IBM Research Report

The Spiral Cache:
A Self-Organizing Memory Architecture

Volker Strumpen
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758

Matteo Frigo
Cilk Arts, Inc.
55 Cambridge Street
Burlington, MA 01803
The Spiral Cache: A Self-Organizing Memory Architecture

Volker Strumpen
IBM Austin Research Laboratory
11501 Burnet Road, Austin, TX 78758

Matteo Frigo∗
Cilk Arts, Inc.
55 Cambridge Street, Burlington, MA 01803

ABSTRACT
We propose a self-organizing cache architecture, the spiral cache, in an attempt to offer scalable performance for large cache capacities and high clock frequencies into physical limits with theoretical guarantees. To demonstrate the practical value of the spiral cache we report results from full-system simulations.

1. INTRODUCTION
Recent work on non-uniform cache architectures (NUCA), including [6, 12, 16, 17, 19, 20], recognizes the fact that the random access machine model [1] with uniform access latencies to each memory location is no longer valid even for single-chip architectures. Because the access latency of a memory array is proportional to its side length, NUCA designs strive to reduce the access latency by breaking a large memory into a set (usually an array) of memory tiles. Furthermore, NUCA designs recognize that, because of the limited speed of signal propagation over long wires, tiles that are physically distant from the processor core incur higher access latencies than tiles that are closer to the processor, and consequently these designs employ dynamic replacement strategies that move most frequently used data into those tiles that are physically closer to the core. Thus, the placement policy is a fundamental aspect of any NUCA design.

The NUCA placement policy can be viewed as a variant of the classic problem of minimizing page faults in demand-page systems [7], but with a twist: while a demand-page system can be approximated with a binary cost function (cheap page in memory vs. expensive page on disk), the access cost of a memory tile on chip should be modelled as a function of the distance of the tile from the processor core. This variant of the problem was studied theoretically by Sleator and Tarjan [28], who proved the following result. Consider a system where the access cost $f(i)$ to tile $i$ is a concave function, e.g., $f(i) = i$ or $f(i) = \sqrt{i}$, then the move-to-front placement policy is 2-competitive. When accessing tile $i$, the move-to-front policy moves the accessed data into front tile 1 and shifts data from tile $j$ to tile $j+1$, for $1 \leq j < i$, in order to make space for the data. Informally, the term 2-competitive expresses that the aggregate cost of a sufficiently long series of accesses under move-to-front is at most twice as large as the cost incurred by any other placement policy, even ones that have perfect knowledge of the future sequence of accesses.

Inspired by the result of Sleator and Tarjan, we propose a non-uniform cache architecture called the spiral cache, which is a tiled, self-organizing memory architecture that uses a move-to-front dynamic placement policy. We lay out the memory tiles in 2D space such that the access cost to the $i$-th tile is a concave function of $i$, so that the competitiveness result applies. In addition, the spiral cache is a pipelined memory architecture, capable of maintaining multiple accesses in flight. The pipeline is implemented as a systolic network, which avoids unnecessary buffer space and yields superior performance over routed networks. As a result, the spiral cache offers competitively low latencies under our spatial memory model compared to any conceivable cache architecture, and high throughput due to pipelining.

Despite the data movements incurred by adaptive cache line replacement across tiles, power densities of our prototype spiral cache design are lower than comparable conventional cache designs. Furthermore, the spiral cache architecture enables adaptive power management to reduce leakage power. Although the spiral cache is an asymptotic design for large caches, our simulation results demonstrate performance improvements up to a factor of 4 when replacing the existing cache hierarchy of contemporary machines with a modestly sized spiral cache of only 2 MB.

This article is structured as follows. In Section 2 we introduce a simple spatial memory model that accounts for signal propagation delays. We study different placement algorithms under the spatial memory model. The insights we glean lead us to the spiral cache architecture in Section 3. Empirical results from full-system simulations can be found in Section 4, and related work is discussed in Section 5.

2. DYNAMIC CACHE PLACEMENT
Given a memory trace, the sequence of addresses accessed during the execution of a program, a cache placement policy maps addresses to cache lines. In cache designs with non-uniform access costs, the placement policy impacts the overall system performance. In this section, we briefly recall some well-known cache placement policies and known results about them, and present less known empirical data.

2.1 1D Linear Memory Model
The spiral cache design that we describe in Section 3 is a 2D systolic array. To simplify the exposition of the basic ideas, however, in this section we consider a NUCA cache comprising a 1D systolic array of tiles, such as the one shown in Figure 1. When processor $P$ issues a load request to a tile, e.g., tile 7, the request signal propagates across tiles 1 through 6 to tile 7, and the data stored in tile 7 propagate in the reverse direction back to $P$.

**Figure 1:** Linear array with memory tiles aligned in the $x$-direction.

In this model, we make the following assumptions: (1) A

---

∗Work done while the author was at the IBM Austin Research Laboratory.
2.2 Placement Algorithms

In the spatial memory model of Section 2.1, the placement algorithm has a direct impact on the average access latency, even if the entire working set fits into the cache and no evictions occur due to conflict misses. In the following, we assume that a memory tile holds one cache line. We illustrate the effect of different placement algorithms on the average access latency by means of the sample trace $T = [A_1, B_2, C_3, C_4, B_5, B_6]$ of addresses $A$, $B$, and $C$.

