Towards a Universal Building Block of Molecular and Silicon Computation

Steven Swanson and Mark Oskin
Deptartment of Computer Science and Engineering
University of Washington

Abstract

The next two decades will bring us silicon computation resources that seem unimaginable today. Advances in lithographic technology will enable devices with ten billion transistors and beyond. Surprisingly, the challenges that face traditional silicon computing are eerily similar to those molecular and other nanoscale devices must confront. The solution we propose for scalable silicon, the computation cache, is adaptable to molecular devices. A computation cache is a uniform substrate built from a simple tile. Each tile encapsulates the functionality to compute a single instruction and self-organize with other tiles to execute entire applications.

This tile serves as a key architectural link between the silicon and non-silicon domains. The network connecting tiles need not be uniform and we envision encapsulating these tiles in specialized packages and bonding agents to form "computational paint." In the longer term, our research into the minimal tile for substrate construction has implications for molecular devices; e.g. if a molecule can be synthesized with this minimum functionality a molecular computational paint will be realizable.

1 Introduction

Computer architects are facing three daunting trends with silicon devices. First, relative to computation speeds, on-chip communication is slowing down. Second, as feature sizes shrink, manufacturing processes are becoming less reliable. Third, the pace at which additional computational realestate becomes available is far outstripping the rate at which designers can effectively utilize it. We foresee analogous challenges with forthcoming nanoscale and molecular devices. Nanotechnology will bring enormous quantities of computational resources, yet data transfer will continue to hinder scaling. Chemically self-assembled devices will, almost by definition, require fault tolerant architectures. Finally, there is no practical way that designers will layout every nanoscale switch, wire, and bond; hence regular, repeatable designs will be required.

For silicon-based computing, we propose an entirely new class of microprocessors, called *computation caches* that directly confronts future technological challenges. A computation cache is best described as a mixed data-instruction cache that performs processing in-place. Traditionally, designers devote a significant portion of processor resources to extracting instruction level parallelism and tolerating memory latency to maximize the performance of a central core. However, if you take the real estate devoted to register renaming, register files, instruction issue and completion you could build a small processor at each word within the instruction cache. If you can effectively use that computational structure, why have a centralized high-performance core at all?

A computation cache is a hardware algorithm for distributed node-based processing. Each node corresponds to a word in a traditional instruction cache and combines a computing engine with network interfacing logic. Similar to a traditional cache, the computation cache fetches and places instruction streams across nodes. A critical difference, however, is that instructions in the computation cache execute in place rather moving to a traditional processor for execution. Once instructions complete execution they explicitly send their results to dependent nodes and remain in place to exploit potential temporal locality. The computation cache only fetches and decodes frequently used instructions once but executes them many times. When instructions are no longer needed, they are replaced – exactly like a conventional instruction cache.

In this paper we outline the computational cache architecture and execution model. While our current effort is motivated towards creating scalable silicon-based systems we believe the core results of our research – the algorithms and techniques for instruction placement, node design, computation and communication will be of interest to the nanoscale device research community. While our two domains (silicon and non-silicon) are currently orders of magnitude apart in terms of scale, they are approaching each other. This raises many exciting research possibilities for both disciplines.

The first part of this paper (Sections 2 and 3) will outline the silicon-based architecture we are currently designing. Section 4 discusses the convergence of scaling trends between silicon and non-silicon devices and how the computation cache architecture applies to both. We follow this with Section 5 which describes some of the modifications necessary for molecular level implementation. Next in Section 6 we discuss related architectural work, and in Section 7 we conclude.

2 Computation Cache

A computation cache (CC) is an hardware algorithm we are developing at the University of Washington for scalable single or multithreaded instruction execution. Just as there are many ways of realizing a Tomasulo-based [1] out-of-order

