Bouncing Threads: Merging a new execution model into a nanotechnology memory

Sarah E. Frost sfrost@nd.edu
Arun F. Rodrigues arodrig6@nd.edu
Charles A. Giefer cgiefer@nd.edu
Peter M. Kogge kogge@wizard.cse.nd.edu

University of Notre Dame
Dept. of Comp. Sci. and Eng.
Notre Dame, IN 46556, USA

Abstract

The need for small, high speed, low power computers as the end of Moore's law approaches is driving research into nanotechnology. These novel devices have significantly different properties than traditional MOS devices and require new design methodologies, which in turn provide exciting architectural opportunities. The H-memory is a design developed for a particular nanotechnology, quantum-dot cellular automata. We propose a new execution model that merges with the H-memory to exploit the characteristics of this nanotechnology by distributing the functionality of the CPU throughout the memory structure.

1 Introduction

The computer industry has had great success in shrinking components according to Moore's law. However, variations of Moore's law also describe the rate of growth of the power dissipation of these chips, especially logic chips, as growing exponenitially and of the cost of fabrication plants, already in the billions of dollars, as growing exponentially. In addition to the problems due to rising heat dissipation requirements and fabrication costs, the current strategy of scaling down current devices is approaching a fundamental physical barrier beyond which the devices no longer make sense. To continue meeting the need for small, high speed, low power computers, the current paradigm needs to be altered.

If one steps back and looks at the total number of transistors in a computer, they are predominantly memory. In fact, memory has traditionally driven the size of devices but not the system microarchitecture. It is in the context of this role of memory that this paper proceeds.

Several devices are being explored to fill the gap left at the end of the roadmap at the nano-scale [4]. The characteristics of these devices are significantly different from the MOS devices that current architectures and designs exploit. As these nano-scale devices are being developed, it is important to develop architectures that exploit the characteristics of the new devices in addition to drawing on wisdom gained in the design of MOS-based computers. For instance, current systems have highlighted problems such as the von Neumann bottleneck, the bottleneck between the CPU and the memory. This paper introduces a memory architecture based on the device characteristics of a particular nanotechnology, quantum-dot cellular automata, and proposes a natural extension of this framework that exploits the characteristics of QCA and addresses the von Neumann bottleneck, basically by doing away with teh cause of the problem, the logic-based CPU chip.

1.1 Quantum Cellular Automata

Quantum-dot cellular automata (QCA) is a novel alternative to the transistor paradigm [6][1]. Its characteristics are very different from MOS devices and offer new architectural opportunities. A QCA cell consists of four quantum dots arranged as sitting on the corners of a square. Two excess electrons are introduced into the cell and are able to move between dots by quantum mechanical tunneling. The electrons repel each other and naturally reside in opposite corners of the cell, leading to two steady states - electrons in the top left and bottom right corners or in the bottom left and top right corners (figure 1). The first has a polarization of -1 and is associated with a binary 0 value, and the second has a polarization of +1 and is associated with a binary 1 value. By placing several of these cells in close proximity in a row, a QCA "wire" is formed. The electrons remain within their own cell, but are able to influence its neighbor cells by the coulombic interaction of the electrons across cell boundaries. If the electrons of a cell are prohibited from tunneling, it can act as a driver cell, driving its neighbors to assume its value, and in this way a value can be propogated down this row of cells as in a wire.

The "clock" controlling such cells has four phases in which an electric field is raised and lowered to control the tunneling within the cells [3]. When the field strength is high, a cell has a definite configuration and can act as a driver cell. Cells in a rising field are assuming or switching to a value from a neighboring driver cell. Cells in an

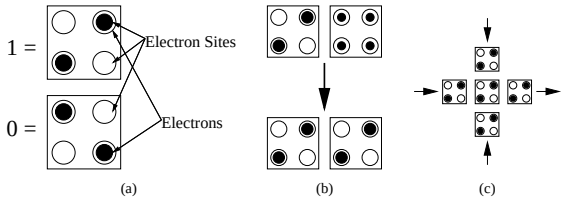

Figure 1. (a) A QCA cell has two stable configurations corresponding to binary 0 and 1. (b) A QCA driving its neighboring cell. (c) A majority gate, the fundamental computational unit.

area with a low field have no configuration and cannot influence the configuration of its neighboring cells, and finally a cell in a region of falling field strength is releasing a defined configuration. Respectively, these phases are referred to as hold, switch, relax and release (figure 2). An array of cells in the same clock phase are referred to as residing in a clock zone. It is important to note that unlike traditional circuits in which the clock is just like any other signal, the QCA clock is a fundamentally different sort of phenomena than the values being carried by the QCA cells. Indeed, a fundamental assumption in designing with QCA is that the continues to cycle independently of the cells in the circuit and their values.