2.2.1 Direct-mapped Placement

The simplest placement algorithm is employed in variations in direct-mapped cache designs: interpret the least significant bits of a line address as the index for the cache line. Figure 2 illustrates the scenario where the mapping from addresses to tile indices is assumed to be $A \rightarrow 7$, $B \rightarrow 10$, $C \rightarrow 2$. Note that this mapping is static, and precludes any control over the distance of the placement from the processor.

We judge the effectiveness of the placement by calculating the average access latency of trace $T$. Assume the cache is initially empty. Access $A_1$ requires a backing store access, which places the fetched data into tile 7, and then incurs 14 clock cycles of cache access latency. The next two accesses $B_2$ and $C_3$ also require backing store accesses, whereas the remaining three accesses are served directly out of the cache. The average access latencies (in cycles) are as follows:

<table>
<thead>
<tr>
<th>access</th>
<th>$A_1$</th>
<th>$B_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
<th>$B_5$</th>
<th>$B_6$</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency</td>
<td>14</td>
<td>20</td>
<td>4</td>
<td>4</td>
<td>20</td>
<td>20</td>
<td>82</td>
</tr>
</tbody>
</table>

The total number of clock cycles spent in access latency is 82, plus the cycles required for 3 backing store accesses. The average access latency, not counting the backing store accesses, is hence $82/6 = 13.7$ cycles per access.

2.2.2 Most-frequent-first Placement

Next, we consider another static, yet more effective placement algorithm. Assume we could map addresses into memory tiles according to their access frequency, such that the most frequently accessed address is mapped closest to the processor to minimize the average access latency. In our trace, the most frequently accessed address is $B$, which we access three times. Hence, we map $B \rightarrow 1$, the second most frequently accessed address $C \rightarrow 2$, and $A \rightarrow 3$, as shown in Figure 3.

Analogous to the accounting of access latencies above, we summarize the access latencies for this placement:

<table>
<thead>
<tr>
<th>access</th>
<th>$A_1$</th>
<th>$B_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
<th>$B_5$</th>
<th>$B_6$</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

The sum of the latencies is 20 clock cycles, and the average access latency is $20/6 = 3.3$ cycles per access, which is more than four times faster than the direct-mapped placement.

2.2.3 Move-to-front Placement

Unlike static placements, a dynamic placement is capable of adapting the mapping to the access pattern of a program’s trace during execution. A popular algorithm is the move-to-front rule [28]: “when accessing an address, move it to the front of the array.” To make space for the new line in the front, we push the current line back toward the tail of the array. Because the placement of a line is now dynamic, we must search the line upon subsequent accesses. Figure 4 illustrates the move-to-front placement.

We first three accesses fetch the data from the backing store and move them into front tile 1. Then, access $C_4$ finds address $C$ in tile 1, incurring the minimal access latency of 2 cycles. Next, access $B_5$ moves the line from tile 2 into front tile 1, effectively swapping the contents of tiles 1 and 2. Finally, access $B_6$ finds address $B$ in tile 1, causing the minimal access latency of 2 cycles. The table below summarizes the access latencies for the placement of Figure 4.

<table>
<thead>
<tr>
<th>access</th>
<th>$A_1$</th>
<th>$B_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
<th>$B_5$</th>
<th>$B_6$</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>14</td>
</tr>
</tbody>
</table>

The sum of the access latencies is 14 clock cycles, and the average access latency is $14/6 = 2.3$ clock cycles per access. It is noteworthy that the move-to-front heuristic produces an even smaller average access latency than the placement based on access frequency, although that placement is based on the knowledge of the entire trace, whereas the move-to-front placement considers one access at a time only.

2.2.4 Transposition Placement

The transposition heuristic [24] is a dynamic placement algorithm that obeys the rule: “when accessing an address, swap it with the item in the immediately preceding tile.” Many hybrids and variations of move-to-front and transposition exist [4]. Figure 5 illustrates the dynamic placement of trace $T$,
assuming that, in case of a miss, a datum is placed in the nearest free tile to the processor.

Figure 5: Linear array with dynamic placement algorithm based on transposition heuristic.

To account for the access latencies, we charge the access latency, and assume that the subsequent transposition incurs no delay. Figure 5 shows the state for accesses $C_4$, $B_2$, and $B_6$ after the transposition. The access latencies, not counting backing store delays, are shown below.

<table>
<thead>
<tr>
<th>access</th>
<th>$A_1$</th>
<th>$B_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
<th>$B_2$</th>
<th>$B_6$</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>24</td>
</tr>
</tbody>
</table>

The average access latency is $24/6 = 4$ clock cycles per access, which is almost twice as large as that of move-to-front.

### 2.2.5 Known Facts About Placement Algorithms

A strong theoretical result [28] is known about the move-to-front heuristic: provided that the access cost $f(i)$ to tile $i$ is a concave function, move-to-front is $2$-competitive. That is the cost of a sufficiently long trace under move-to-front is at most twice as large as that of any other placement policy. In our 1D example, $f(i) = 2i$ is concave; in our 2D spiral cache in Section 3, we have $f(i) = O(\sqrt{i})$, which is also concave.

The factor of $2$ in the competitiveness result depends on the chosen cost function for list accesses, and can be larger under different assumptions. However, experiments show that $2$-competitiveness is actually conservative, and move-to-front often performs better in practice. More recent theoretical work [2, 3] attempts to close this gap.

The transposition heuristic is not $2$-competitive; the counterexample by Bentley and McGeoch [8] demonstrates that the transposition heuristic can have very poor amortized performance. Although the transposition method has been shown to use strictly fewer comparisons than move-to-front asymptotically [27], its convergence to the asymptotic limit is much slower than for move-to-front [10].

