Getting the most out of Rcpp

High-Performance C++ in Practice

Matt P. Dziubinski

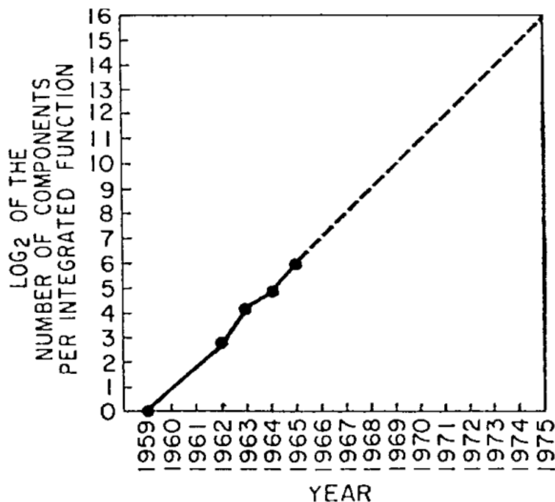
R/Finance 2016

matt@math.aau.dk // @matt_dz
Department of Mathematical Sciences, Aalborg University
CREATES (Center for Research in Econometric Analysis of Time Series)

Outline

- Performance
  - Why do we care?
  - What is it?
  - How to
    - measure it - reason about it - improve it?

Why?



Moore, Gordon E. (1965). "Cramming more components onto integrated circuits". Electronics Magazine.

Spending Moore's Dividend

"Grove giveth and Gates taketh away." – anon.

James Larus
larus@microsoft.com
Microsoft Research

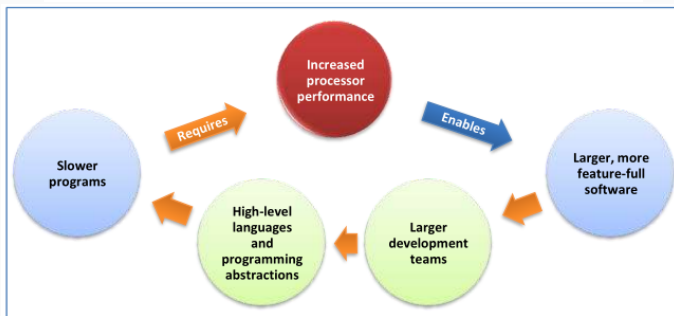


Figure 1. Cycle of innovation in computer industry.

"Spending Moore's Dividend," James Larus, Microsoft Research Technical Report MSR-TR-2008-69, May 2008.

Transformation Hierarchy

Problem

Algorithm

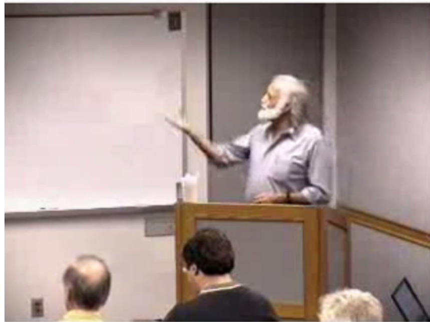
Program

ISA (Instruction Set Arch)

Microarchitecture

Circuits

Electrons



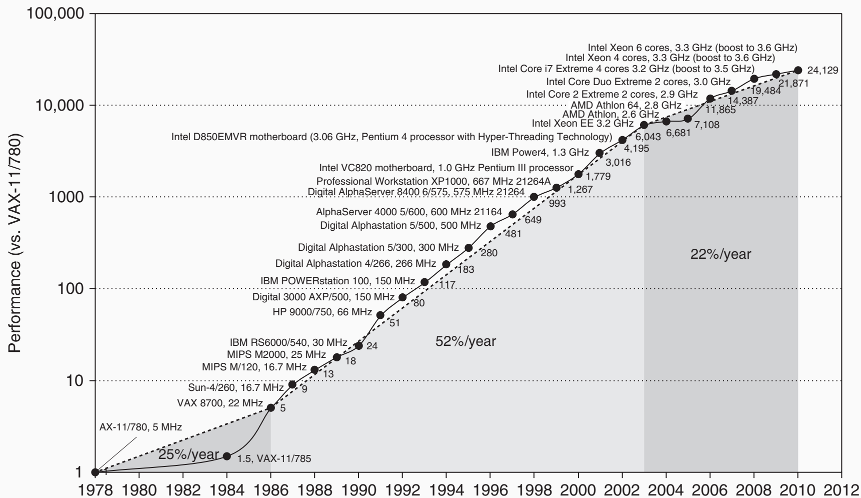
Yale N. Patt, *Microprocessor Performance, Phase 2: Can We Harness the Transformation Hierarchy*
https://youtube.com/watch?v=0fLlDkC625Q

Until Recently (Phase I)

- ***Maintain the artificial walls between the layers***
- ***Keeps the abstraction layers secure***
  - ***Makes for a better comfort zone***
- ***(Mostly) Improves the Microarchitecture***
  - ***Pipelining, Caches***
  - ***Branch Prediction, Speculative Execution***
  - ***Out-of-order Execution, Trace Cache***
- ***Today, we have too many transistors***

