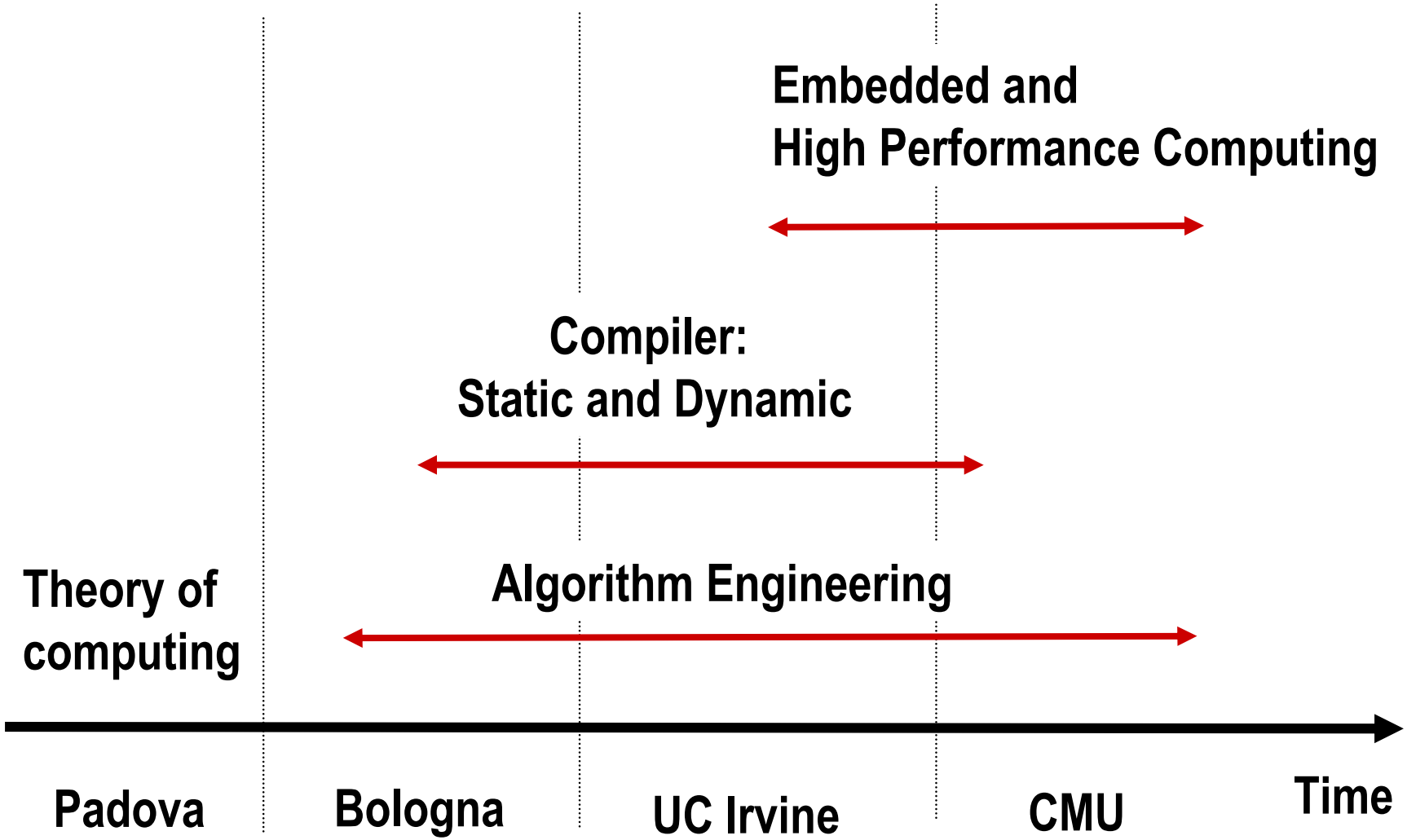
Vertical Compiler Analysis and Optimization Techniques

Paolo D'Alberto

Electrical and Computer Engineering
Carnegie Mellon University

Personal Research Background



Divide-and-Conquer Algorithms

- Most scientific applications are Matrix Algorithms
  - (BLAS3, linear algebra, LAPACK)
- *D&C algorithms formulate MA naturally*
- *D&C algorithms can be implemented as:*
  - *Blocked: implementation based on loop nests*
  - *Recursive: implementation based on recursive calls*
  - *Hybrid: composition of both*

Blocked D&C Algorithms

- Most matrix-computations libraries are blocked
  - Code legacy (Fortran)
  - Availability of advanced compiler optimizations
    - Tiling (Improving cache locality)
    - Software pipelining (hiding latency/exploit ILP)
    - Register allocation (Reducing cache/memory accesses)

- Self-installing libraries
  - ATLAS, PhiPac, SPIRAL, FLAME …
  - Architectures are represented by few parameters,
  - The codes are tailored for those parameters
  - The library is built and installed