Experimental studies [4, 8] report that the move-to-front heuristic outperforms the transposition heuristic in all of their experiments, although instances are known where the transposition heuristic is marginally faster than move-to-front. On the other hand, the comparison in [4] of 18 heuristic families and more than 40 algorithms shows that a variation of move-to-front slightly outperforms vanilla move-to-front.

### 2.3 Empirics of Move-to-Front

To illustrate the behavior of move-to-front placement in practical situations, we study the memory behavior of ten important algorithms, see Figure 6, when executed on a memory array with one basic data type, such as an integer number, per tile. Placement in the array is managed with the move-to-front heuristic, as illustrated in Figure 4: each access to tile $i$ moves its datum to front tile 1 and shifts all data in front of tile $i$ by one tile toward the tail end. The array is sized to accommodate the entire working set.

For each algorithm, Figure 6 plots the number of memory accesses serviced by each tile. Specifically, an access to tile $i > 1$ causes its datum to be moved to front tile 1. If the datum is reused subsequently, it will be found in some tile $j \geq 1$. Thus, $j - 1$ is the number of distinct data accessed between subsequent accesses to the same datum. Quantity $j$ is known as the stack distance [22], and as such Figure 6 plots the density of the stack distance for each algorithm.

The distribution of the stack distances offers some insight into the “cache friendliness” of an algorithm. Consider a machine with LRU cache of size $m$. Accesses at stack distance $j \leq m$ hit in the cache, and accesses at distance $j > m$ are cache misses. Thus, the stack distance gives a complete description of the behavior of an algorithm under LRU caches of any size. Because a LRU cache of size $m$ is $2$-competitive with any cache of size $m/2$, we also have an approximate description of the behavior of an algorithm under any other replacement policy.

Ideally, we would like the stack distance to decrease quickly, so that a cache with $m$ tiles will serve the majority of accesses, independent of $m$. In Figure 6, however, only the shortest path computation exhibits such a cache friendly behavior. Particularly cache unfriendly algorithms are the iterative matrix multiplication, stencil computation, and longest common subsequence computation. The matrix multiplication in Figure 6, for example, can only exploit a cache if it is large enough to cover the first 90,000 tiles. If the cache covers just one tile less, more than half of all accesses will miss. In other words, no cache of a given size can be effective for all problem sizes. Algorithms like quicksort and minimum spanning tree are still reasonably cache friendly, because they exploit only a few outliers of tiles with large numbers of accesses deeper in the array. In contrast, the access distributions of the QR decomposition, SVD, and FFT deviate significantly from an exponential distribution, and can be classified as moderately cache unfriendly. On memory bound machines, cache friendliness translates into higher performance.

### 3. Spiral Cache Architecture

In this section we present the architecture of the spiral cache. Our design is based on the spatial memory model, exploits the dimensionality of Euclidean space to support a near-optimal worst case access time, uses the move-to-front heuristic to reduce average access time, and features a conflict-free systolic data flow capable of pipelining multiple memory accesses.

#### 3.1 Basic Spiral Cache Architecture

The basic architecture of a 2D spiral cache is shown in Figure 7. The spiral cache can be viewed as as the linear array of Figure 1 wrapped around tile 1, such that the linear array forms an Archimedes spiral in Manhattan layout. An unspecified processor core is connected to the front end of the spiral at tile 1. The tail end, in this case at tile 49 of the 7×7 matrix of tiles, connects to the backing store, e.g. DRAM modules.

Each tile of the spiral cache comprises a conventional cache-like array, e.g. a direct-mapped cache. An efficient tile design balances the size of the array such that the propagation delay...
Matrix multiplication of two dense $300 \times 300$ matrices $C = AB$. The 3-fold nested loop exhibits a characteristic reuse pattern at tiles $i \equiv 1$, 600, 90,900. At $i \equiv 1$, we find $3 \times 300^2$ accesses after cold misses, at $i \equiv 600$, reuse of row elements of $A$, and at $i \equiv 90,900$ reuse of elements of $B$.

QR decomposition of an $800 \times 300$ matrix with Givens rotations against the diagonal elements. We store the angle of the rotations for subsequent multiplications by $Q$ or $Q^T$ in the lower triangular part of the decomposed matrix.

SVD computes the singular value decomposition of a dense symmetric $350 \times 350$ matrix using a two-sided Jacobi algorithm.

Stencil computation solving heat equation on 1-dimensional spatial domain with 3-point stencil. The computation uses toggle arrays, and iterates 500 time steps over 100,000 space points. Due to the toggle arrays, reuse occurs at tiles $i \equiv 2$ and 199,999 only.

LCS is an iterative dynamic program computing the length of the longest common subsequence of two sequences of lengths 10,000 and 190,000. The tableau uses linear space of size $10,000 + 190,000$.

FFT is an iterative version of out-of-place fast Fourier transform with vectors of $2^{16}$ complex numbers.

Quick sort sorts an array of 400,000 randomly chosen integer numbers using an in-place algorithm with randomized pivot selection.

Minimum spanning tree computation based on Kruskal’s algorithm using a disjoint-set data structure with union-by-rank and path-compression heuristics is applied to a graph with 30,589 vertices and 86,892 arcs.

Single-source shortest paths computation using Dijkstra’s algorithm with a binary heap implementation of a priority queue; applied to a graph with 30,589 vertices and 86,892 arcs.

