffs with respect to speed, area and power.

Another designer might find an application does lots of reads and other times lots of writes. In this instance, a quad-port memory, which means four bi-directional ports, would be preferable. By convention, if the ports are bi-directional they are called dual, tri, quad, and so on. For example, a quad-port memory is a superset, and it can be used as a four port memory or with various ports used bi-directionally. Perhaps an application is doing only updates, which is a special case where the application does a read-modify-write. For example, to update a counter you would read from an address and immediately write back to the same address a few cycles later. Can a memory be built to exploit this special case?

1. Increasing the Portfolio of Memories Available for SoC Applications

Until now, using customized multiport memories was discouraged because both the cost and the amount of time required to design, develop and verify a new memory were prohibitive. This is no longer the case with algorithmic memories. Algorithmic memories can be created very rapidly by combining existing memory cores with previously verified algorithms. In principle, a memory synthesis tool could analyze existing memory offerings from any vendor and select the right memory core and the right combination of algorithms for a particular set of memory requirements. Using this automated memory synthesis platform, a new custom memory could be created within a couple of days as shown in Figure 5. Note that all higher performance algorithmic memories can be created from single-port memories. However, if the set of available physical memory libraries

increase -- Figure 5 demonstrates a case where both a single-port (1RW) and a dual port memory (2RW) are available in the physical library --the algorithmic memories can use the richer set of available physical memory libraries, and generate more area and power optimal higher performance memories.

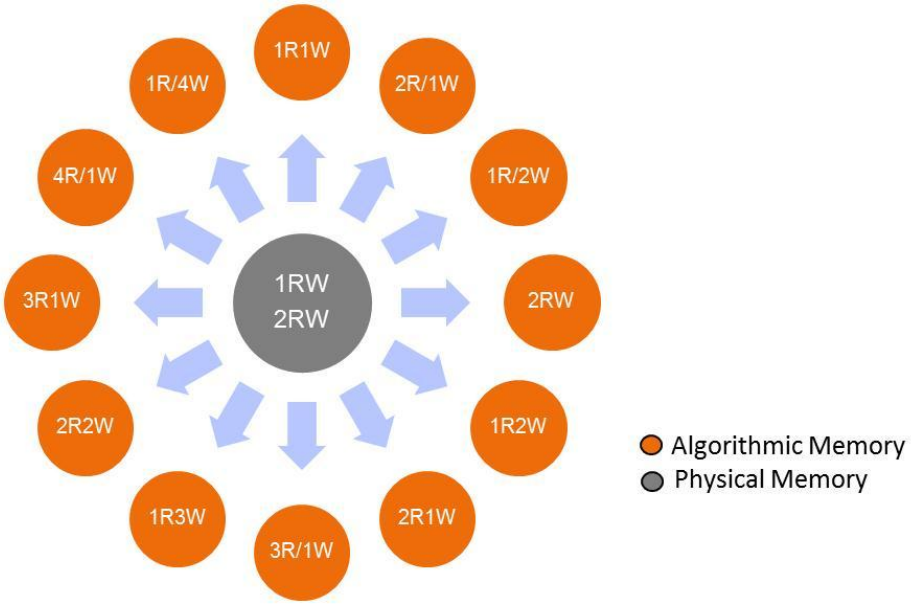

Figure 5: Algorithmic memories can be generated from a small set of base physical memories and provide a broad portfolio of customized memories with any combination of read and write interfaces.

VII. Synthesizing New Memory Solutions

1. Specification of a Custom Algorithmic Memory

How would an algorithmic memory synthesis platform work? A system architect would need to specify the desired characteristics of the new memory such as the number of read and write interfaces, the operating clock frequency, and any area and power requirements, as shown at the left of Figure 6. The synthesis platform would need to perform very rapid analysis and estimations of potential solutions since it must sort through a large body of commercially available memory macros and determine the best matching of physical memory and algorithms.

2. Analysis of a Custom Algorithmic Memory

This phase of processing could be done working with abstract models of the available memory macros. For example, all memory macros are considered as "building blocks" and might be characterized in a common format, such as an ASCII representation, to capture each memory's data width, address depth, operating clock frequency and power consumption, as shown at the bottom of Figure 6. Likewise, every available algorithm could be characterized or mapped into a database for analysis based on whether it accelerates reads or writes or both, the number of ports it supports, the number of additional bits required to accelerate the memory, etc. Working with this high level information, the synthesis platform could rapidly analyze various combinations of

memory macros and algorithms to find a set of potential algorithmic memory solutions and their estimated speed, area, and power characteristics.