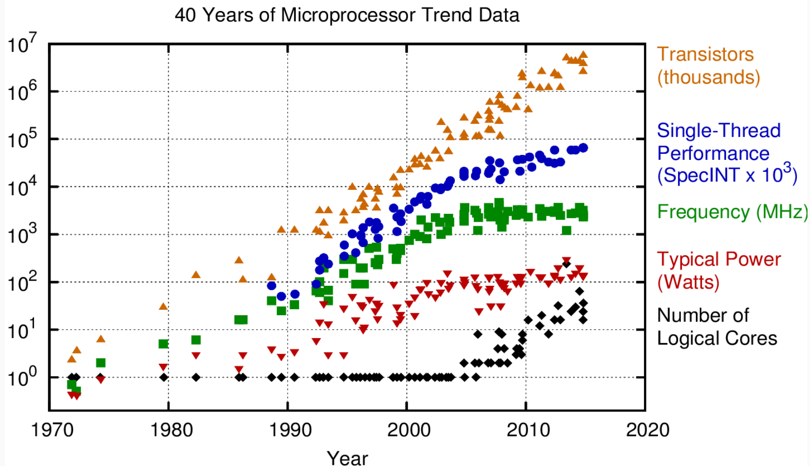
Yale N. Patt, *Microprocessor Performance, Phase 2: Can We Harness the Transformation Hierarchy*
https://youtube.com/watch?v=0fLlDkC625Q

CPU Performance Trends



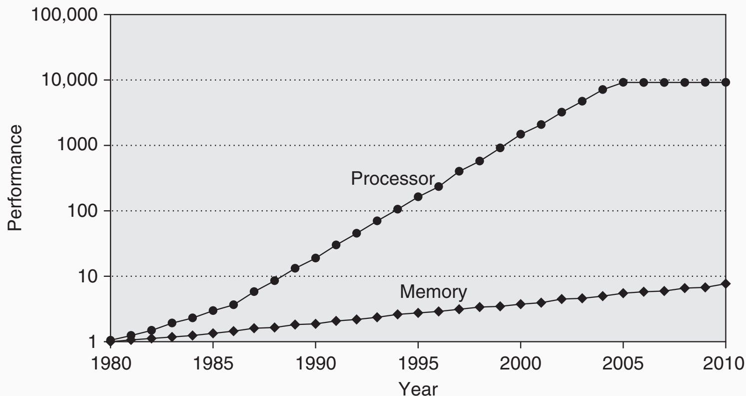
Hennessy, John L.; Patterson, David A., 2011, ”Computer Architecture: A Quantitative Approach,” Morgan Kaufmann.

40 Years of Microprocessor Trend Data



https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Processor-Memory Performance Gap



the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access

Hennessy, John L.; Patterson, David A., 2011, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann.

Computer Architecture is Back: Parallel Computing Landscape https://www.youtube.com/watch?v=On-k-E5HpcQ

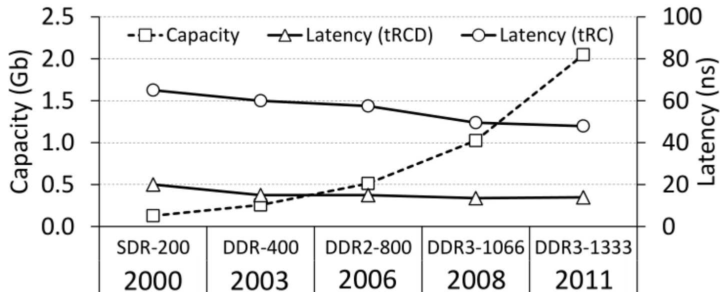


Figure 1.1: DRAM Capacity & Latency Over Time [27, 111, 207, 232]

D. Lee: Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity. http://arxiv.org/abs/1604.08041 (2016)

D. Lee et al., “Tiered-latency DRAM: A low latency and low cost DRAM architecture,” in HPCA, 2013.

Emerging Memory Technologies - Further Down The Hierarchy

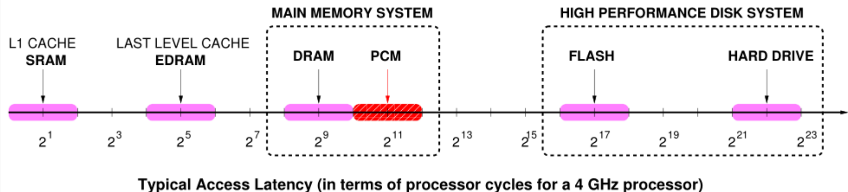


Figure 1: Latency of different technologies in memory hierarchy. Numbers accurate within a factor of two.

Qureshi et al., “Scalable high performance main memory system using phase-change memory technology,” ISCA 2009.