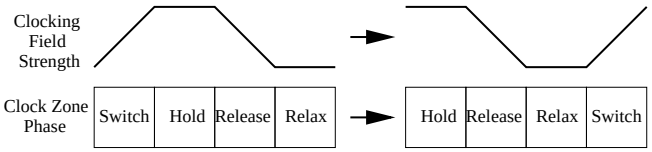

Figure 2. The four phases of the QCA clock: hold, switch, relax, release.

The fundamental computational block is the majority gate. A center cell acts as a "device cell", with neighbors on three of its edges acting as inputs, and a neighboring cell on the fourth edge acting as the output. Placing the cells in appropriate clocking zones forces the device cell to assume the value of the majority of its inputs, and allows this device cell to drive the output cell. With inputs A,B,C, the majority gate performs the function AB+BC+AC. By fixing one of the inputs to be a logical 1, the majority gate acts as a two input OR gate. Similarly, fixing one input to be a logical 0 leads to an AND gate.

Offsetting the cells in a wire so they are in a line corner to corner forces the value being propagated to be inverted at each cell. This allows an inverter to be built. With the majority gate, fixed inputs and inversion, QCA is able to perform any logical function. The device has been experimentally verified using metal dots [7]. In a molecular implementation [5], a QCA cell will be approximately 1.4 nm on a side and operate at room temperature. QCA has the potential for extremely dense logic and memory.

2 H Memory Framework

The goal of the "H" framework is a dense memory native to QCA [2]. The basic design landscape for QCA is different from the CMOS landscape. The design is driven by three related considerations that come out of the basic device characteristics of QCA. The first consideration is the inherent, fine-grained pipelining of QCA "wires". The second is the tight connection between layout and timing, and the third is that the data is always in motion. The H-memory is a design that grows naturally out of this landscape. The memory consists of a recursive H-tree structure (figure 3). Data words are stored in memory macros at the leaves of the

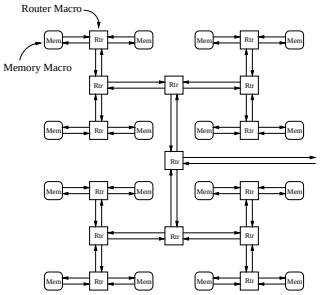

Figure 3. Recursively built H-structure built with routers forming interior nodes and memory macros forming leaves.

tree, and the internal nodes can be thought of as routers. Accesses to the data are effected via two parallel parcels. The first parcel, the address/data parcel, consists of the address to be accessed, followed by the operation to be performed (read or write), and finally the data to be written, if appropriate. The select parcel accompanies the data parcel and is used to signal the presence of meaningful bits on the data line (figure 4).

The parcels travel through the memory together in lockstep. The data parcel goes to every memory macro in the structure. The select parcel, though, arrives only at the desired address. The leading edge of the select parcel activates each router as the parcels encounter them. The routing decision, left or right to reach the desired memory macro, is made based on the bit in the address/data parcel that arrives with the first bit of the select parcel. After the select parcel leaves the router, the first bit of this parcel is stripped off, leaving the next bit of the address to determine the direction of the select parcel at the next router. In this manner, the address is effectively stripped off as well. When the parcels reach the desired memory macro, the operation code (read or write) arrives with the leading edge of the select parcel (figure 4). In this way, every leaf is uniquely addressable, memory accesses can be pipelined on a fine-grained level,

and the structure provides the opportunity for very dense storage.

2.1 The Memory Macro and Router Macro

The H-framework provides an efficient means of organizing and accessing data stored in memory macros. The design of the memory macro itself takes advantage of the design landscape of this nanotechnology as well. The basic storage mechanism is a spiraled wire which acts as a shift register due to the inherent latching in QCA wires. With simple control logic, each bit in the memory macro is either ”refreshed” by being allowed to re-enter the spiral or replaced by a bit of a new data word during a write operation.