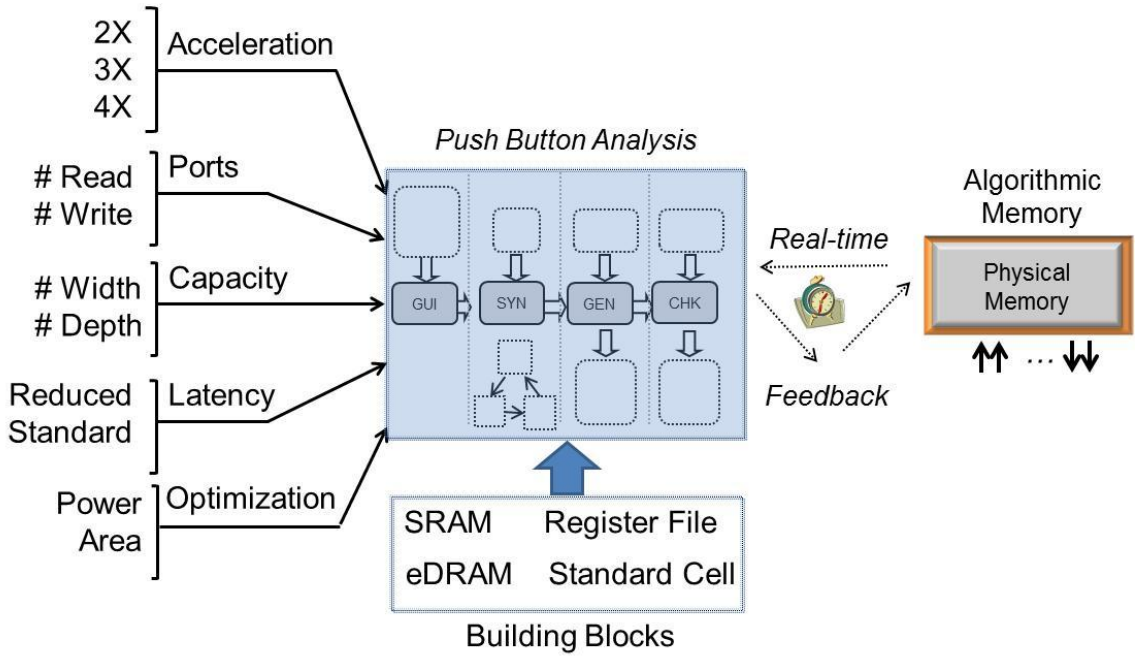

Figure 6: An algorithmic memory synthesis platform can analyze and estimate the resulting area, power and speed of custom memory configurations in seconds, and generate it in a matter of days.

3. Estimation of a Custom Algorithmic Memory

How are the area and power tradeoffs for an algorithmic memory estimated? Note that estimating the area and power consumption for a complex memory with circuits is tedious, but there is essentially only one brute force way to do it. For example, building a 4R4W memory requires laying out enough transistors onto a cell to support four inputs and four outputs to derive the actual area, and perform simulations to estimate the power consumption. With algorithmic memory, however, there are many ways to build one, each with its own advantages and disadvantages.

For example, to build a 4R memory, we might start by building a 2R memory. This 2R algorithmic memory could then be modified to create a 3R memory and then the 3R memory modified to form a 4R memory. Furthermore, the 4R memory could form the basis for a 4R1W memory and additional write acceleration algorithms could be added to support more write ports to form a 4R4W memory. The underlying physical memory may only be doing just 1R. Another option could be to build a 4W memory, and then add read acceleration algorithms to form a 4R4W memory. Thus there are many ways to construct an algorithmic memory, each with their own area and power trade-offs.

The point is that algorithmic memory can be constructed recursively, and it is not necessary to a priori estimate the area and power tradeoffs of every configuration of algorithmic memory. Due to this fact, the estimation of the area and power tradeoffs can

be done in software, and the task is highly parallelizable, and can be performed in real-time in a matter of seconds.