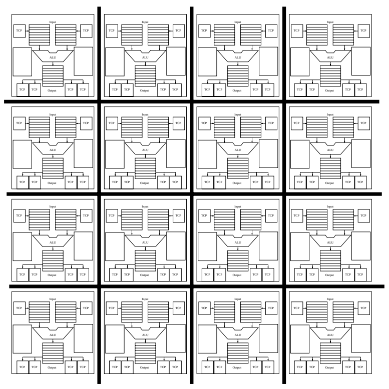
Figure 1: Scalable Tiled Architecture

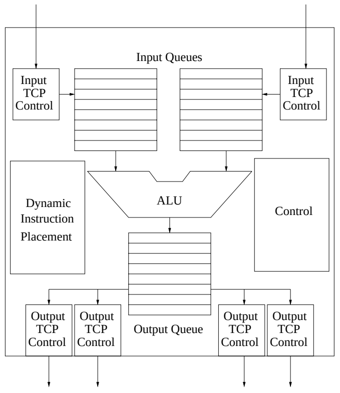

Figure 2: Silicon-based Tile

execution core, there are many possible computation cache-based architectures. In this section we describe our early results in developing one such computation cache architecture. The architecture is based around a uniformly constructed two-dimensional mesh of computation nodes. Figures 1 and 2 depict a small section of this mesh and the node, respectively. Each node has four key components:

- The functional unit.
- An interface to the chip network.
- Operand input and output queues.
- Dynamic configuration logic.

We will discuss each of these in turn.

2.1 The Functional Unit

The computational cache functional unit is a basic 64-bit arithmetic logic unit. It supports the conventional operations such as ADD, SUB, OR, etc., and we expect it to operate within a single clock cycle. The functional unit consumes operands from the input queues and places results in the output queues. On any cycle it can produce a single result if all of the needed operands are present in one corresponding set of slots in the input queues, and there is space in the output queue. Precisely what operation the functional unit performs is determined from the dynamic configuration logic that we describe below.

2.2 Chip Area Network (CAN)

The Chip Area Network (CAN) is a switched network that connects all of computational cache's components. The CAN provides delivery of operands, instructions, and control packets. Addresses on the network consist of a location in the computation cache and a port number (e.g. the operand queue number). Each tile has an input and an output interface to the CAN. While our current investigation will focus on an in-order delivery reliable network infrastructure this is not critical to the implementation of the computational cache hardware algorithm. In fact, central to supporting future process technologies will be supporting unreliable and un-ordered network delivery.

2.3 Operand Queues

Each tile contains input and output operand queues. The queues provide buffering between dependent nodes. In our design there are three input queues: two for the arguments to the instruction and one for control messages. The contents of the three input queues are always synchronized: The nth entry in each of the queues will combine to produce the nth result value. Keeping the queues synchronized is simple using either an in-order delivery network, or simple packet sequence numbering.

The latency for computing a result and delivering it to its destination can vary widely due to congestion in the network and the relative location of the instructions. Operands may arrive at one input queue more quickly than the other. The operand queues help hide these inconsistencies by buffering operands. Our initial design uses 16 entry queues.

Nodes in the computational cache communicate with each other using a protocol similar to TCP windowing. When the execution begins, all operand queues are empty, and a tile, A, can send 16 values to another tile, B, without fear of overfilling B's input queues. As B consumes values, it sends acknowledgments (ACKS) to A so it can send more values. Many optimizations that TCP uses are applicable, including selective acknowledgments.

The queuing mechanism also provides an elegant and completely distributed way to stop computation in the cache. If the tiles stop sending ACKs, computation will gradually halt.

2.4 Dynamic Configuration Logic

Unlike a traditional instruction cache, the computation cache does not use a simple hash of an instruction's address to decide on the location of an instruction within the overall mesh. Instead, instructions are initially placed anywhere within the

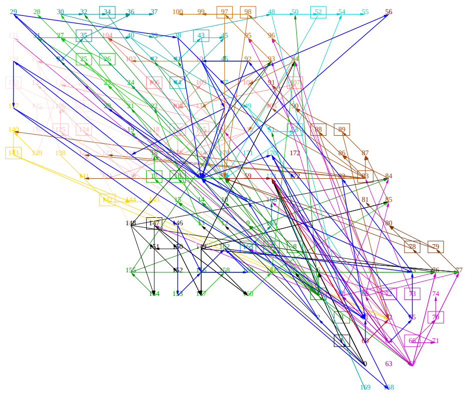
Figure 3: Chaotic decentralized placement

Benchmark	Ideal	In-order	Block-greedy
bzip	4.2	1.9	1.1
crafty	90.4	12.4	13.9
mcf	22.9	12.6	17.7
parser	19.9	8.8	10.7
vpr-place	134.8	68.1	72.4

Table 1: Initial evaluation of cost of chaotic dynamic placement. Here an "infinitely connected" idealized placement is compared against our dynamic placement algorithms. The numbers represents the maximum available IPC given a particular scheduling algorithm.

mesh, and use simple heuristics to optimize placement during computation based on the spatial locality of instruction dependencies and observed program behavior. Each tile includes logic for this automatic configuration process, including support for exchanging information with adjacent tiles and forwarding instructions to their final destination.