The router macro design also takes advantage of the characteristics of QCA, exploiting both the majority gate, the native computational gate of QCA, and the self-latching of wires, another fundamental property of QCA. The router allows the data parcel to go out both the left and right output paths, and makes a simple decision based on the appropriate address bit whether to send the select parcel down the left or right branch. The same router design can be used throughout the memory since the router itself needs no information about its whereabouts in the structure. Its function is a simple binary decision, left or right, and then taking advantage of the self-latching of the wires, the decision can be used until the entire parcel has passed through the router.

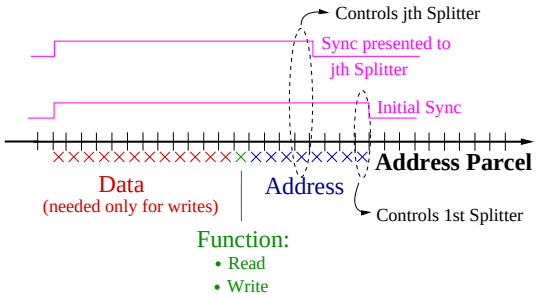

Figure 4. Address and select parcel relationship. The leading edge of the select parcel signals the bit on which to base the routing decision. At the j-th router encountered, the j-th bit in the address is referenced.

The return path from the memory macro back to the H-memory root is very simple. On a read, the data returns to the root along a route parallel to its leaf-ward path. If a memory macro is not being accessed, it outputs logic zero on each clock cycle. At each router, then, where two subtrees join, a simple OR operation can merge the two return paths that meet at that router. This straightforward strategy is reasonable because in the basic case in which the H-structure is a traditional memory with an external CPU, it is impossible for accesses to collide inside the H-structure since all leaves are a uniform distance from the root, all accesses start at the root, and all accesses take the same amount of time to execute at the leaf. Therefore, in the traditional memory case, no collision handling is required inside the memory.

The H-memory has the potential for very dense storage. For instance, the basic H-memory with one 64 bit word per leaf can store 1 GB in an area of $4.4\text{x}10^(12)$ cells, or $0.08cm^2$, assuming a 1.4 nm separation between the centers of two adjacent cells, consistent with a molecular implementation of QCA. However, in traveling from entering the root to the memory leaf, a significant latency cost in incurred. The latency depends on the number of routers and the length of ”wire” the parcels must travel through as well as the potential synchronization delay to guarantee the access arrives at the memory leaf in step with the constantly moving word stored in the memory macro. The number of routers an access must travel through grows as the log of the number of leaves in the structure, comparable to the height of a binary tree. The latency for memories greater than 1 MB, though, is dominated by the wire length which grows on the order of the squareroot of the number of leaves. To reach the start of a word in this gigabyte memory from the root would require 882 clock cycles. With a 1 THz clock, this would be less than 1 nanosecond (.88 ns). This timing is competitive with current memories whose access time is on the order of tens of nanoseconds, but this is only a single order of magnitude improvement despite the terahertz clock. However, the QCA H-structure offers several possibilities to improve this time, such as taking advantage of the fine grained pipelining available. A more exciting alternative is to eliminate the need for the latency by moving the functionality of the CPU into the memory structure itself, distributing the functions among the routers. This eliminates the latency by removing the need to travel long distances between a central processing unit and the data to

There are several straightforward techniques that can be applied to mask this latency. Foremost is the opportunity for the fine-grained pipelining of accesses into the memory itself. Also, there are alternative leaf configurations that allow storing more than one word per leaf that can minimize the number of leaves needed to store a certain capacity, and so reduce the length of wire an access must travel through. In addition, techniques currently employed to mask memory latencies can be applied, such as implementing a memory hierarchy and caching, and implementing a ”sub-tree” mode read analogous to page mode read.

The most exciting possibility for lessening the effect of the long latency lies in eliminating the need to travel from and to the root on every access by distributing the functionality of the CPU throughout the H-structure itself. This eliminates both the bottleneck of the central processing unit and the long latencies involved with traveling between the CPU and memory. A new execution model, one ”native” to the H-framework, can take fully explore and exploit the H-memory.

3 Execution Model

The most exciting possibility for lessening the effect of the long latency lies in eliminating the need to travel from and to the root on every access by distributing the functionality of the CPU throughout the H-structure itself. This eliminates both the bottleneck of the central processing unit and the long latencies involved with traveling between the CPU and memory. A new execution model, one ”native” to the H-framework, can take fully explore and exploit the H-memory.