4. How is Algorithmic Memory Generated and Delivered?

Using the memory synthesis tool, a custom algorithmic memory can be generated from a combination of the various algorithms. Before generating a memory, the tool selects the most area, power optimal memory (as specified by the user) from the candidate list of all algorithmic memories that have been estimated. After generation, the tool also runs through a comprehensive suite of dynamic simulations and formal proofs to exhaustively verify the algorithmic memory (100% verification). At the end of the process, what would be the output of the synthesis platform, or more to the point, what constitutes an algorithmic memory?

Recall that circuit definitions for the algorithms are just register-transfer level (RTL) logic, like any other logic. Then a logic synthesis tool such as Design Compiler from Synopsys [12] or RTL Compiler from Cadence [13], is used to create an intermediate format code, i.e. a gate level netlist that is specific for a foundry and technology node. Thus only in this final stage would the synthesis platform need to use detailed information for a specific vendor, process, and node to synthesize the algorithmic memory, close timing, and perform formal verification as shown in Figure 6. Algorithmic memory technology -- because it is RTL based -- can significantly shorten memory development time. Custom memory development, including architecture, physical design, testing and characterization, can typically range between 6 to 12 months. With algorithmic memory, a customized memory (built on existing memory macros) can be synthesized in a matter of days.[5]

Algorithmic memories would be delivered as soft intellectual property (soft IP), because only intermediate formats (i.e. gate level netlists) are generated. The chip designers would then integrate the intermediate format memory with the rest of the chip's intermediate format components. This has the advantage of allowing chip designers, who have more system knowledge of their chips, to make decisions about how the routing and placement is done.

VIII. Conclusion

In summary, algorithmic memory provides a means to eliminate memory performance bottlenecks in any SoC design and thereby improve overall application performance. It addresses the challenge of memory performance at a higher level and allows system designers to rapidly create customized memory solutions that are optimized for a specific application. New and customized memories can be designed and synthesized in a few days and require no further silicon verification. The resulting memories use a standard SRAM interface with identical pinouts and integrate into a normal ASIC design flow. Since it is RTL based, algorithmic memory technology is process, node, and foundry independent, and applies to a variety of SoC implementations such as ASICs, ASSPs,

[5] A current working platform MemoSyn™ can architect and evaluate custom memory in less than 10 seconds, and synthesize memory in less than two days.

GPPs and FPGAs. Algorithmic memories allow system architects to treat memory performance as a configurable characteristic with its own set of tradeoffs with respect to speed, area and power. While not a panacea it empowers us with new techniques to overcome the processor embedded memory gap, and further unlock SoC performance.

References

[1] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th Edition ed., Morgan Kaufmann Publishers, Inc., 2011.

[2] "DesignWare Embedded Memories," [Online]. Available: http://www.synopsys.com/IP/EmbeddedMemories/Pages/default.aspx.

[3] "ARM Embedded Memory IP," [Online]. Available: http://www.arm.com/products/physical-ip/embedded-memory-ip/index.php.

[4] "Dolphin Memory Products," [Online]. Available: http://www.dolphin-ic.com/memory-products.html.

[5] S. Ben-David, A. Borodin, R. Karp, G. Tardos and A. Wigderson, On the power of randomization in on-line algorithms, New York: Springer, 1994.

[6] "Magellan Formal Verification Tool," [Online]. Available: http://www.synopsys.com/tools/verification.

[7] G. Varghese, Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices, Morgan Kaufmann, 2004.

[8] S. Iyer, "Load Balancing and Parallelism for the Internet," Stanford University, PhD Thesis, 2008.

[9] K. Kennedy and A. Randy, Optimizing Compilers for Modern Architectures: A Dependence-based Approach, Morgan Kaufmann, 2001.

[10] G. Rivera and C. Tseng, Data Transformations for Eliminating Conflict Misses. In Proc. of ACM SIGPLAN 1998 Conference.

[11] R. Rao, "Pseudo-Randomly Interleaved Memory," in *In Proceedings of the 18th annual international symposium on computer architecture*, 1991.

[12] "Synopsis Design Compiler," [Online]. Available: http://www.synopsys.com/tools/implementation/rtlsynthesis/dcgraphical/Pages/default.aspx.

[13] "Cadence Logic Synthesis," [Online]. Available: http://www.cadence.com/products/ld/Pages/default.aspx.

[14] S. Iyer. [Online]. Available: http://www.memoir-systems.com/media/Memoir-AlgMemory-Whitepaper.pdf.