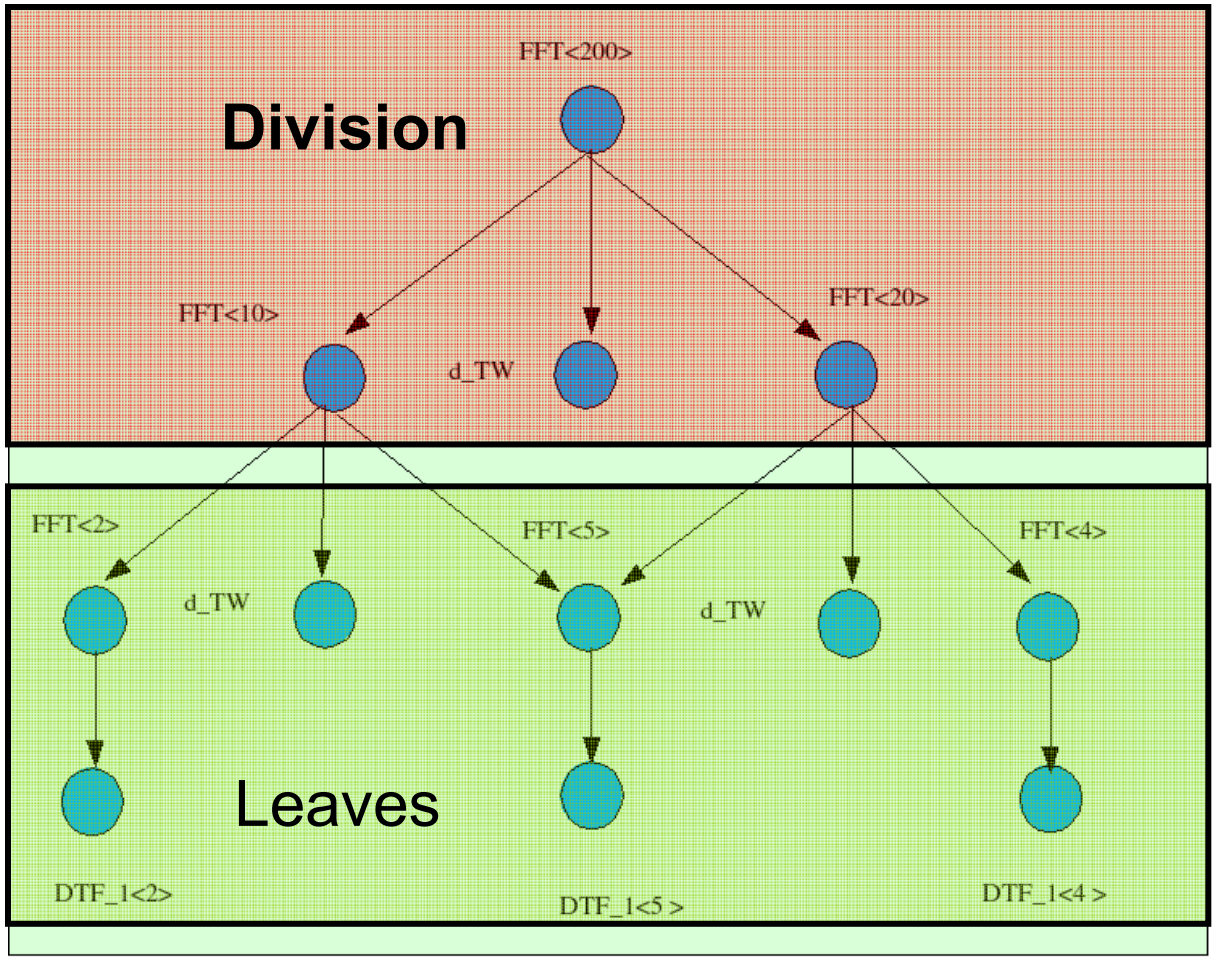
Recursive D&C algorithms

- Natural representation of matrix algorithms
  - R D&C algorithms are top-down solutions
- Inherent Data/Computation locality at multiple levels:
  - Division: The problem is divided in sub problems
  - Leaf computation: The problem is solved locally
    - Blocked algorithms

Recursive vs. Blocked

- Blocked algorithms achieve high performance (but)
  - The installation process is increasingly difficult
    - Because architectures are becoming increasingly complex
  - Tiling is not done for all memory levels (L1, RF)
- Recursive algorithms achieve good performance (with)
  - No tuning
  - Inherent hierarchical tiling
  - Portability, retargettability
- Hybrid: Recursive + Blocked are ubiquitous
  - FFTW, SPIRAL, LAPACK, ScaLAPACK