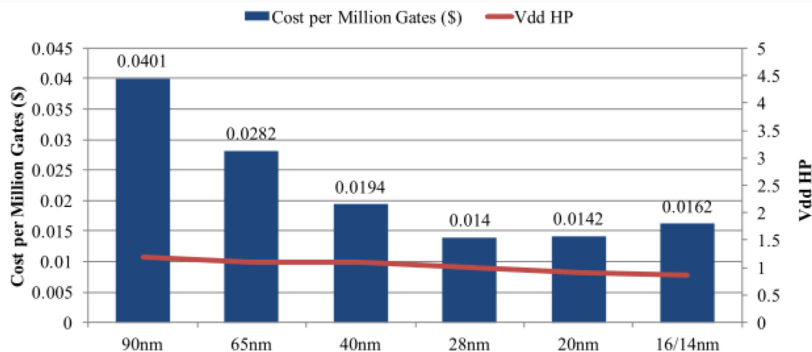


Figure 1.1: Trends in Cost per Gate and Nominal Vdd for Advanced HP (High-Performance) Process Nodes – Cost per gate data from [59] and nominal Vdd data from International Technology Roadmap of Semniconductors (ITRS).

Lee, Yunsup, "Decoupled Vector-Fetch Architecture with a Scalarizing Compiler," EECS Department, University of California, Berkeley. 2016. http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-82.html

What?

Performance: The Early Days

Origins of Matrix Methods

JIUZHANG SUANSHU – is a Chinese manuscript dating from approximately 200 BC and containing 246 problems intended to illustrate methods of solution for everyday problems in areas such as engineering, surveying, and trade. (To the left, we see the opening of Chapter 1.) Chapter 8 of this ancient document details the first known example of matrix methods with a method known as *fangcheng*. The method of solution, described in detail, is what would become known centuries later as Gaussian elimination. The coefficients of the system are written as a table on a "counting board." The text also compares two different fangcheng methods by counting the number of counting board operations needed in each method. [46, 7]

A. Greenbaum and T. Chartier. "Numerical Methods: Design, analysis, and computer implementation of algorithms." 2010. Course Notes for Short Course on Numerical Analysis.

Algorithms Classification Problem

ON THE COMPUTATIONAL COMPLEXITY OF ALGORITHMS

BY

J. HARTMANIS AND R. E. STEARNS

I. **Introduction.** In his celebrated paper [1], A. M. Turing investigated the computability of sequences (functions) by mechanical procedures and showed that the set of sequences can be partitioned into computable and noncomputable sequences. One finds, however, that some computable sequences are very easy to compute whereas other computable sequences seem to have an inherent complexity that makes them difficult to compute. In this paper, we investigate a scheme of classifying sequences according to how hard they are to compute. This scheme puts a rich structure on the computable sequences and a variety of theorems are established. Furthermore, this scheme can be generalized to classify numbers, functions, or recognition problems according to their computational complexity.

Hartmanis, J.; Stearns, R. E. (1965), "On the computational complexity of algorithms", Transactions of the American Mathematical Society 117: 285–306.

Algorithms Classification Problem

The computational complexity of a sequence is to be measured by how fast a multitape Turing machine can print out the terms of the sequence. This particular abstract model of a computing device is chosen because much of the work in this area is stimulated by the rapidly growing importance of computation through the use of digital computers, and all digital computers in a slightly idealized form belong to the class of multitape Turing machines. More specifically, if $T(n)$ is a computable, monotone increasing function of positive integers into positive integers and if α is a (binary) sequence, then we say that α is in complexity class S_T or that α is T-computable if and only if there is a multitape Turing machine $\mathscr{T}$ such that $\mathscr{T}$ computes the nth term of α within $T(n)$ operations.

Hartmanis, J.; Stearns, R. E. (1965), "On the computational complexity of algorithms",

Transactions of the American Mathematical Society 117: 285–306.

O(N)

http://en.cppreference.com/w/cpp/algorithm/find

Complexity: At most `last - first` applications of the corresponding predicate.

O(N·$log(N)$)

http://en.cppreference.com/w/cpp/algorithm/sort

Complexity: $\mathcal{O}(N \log(N))$ (where N `== last - first`) comparisons.

http://en.cppreference.com/w/cpp/algorithm/lower_bound

Complexity: At most $\log_2(last - first) + \mathcal{O}(1)$ comparisons.

log(N)

http://en.cppreference.com/w/cpp/container/set/find

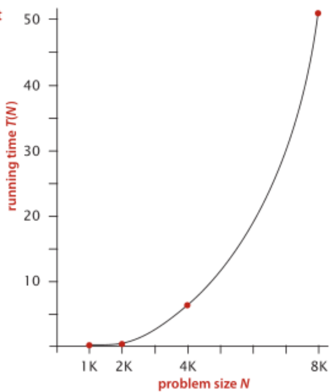
`a.find(k)`	`iterator`; `const_-iterator` for constant `a`.	returns an iterator pointing to an element with the key equivalent to `k`, or `a.end()` if such an element is not found	logarithmic

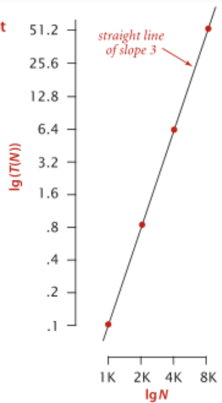
Scientific method The very same approach that scientists use to understand the natural world is effective for studying the running time of programs:

- *Observe* some feature of the natural world, generally with precise measurements.
- *Hypothesize* a model that is consistent with the observations.
- *Predict* events using the hypothesis.
- *Verify* the predictions by making further observations.
- *Validate* by repeating until the hypothesis and observations agree.