The configuration hardware will be flexible enough that multiple configuration algorithms could be used on the same chip. By using different algorithms, parts of the computational cache could specialize themselves for vector operations or loops while others might optimize for operating system activities.

Placing instructions dynamically, without explicit instruction from the compiler, is central to being fault tolerant, but potentially detrimental to performance. Because the goal of the computational cache is scalable execution performance, there can be no large, globally accessed structures. Consequently, the placement algorithm the computation cache uses must only use local information. The algorithm running on a node will only have information about the local node, its neighbors, and the nodes it exchanges operands with. If such algorithms do not exist, the computation cache will never achieve its goal. Thus our initial investigation has focused on simple hardware algorithms for dynamic placement, and whether they can lead to reasonable performance.

Figure 3 depicts the placement of a function (from *bzip2*) within a mesh of nodes by a simple stateless algorithm. We quantified the impact of this process and how much it will slow down execution relative to an idealized placement for various applications. As yet, we do not know the performance relative to a realistic "ideal" placement into a computational cache. However, we can make a highly pessimistic assessment by comparing our placement algorithms to a completely connected mesh of computation, so instructions always receive data from adjacent tiles (think of this as a computational cache existing in a highly-dimensional physical space). Table 1 illustrates our results for a subset of the SPEC2000 benchmark suite. The data demonstrate that choosing the best placement algorithm for an application yields a 57% performance penalty from an "ideal" schedule. Choosing the best overall placement algorithm (*block-greedy*) for all applications yields a 61% performance penalty. Work is ongoing to better assess the performance impact of dynamic placement against a realistic "ideal."

3 Execution Model

The computational cache executes programs by passing messages that contain operand values and control information between dependent nodes. There are no registers and, therefore, no notion of a named value (except via memory references). Nor is there a notion of "program order"; instructions execute when their operands are available – whenever that may be.

This execution model has much in common with previous dataflow machines. However, dataflow designs traditionally rely upon a centralized result buffer to store tagged intermediate values. The constraints that future process technologies will place on communication across a silicon device make these structures infeasible and hence require a radically different design.

As a result, program execution within a computational cache is far more complicated than execution on a traditional dataflow or Von Neumann processor. Traditional processors encode data-flow information in register names and control information in branches and destination addresses. The computational cache manages control-flow and data dependencies explicitly. The execution model must, therefore, include these aspects of the program as first-class entities.

To demonstrate how a computational cache actually computes, we will illustrate execution with a simple function. Figure 3 contains C source code for a function that interleaves the elements of two arrays. Figure 5 contains the data and control flow graphs for this function. A binary encoding of these graphs is the executable format for the computation cache. The next sections describe control-flow and data-flow in the computational cache execution model and a discussion of how they operate in the example.

```
for(i = 0; i < 10; i++) {
    if ((i % 2) == 0) {
        c[i] = a[i >> 1];
    } else {
        c[i] = b[i >> 1];
    }
}
```

Figure 4: Source code for the example.

3.1 Control and Speculation

The computation cache uses three kinds of *control messages* to manage control flow and speculation: *enable*, *bless*, and *poison*. Conceptually, the messages pass from one basic block to another. An enable message signals the basic block to start executing speculatively and establishes a contract between the sending and receiving block. The contract ensures that the sending block will eventually send a corresponding poison or bless message.

A poison message signals that the basic block executed on a wrong path, while a bless message signals correct path execution. In either case, the basic block must pass these messages along to follow up on any enable messages it sent to other blocks. In this way, computation moves through control flow graphs in two "waves"; an enable wave to initiate speculative computation, followed by a poison-bless wave to commit the results. In principle there is no limit to how many enable message can be outstanding at one time, but in practice hardware limitations bound the number of outstanding enables.

The tiles in the computation cache do not manipulate basic blocks and the instruction stream does not explicitly encode block boundaries. Instead, each tile has additional inputs and outputs for the control flow messages. The enable, poison, and bless messages flow from instruction to instruction. Each basic block has two distinguished instructions, the *control head* and *control tail*. The control head is responsible for distributing the messages to all the instructions in the block. The control tail passes messages onto the control heads of the subsequent basic blocks.