Talk = **Division** + Leaves Optimizations
JuliusC



Modelling D&C Algorithms: *The Art of Divide*

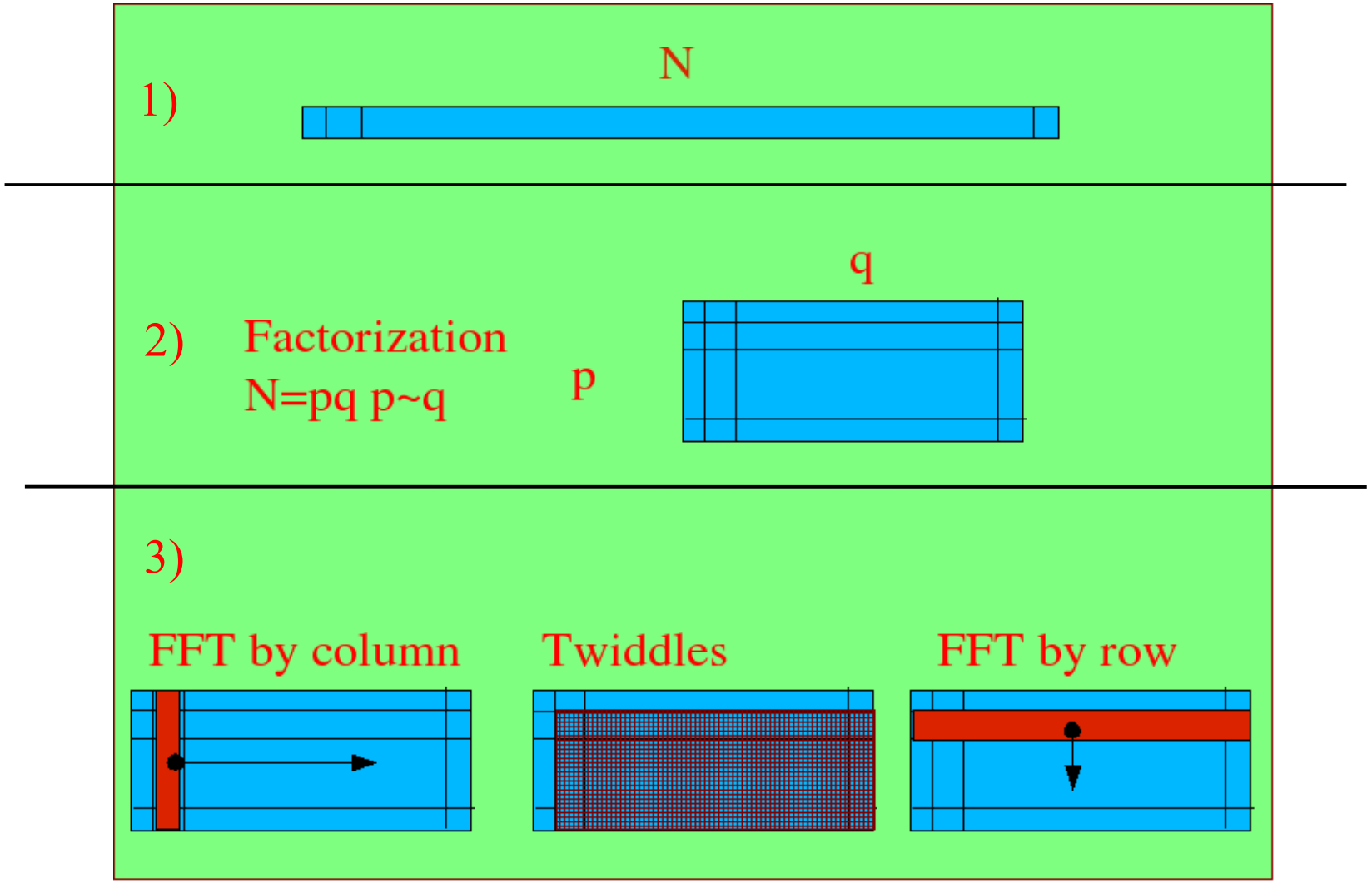
Recursive D&C algorithms consist of

- Division process
  - The division requires work
  - The division shapes up the computation
  - The problem is divided into smaller problems