Robert Sedgewick and Kevin Wayne, ”Algorithms,” 4th Edition, Addison-Wesley Professional, 2011.

Analysis of Algorithms - Problem Size N vs. Running Time T(N)





Robert Sedgewick and Kevin Wayne, "Algorithms," 4th Edition, Addison-Wesley Professional, 2011.

Definition. We write $\sim f(N)$ to represent any function that, when divided by $f(N)$, approaches 1 as N grows, and we write $g(N) \sim f(N)$ to indicate that $g(N)/f(N)$ approaches 1 as N grows.

function	tilde approximation	order of growth
$N^3/6 - N^2/2 + N/3$	$\sim N^3/6$	N^3
$N^2/2 - N/2$	$\sim N^2/2$	N^2
$\lg N + 1$	$\sim \lg N$	$\lg N$
3	~ 3	1

Typical tilde approximations

Robert Sedgewick and Kevin Wayne, "Algorithms," 4th Edition, Addison-Wesley Professional, 2011.

Proposition C. (Doubling ratio) If $T(N) \sim a N^b \lg N$ then $T(2N)/T(N) \sim 2^b$.

Proof: Immediate from the following calculation:

$$
\begin{aligned}
T(2N)/T(N) &= a(2N)^b \lg(2N) / a N^b \lg N \\
&= 2^b (1 + \lg 2 / \lg N) \\
&\sim 2^b
\end{aligned}
$$

Robert Sedgewick and Kevin Wayne, ”Algorithms,” 4th Edition, Addison-Wesley Professional, 2011.

Rcpp Example - Pass & Sum Experiment - C++ Code I

```
// [[Rcpp::plugins(cpp11)]]
#include <algorithm> // std::accumulate
#include <cstddef> // std::size_t
#include <iterator> // std::begin, std::end
#include <vector> // std::vector
//#include <Rcpp.h> // note: not w/ <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppEigen)]]
#include <RcppEigen.h>

// [[Rcpp::export]]
double sum_Rcpp_NumericVector_for(const Rcpp::NumericVector input) {
    double sum = 0.0;
    for (R_xlen_t i = 0; i != input.size(); ++i)
        sum += input[i];
    return sum;
}
```

Rcpp Example - Pass & Sum Experiment - C++ Code II

```
// [[Rcpp::export]]
double sum_Rcpp_NumericVector_sugar(const Rcpp::NumericVector input) {
    return sum(input);
}

// [[Rcpp::export]]
double sum_std_vector_for(const std::vector<double> & input)
{
    double sum = 0.0;
    for (std::size_t i = 0; i != input.size(); ++i)
        sum += input[i];
    return sum;
}

// [[Rcpp::export]]
double sum_std_vector_accumulate(const std::vector<double> & input) {
    return std::accumulate(begin(input), end(input), 0.0);
}
```

Rcpp Example - Pass & Sum Experiment - C++ Code III

```
// [[Rcpp::export]]
double sum_Eigen_Vector(const Eigen::VectorXd & input) {
    return input.sum();
}

// [[Rcpp::export]]
double sum_Eigen_Map(const Eigen::Map<Eigen::VectorXd> input) {
    return input.sum();
}

// [[Rcpp::export]]
double sum_Eigen_Map_for(const Eigen::Map<Eigen::VectorXd> input) {
    double sum = 0.0;
    for (Eigen::DenseIndex i = 0; i != input.size(); ++i)
        sum += input(i);
    return sum;
}
```

Rcpp Example - Pass & Sum Experiment - C++ Code IV

```
// [[Rcpp::export]]
double sum_Arma_ColVec(const arma::colvec & input) {
  return sum(input);
}

// [[Rcpp::export]]
double sum_Arma_RowVec(const arma::rowvec & input)  {
  return sum(input);
}
```

Note:

types with reference semantics - pass-by-value (`const`)

types with value semantics - pass-by-reference (-to-`const`)

Rcpp Example - Pass & Sum Experiment - R Benchmark

```r
size = 10000000L
v = rep_len(1.0, size)

library("rbenchmark")

benchmark(sum_Rcpp_NumericVector_for(v), sum_Rcpp_NumericVector_sugar(v),
          sum_std_vector_for(v), sum_std_vector_accumulate(v),
          sum_Eigen_Vector(v), sum_Eigen_Map(v), sum_Eigen_Map_for(v),
          sum_Arma_ColVec(v), sum_Arma_RowVec(v),
          order = "relative",
          columns = c("test", "replications", "elapsed", "relative",
                      "user.self", "sys.self"))
```

```
                          test replications elapsed relative user.self
6                sum_Eigen_Map(v)          100    0.64    1.000      0.64
2 sum_Rcpp_NumericVector_sugar(v)          100    0.67    1.047      0.67
9               sum_Arma_RowVec(v)         100    0.67    1.047      0.67
8               sum_Arma_ColVec(v)         100    0.68    1.062      0.67
7             sum_Eigen_Map_for(v)         100    1.41    2.203      1.41
4     sum_std_vector_accumulate(v)         100    4.80    7.500      2.55
3            sum_std_vector_for(v)         100    4.82    7.531      2.49
5              sum_Eigen_Vector(v)         100    4.88    7.625      2.92
1   sum_Rcpp_NumericVector_for(v)          100    6.68   10.438      6.67
```