To reduce the number of messages the control head sends, we allow the control messages to piggyback on operand messages. The control head only needs to send control messages to enough instructions to ensure that all the instructions in the block will eventually see them.

3.2 Instruction Execution

Instructions in the computation cache operate on both control and data values. Thus each instruction contains two parts: a control operation and a data operation. Data operations correspond to normal computations (ADD, SUB, LOAD, etc.). They consume values from the input queues and produce a value in the output queue.

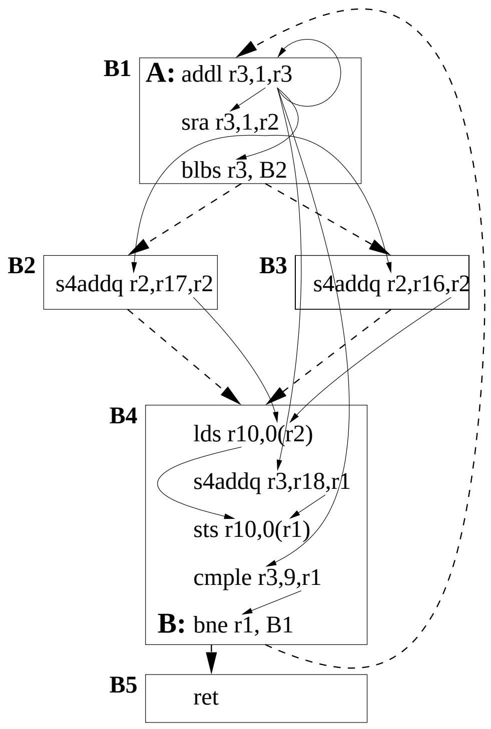

Figure 5: The data- and control-flow graphs for the example. Data-flow edges are solid. Control-flow edges are dashed.

The control operation in an instruction depends on its place within a basic block. Control head and tail instructions use operations that distribute control messages to other instructions in the block and the subsequent control heads, respectively. The remaining instructions simply forward control messages to the instructions that consume their results.

Branches are always control tails and use a special version of the tail operation. They immediately forward enable messages to the following control heads when the enable arrives. If a branch receives a poison message, it poisons both sides of the branch. For a bless message, it blesses the taken path and poisons the non-taken path. This means branches are the only instruction whose data and control operands interact.

The control and data operations execute independently of each other. For instance, a control head instruction might distribute control messages before both operands for the corresponding data operation have arrived.

3.3 Discussion

The computation cache execution model leads to some interesting behavior. For instance, instruction A (in Figure 5) depends only on itself. On the first iteration, it will receive the initial value and enable from the control tail of the previous block (not shown). Next, A will fire producing the first index value. The enable message flows down both sides of the branch and will return to instruction A via a back edge. Then instruction A produces a new value.

It is likely that the enable message will move much faster than the operand values because no computation is required (careful compiler analysis can further reduce the time needed). As computation proceeds, the branch (instruction B) receives a bless message and the direction of the branch will become known. For the first 10 iterations, the branch sends a bless message to A, and poisons the instructions in block $B5$. Instruction A, in turn, blesses its results.

Eventually, the branch will fall through and then B will poison instruction A. The poison message will circulate until it poisons all the wrong-path executions.

4 Silicon vs. Non-silicon

The computational cache is a hardware algorithm designed for lithographic silicon technology that will not be available for another decade. Hence, details such as whether a node's input queues are 16 or 32 entries are not the type of minutia we are focusing our attention on at the moment. Our goal is to develop the scalable algorithms and execution models for a silicon process technology that is radically different then what is available today. As we progressed in this work, we soon realized a convergence of properties between future lithographic silicon and emerging nanoscale devices.

Despite the fact that even future generations of lithographic silicon will be orders of magnitude larger than nanoscale molecular devices, architectures for both technologies will need similar characteristics to be successful: They must be relatively simple to design, they must be scalable in the face of expensive communication, and they must be fault tolerant. We address each of these issues in turn, explaining why they are important and demonstrating that the computation cache paradigm meets all three constraints in both silicon and non-silicon domains.