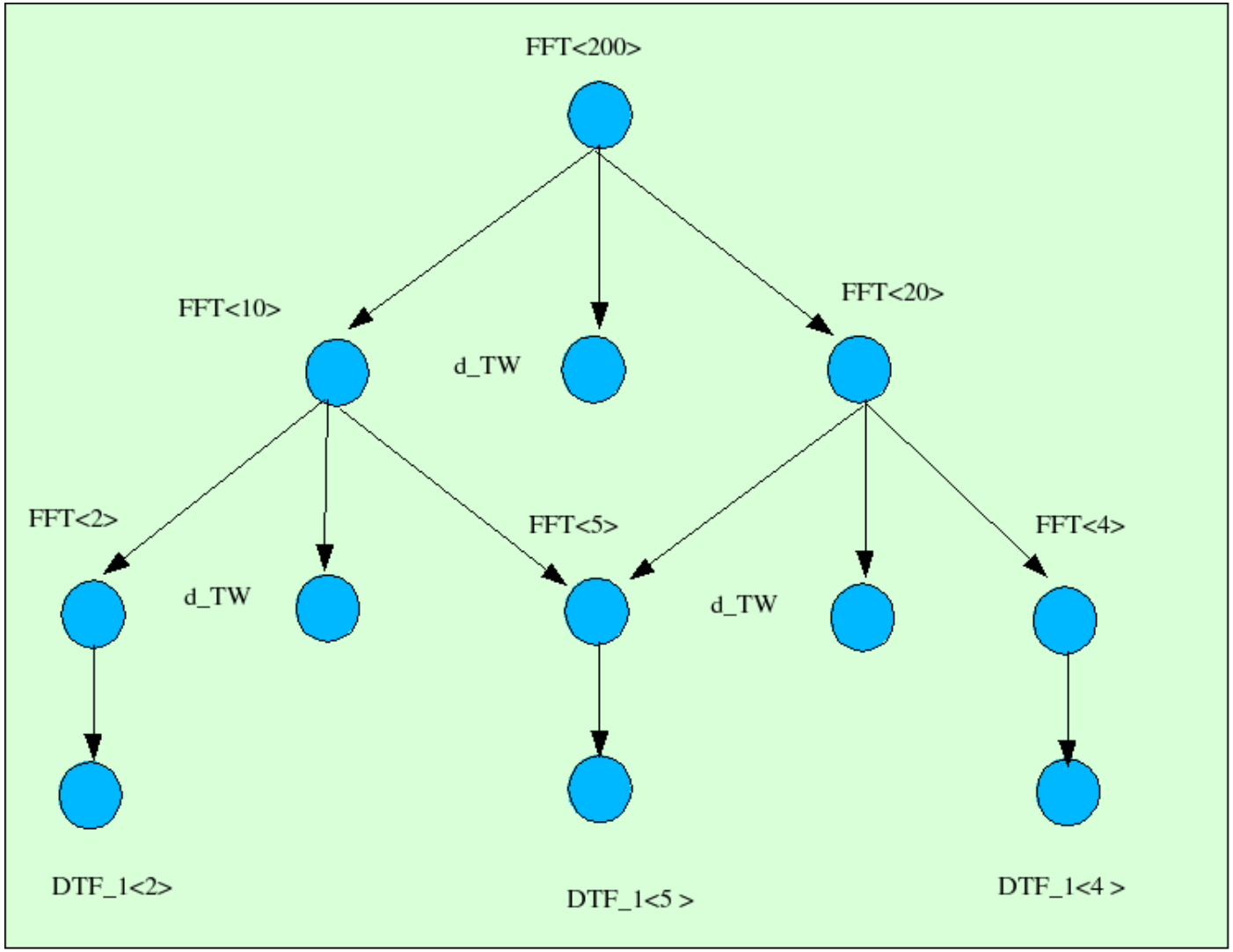
    (what is the size of the leaf computations?)

- Leaf computation
  - The problem is solved directly

Example: FFTV [Vitter et al.]



Example: Recursive-DAG for FFTV



Recursion-DAG

- We propose to model the runtime division process using a DAG: the Recursion-DAG

- Recursion-DAG ~ Call-Graph unfolding
  - A node stands for a problem of a specific size
  - A problem division is represented by arcs
  - Outgoing arcs (from a node) are ordered to maintain the original division process order.

Recursion-DAG as Model of Computation

- Recursion-DAG models the computation
  - It represents an abstraction of the execution unfolding
  - It depends on the division strategy and input size
  - It extracts sub-problems with common size
  - Division work reduction (We cache the work)

- Recursion-DAG is practical
  - E.g., for square matrix multiply it is logarithmic (in problem size)
  - Square matrix multiplications is basic operation for matrix factorizations such as LU and solution of systems

JuliusC

- Builds a recursion-DAG
  - Annotates recursive call parameters
  - These annotations are used to drive the interpretation of the code
- We used JuliusC to analyze and test applications (6)
  - From graph theory, linear algebra and number theory
  - Four presented in the following
  - We summarize the (potential) performance by the Reuse Ratio
- ***R*** = Number of function calls / Number of recursion-DAG nodes

Recursion-DAG: Considerations

- Large Reuse Ratio
  - Potential for work reuse (in the division process)
  - Relative small size of the recursive-DAG
  - Relative small effect in building the recursive-DAG at run time
- Small Reuse Ratio
  - The algorithm division process yields little regularity
  - The leaf computations are all different