- `sum_Rcpp_NumericVector_sugar` faster than `sum_Rcpp_NumericVector_for`
- `sum_std_vector_for`, `sum_std_vector_accumulate`, and `sum_Eigen_Vector` slow
- `sum_Eigen_Map_for` slower than `sum_Eigen_Map`

`sum_Rcpp_NumericVector_sugar` faster than `sum_Rcpp_NumericVector_for`

Why?

```
// [[Rcpp::export]]
double sum_Rcpp_NumericVector_for(const Rcpp::NumericVector input) {
    double sum = 0.0;
    for (R_xlen_t i = 0; i != input.size(); ++i)
        sum += input[i];
    return sum;
}

// https://github.com/RcppCore/Rcpp/
// /blob/master/inst/include/Rcpp/sugar/functions/sum.h

        R_xlen_t n = object.size();
        for (R_xlen_t i = 0; i < n; i++)
```

Alternative `sum_Rcpp_NumericVector_for`:

```
for (R_xlen_t i = 0, n = input.size(); i != n; ++i)
```

Rcpp Example - Pass & Sum Experiment - New Results

```
                          test replications elapsed relative user.self
6               sum_Eigen_Map(v)          100    0.67    1.000      0.68
1  sum_Rcpp_NumericVector_for(v)          100    0.69    1.030      0.69
9              sum_Arma_RowVec(v)          100    0.69    1.030      0.69
2 sum_Rcpp_NumericVector_sugar(v)          100    0.70    1.045      0.70
8              sum_Arma_ColVec(v)          100    0.70    1.045      0.67
7           sum_Eigen_Map_for(v)          100    1.48    2.209      1.48
3          sum_std_vector_for(v)          100    4.90    7.313      2.89
4   sum_std_vector_accumulate(v)          100    5.01    7.478      2.67
5            sum_Eigen_Vector(v)          100    5.17    7.716      2.84
```

`sum_Rcpp_NumericVector_sugar` now on par w/ `sum_Rcpp_NumericVector_for`

Next:

Why is `std::vector` slow - and `sum_Eigen_Vector` even slower than `sum_Eigen_Map_for`?

Hypothesis: (Implicit) conversions — passing means copying — linear complexity.

Rcpp Example - Doubling Ratio Experiment - std::vector

Hypothesis: (Implicit) conversions — passing means copying — linear complexity.

How to check? Recall doubling ratio experiments!

C++:

```cpp
#include <vector>

// [[Rcpp::export]]
double pass_std_vector(const std::vector<double> & v) {
  return v.back();
}
```

R:

```r
pass_vector = pass_std_vector

v = seq(from = 0.0, to = 1.0, length.out = 10000000)
system.time(pass_vector(v))

v = seq(from = 0.0, to = 1.0, length.out = 20000000)
system.time(pass_vector(v))

v = seq(from = 0.0, to = 1.0, length.out = 40000000)
system.time(pass_vector(v))
```

Timing results: `0.04, 0.08, 0.17`

Rcpp Example - Doubling Ratio Experiment - Rcpp::NumericVector

C++:

```
#include <Rcpp.h>

// [[Rcpp::export]]
double pass_Rcpp_vector(const Rcpp::NumericVector v) {
  return v[v.size() - 1];
}
```

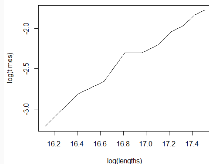
R:

```
pass_vector = pass_Rcpp_vector

v = seq(from = 0.0, to = 1.0, length.out = 10000000)
system.time(pass_vector(v))

v = seq(from = 0.0, to = 1.0, length.out = 20000000)
system.time(pass_vector(v))

v = seq(from = 0.0, to = 1.0, length.out = 40000000)
system.time(pass_vector(v))
```

Timing results: `0, 0, 0`

Rcpp Example - Timing Experiments - std::vector

```r
pass_vector = pass_std_vector

time_argpass = function(length) {
    show(length)
    v = seq(from = 0.0, to = 1.0, length.out = length)
    r = system.time(pass_vector(v))["elapsed"]
    show(r)
    r
}

lengths = seq(from = 10000000, to = 40000000, length.out = 10)
times = sapply(lengths, time_argpass)

plot(log(lengths), log(times), type="l")
```




```r
lm(log(times) ~ log(lengths))

Coefficients:
 (Intercept)  log(lengths)
     -19.347         1.005
```

C++ - Copies vs. Views

- implicit conversions - costly copies - worth avoiding
- analogous case: `std::string` vs. `boost::string_ref`
- C++17: `std::string_view`

http://www.boost.org/doc/libs/master/libs/utility/doc/html/string_ref.html
http://en.cppreference.com/w/cpp/experimental/basic_string_view

http://en.cppreference.com/w/cpp/string/basic_string_view

Computer Architecture: A Science of Tradeoffs

"My tongue in cheek phrase to emphasize the importance of tradeoffs to the discipline of computer architecture. Clearly, computer architecture is more art than science. Science, we like to think, involves a coherent body of knowledge, even though we have yet to figure out all the connections. Art, on the other hand, is the result of individual expressions of the various artists. Since each computer architecture is the result of the individual(s) who specified it, there is no such completely coherent structure. So, I opined if computer architecture is a science at all, it is a science of tradeoffs. In class, we keep coming up with design choices that involve tradeoffs. In my view, "tradeoffs" is at the heart of computer architecture." — Yale N. Patt

Software Performance Optimization - Analogous!

