Simple mathematical model of cache behavior

Mathematical model of cache behavior

- Simple mathematical model
  - Input:
    - A run of a (single-threaded) procedure with particular data
      - Often, a generalization to any run with similarly-sized data is valid
    - C = Cache size
  - Output: The total number of cache misses during the run
    - Estimation of the required main-memory **throughput**
      - Does not estimate **latency** effects
    - A statistic over the total run time – cannot identify bottlenecks
    - Start/stop effects: Assume the procedure runs in an infinite loop
      - The initial set of addresses present in the cache equals to the final set
  - Assumptions
    - All memory accesses of the same size
    - Cache line size is equal to the access size (i.e., spatial locality has no effect)
    - Fully associative cache
    - Perfect LRU replacement strategy
  - Many statistical details are ignored, the results are only approximate

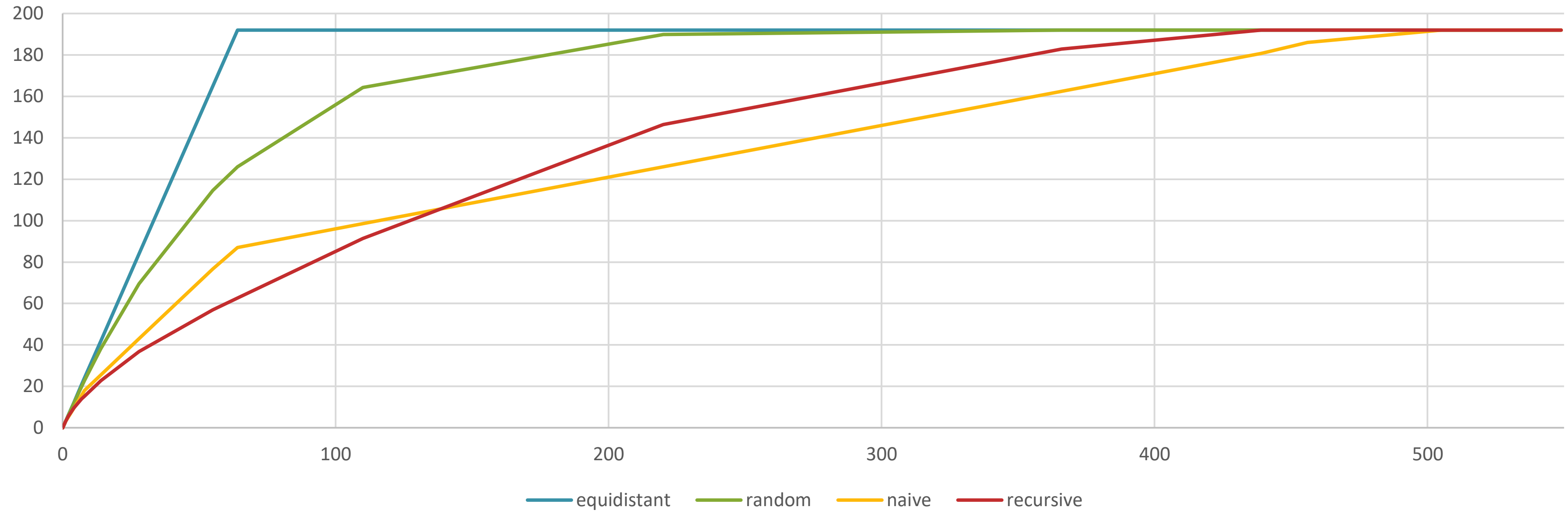
Mathematical model of cache behavior

- Notation:
  - $m(t_1, t_2)$ = the number of different addresses accessed inside (t_1, t_2)
    - Time points t_1, t_2 measured in arbitrary units; only one memory access at a time
    - Note: m satisfies triangle inequality – it is a distance measure on the time axis
- Perfect LRU replacement strategy
  - The oldest entry in the cache is evicted
- Equivalent formulation:
  - If t_1, t_2 are adjacent accesses to the same address *a*...
    - i.e. there is no access to *a* inside (t_1, t_2)
  - ... then there is a cache miss at t_2 iff $m(t_1, t_2) \geq C$
  - Proof:
    - In any moment $t \in (t_1, t_2)$:
      - The cache entries accessed inside (t_1, t) are younger than *a*
      - The entries for all the other addresses are older than *a*
    - *a* will be evicted at a time $t \in (t_1, t_2)$ such that
      - there is an access at time t to an address not accessed inside (t_1, t)
      - $1 + m(t_1, t) = C$, i.e. the cache contains exactly *a* and the addresses accessed inside (t_1, t)
    - If $m(t_1, t_2) < C$ then there is no such eviction of *a*