JuliusC: Reuse Ratio

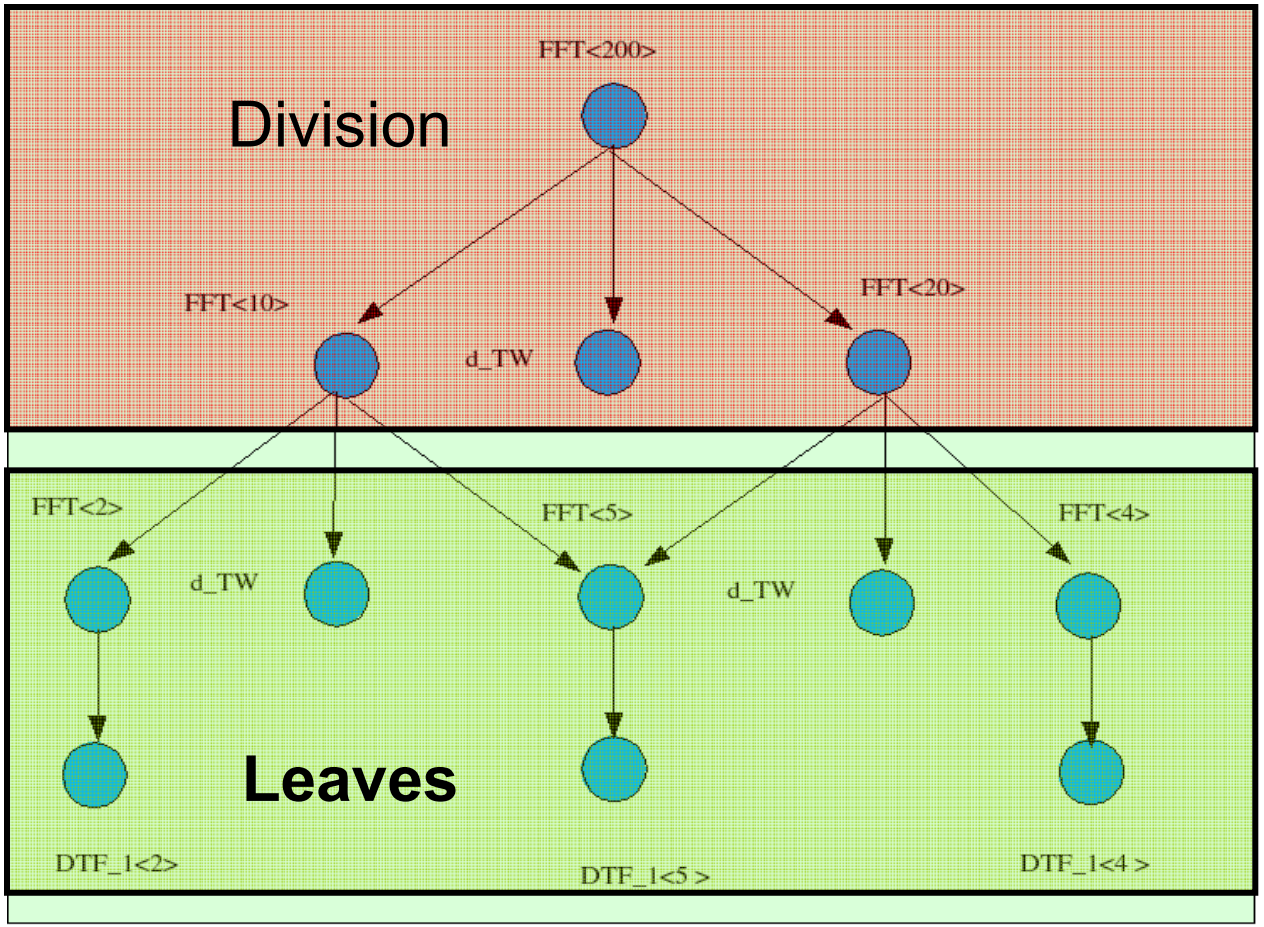
- Binomial(n,p)
  - (10,5) ***R***=14
  - (20,10) ***R***=3053
- TC(n) ***- matrix nxn***
  - ***(750) R=60728***
  - ***(7500) R=23967500***
- FFT(n)
  - (128) ***R=34;***
  - (5000) ***R=2076***
  - (65536) ***R=12683***
- MM(n) ***-matrix nxn***
  - ***(100) R=1950;***
  - ***(1123) R=2765800***

Notice that we have reuse even for non power of two inputs; that is, the reuse ratio is a property of both the application and the problem size.

Recursion-DAG: Applications

- Effects on application performance (previous work)
  - To maximize performance FLOPS/MIPS
- Effects on compiler performance
- Effects on profiling tools
  - How many times a function is called
- Effects on the debugging of R D&C algorithms:
  - Infinite loops can be found/removed easily

Talk = Division + **Leaves Optimizations**
STAMINA Cache Line Size Optimization



Modelling D&C Algorithms: The bulk of the computation, Leaves

Recursive D&C algorithms consist of Division + Leaves

- The leaves are parameterized loops
  - Loops with parameterized bounds
  - Memory references with parameterized index computations
- These parameters belong to a family of values

Leaves, Loops and cache utilization

- Loop optimizations for cache utilization

- Caches are important in modern systems
  - A key performance determinant
  - An important part of the overall energy equation:
    - large and growing size and associativity, multi-port access
  - Increasingly critical for data-intensive applications
- 'Adaptive' caches can improve performance
  - By changing line and/or fetch size based on runtime behavior
  - Varying associativity, etc
  - *"Optimal"* is application specific

Adaptive Memory System

- **Problem**: How to control adaptive cache line size to maximize performance *and* minimize energy consumption?
- **Approach**: use a compiler to generate code directing the adaptation at run time
- Issues in such a compilation approach
  - What application characteristics do we need to measure?
  - When can we statically determine an optimal line size?
  - How can we “generate” code based on the static analysis?