The multiplicity of tradeoffs:

- Multidimensional
- Multiple levels
- Costs and benefits

$$\text{minimize}_{x \in \mathbb{R}^d} \quad f(x) = \sum_{j=1}^{N} f_j(x)$$

Algorithm	Time per iteration	Error after T iterations	Error after N items
Newton	O(d²N+d³)	${C_I}^{2^T}$	${C_I}^2$
Gradient	O(dN)	${C_G}^T$	C_G
SGD	O(d) (or constant)	$\frac{C_S}{T}$	$\frac{C_S}{N}$

Ben Recht, Feng Niu, Christopher Ré, Stephen Wright. ”Lock-Free Approaches to Parallelizing Stochastic Gradient Descent” OPT 2011: 4th International Workshop on Optimization for Machine Learning
http://opt.kyb.tuebingen.mpg.de/slides/opt2011-recht.pdf

Gradient computation - accuracy vs. function evaluations

$f : \mathbb{R}^d \rightarrow \mathbb{R}^N$

- Finite differencing:
  - forward-difference: $O(\sqrt{\epsilon_M})$ error, d O(Cost(f)) evaluations
  - central-difference: $O(\epsilon_M^{2/3})$ error, 2d O(Cost(f)) evaluations
    w/ the *machine epsilon* $\epsilon_M := \inf\{\epsilon > 0 : 1.0 + \epsilon \neq 1.0\}$
- Algorithmic differentiation (AD): precision - as in hand-coded analytical gradient
  - rough forward-mode cost d O(Cost(f))
  - rough reverse-mode cost N O(Cost(f))

Least Squares - Trade-offs

LS Algorithm	Flop Count
Normal equations	$mn^2 + n^3/3$
Householder QR	$n^3/3$
Modified Gram-Schmidt	$2mn^2$
Givens QR	$3mn^2 - n^3$
Householder Bidiagonalization	$4mn^2 - 2n^3$
R-Bidiagonalization	$2mn^2 + 2n^3$
SVD	$4mn^2 + 8n^3$
R-SVD	$2mn^2 + 11n^3$

Figure 5.5.1. *Flops associated with various least squares methods*

Golub & van Van Loan (2013) "Matrix Computations"

Trade-offs: FLOPs (FLoating-point OPerations) vs. Applicability / Numerical Stability / Speed / Accuracy

Example: Catalogue of dense decompositions

http://eigen.tuxfamily.org/dox/group__TopicLinearAlgebraDecompositions.html

Trade-offs: Costs and Benefits

That point is that people work very hard to attain every microsecond of speed that a computer demonstrates, and there are two major problems facing an implementor when he embarks on producing a Lisp system: the first problem is the myriad of decisions to be made, the interactions of various parts of the Lisp system when they are brought together, the unfortunate choice in one aspect of the system turing around and influencing, badly, the performance of another; the second problem is that writing a Lisp system is a monumental undertaking, and this undertaking is executed within the context of living a life as well. And, although an implementor might start out with large goals and spectacular intentions, the time it takes to do the thorough job required to produce an excellent Lisp system will bring many obstacles and intrusions, impediments and obstructions, and in the end, Time will have won out, in that every microsecond the implementor grabs from the hands of Time are bought with hours or days or weeks or months of effort expended by the implementor.

Gabriel, Richard P. (1985). Performance and Evaluation of Lisp Systems. Cambridge, Mass: MIT Press; Computer Systems Series.

Costs and Benefits: Implications

- Important to know what to focus on
- Optimize the optimization: so that it doesn't always take *hours or days or weeks or months…*

How?

Asymptotic growth & "random access machines"?

The Cost of Address Translation: http://arxiv.org/abs/1212.0703

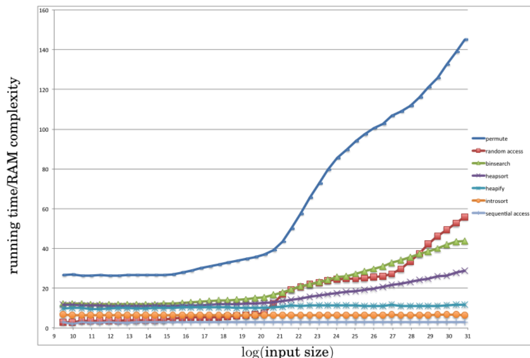


Fig. 1. The abscissa shows the logarithm of the input size. The ordinate shows the measured running time divided by the RAM-complexity (normalized operation time). The normalized operation times of sequential access, quicksort, and heapify are constant, the normalized operation times of the other programs are not.