Mathematical model of cache behavior

- Notation:
  - A = the set of addresses accessed by the procedure
  - T = the running time of the procedure
  - $m(w)$ = the average value of $m(t, t+w)$ across all $t \in [0, T)$
    - i.e., how many addresses are accessed during a time window of size w
    - well-defined due to the assumed infinite cycle over the measured procedure
    - $m(w)$ is non-decreasing and concave
    - for $w \geq T$, $m(w) = |A|$
- The $m(w)$ function is a mathematical measure of temporal locality
  - Lower values indicate better temporal locality

Example



- The $m(w)$ function for 8*8*8 matrix multiplication
  - $T = 8 * 8 * 8 = 512; |A| = 3 * 8 * 8 = 192$
  - Equidistant: every address accessed every 64 iterations
    - Not really exists as a matrix-multiplication algorithm
    - Equidistant is always the **worst** algorithm wrt. cache
  - Random: iterations randomly permuted
    - Expectably worse than all the algorithms in use
  - Naive: three nested loops
  - Recursive: decomposed via 8 4*4*4 into 64 2*2*2 multiplications

Estimating m(w)

- $m(w)$ for an equidistant algorithm
  - For every address $a \in A$, assume periodic access every d_a time units
  - Let $H_a(w)$ = 1 if the address a is accessed during a time window of size w
    - $H_a(w)$ = 0 otherwise
    - This is a random variable depending on the placement of the window
  - The expected value of $H_a(w)$ is:
    - $\mathbf{E}\left(H_a(w)\right) = \min\left(\frac{w}{d_a}, 1\right)$
  - Let $N(w) = \sum_{a \in A} H_a(w)$, i.e. the number of different addresses accessed
  - $m(w)$ is just the average of $N(w)$ across all window placements
    - $m(w) = \mathbf{E}\left(N(w)\right) = \sum_{a \in A} \mathbf{E}(H_a(w)) = \sum_{a \in A} \min\left(\frac{w}{d_a}, 1\right)$

Estimating m(w)

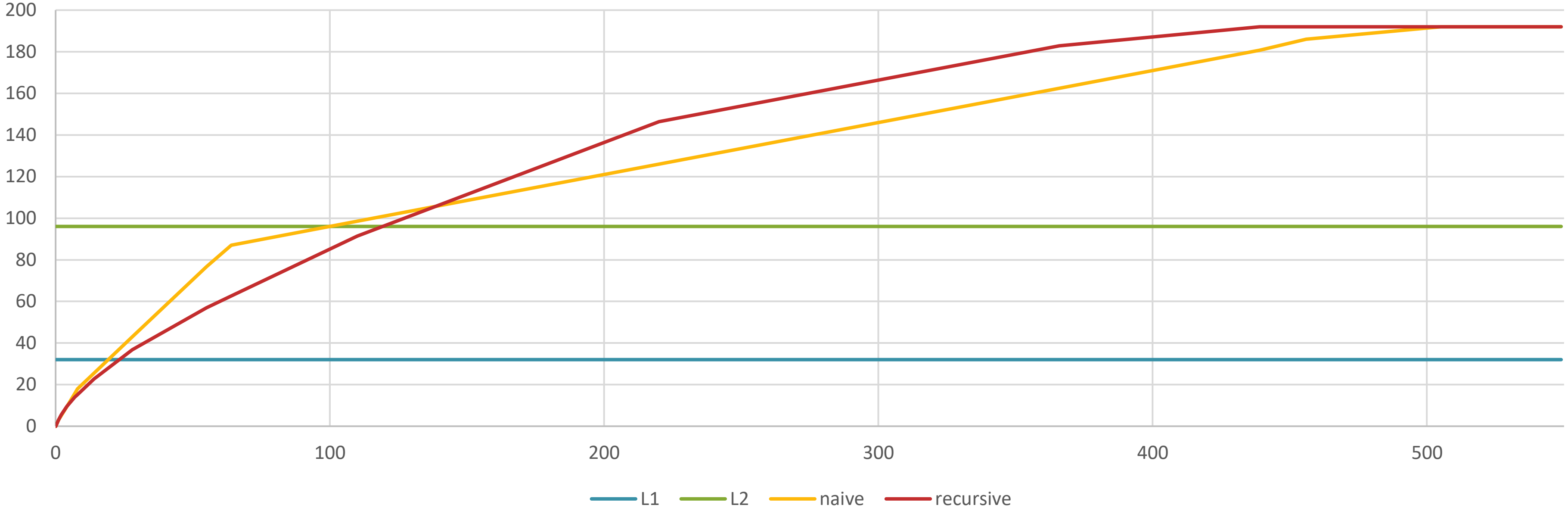
- $m(w)$ in general
  - The intervals between adjacent accesses to the same address may vary
  - The d_a is, in general, a random variable dependent on window placement
  - The correct general formula for the expected value of $H_a(w)$ is:
    - $\mathbf{E}\big(H_a(w)\big) = \frac{\mathbf{E}(\min(w,d_a))}{\mathbf{E}(d_a)}$
      - Based on the fact that wide d_a is encountered more frequently
    - $m(w) = \mathbf{E}\big(N(w)\big) = \sum_{a \in A} \mathbf{E}(H_a(w)) = \sum_{a \in A} \frac{\mathbf{E}(\min(w,d_a))}{\mathbf{E}(d_a)}$
  - Note: If the random variables $H_a(w)$ are independent for different $a \in A$
    - This is not a realistic assumption for most algorithms, but it still works here
    - Then, for large $|A|$, $N(w)$ can be approximated by a normal distribution (by CLT)
      - The variance will be relatively low, $\sigma^2 \leq |A|/4$, i.e. the std. dev. $\sigma \leq \sqrt{|A|}/2$
      - This observation will soon be useful...