Merging processing and memory in such a close fashion will require new programming and execution models. The ”H” structures will require an execution model which takes advantage of characteristics of QCA. One such execution model is a ”bouncing thread” model in which computation is comprised of a large number of threads. The state of these threads is small enough that it can be carried with them as they move through the H structure. Synchronization is provided at the word level and control is distributed through the H structure.

A ”Bouncing threads” model is well suited to QCA because it takes advantage of the self-latching qualities of QCA cells. Because a QCA ”wire” can hold the state of a thread, we do not need to refer to memory to recall thread state or construct an expensive centralized register file. This lack of centralized structures allows support for a large number of threads executing in parallel, thus concealing latency. Finally, because logic is inexpensive, it can be replicated through the H structure.

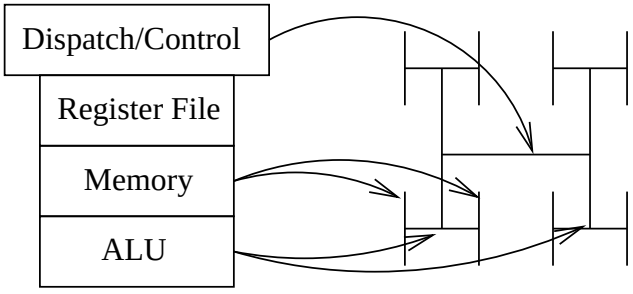

Figure 5. Distribution of the CPU functionality throughout the internal nodes of the H-structure

To support this execution model we integrate arithmetic logic and control into the routers of the H memory. The operations performed by each router on a thread traveling in an H memory are similar to the operations performed by each pipeline stage on an instruction traveling through a classical CMOS processor. A thread which has just fetched a new instruction may have that instruction decoded at one router, be directed to the proper memory word by the next router, and fetch the relevant data from that word into its carried state. On the return path, one router may perform an arithmetic operation on the thread’s stage and the next router will update the thread’s program counter and begin directing it to fetch the next instruction. Unlike a classical pipeline, however, control, execution and memory are woven together. There is no equivalent to a ”register fetch” stage, because thread state is carried with the thread, not stored in centralized register files.

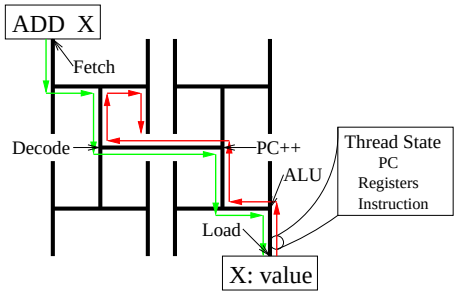

Figure 6. How a thread travels throught the memory from instructions to data.

The key characteristic of QCA technology which necessitates bouncing threads is locality. Because latency is directly proportional to distance, having single centralized structures, such as a traditional register file, would require constant high latency transfers of data to and from the centralized structure. With a bouncing thread model, we move the thread state to the data, avoiding the need for any centralized structures. Because QCA wires are self-latching, transmitting the thread state does not require additional storage structures.

Because the control logic for bouncing threads is distributed and stateless (the state being a part of the thread itself rather than being maintained separately), it is possible to support large numbers of threads with little overhead. Instead of a central processor scheduling threads and managing context switches between them, threads are physically spread through the machine. Multiple threads can be executing at the same time, and the ”scheduling” of execution units is performed naturally by the routing logic. Additionally, the design of the memory macro ensures that all memory accesses are atomic, making synchronization easier.

4 Specific Implementation of Execution Model

The ”bouncing threads” execution model supports a wide range of possible implementations. One ISA in particular that we studied, called Simple24, has the appropriate mix of simplicity and functionality to aptly demonstrate the bouncing thread execution model. Simple24 is a bare-bones accumulator based ISA. In addition to the standard memory, control, and arithmetic instructions, Simple24 has support for fine grained multithreading and synchronization. It is constructed to demonstrate the fundamentals of the bounc-

ing threads model, so it lacks support for virtual memory, interrupts, and other characteristics of a full featured processor.

4.1 ISA variants

An accumulator based ISA was chosen because it simplified decoding and control logic and allowed a very small thread state. Other ISA models, such as VLIW or stack-based, would require more complex decode and control logic or would require thread state to become unmanageably large.

4.2 Thread State

The state associated with each thread which must be carried consists of an accumulator and immediate value, the opcode of the current instruction, an address, and a program counter. The accumulator and immediate value are the only programmer visible state. The address register is used to route the thread through the H-structure. The opcode and program counter are used for control of the program.