4.1 Simple Computing Elements

Future silicon technologies will provide so many transistors that our ability to engineer them correctly or even find uses for them comes into question. Signs of this trend already surround us. Modern designs contain ever-larger on-chip caches and design schedules frequently slip due to verification problems. Furthermore, while transistor densities consistently track Moore's law, manufacturers' ability to engineer complexity is far less, leading to the so-called *productivity-gap*.

Engineers working with nanoscale devices face the same complexity limitations, albeit for somewhat different reasons. The nature of chemical assembly will keep the complexity of molecular devices low. For instance, reliably synthesizing a single molecule the size of a modern processor is infeasible. Self-assembly will mean that precise placement of different types of molecules in irregular patterns will be difficult if not impossible. Hence, initial architectures for nanoscale devices rely upon regularly constructed substrates [2, 3].

For silicon and nanoscale devices, the solution is the same: build a large number of simple computing elements that combine into a scalable computing substrate. The small, individual elements will be simpler, easier to verify and less difficult to produce. The computation cache takes exactly this approach with its sea of only a handful of different tile types.

4.2 Expensive Communication

Silicon devices have entered the era of expensive communication. While transistors and computation are becoming nearly free, the relative cost of communication increases steadily. Since the speed of light limits communication across a chip, it is unavoidable that long wires on the critical path will soon become prohibitively expensive in terms of either clock cycles or clock speed.

While molecular nanoscale devices will provide an enormous amount of computing power per unit area, utilizing this resource effectively will require sufficient communication capability. While dense computation “macro blocks” that require little outside communication, yet still compute a sufficiently complex function may be realized, precisely how useful they will be for real applications is unknown. It may be that future non-silicon systems will have computational density to communication latency characteristics similar to silicon. This will particularly plague those systems that rely upon lithographic scale interfacing logic.

The computational cache does not eliminate long-distance communication (sometimes data must cross the entire chip), but it does remove it from the critical path for cycle time. Our design contains none of the large associative structures common in modern CPUs (e.g. instruction queues and renaming logic), so the computation cache can run at high frequencies. Also, in the common case, operands in the computation cache need travel only a short distance, often reaching their destination in one or two cycles. This is a substantial savings compared to the three or four cycles needed to read from the register file in next generation processors.

Furthermore, the computation cache does not require *any* long wires. While “long-haul” connections would provide improved performance for the CAN, they are not critical to the design. The hardware algorithms require only nearest neighbor communication.

4.3 Fault Tolerance

Designing with smaller transistors and thinner wires means more manufacturing defects and increased susceptibility to cosmic rays and other kinds of interference. Already, processors include error correcting codes (ECC) on memory structures throughout the chip, and fault-tolerant architectures are an active area of research [4].

Non-silicon technologies face similar challenges for two reasons. First, defects are unavoidable in many non-silicon technologies due to the self-assembly manufacturing process. Second, it is conceivable that the imprecise manufacturing process will create irregular interconnect topologies. A fixed mapping of computation onto nanoscale devices thus becomes impossible in such a setting. Current proposals use a post-manufacturing application compilation phase to handle this [2]. Even these approaches are "static" and rely upon redundancy at the molecular level to overcome transient faults. The architecture itself cannot re-map computation in a dynamic, ad-hoc fashion.

The key to the computational cache's tolerance of faulty hardware is the dynamic code layout that precedes computation. The mapping of instructions onto processing elements is not fixed, so the instruction placement algorithm can ignore defective tiles and computation will just flow around them.

In the next section we discuss our preliminary ideas on the modifications to the computation cache architecture previously described to support nanoscale chemically self-assembled devices.

5 Molecular Implemention

We believe (quite optimistically, perhaps) that the design of conventional architecture and the design of nanoscale devices will converge. That is, one day, we will be able to chemically assemble a complex molecule designed for computation. Currently, researchers can engineer and assemble a molecular switch. If they succeed in expanding this capability several thousand times in numbers and complexity, just as nature started out small and arrived at the diversity of life we see around us, building a molecular tile for the computation cache would be feasible.

In this section we explore this possibility. Implementing a tile in a single molecule will require modifications to our base design. There are several ways to reduce the complexity of the tile to make it easier to synthesize. We discuss two: narrow bit-width design and function-specific tiles. In addition we sketch a design for a randomly constructed CAN.

5.1 Narrow Bit-width Design

To be useful for modern applications the computation cache must use 64 bits as its architectural word length. Internally, however, its functional units can operate on any width. A computation cache with elements that operate on small (4 or 8 bit) slices of 64 bit words would be simpler in two areas: communication and computation. In both cases, we can vary the slice width to trade off complexity for increased latency.