- Estimating number of cache misses
  - C – the size of the cache
  - For an access to an address $b \in A$
    - assuming the previous access is at the distance d_b
    - the address b will be evicted and thus a cache miss will occur if $N(d_b) \geq C$
      - $N(d_b)$ is a random variable dependent on the position of the access
      - However, due to the narrow variance of $N(w)$, the formula $N(d_b) \geq C$...
      - ... may be simplified to $m(d_b) \geq C$, which is still random due to d_b
  - The total frequency of cache misses (wrt. unit of time) is then estimated as
    - $X(C) = \sum_{b \in A} \frac{\mathbf{P}(m(d_b) \geq C)}{\mathbf{E}(d_b)}$
      - the $\mathbf{E}(d_b)$ factor accounts for the frequency of memory accesses to b

Estimating the frequency of cache misses

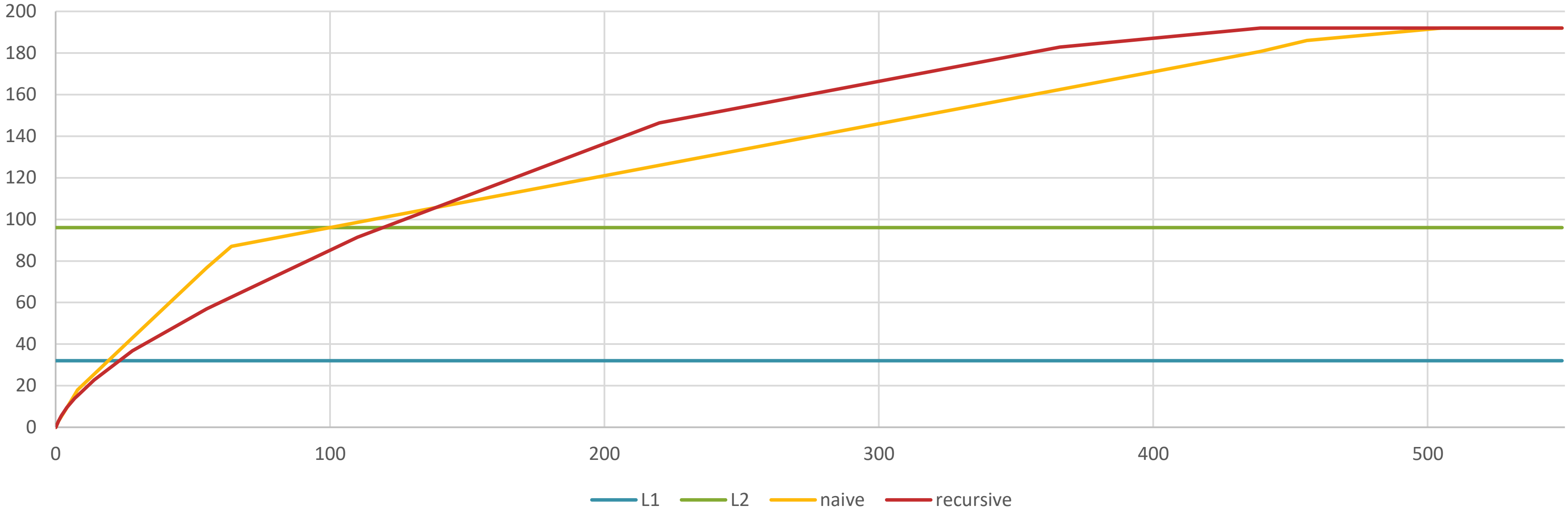
- Computing $X(C)$ from $m(w)$
  - Trick: Compute the derivative of $m(w)$:
    - $\frac{\partial}{\partial w} m(w) = \sum_{a \in A} \frac{\frac{\partial}{\partial w}\mathbf{E}(\min(w,d_a))}{\mathbf{E}(d_a)} = \sum_{a \in A} \frac{\mathbf{P}(w \le d_a)}{\mathbf{E}(d_a)}$
  - $m(d_b)$ is increasing (except when equal to $|A|$)
    - therefore $w \le d_a$ is equivalent to $m(w) \le m(d_a)$
  - Combined:
    - $\frac{\partial}{\partial w} m(w) = \sum_{a \in A} \frac{\mathbf{P}(m(w) \le m(d_a))}{\mathbf{E}(d_a)}$
  - This is similar to the definition of $X(C)$:
    - $X(C) = \sum_{b \in A} \frac{\mathbf{P}(m(d_b) \ge C)}{\mathbf{E}(d_b)}$
      - with the substitution $C = m(w)$
  - Finally:
    - $X(C) = \frac{\partial m(w)}{\partial w}(m^{-1}(C))$
    - This is only an approximative formula
      - not applicable for small $C \ll \sqrt{|A|}$