The accumulator based ISA and minimal amount of thread metadata allows for a small thread state. This organization of data is well suited for the serial decoding found in the H-memory. This minimal overhead will enable mulitithreading at a

4.3 Thread Control & Synchonization

Thread control and synchronization is provided through a few special instructions which are uniquely suited to the H-memory.

A new thread is created with a FORK instruction. A FORK instruction creates a new thread at a memory node. This new thread contains the same state as the parent thread, except its accumulator register is set to zero. By following a FORK with a branch, behavior similar to the POSIX `fork()` system call is achieved. This allows thread creation very low overhead.

Because of the physical design of the H memory macro, all memory accesses are atomic. To allow low overhead synchronization it is easy and efficient to implement a simple empty/full synchronization scheme at each word. In this scheme, each memory work is equipped with an additional synchonization bit, initially set low. A variant of the STORE instruction, PUT, performs a STORE and also sets the synchonization bit high. A varient of the LOAD, GRAB will only load data if the synchonization bit is high. If the synchonization bit is low, the GRAB instruction will block until anothe thread sets the synchonization bit high.

5 Component Sizing

The memory macro and basic router macro have been designed and their operation simulated providing a reliable frame of reference for the sizes and latencies of components. In the basic memory with an external CPU, two components are needed, a memory macro and a simple router macro. The memory macro consists of two elements: the basic storage element which is a wire spiral, and the control logic. For the bouncing threads execution model and the ISAs explored, four additional types of routers are needed. They are a center router, the decode router, the PC-inc router, and finally an execute router, not needed for the programs simulated. The center router is responsible for thread dispatch and communication with devices outside the H-memory. The decode router is comparable to the decode stage in a simple pipeline. The PC-inc router increments a thread's program counter, and the execute router would perform complex ALU operations. In this framework, the memory macro would change only minimally, adding a full/empty bit to guarantee atomic operations at the memory leaves, and some additional logic outside the memory leaf to reconstitute the thread after a memory access.

5.1 Traditional Memory Components

The basic memory macro consists of the spiral and the control logic. The control logic occupies an area of 1914 cells. The size of the spiral depends on the number of bits it holds and grows as a function of the square root of the number of bits stored.

5.2 Normal Router

The simple router is 56 cells along its widest side which dictates the width of the gap between successive portions of the recursive structure. For the bouncing threads execution model, a more sophisticated return path is needed to allow the thread to go either up or down the tree. A first cut router incorporating routing on the return path approximately is approximately double the size of the simple router. This design incorporates three of the simple routers (one router going down the tree, and one router for each of the returning subtrees) and by sharing clocking zones, only doubles the size of the router. However, this naive approach can be substantially improved upon. This router incorporating the route-on-return is termed a "normal" router.

5.3 Center Router

The center router, responsible for I/O and initial thread dispatch has the potential to become rather complex. However, there is only one center router, and it has available to it the area from the center of the memory out to the edge of the memory. Since this space is not occupied by anything else, the center router can expand into it without penalty. Also, there is only one center router. For a reasonable sized H-structure, the size of the center router is insignificant.

5.4 Decode Router

The decode router performs a relatively simple function whose exact nature will depend on the specifics of the ISA being implemented. Its function will be analogous to the decode stage of a simple pipeline, which could mean directing the thread to the appropriate functional router (e.g. the execute router) or it could be just a normal router if the op-code itself acted as the directions to the appropriate router. The serial nature of the router and function keeps the area requirement low. It is reasonable to assume the size of this router would be similar to the normal router, and in the worst case certainly no more than twice its size.

5.5 PC Increment Router

The PC-inc router logic is a simple serial addition. The few gates needed to implement this function can be incorporated into the empty space available in the router taking advantage of the clocking zones already in place. It can reasonably be expected to be the same size as the normal router.

5.6 Bouncing Thread Memory Macro

Finally, the implementation of the bouncing threads model requires additional logic around the memory macros to maintain the thread state during memory accesses and to reconstitute the thread on completion of the access. The main function of this logic is to store the parts of the thread state not directly involved in the memory access. The size of the required logic will be roughly a function of the size of the thread state. The functional aspect will be quite simple, being to precisely synchronize the result of the memory access and its addition to the thread state. Assuming only one word is stored in each memory macro, the overall size of the bouncing thread memory macro can be estimated as being no bigger than twice the size of the basic memory macro. For larger words, such as 128 bytes, this will be an overestimate, but it is a comfortable upper bound.