Maximum flow with lift-to-front algorithm [14, Sec. 27.5] computes the maximum flow of a graph with 32,774 vertices and 49,159 arcs.

Figure 6: Memory access distributions of ten algorithms.
of the wires connecting neighboring tiles equals the access latency of the tile's array. We assume that a cache line is the unit of communication between tiles.

The interconnection networks shown in Figure 7 implement the two functionalities needed for move-to-front placement of cache lines into tile caches: (1) move a cache line to the front, and (2) push back existing cache lines to make space for the line that is moved to the front. We dedicate the spiral network to the push-back operation. The spiral network connects tile $i$ to tile $i+1$ in Figure 7. This choice enables us to move one new data item into front tile 1 during every duty cycle (more on the definition of a duty cycle later), because a fully occupied spiral cache can perform one push-back swap in each tile per duty cycle. To support the search for and communication of a line to front tile 1, we introduce a second network, the grid-style move-to-front network of next neighbor connections indicated by the arrows in Figure 7.

From a high-level perspective the operation of the move-to-front network is straightforward. Assume that the processor issues a load operation for a line stored in tile 49. The processor issues the request into tile 1, from where it travels across the diagonal toward corner tile 49. The line is found in tile 49, and moves to front tile 1 in an $xy$-routing style via tiles $48 \to 47 \to 46 \to 23 \to 8$. The travel time along path $(1, 9, 25, 49, 48, 47, 46, 23, 8, 1, 29, 12, 3, 40, 46, 45, 44, 43, 42)$ involves 11 hops, or 11 cycles according to our spatial memory model. The analogous access latency in a linear array of 49 tiles would be $2 + 49 = 51$ cycles. Thus, the 2D spiral organization reduces the access latency to about the square root of the number of tiles. In the following, we refer to the access latency of a tile with the largest Manhattan distance from tile 1 as the worst-case access latency. We find that, in general, a $k$-dimensional spiral cache consisting of $N$ tiles has a worst-case access latency of $\Theta(\sqrt{N})$. In particular, a linear array has the largest worst-case access latency of $\Theta(N)$.

The move-to-front network of the spiral cache is structured such that the latency of all tiles on a ring is the same, where a ring consists of all tiles with the same Manhattan distance from the processor. We refer to such a distance as the radius of the ring. In particular, tile 1 has a latency of 2 duty cycles, and tiles 2–9 have 5 duty cycles, tiles 10–25 have 8 duty cycles, and tiles 26–49 have 11 duty cycles latency. In general, tile $i$ of the spiral cache in Figure 7 has latency $t_i = 2 + 3(r_i - 1)$, where $r_i$ is the radius of the ring of tile $i$. Since the radius $r_i$ on the Manhattan layout is within a constant factor of that of a true Archimedes spiral, we find from the geometry of the latter that $r_i = \Theta(\sqrt{i})$, and thus $t_i = \Theta(\sqrt{i})$.

### 3.2 Geometric Retry

A characteristic feature of the move-to-front heuristic is that it compacts the working set at the front of the spiral, and keeps the most recently accessed cache lines in front tile 1. Because of this property, it is desirable to employ a lookup strategy that incurs a latency proportional to the radius of the tile that contains the item being looked up, rather than incurring a latency proportional to the radius of the spiral. Specifically, we would like to incur a latency of 1 cycle whenever the requested line is found in tile 1, which should be a frequent case according to Figure 6. We now describe one way to attain this goal.

Recall that the move-to-front placement causes newly requested cache lines to move to the front, and older lines to be pushed back along the spiral to make space for newly requested lines. Since the processor is unaware of the tile location of a line, each request must search for the tile. Sleator and Tarjan’s [28] competitiveness result of the move-to-front heuristic for a linear array assumes that this search scans the tiles from front tile 1 toward the tail end of the spiral network. However, in a 2D spiral cache we wish to exploit its higher dimensionality and scan the area in a radial fashion, so that the worst-case latency with $n$ tiles is $\Theta(\sqrt{n})$ rather than $\Theta(n)$. In a 3D spiral cache design, we scan through “spheres” rather than rings.

The problem with scanning as a search strategy is the unpredictable flow of data moving toward front tile 1 when multiple accesses are in flight. In order to avoid the area and time penalty associated with buffering and flow control mechanisms, we propose a different search strategy based on the well-known principle of geometric retry. Figure 8 illustrates our search strategy with the geometric retry rule: “If an item is not found in the area of radius $2^s$ retry the search in the area with radius $2^{s+1}$.” The initial radius is $2^0 = 1$ for $s = 0$. This represents a lookup in tile 1. If the lookup in tile 1 fails, we search all tiles within radius $2^1 = 2$, that is all tiles 2–9 at radius 2, plus tile 1 at radius 1. If this search fails again, we double the search radius to $2^2 = 4$. This search covers the entire spiral cache in Figure 8. If the search of the entire spiral cache fails, the requested line is not in the cache and the processor must fetch it from the backing store.