Narrow communication paths will be necessary in any non-silicon designs that include a completely ad-hoc interconnection network. Full 64 bit parallel communication between tiles will be nearly impossible in such a system because the probability of randomly creating 64 connections between two cells is almost nil. However, the chances of creating one or two connections are quite good.

5.2 Function-specific Tiles

Because they perform a multitude of operations, functional units would constitute a large share of the complexity in a molecular computation cache tile. Since the cache configures tiles once and leaves the instruction in place to re-execute many times, we can simplify the molecular functional unit by having a specialized molecule for each operation. There are at least two models for a molecule-based computational cache using specialized function-specific tiles.

In one scheme, the dynamic configuration process would leave each tile with an operation-specific receptor. A solution containing all types of functional units would flow over the cache and the appropriate functional-unit would bind to the tiles that need them.

Alternately, we could specialize the entire tile instead of just the functional unit. Since dynamic configuration would be very difficult with fixed-function tiles, an alternative programming method is required.

Borrowing from biology, a strand of mRNA could represent a program. A process similar to the translation of mRNA into proteins would translate mRNA into a component of a computation cache. Each sequence of 4 bases (codon) would represent a different operation. Molecules similar to those that assemble proteins would assemble computational "proteins," strings of tiles that would interconnect to form the computational cache.

Both of these schemes sacrifice one of the computational cache's greatest strengths: reconfigurability. Reprogramming would require dismantling significant portions of the cache and then reassembling them. This would probably be a lengthy process. In spite of this, both methods could still construct special-purpose molecule-based computing devices. Indeed, given recent advances in handling large biological molecules it might one day be possible to synthesize a specialized molecular computing device with a table-top machine.

5.3 Random Interconnect

Our initial design for a computation cache uses a rectilinear grid of tiles. The interconnect in this design is fairly simple, since the coordinates of a tile provide unique names and make routing almost trivial. Furthermore, we can rely on many years of research into interconnect fabrics for guidance.

Such an orderly structure may not be possible in a nanoscale molecular implementation. Tiles may or may not be connected to all of the neighbors, the number of neighbors could vary, and connections may fail unexpectedly. Fortunately, a similar problem is already under careful study.

Routing in wireless sensor networks shares many characteristics with routing in this randomly connected CAN. For instance, in both cases connections are unreliable and may change unexpectedly, no central control point exists, the network must configure itself without supervision, and the network must be able to reconfigure itself for new applications. Already there are effective, fully distributed algorithms for routing, data collection and dispersal as well as rudimentary self-configuration [5, 6].

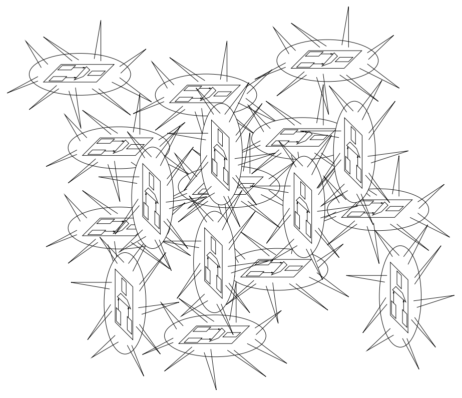

Figure 6: Randomly connected computational paint

As a transition phase between silicon and molecular computing we envision packaging our silicon-based tile in a specialized "sea urchin"-like package. By using several thousand of these mixed with a bonding agent we envision creating a computing paint. The result is a randomly constructed substrate with connections formed haphazardly between tiles (Figure 6).

6 Related Work

Tile-based computing has gained significant interest among the architecture community. Among several research efforts there are three that we wish to highlight here. The first is the RAW [7] reconfigurable wires project from MIT. The second is the Smart Memories [8] tiled architecture from Stanford. The third is the GPA processor from the University of Texas [9]. In addition there is a wealth of decoupled, dataflow and other related work that is applicable.

The RAW [7] tiled architecture is a scalable computational mesh built from independently operating R2000-like cores interconnected with a simple synchronous [10] communication network. Each node in the mesh contains a data and instruction memory, register file, and pipelined processor. A compiler statically schedules instruction streams for each tile and interconnection switch. The RAW machine performs best on highly regular and statically schedulable codes. Irregular behavior, such as a long latency cache miss can stall the entire RAW machine.