5.7 Comparison

To simplify the calculation, all of the bouncing thread routers were considered to be twice the size of the normal router, and each bouncing thread router takes four additional clock cycles to travel through. This accounts for the added complexity at some of the routers, and is again an overestimate since for the larger memories, the bulk of the routers will be normal routers.

To form a valid comparison, the access time is calculated as being the time for a request to travel from the root to the memory macro, assuming the worst synchronization case in which the request has to wait a time equal to the number of bits stored in the spiral to arrive at the memory macro in sync with the start of the stored word, and back to the root. The area grows faster for the bouncing threads H-structure since the initial seed is larger. Even with the added complexity, the access time for the bouncing threads memory is roughly a constant ratio slower than the basic H-memory (figure 8). A 24 bit word was used in the simulations of the execution model. For the same sized memory, a larger word is more efficient in terms of density and latency since fewer memory macros are needed, and the tree is shallower. The latency is in the tens of thousands of cycles. However, a clock rate on the order of one to ten terahertz is expected for the molecular implementation. In addition, the bouncing threads memory will on average avoid this long latency by taking advantage of the locality of data and instructions.

Because of the added complexity at the added complexity at the routers, the H-structure that supports the bouncing threads execution model has a greater area that depends mainly on the word size. For the 24 bit word, the memory is roughly 40% larger, while the 128 bit bouncing threads memory is 30% larger than its basic H-memory counterpart. For smaller sized words and larger memories, the area of the memory is dominated by the size of the routers. Even with the inefficient 24 bit word, the bouncing thread H-memory exceeds the projected density of CMOS SRAM through the end of the roadmap, and exceeds the DRAM projection past 2012. The 128 bit word bouncing memory exceeds the density of DRAM past the end of the roadmap (figure 7).

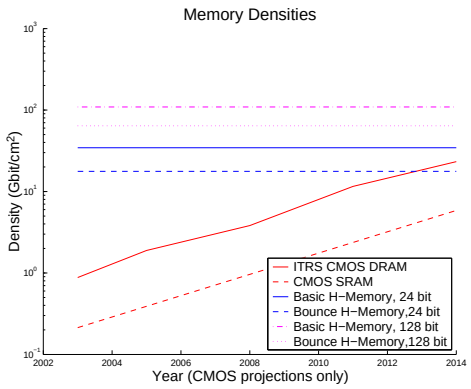

Figure 7. Potential densities of the basic H-Memory, bouncing threads H-Memory and projected CMOS DRAM and SRAM.

6 Simulation Results

To test and demonstrate the operation of the execution model, a simulator was built to explore design optimizations and several ISA configurations. The simulator allows the thread paths for an actual executable program to be seen.

Three ISAs were implemented on this architecture. The first ISA is Simple12, a simple accumulator based ISA. The second ISA implemented was stack-based. Finally, the simulations culminated in the Simple24 ISA.

Simple12 is a simple, easily implemented ISA which

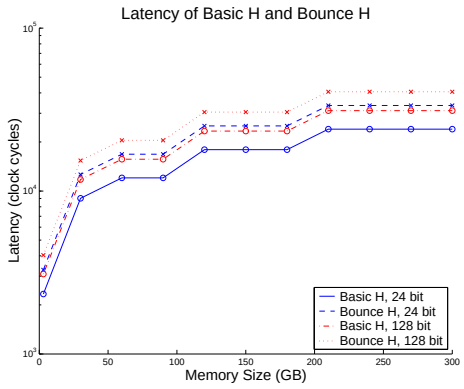

Figure 8. Access time in clock cycles. The clock rate for a molecular implementation is projected to be on the order of 1-10 THz.

allows straightforward implementations of enhancements such as microthreads and an ”instruction cache”. However, it also has several significant limitations such as supporting only one accumulator, allowing self-modifying code, and lacking support for the use of immediate values.

Due to the limitations of Simple12, a stack-based ISA was explored. A stack-based ISA would be a good choice to keep the size of the thread small and the logic efficient. It allows for a larger address space to be tested along with testing some other design ideas such as vector operations and severe microthreading. The primary limitation of this ISA, however, is that multithreading is difficult when dealing with a stack. Since there is no operating system to control memory management, implementing multithreading with a stack in hardware would require too much logic and would undermine the simplicity of this execution model.