The data flow through the spiral cache during the scanning process is illustrated in Figure 8 by means of the fat arrows. The case with retry radius $2^0 = 1$ is trivial, and retry radius $2^1 = 2$ is a small version of the larger scenario exhibited by retry radius $2^2 = 4$. We explain the communication pattern for the top right quadrant of retry radius $2^2$. The other quadrants operate analogously. The request propagates from tile 1 outwards via the diagonal across tiles 7 and 21 to tile 43 in the top right corner. There, the address is sent westwards to tile 44 and southwards to tile 42. The westward path continues until tile 46, where it turns southwards to tile 1. The southward path is followed until tile 40, where it turns westwards to tile 1. In each of the tiles on the southward path a westwards path is split off. From tile 42, the westward path traverses tiles 21 and 22, and turns southwards at tile 23, and from tile 41, the westward path traverses tiles 20 and 7, and turns southwards at tile 8. Each tile of the quadrant is visited, and a lookup can be performed in each tile’s cache with the requested address. The geometric retry does not change the asymptotic bounds

![Figure 7: Spiral cache in 2-dimensional space.](image-url)
Geometric retry at most doubles the worst-case access latency. During any cycle, tiles in different timelines process different requests. If a tile in a timeline processes a request, then all diagonals have arrived at corner tiles 49, 43, 37, and 31 during cycle 0. Assume that a request travelling along the diagonals has arrived at corner tiles 49, 43, 37, and 31 during cycle 0. As discussed in Section 3.2, each tile on the western and eastern boundary splits a request into a horizontal and a vertical copy. Consequently, it cycle 1 the request arrives at tiles 26, 48, 44, 42, 30, 36, and 38, which are the tiles that comprise timeline 1. Furthermore, the request is no longer in timeline 0, maintaining invariant 1. Similarly, in cycle 2 the request visits the 12 tiles of timeline 2. In cycle 3, multiple requests arrive at tiles 46, 40, 34, and 28. There are no conflicts that could destroy the systolic pattern, however, because multiple incoming requests on timeline 3 were in timeline 2 at the previous cycle. By invariant 2, they are the same request and can be merged. For the same reason, tiles 23, 8, 4, 15, and 1 operate conflict-free, because each of multiple incoming requests carries the same address during a cycle, and the tiles pass this address along to the neighboring tile connected to their output.

3.3 Systolic Design

In the following we show how the basic spiral cache architecture augmented with the geometric retry mechanism extends to a conflict-free systolic architecture that supports both low access latency and high throughput at the same time.

Let a timeline be the subset of tiles of the spiral cache with the same Manhattan distance from the processor. The structure of the spiral cache maintains these three invariants:

1. During any cycle, tiles in different timelines process different requests.
2. During any cycle, if two tiles in a timeline each process a request, then the requests carry the same address.
3. If a tile in a timeline processes a request, then all diagonal tiles in the same timeline process the same request.

Rather than formally proving these invariants, we illustrate them by means of an example. Figure 9 shows the timelines traversed by a request travelling from the outer corners toward front tile 1. Assume that a request travelling along the diagonals has arrived at corner tiles 49, 43, 37, and 31 during cycle 0. As discussed in Section 3.2, each tile on the western and eastern boundary splits a request into a horizontal and a vertical copy. Consequently, it cycle 1 the request arrives at tiles 26, 48, 44, 42, 30, 36, and 38, which are the tiles that comprise timeline 1. Furthermore, the request is no longer in timeline 0, maintaining invariant 1. Similarly, in cycle 2 the request visits the 12 tiles of timeline 2. In cycle 3, multiple requests arrive at tiles 46, 40, 34, and 28. There are no conflicts that could destroy the systolic pattern, however, because multiple incoming requests on timeline 3 were in timeline 2 at the previous cycle. By invariant 2, they are the same request and can be merged. For the same reason, tiles 23, 8, 4, 15, and 1 operate conflict-free, because each of multiple incoming requests carries the same address during a cycle, and the tiles pass this address along to the neighboring tile connected to their output.

Figure 8: Geometric retry in 2-dimensional spiral cache.

Figure 9: Timelines of systolic data flow in 2-dimensional spiral cache.
requests. In the absence of geometric retry, this is easy to see. Each request is sent from tile 1 via the diagonals to the corners, and moves via the timelines back to front tile 1. Viewing each tile on the outwards pointing diagonals and each timeline as a pipeline stage, the 7 × 7 spiral cache in Figure 9 has 10 stages. This spiral cache generates a throughput of one request per cycle, and maintains 10 requests in flight. In general, an $\sqrt{N} \times \sqrt{N}$ spiral cache as shown in Figure 9 with odd $\sqrt{N}$ has $[\sqrt{N}/2] + 2[\sqrt{N}/2]$, or approximately $3\sqrt{N}/2$ pipeline stages.

Geometric retry introduces the complication that tiles on the diagonal may receive both an inward request from the previous timeline, and an outward request from the processor. If the retry radius of the outward request equals the tile radius, then we have a violation of invariant 1, because we have two requests with different addresses in the same timeline. To restore invariant 1, we simply increase the retry radius of the outward request. It is not possible to increase the retry radius at the corner tiles, but these tiles are conflict-free anyway since they cannot receive any inward requests.

![Figure 10: Tile architecture with cache, push-back network, and move-to-front network.](image)

Finally, we describe how to schedule the move-to-front and push-back accesses to the caches within each tile. Besides the networks, the cache memory within each tile constitutes a contended resource. Figure 10 shows the block diagram of the tile architecture. The control unit coordinates the activities on the push-back and move-to-front networks and the accesses to the tile cache.