Asymptotic - growing problem size - but for large data need to take into account the costs of actually bringing it in - communication complexity vs. computation complexity, including overlapping computation-communication latencies

Complexity - constants, microarchitecture?

"Array Layouts for Comparison-Based Searching"
Paul-Virak Khuong, Pat Morin
http://cglab.ca/~morin/misc/arraylayout-v2/

- "With this understanding, we are able to choose layouts and design search algorithms that perform searches in 1/2 to 2/3 (depending on the array length) the time of the C++ std::lower_bound() implementation of binary search

Complexity - constants, microarchitecture?

"Array Layouts for Comparison-Based Searching"
Paul-Virak Khuong, Pat Morin
http://cglab.ca/~morin/misc/arraylayout-v2/

- "With this understanding, we are able to choose layouts and design search algorithms that perform searches in 1/2 to 2/3 (depending on the array length) the time of the C++ std::lower_bound() implementation of binary search
- (which itself performs searches in 1/3 the time of searching in the std::set implemementation of red-black trees).

Complexity - constants, microarchitecture?

”Array Layouts for Comparison-Based Searching”
Paul-Virak Khuong, Pat Morin
http://cglab.ca/~morin/misc/arraylayout-v2/

- ”With this understanding, we are able to choose layouts and design search algorithms that perform searches in 1/2 to 2/3 (depending on the array length) the time of the C++ std::lower_bound() implementation of binary search
- (which itself performs searches in 1/3 the time of searching in the std::set implemementation of red-black trees).
- It was only through careful and controlled experimentation with different implementations of each of the search algorithms that we are able to understand how the interactions between processor features such as pipelining, prefetching, speculative execution, and conditional moves affect the running times of the search algorithms.”

Reasoning about Performance: The Scientific Method

Requires - enabled by - the knowledge of microachitectural details.

- *Develop alternative hypotheses*. Hypothesis 1: The program has synchronization bottlenecks. Hypothesis 2: The program is taking too many cache misses. Having multiple hypotheses gives parallelism to the following steps and helps keep us from getting too attached to one hypothesis.
- *Develop one or more experiments that can exclude or corroborate an alternative hypothesis*. Experiment 1: Add code in every critical section that stalls for time T and counts how often it is executed. Can you develop experiments for Hypothesis 2?
- *Predict experimental results before running the experiment*. If Hypothesis 1 is true, a P-processor program that executes S stalls should slow by much more than $T \times S/P$. If not, Hypothesis 1 is excluded.
- *Run experiments*. If the program runs only $T \times S/P$ slower, then Hypothesis 1 is excluded. We can continue with experiments for our alternatives or return to the first step and develop new hypotheses. If the program does run much more slowly, then Hypothesis 1 is corroborated. We should develop refined hypotheses (e.g., the synchronization bottleneck is for data structure A or B) and return to the first step.

Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, Chapter 2 "Methods" from "Readings in Computer Architecture," Morgan Kaufmann, 2000.

Prefetching benefits evaluation: Disable/enable prefetchers using likwid-features: https://github.com/RRZE-HPC/likwid/wiki/likwid-features

Example: https://gist.github.com/MattPD/06e293fb935eaf67ee9c301e70db6975

Microarchitecture

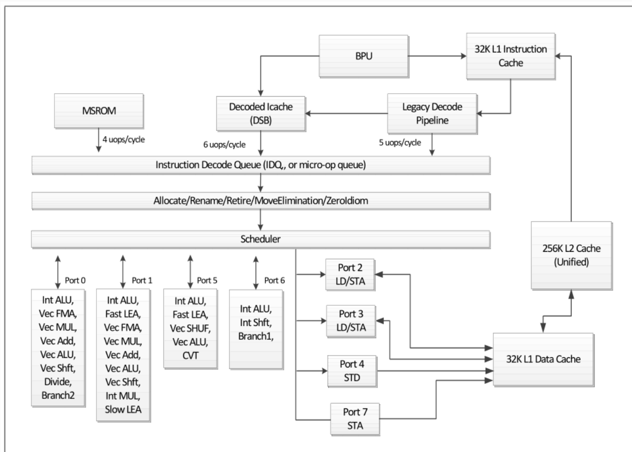


Figure 2-1. CPU Core Pipeline Functionality of the Skylake Microarchitecture

Intel® 64 and IA-32 Architectures Optimization Reference Manual
https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

Pervasive CPU Parallelism

pipeline-level parallelism (PLP)

instruction-level parallelism (ILP)

memory-level parallelism (MLP)

data-level parallelism (DLP)

thread-level parallelism (TLP)

Instruction Level Parallelism & Loop Unrolling - Code I

```
#include <cstddef>
#include <cstdint>
#include <cstdlib>
#include <iostream>
#include <vector>
#include <boost/timer/timer.hpp>
```

Instruction Level Parallelism & Loop Unrolling - Code II

```
typedef double T;


T sum_1(const std::vector<T> & input) {
    T sum = 0.0;
    for (std::size_t i = 0, n = input.size(); i != n; ++i)
        sum += input[i];
    return sum;
}

T sum_2(const std::vector<T> & input) {
    T sum1 = 0.0, sum2 = 0.0;
    for (std::size_t i = 0, n = input.size(); i != n; i += 2) {
        sum1 += input[i];
        sum2 += input[i + 1];
    }
    return sum1 + sum2;
}
```