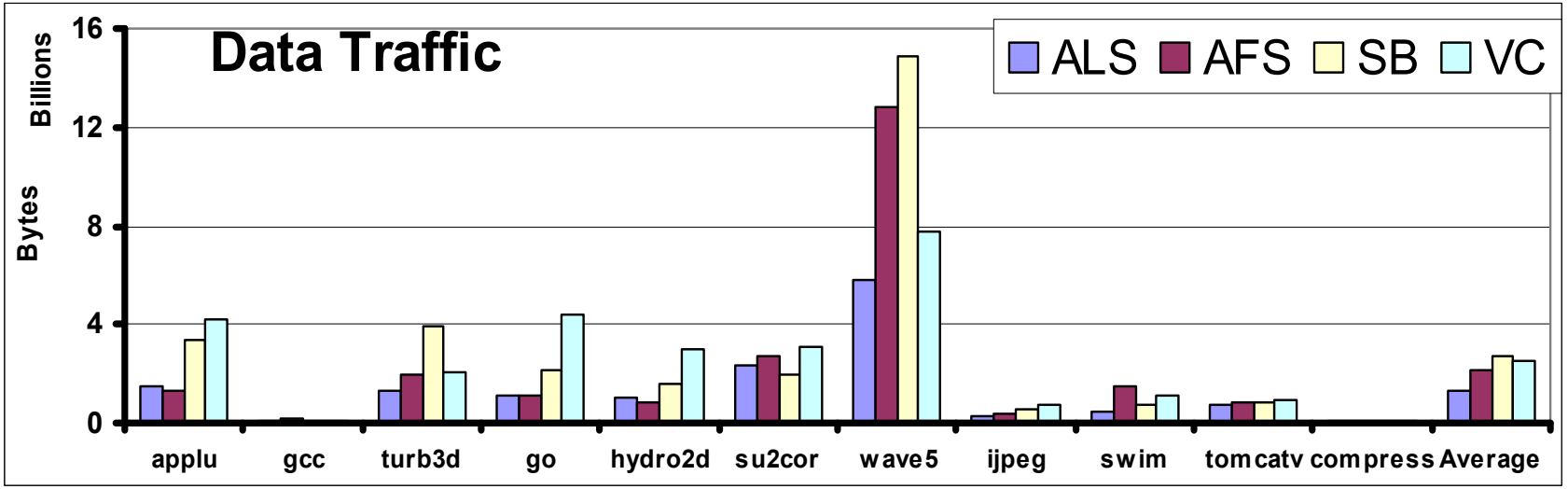
Rewards of Line Size Adaptation

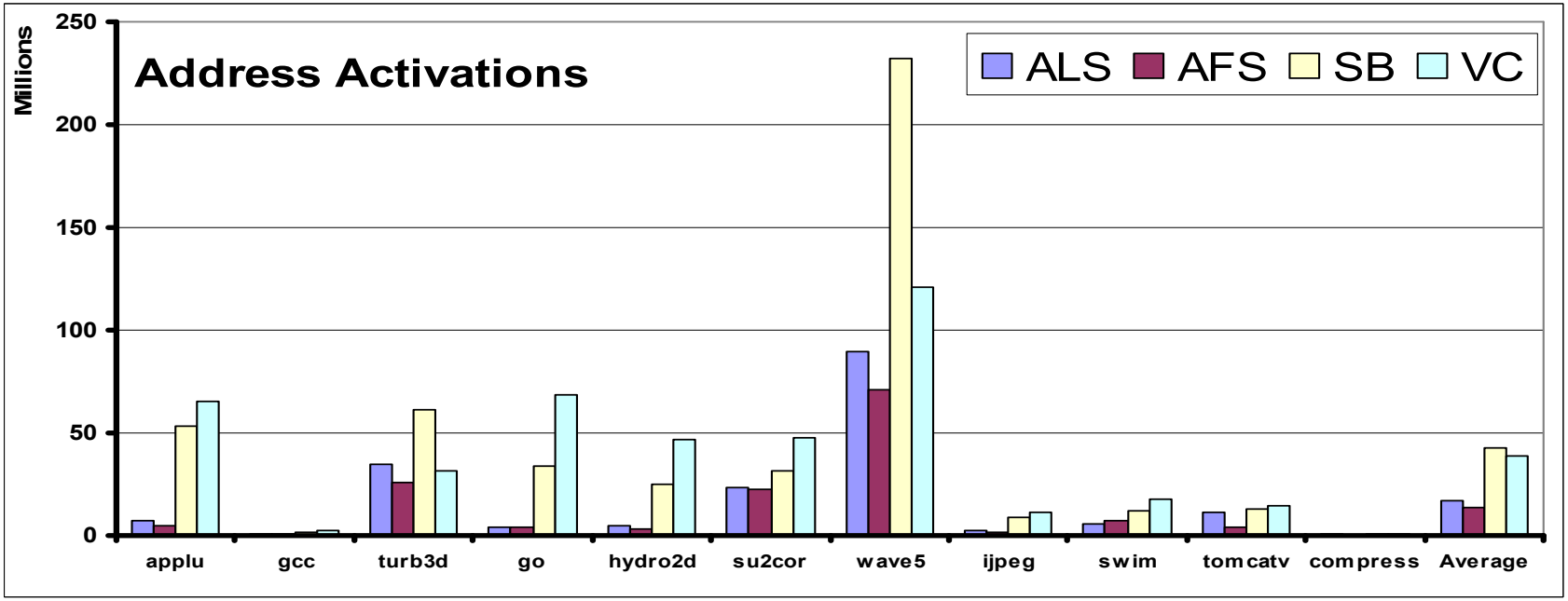
- Miss rate is reduced resulting in:
  - Fewer fetches from next-level cache or memory
  - Less transfer traffic
  - Fewer processor stalls
  - The code is, of course, left unchanged
- What is the effect on energy dissipation?