Frequency of cache misses



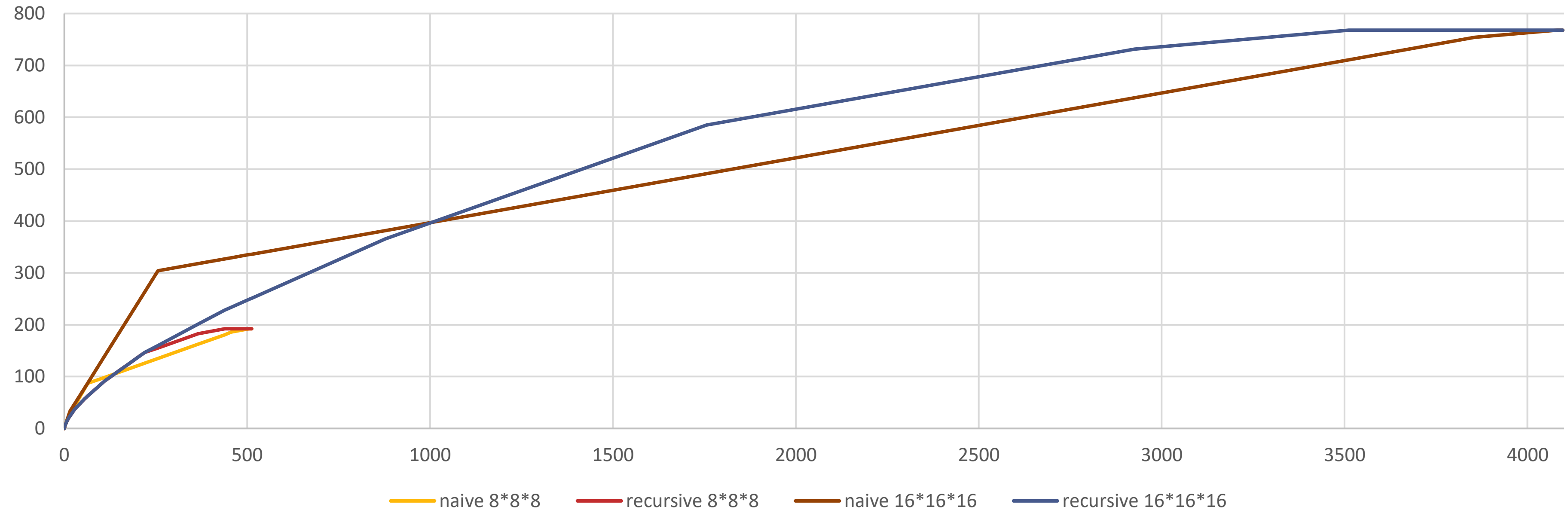
- Frequency of cache misses
  - $X(C) = \frac{\partial m(w)}{\partial w}(m^{-1}(C))$
  - Example 8*8*8 matrix multiplication
    - For a L1 cache of size 32 (matrix elements), the recursive algorithm is better
    - For a L2 cache of size 96, the naive algorithm is better
      - The derivative is important, not the time-axis position

Frequency of cache misses



- Two approaches to cache-miss optimization
  - Cache-aware
    - Make a turn in $m(w)$ every time it approaches a cache-level size
      - The new derivative will be kept until approaching the next level
    - Manipulating $m(w)$ while keeping the algorithm working may be hard or impossible
  - Cache-oblivious
    - Keep the $m(w)$ curve smoothly turning throughout the whole domain
    - For recursive algorithms, the curve is often almost independent of T and $|A|$

Frequency of cache misses



- So far, we assumed algorithm execution for particular input data
  - If we run the algorithm with different data of the same size
    - For many problems, $m(w)$ depends only on the size of data
      - Matrix multiplication and other numerical problems
    - In general, $m(w)$ may significantly vary depending on the data
      - E.g., search algorithms depend on statistical distribution of keys
  - If we run the algorithm with significantly different data size $|A|$
    - The $m(w)$ curve always converges to $|A|$
    - For recursive algorithms, the curve beginnings for different $|A|$ will be similar