![Figure 11: Illustration of micro-pipelining of push-back and move-to-front cache accesses (swap and lookup) and next-neighbor communications in each tile.](image)

Since the systolic design permits one move-to-front lookup and one push-back per cycle, we introduce a micro-pipeline with a duty cycle consisting of two clock cycles. During the first clock cycle, we perform a swap operation as part of the push-back functionality, and during the second clock cycle we perform the cache lookup associated with the search functionality of a request moving to the front. Figure 11 illustrates the pipelining of cache accesses and next-neighbor communications from the perspective of one tile. We feel that a swap operation should be incorporated in the memory array design, because it uses the address efficiently: (1) apply the push-back address, (2) read the cache contents, and (3) write the push-back data. If the swap operation cannot be fit into a single clock cycle, the swap functionality can be implemented by means of a separate read after a write operation within two clock cycles, prolonging the duty cycle of the micro-pipeline to three clock cycles.

### 3.4 Space-Filling Spiral Caches

We introduce alternative geometries for spiral cache designs which trade flexibility of layout for space utilization.

The effectiveness of the spiral cache hinges on the ability to keep most recently used lines in close physical proximity of the processor as long as possible. The spiral geometry is not the only one that ensures this property, however. The essence of the spiral cache lies in the existence of a total order of tiles, encoded in the spiral network in Figure 7, and of a move-to-front network, such that the distance of a tile from the processor in the move-to-front network is a bounded by a monotonic convex function of the distance of the tile in the spiral network. Any combination of two networks with these properties is a possible architecture for a spiral cache. For example, the 1-quadrant design on the left in Figure 12 may serve as building block to assemble the rectangular 2-quadrant design or the L-shaped 3-quadrant design.

![Figure 12: Spiral cache designs based on 2D quadrants.](image)

In the spirit of the quadrant design, we may also extend the spiral cache to 3D architectures. Figure 13 shows a 1-octant design of a 4 × 4 × 4 spiral cache. Multiple octants can be connected analogous to 2D quadrants to fill a 3D space.

The constraints on the geometry of the spiral cache can be relaxed even further, if we are willing to trade micro-pipelining efficiency of a duty cycle for flexibility in aspect ratio. We may stretch the diagonal across multiple tiles rather than connecting next neighbors only. Considerable flexibility with respect to the aspect ratio is available in contemporary VLSI technologies, where higher metal layers can be employed to implement faster, longer wires.

### 3.5 Power Management

The goal of the spiral cache is to provide a large cache with low access latencies. Large caches can cope with large working sets, but can be inefficient if small working sets occupy a small portion of the cache only. The structure of the spiral cache enables us to adjust the size of the active cache area dynamically as a function of the working set size.
The spiral network imposes a linear structure on spiral cache designs. This linear structure identifies the head and the tail for the move-to-front placement algorithm. The move-to-front heuristic has the effect of compacting the working set of a program, or of multiple programs in a multiprogrammed environment, at the head of the spiral. This compaction effect is particularly visible for working sets that are smaller than the capacity of the spiral cache. For those, the spiral cache is cut into two disjoint pieces: the active tiles at the head of the spiral which contain the working set, and the inactive tiles at the tail, whose tile caches remain unused. We exploit the compaction effect to reduce the power consumption, including the leakage power, of large spiral caches.

A simple algorithm for the power management of a tile in the spiral cache toggles the power supply of the memory array between on and off, while keeping the move-to-front and push-back networks active. Algorithm 1 assumes that each tile maintains two counters a hit counter and a push-back counter. During each duty cycle, each tile updates the relevant counter.

Algorithm 1 Power management algorithm for each tile.

\begin{verbatim}
if array power is on then
  if move-to-front lookup hits then
    increment hit counter
  end if
  if receive push-back on spiral network then
    increment push-back counter
  end if
  if hit rate < hit threshold
    ∧ push-back rate < push-back threshold then
      push-out dirty data
      turn array power off
  end if
else (array power is off)
  if receive push-back on spiral network then
    increment push-back counter
  end if
  if push-back rate > threshold then
    turn array power on
  end if
end if
\end{verbatim}

When the array of a tile is powered on, the tile counts the number of hits due to move-to-front lookups and the number of lines received from the spiral network. If the rate of hits and push-ins (over a period of time) is less than a given threshold, the tile does not contribute constructively to the program execution. Thus, we should remove the tile's memory array from the active tile set. Before we can do so, however, we must evict all dirty cache lines. This can be done with the existing infrastructure by pushing dirty lines out toward the tail end of the spiral during duty cycles when the tile does not receive a push-in via the spiral network. Once the array is clean, it can be powered off safely.

When a tile has its memory array powered off, it monitors the push-back activity on the spiral network by means of the push-back counter. If the number of push-backs over a period of time exceeds a given threshold, the tile could contribute its memory array constructively to the program execution. In this case, the tile powers up its memory array, and resumes storing push-in lines and performing lookups due to requests arriving on the move-to-front network.

4. FULL-SYSTEM SIMULATIONS

We present full-system simulations [11] comparing two PowerPC [23] based system configurations using the memory hierarchies shown in Table 1 with modified versions that replace the L2 and L3 caches with a cycle-accurate RTL model of a 2 Mbyte spiral cache. The spiral cache is a 1-quadran
design with 16 tiles as shown on the left in Figure 12, with a direct-mapped cache of 128 Kbyte capacity per tile and a line size of 128 bytes. The 16 tiles of this spiral cache offer 16-way associativity, with each set spanning all 16 tiles and move-to-front replacement within each set. We have chosen a conservative spiral cache design with a 3-stage micro-pipeline and 3 processor clock cycles per duty cycle, cf. Section 3.3. To mask the best-case spiral cache latency of 3 clock cycles when accessing front tile 1, we employ a 32 Kbyte L1 data cache with 2-way set-associativity and write-through policy in front of the spiral. Furthermore, a separate 32 Kbyte 2-way set-associative L1 instruction cache fetches instructions via the spiral cache. Both L1D and L1I caches are inclusive with respect to the spiral cache.