Energy Effects

- To ensure total energy reduction need to
  - reduce memory accesses and memory traffic
- The choice of the line size affects both memory accesses and memory traffic
  - Minimizing for either one alone will not give minimal energy dissipation
- A tradeoff is possible and practical, at compile time, when they are “quantified” accurately and fast

Data Traffic and Address Activations





Parameterized Loop Nests Analysis for Direct Mapped Data Cache

1. Every memory reference is symbolically equated with every possible source of interference
2. Symbolic solution of the equation is sought
   - Is there a solution?
   - Existence condition for a solution
3. Trade-off between spatial reuse and interference
   - If there is a solution, how can we estimate the number of solutions without counting them?
   - Bounding the misses by interference
4. Misses = Contribution from each reference

Interference and Reuse, *explanation*

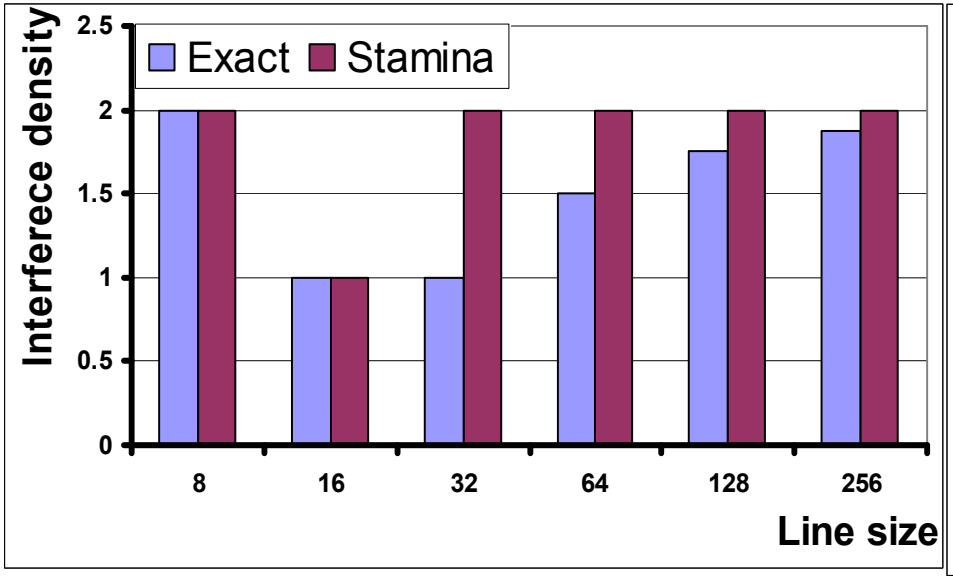
- An interference equation represents the set of iterations in the loop nest where two references interfere
  - Where there is a miss
- Existence conditions for solutions of the equation:
  - Find at least one iteration for which the equation is satisfied
- Bounding the misses due to interference
  - If there is a solution, we propose, "*the interference density",* to bound the ratio of the iteration solutions over the total number of iterations

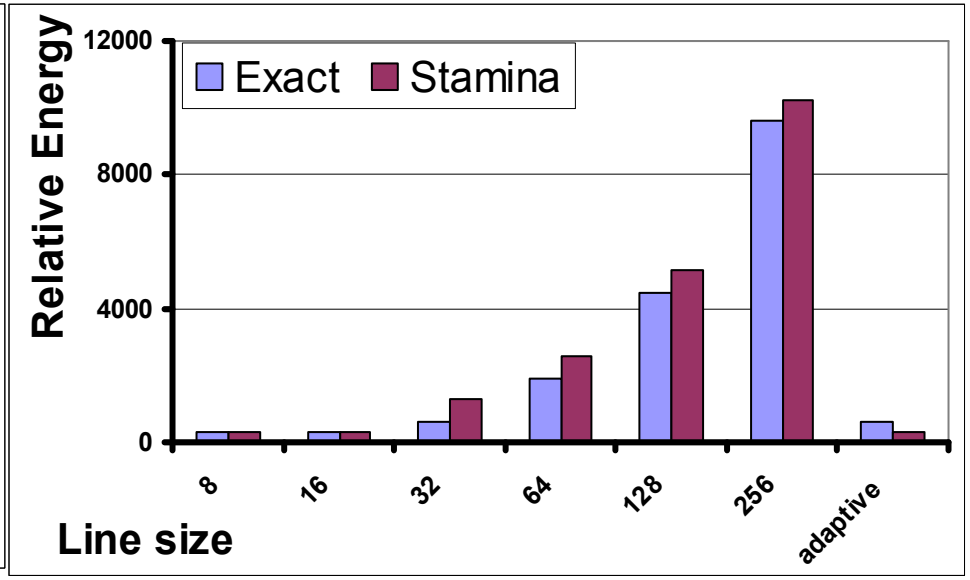
Interference Density, *explanation*

- The interference density is a straightforward quantity to determine
  - Function of the coefficients of the interference equation
- It is independent from the loop nest and the definition domain
- It is a *good* upper bound to the cache miss ratio due to interference equation
  - (but only when it is known if there is interference)