The final ISA, a modification of the Simple12 ISA tentatively named Simple24, better supports multithreading. Among the modifications are: increasing the word size from 12 bits to 24 bits, increasing the address space to 16 bits (64 kB), and allowing different addressing modes. The feature of different addressing modes is key to multithreading because it eliminates the need for self-modifying code and can achieve much of the functionality of having an index register.

A maxfinder program was primarily used as a benchmark to test all of the ISA’s as well as several enhancements of the Simple12 ISA. The ”I-Cache” feature allows several instructions to be prefetched so that thread time in the instruction part of memory is reduced. The ”I-cache” is treated simply as additional thread state. A slight modification of this is a ”Smart I-Cache” where the prefetching stops at a jump or branch so that unused instructions are not fetched, avoiding carrying excess state with the thread. The ”Write Through” feature was the first implementation of ”microthreading.” Since stores to memory do not affect the thread state, this feature sent off a separate thread to the memory leaf to execute the store and then terminate itself. This freed up the main thread to continue with the program execution. Finally, the ”Router Configuration” allowed different types and arrangements of routers to be examined, exploring what functions to place in which routers.

In the simulations, several variables were explored including functional router placement, I-cache size and type, and the use of write through microthreads. The first cut simulation took just over 8000 cycles to execute. The final simulation with a 20 instruction I-cache, write through microthreads, and the advanced routers that could route threads both up and down the tree took just over 3000 cycles. This can be even further improved upon since the program’s data and instructions were spatially separated, and the thread had to travel far up the tree to go from data to instruction. The thread passes through many routers as it travels, which can be beneficial but results in longer latency. On the the one hand, passing though several routers allows for much of the logic to be distributed amongst these routers, resulting in simple logic at smaller and faster routers, allowing a ”superpipelined” model to be approached. To offset the longer latency, it may be beneficial to research a creative integration of program data and text to reduce how far up the tree threads must travel.

It is important to note that due to the recursive structure of the H-framework, the size of the memory does not effect the operation of the ISA since a thread usually operates in only a portion, a sub-tree, of the memory. While the maxfinder program is a relatively simple one, it is complex enough to indicate general strategies. With more complex programs, the benefits of microthreads such as the write through function and the I-cache will become even more apparent. Reducing the running time of this maxfinder to less than half the original time demonstrates the power of the fine-level pipelining and multithreading available with the H-Memory and the bouncing threads execution model.

7 Other Work and Conclusion

This is an exciting execution model that offers a wealth of variations and enhancements. Some of the variations being initially explored were mentioned briefly such as making use of microthreads to execute stores or as a pre-fetch mechanism, incorporating sub-tree operations analogous to vector operations, implementing instruction caches with various protocols, and takind advantage of the relative ease of explicit multithreading in this execution model.

In addition, the process that led to this execution model, that of defining the design landscape (the characteristics of a single device and a collection of devices), illustrates the value of explicitly designing to exploit the characteristics of a particular device. Not only does this lead to efficient designs, but it leads to architectures and execution models that address problems faced by traditional architectures.

References

[1] S. C. Benjamin and N. F. Johnson. A possible nanometer-scale computing device based on an adding cellular automaton. *Applied Physics Letters*, 1997.

[2] S. E. Frost, A. F. Rodrigues, A. W. Janiszewski, R. T. Rausch, and P. M. Kogge. Memory in motion: A study of storage structures in qca. In *1st Workshop on Non-Silicon Computation (NSC-1), held in conjunction with 8th Int. Symp. on High Performance Computer Architecture (HPCA-8), Boston, MS*.

[3] K. Hennessy and C. Lent. Clocking molecular quantum-dot cellular automata. To appear in the Journal of Vacuum Science Tech.

[4] ITRS. International technology roadmap for semiconductors 2000 update. Technical report, ITRS, 2000.

[5] C. Lent. Molecular electronics: Bypassing the transistor paradigm. *Science*, 2000.

[6] C. S. Lent and P. D. Tougaw. A device architecture for computing with quantum dots. *Proceedings of the IEEE*, 1997.

[7] A. O. Orlov, I. Amlani, G. Toth, C. S. Lent, G. H. Bernstein, and G. L. Snider. Experimental demonstration of a binary wire for quantum-dot cellular automata. *Applied Physics Letters*, 1999.