Smart Memories [8] is a proposal for a universal computing substrate. It is based on a tiled architecture that replicates a coarsely configurable processing core and memory in a grid communication structure. Similar to RAW, each node in the grid executes an independent instruction stream. Also similarly, a fine-grained switch communication network is integrated on-chip for fast near-neighbor communication. Native programming for a Smart Memories device is similar to development for traditional parallel systems. Impressively, the Smart Memories effort has been able to efficiently emulate existing architectures, such as Imagine [11] and Hydra [12], on top of their mesh. The computation cache differs from both RAW and Smart Memories in that it uses a much simpler computation unit. This provides a much better match between traditional programming techniques and the underlying hardware, i.e. the computation cache does not rely on static scheduling or parallel programming languages.

The GPA [9] processor from the University of Texas at Austin is an unique hybrid of dataflow and Von-Neumann architectures. Basic-blocks of instructions are brought into a processor for execution, similar to a classic Von-Neumann design. However, once in the core these instructions execute in a data-flow / systolic array like fashion. The GPA project is an interesting alternative for future silicon systems; however, it remains to be seen if this hybrid architecture scales to extremely large mesh sizes. In particular, this design retains a costly fetch and decode process to perform computation, and except for limited circumstances, instructions must be decoded each time they execute.

Decoupled architectures rely on compiler support to expose parallelism to independently communicating processor nodes [13, 14]. The classic decoupled design splits applications into one or more instruction streams along functional boundaries. For instance, one stream may focus on memory input-output, while another on execution. This compiler directed approach exposes instruction level parallelism inherent in a sequential instruction stream. Unfortunately, it is difficult to generate several independent instruction streams and those that are executed tend to be tightly coupled. This coupling creates sequentiality that prevents one stream from executing far ahead of another, called *slip* in decoupled architectures. The computation cache divides the program, but along different lines. The enable and poison/bless waves effectively decouple calculation of control path from the actual execution of instructions.

Multiscalar [15] from the University of Wisconsin is a unique design that tries to speculate far ahead within a single thread of computation. The architecture consists of a ring of processors that speculatively execute tasks; short splices of computation detected with a compiler. While we are currently exploring decentralized approaches to handling speculative memory operations, the substantial work that went into this on the Multiscalar design could be adapted for computation caches.

Dataflow computing has a long history [16, 17, 18, 19, 20, 21]. In some philosophical sense, the concept behind computation caches is to take the best of dataflow and superscalar designs and build a programmable substrate that decentralizes all of it. Centralization, of anything, in future processing technologies will limit scalability. Dataflow computing suffered from the need for a centralized tagged memory space. Additionally, the efficient programming interface (via languages like val [22]) was awkward and difficult for industry to accept. However, if the bottlenecks that underpin past dataflow systems were removed it would be a compelling solution to future processor designs. The computation cache tries to fulfill this vision by removing the centralized hardware structures and allowing programs written in commonly used languages to execute in dataflow fashion.

7 Conclusion

Computational caches are a unique forging of two traditional computer architecture concepts: an instruction cache and a processor. This combination will enable a scalable silicon-based computing substrate since it directly confronts the three architectural challenges of distribution, fault-tolerance, and complexity that we currently face. While our current research focuses on scalable silicon-based systems we believe the most lasting contribution of our work will be the techniques developed for dynamic-placement, self-adaptation and migration of instruction streams. In effect, a computation cache is a hardware algorithm for efficient distributed instruction-stream processing across irregularly constructed and connected computing agents. The software-hardware algorithms we develop for this will open up new avenues of research in ubiquitous "grid computing", computational fabrics and paint, and non-silicon self-assembled nanodevices. The challenges we currently face with scaling silicon into the next decade are the same challenges these alternative technologies will have to confront. As our research progresses we will look for opportunities to apply our ideas to these emerging domains.

References

[1] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," in *IBM Journal*, 1967.

[2] S. C. Goldstein and M. Budiu, "Nanofabrics: Spatial computing using molecular electronics," in *International Symposium on Computer Architecture*, 2001.

[3] A. Dehon, "Array-based architecture for mulecular electronics," in *Workshop on Non-silicon Computing in conjunction with the International Symposium on High Performance Computer Architecture*, 2002.