Example: Matrix Multiply

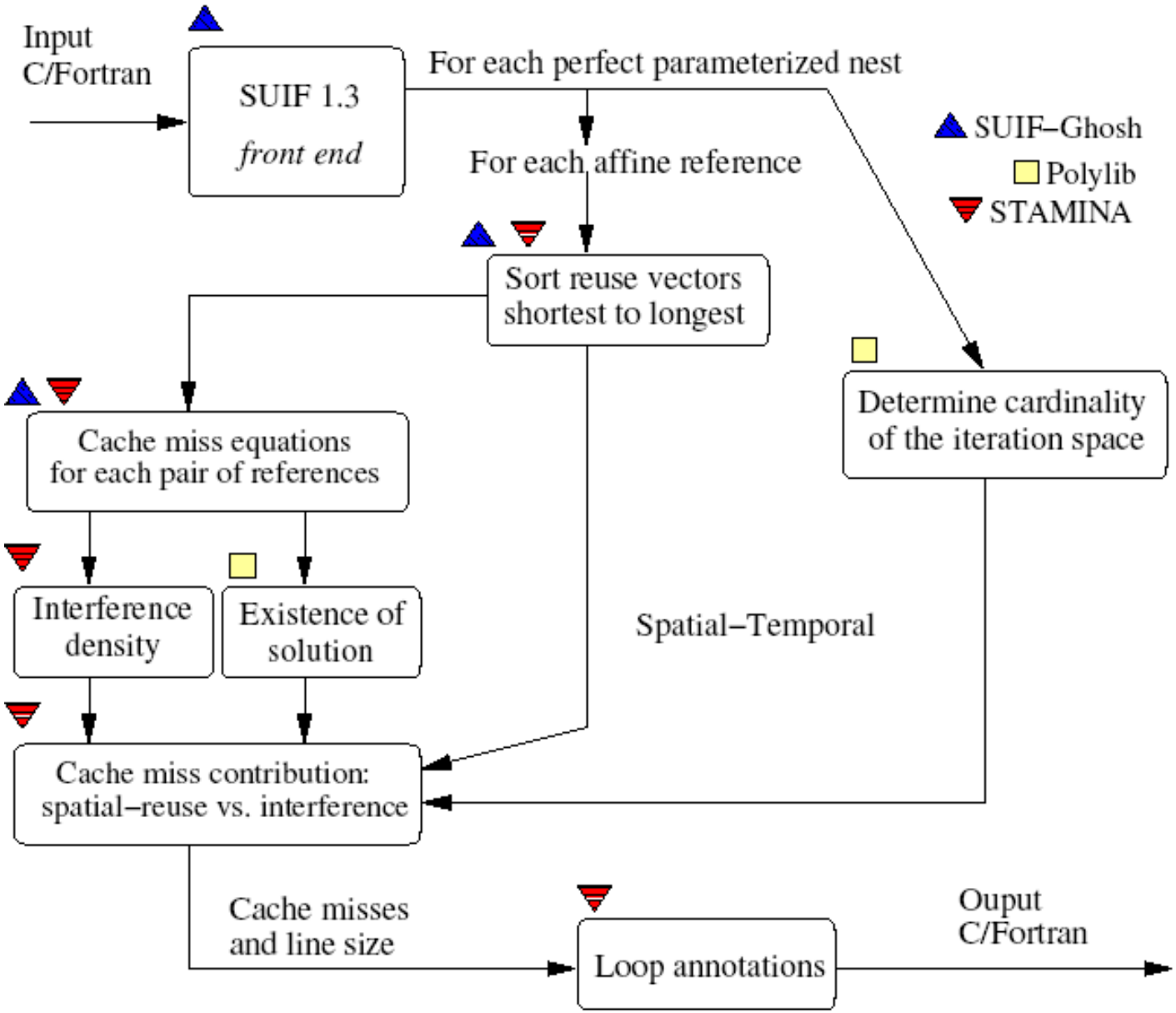
- ijk-Matrix Multiply
- STAMINA takes about 2min for *untiled* and 8hrs for tiled
- The tiled MM cannot be analyzed by CMEs
- Comparison with the “exact version”





MM without tiling

Implementation (organization)



Conclusions

- Architectural adaptation presents an opportunity to maximize performance based on application and data needs
- Energy consumption can be optimized within the same framework
- Compiler analysis and its integration with the runtime system is needed to achieve this
  - This work enables optimum tradeoff between conflict and reuse based on static analysis of nested loops
  - The result is a possible trade-off between energy and performance or energy optimization
- The model is validated by the experimental results

Future Applications: JuliusC and Static Analysis

- Measure of space complexity
  - Inference of problems size by symbolic analysis
- Measure of time complexity
  - Inference of number of operations by symbolic analysis
- Measure of Cache utilization
- Mapping processors <-> sub-problem
  - Work load balancing
  - Mapping investigation
- Matrix Languages
  - We can integrate SPIRAL/FFTW in a compiler environment

Thank you !