<table>
<thead>
<tr>
<th>System</th>
<th>L1D</th>
<th>L2</th>
<th>L3</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cap [KB]</td>
<td>Lat [Cyc]</td>
<td>Cap [KB]</td>
<td>Lat [Cyc]</td>
</tr>
<tr>
<td>Power4</td>
<td>32</td>
<td>1</td>
<td>512</td>
<td>4</td>
</tr>
<tr>
<td>Power†</td>
<td>32</td>
<td>1</td>
<td>256</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 1: Memory configurations for full-system simulations. Power† is a research design with relatively fast main memory (DRAM) that limits the effectiveness of its caches.

Each simulation boots the Linux operating system, loads the application program, switches the simulator into timing accurate mode, purges the L1 and spiral caches to initialize gathering of cache statistics, records the start cycle, executes the application, and records the end cycle. In the following we discuss two types of experiments. First, we micro-benchmark the memory performance, and, second, we analyze the performance of the ten applications shown in Figure 6.

4.1 Memory Microbenchmarking

We gauge the performance of different memory subsystems by means of a simple microbenchmark, consisting of a loop that
repeatedly loads \( n \) cache lines with one load instruction per line, and sweeps over \( m \). Given the cache line size of 128 bytes, the number of bytes \( n \) transferred in the loop body is \( n = 128m \), where 2 KB \( \leq n < 64\) MB. The PowerPC cores are capable of maintaining sufficiently many outstanding loads for this benchmark to be memory bound.

Figure 14 compares the access latencies of the conventional Power4 and the Power\( \dagger \) systems with the modified spiral cache versions. We find that the spiral cache (‘spiral cache’) reduces the step function (‘conventional cache’), typically observed with conventional memory hierarchies, to a gentle slope. This behavior indicates that much larger cache sizes will be effective up to a size where the slope transitions smoothly into the backing store (DRAM) latency.

To expose the pipelining capability of the spiral cache, we throttled the maximum number of outstanding requests artificially, and include the access latencies for at most one (‘1-access’), two (‘2-access’), and four outstanding requests (‘4-access spiral cache’) in Figure 14. We observe that the pipelining capability boosts the spiral cache performance significantly. In particular, pipelining two or more accesses hides the additive effect of the sequential lookups in the 1-access spiral cache and the backing store in case of a spiral cache miss for \( n > 2 \) MB completely. Achieving the same effect in a conventional memory hierarchy requires additional logic design effort to split and bypass tag and data accesses. Furthermore, the larger the number of outstanding requests, the lower the access latency as the sweep size increases but fits into the spiral cache.

### 4.2 Application Analysis

We compare the ten applications of Figure 6 when executed on the conventional memory configurations with executions on the spiral cache. The problem sizes for the applications were chosen such that their working set sizes are approximately 2 Mbytes for both the access distributions presented in Section 2.3 and the full-system simulations reported here. Thus, the working sets fill the spiral cache, overflow the L2 cache of the Power4, and fit comfortably into the L3 cache of the Power\( \dagger \).

Table 2 reports runtimes and speedups of all applications. We find that the Power4 system exhibits significantly improved performance by a factor of 1.4–3.4 for half of the applications, matrix multiplication, QR, stencil computation, FFT, and maximum flow. For the other half, the spiral cache reduces the number of clock cycles only marginally. On the Power\( \dagger \), the large L3 cache and the relatively fast backing store of the original cache configuration cause the speedups due to the spiral cache to be no larger than 1.3. It is notable, however, that the spiral cache does not cause a performance degradation for any of the applications on either system.

Table 3 includes hits in the L1D cache. Interesting to note is the fact that the L1D cache is not used at all by the applications. Various of the applications use the L2 cache as the default resource for loads, which exhibit a smoothening effect of the much larger tile sizes of the spiral cache. Interesting to note is the fact that the L1D data cache serves several orders of magnitude more accesses than the remaining tiles for most applications.

We conclude that the spiral cache approximates the idealized access distributions of Figure 6 reasonably well. We expect the quality of the approximation to improve for spiral caches with larger numbers of smaller tiles, although at the expense of increased access latencies in terms of number of clock cycles. However, smaller tiles enable faster clock frequencies, which may amortize an increased clock cycle count.

Our discussion in Section 2.3 revealed that most of our
ten applications are by no means cache friendly. We briefly discuss the memory behavior of alternative cache oblivious algorithms. We have implemented cache oblivious versions of matrix multiplication, QR decomposition, SVD, stencil computation, LCS, and FFT. The exponential access distributions across the spiral cache tiles in Table 5 are testament to the quality of the cache oblivious algorithms. They also demonstrate the compaction effect mentioned in Section 3.5, which enables effective power management of the spiral cache.