[4] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, (Haifa, Israel), pp. 196–207, IEEE Computer Society TC-MICRO and ACM SIGMICRO, November 16–18, 1999.

[5] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," in *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCOM)*, 2000.

[6] G. Pei, M. Gerla, and X. Hong, "Lanmar: Landmark routing for large scale wireless ad hoc networks with group mobility," 2000.

[7] W. Lee *et al.*, "Space-time scheduling of instruction-level parallelism on a Raw machine," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-VIII*, October 1998.

[8] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: A modular reconfigurable architecture," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, (Vancouver, British Columbia), pp. 161–171, IEEE Computer Society and ACM SIGARCH, June 12–14, 2000. *Computer Architecture News,* 28(2), May 2000.

[9] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler, "A design space evaluation of grid processor architectures," in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, 2001.

[10] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Argarwal, "Baring it all to software: Raw machines," *IEEE Computer*, 1997.

[11] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, and B. Towles, "IMAGINE: Signal and image processing using streams," in *Hot Chips 12: Stanford University, Stanford, California, August 13–15, 2000* (IEEE, ed.), (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA), pp. ??–??, IEEE Computer Society Press, 2000.

[12] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Willey, M. Chen, M. Kozyrczak, and K. Olukotun, "The Stanford Hydra CMP," in *Hot Chips 11: Stanford University, Stanford, California, August 15–17, 1999* (IEEE, ed.), (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA), pp. ??–??, IEEE Computer Society Press, 1999.

[13] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture* [23], pp. 112–119. *Computer Architecture News,* 10(3), April 1982.

[14] G. Tyson, M. Farrens, and A. R. Pleszkun, "MISC: A multiple instruction stream computer," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, (Portland, Oregon), pp. 193–196, IEEE Computer Society TC-MICRO and ACM SIGMICRO, December 1–4, 1992. SIG MICRO Newsletter 23(1–2), December 1992.

[15] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture* [24], pp. 414–425. *Computer Architecture News,* 23(2), May 1994.

[16] F. J. Burkowski, "Instruction set design issues relating to a static dataflow computer," in *Proceedings of the 9th Annual Symposium on Computer Architecture* [23], pp. 101–111. *Computer Architecture News,* 10(3), April 1982.

[17] Y.-C. Hong, T. H. Payne, and L. B. O. Ferguson, "Graph allocation in static dataflow systems," in *Proceedings of the 13th Annual International Symposium on Computer Architecture* [25], pp. 55–64. *Computer Architecture News,* 14(2), June 1986.

[18] J. Sargeant and C. C. Kirkham, "Stored data structures on the Manchester dataflow machine," in *Proceedings of the 13th Annual International Symposium on Computer Architecture* [25], pp. 235–242. *Computer Architecture News,* 14(2), June 1986.

[19] K. Kawakami and J. R. Gurd, "A salable dataflow structure store," in *Proceedings of the 13th Annual International Symposium on Computer Architecture* [25], pp. 243–250. *Computer Architecture News,* 14(2), June 1986.

[20] D. E. Culler and Arvind, "Resource requirements of dataflow programs," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, (Honolulu, Hawaii), pp. 141–150, IEEE Computer Society TCCA and ACM SIGARCH, May 30–June 2, 1988. *Computer Architecture News,* 16(2), May 1988.

[21] K. M. Kavi, A. R. Hurson, P. Patadia, E. Abraham, and P. Shanmugam, "Design of cache memories for multi-threaded dataflow architecture," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture* [24], pp. 253–264. *Computer Architecture News,* 23(2), May 1994.

[22] J. McGraw, "The val language: Description and analysis." TOPLAS 4(1):44-82, 1982.

[23] IEEE Computer Society and ACM SIGARCH, *Proceedings of the 9th Annual Symposium on Computer Architecture*, (Austin, Texas), April 26–29, 1982. *Computer Architecture News,* 10(3), April 1982.

[24] ACM SIGARCH and IEEE Computer Society TCCA, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, (Santa Margherita Ligure, Italy), June 22–24, 1995. *Computer Architecture News,* 23(2), May 1994.

[25] IEEE Computer Society TCCA, ACM SIGARCH, and the Information Processing Society of Japan, *Proceedings of the 13th Annual International Symposium on Computer Architecture*, (Tokyo, Japan), June 2–5, 1986. *Computer Architecture News,* 14(2), June 1986.