The remaining applications can be grouped into four classes (1) those where neither the spiral cache nor a cache oblivious algorithm improve performance, (2) where the spiral cache and not the algorithm improve performance, (3) where not the spiral cache but the cache oblivious algorithm improves performance, and (4) those where both the spiral cache and the cache oblivious algorithm offer a performance boost. The LCS computation falls into the first class, which is apparently not memory bound. The FFT falls into the fourth class. On the Power4 system both the spiral cache and the cache oblivious algorithm improve the performance by a factor of two, and together result in a speedup close to 4. The same qualitative effect appears on the Power1 system, although with smaller speedups. The QR decomposition falls into the second class. The SVD is an example for the third class, where the spiral cache is ineffective but the cache oblivious algorithm yields a speedup of about 2. The remaining applications, matrix multiplication and stencil computation, benefit from both the spiral cache and the cache oblivious algorithm, although the combination of both does not yield the ideal product of the speedups as found for the FFT. Thus, these applications fall between the second and third class.

We conclude from our full-system simulations that even a relatively small spiral cache can offer performance improvements up to a factor of 4. None of our simulations exhibts a performance degradation due to the spiral cache. The re-
duced access latencies provided by the move-to-front heuristic and the pipelining capability contribute to larger performance gains for the naive, iterative algorithms. However, our study in Section 2.3 shows that this cannot be the case if the working sets do not fit into the spiral cache. In contrast, cache oblivious algorithms complement the spiral cache nicely. The pipelining capability of the spiral cache contributes improved throughput to the performance gain, as demonstrated in Section 4.1. The full potential of the spiral cache is expected to unfold for significantly larger cache capacities and larger working sets. Unfortunately, our current simulation infrastructure is not suited for exploring a significantly larger design space than presented above.

5. RELATED WORK

The spiral cache may be classified as a non-uniform cache architecture (NUCA) [6, 12, 16, 17, 19, 20, 25]. The discussion of NUCA in [20] provides an introductory study of the design space of such cache architectures including the dynamic placement of cache lines. A “generational promotion policy” for data movement is proposed, which happens to be the well-known the transposition heuristic [24]. As discussed in Section 2.2, the transposition heuristic is not competitive, causing particularly poor performance for certain traces, and is generally not superior to the move-to-front heuristic. Furthermore, the structure of the spiral cache enables a systolic design, which does not suffer the buffering and switching overheads of the wormhole-routed network proposed for the D-NUCA architecture in [20].

The NuRAPID cache [12] proposes an alternative NUCA architecture that sequentializes tag and data accesses, and decouples data placement from tag placement. The design is an array of associative distance groups. The paper studies replacement heuristics across groups including transposition and move-to-front, there called “next-fastest” and “fastest promotion policy.” The empirical evaluation shows that in NuRAPID simulations transposition and move-to-front are the fastest of the considered replacement policies, and the transposition heuristic outperforms move-to-front by a negligible margin. The NuRAPID cache does not permit pipelining of multiple accesses, leading the authors to conclude that their sequential access design is justified by the marginal performance benefits over the NUCA design [20]. We do not share this conclusion. In contrast, the pipelining capability of the spiral cache is a major reason for its performance, cf. Section 4.1. However, the reported energy reduction of 77% compared to the NUCA design is significant. This observation coincides with our preliminary evaluation of the energy consumption of a hardware prototype of the spiral cache.

It is widely acknowledged that for fully or set-associative caches the most effective replacement policy for data within a set is least recently used (LRU) [18]. It is noteworthy that in the context of caching, LRU and and move-to-front can be viewed as instances of the same algorithm from the perspective of competitiveness results [13]. In this spirit, a spiral cache with N tiles can be viewed as an N-way set-associative cache. A set consists of the cache lines of the same index in each tile, and the move-to-front heuristic implements an LRU stack across the tiles. If we were to use a k-way set-associative cache in each tile rather than a direct-mapped cache, we could increase the associativity to a (kN)-way set associative cache.

In the past, several cache architectures have been proposed that covered a subset of aspects of the spiral cache. Earlier systolic designs [15, 21] achieve the same throughput than the spiral cache, but incur worst-case latency for each access: a request must travel to the far end of the memory, and then traverse each block (or tile in our terminology) on the way back toward the processor. In contrast, the spiral cache is designed to avoid the high-latency problem. The pipelined hierarchical memory architecture proposed in [9] is an attempt to reduce access latencies while offering the high throughput inherent in a pipelined design. This architecture requires relatively large buffers to control the flow of data through a 1D hierarchy of memory tiles. The spiral cache does not need buffers for flow control, and is not limited to 1D designs. Just the opposite, the spiral cache exploits the dimensionality of Euclidean space to reduce the worst-case access latency.

6. CONCLUSIONS

We presented the spiral cache, a new self-organizing, tiled cache architecture. The spiral cache is a pipelined systolic array, capable of maintaining multiple memory operations in flight. The design exploits the dimensionality of Euclidean space, such that a 3D implementation with N tiles has a worst-case access latency of $\Theta(\sqrt[3]{N})$, compared to $\Theta(\sqrt{N})$ for a 2D implementation and $\Theta(N)$ for a 1D implementation. The best-case access latency is that of one tile. Depending on the structure of the application program, but in particular for cache oblivious algorithms, the vast majority of accesses
incur best-case access latency, thanks to the move-to-front placement policy.

We reported results from full-system simulations that indicate the potential for building large spiral caches with small access latencies. Many questions remain unanswered, however, especially regarding the exact power/latency tradeoffs of the spiral caches. Because simulators can only provide rough answers to these questions, we are currently in the process of building a hardware prototype of the spiral cache.

Acknowledgements. We acknowledge our collaborators Fadi Gebara, JB Kuang, Rahul Rao, and Jeremy Schaub of our hardware design team, and Ahmed Gheith who helped us integrating the spiral cache model into the Mambo simulator.

7. REFERENCES