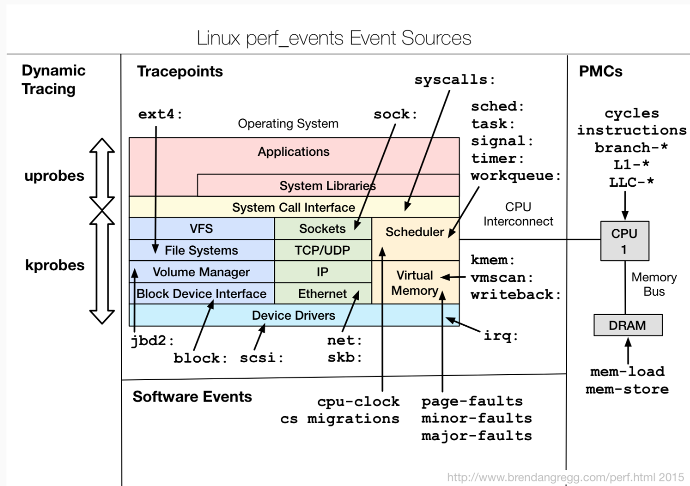
```
int main(int argc, char * argv[]) {
    std::size_t n = (argc >= 2) ? std::atoll(argv[1]) : 10000000;
    std::size_t f = (argc >= 3) ? std::atoll(argv[2]) : 1;
    std::cout << "n = " << n << '\n'; // iterations count
    std::cout << "f = " << f << '\n'; // unroll factor
    std::vector<T> a(n, T(1));
    boost::timer::auto_cpu_timer timer;
    T sum = (f == 1) ? sum_1(a)
          : (f == 2) ? sum_2(a)
          : 0;
    std::cout << sum << '\n';
}
```

```
make vector_sums CXXFLAGS="-std=c++14 -O2 -march=native"
LDLIBS=-lboost_timer
```

```
$ ./vector_sums 1000000000 2
n = 1000000000
f = 2
1e+09
0.466293s wall, 0.460000s user + 0.000000s system = 0.460000s CPU (98.7%)

$ ./vector_sums 1000000000 1
n = 1000000000
f = 1
1e+09
0.841269s wall, 0.840000s user + 0.010000s system = 0.850000s CPU (101.0%)
```

PERF

- https://perf.wiki.kernel.org/
- http://www.brendangregg.com/perf.html



perf Results - sum_1

```
Performance counter stats for './vector_sums 1000000000 1':

  1675.812457      task-clock (msec)         #    0.850 CPUs utilized
           34      context-switches          #    0.020 K/sec
            5      cpu-migrations            #    0.003 K/sec
        8,953      page-faults               #    0.005 M/sec
5,760,418,457      cycles                    #    3.437 GHz
3,456,046,515      stalled-cycles-frontend   #   60.00% frontend cycles id
8,225,763,566      instructions              #    1.43  insns per cycle
                                             #    0.42  stalled cycles per
2,050,710,005      branches                  # 1223.711 M/sec
      104,331      branch-misses             #    0.01% of all branches

1.970909249 seconds time elapsed
```

perf Results - sum_2

```
Performance counter stats for './vector_sums 1000000000 2':

  1283.910371      task-clock (msec)         #    0.835 CPUs utilized
           38      context-switches          #    0.030 K/sec
            3      cpu-migrations            #    0.002 K/sec
        9,466      page-faults               #    0.007 M/sec
4,458,594,733      cycles                    #    3.473 GHz
2,149,690,303      stalled-cycles-frontend   #   48.21% frontend cycles id
6,734,925,029      instructions              #    1.51  insns per cycle
                                             #    0.32  stalled cycles per
1,552,029,608      branches                  # 1208.830 M/sec
      119,358      branch-misses             #    0.01% of all branches

1.537971058 seconds time elapsed
```

GCC Explorer: sum_1

http://gcc.godbolt.org/

```cpp
#include <cstddef>
#include <cstdint>
#include <vector>

typedef double T;

T sum_1(const std::vector<T> & input)
{
    T sum = 0.0;
    for (std::size_t i = 0, n = input.size(); i != n; ++i)
        sum += input[i];
    return sum;
}
```

```asm
1  sum_1(std::vector<double, std::allocator<double> > const&):
2          mov     rcx, QWORD PTR [rdi]
3          mov     rdx, QWORD PTR [rdi+8]
4          sub     rdx, rcx
5          sar     rdx, 3
6          je      .L4
7          xor     eax, eax
8          vxorpd  xmm0, xmm0, xmm0
9  .L3:
10         vaddsd  xmm0, xmm0, QWORD PTR [rcx+rax*8]
11         add     rax, 1
12         cmp     rax, rdx
13         jne     .L3
14         ret
15 .L4:
16         vxorpd  xmm0, xmm0, xmm0
17         ret
```

GCC Explorer: sum_2

http://gcc.godbolt.org/

```cpp
#include <cstddef>
#include <cstdint>
#include <vector>

typedef double T;

T sum_2(const std::vector<T> & input) {
  T sum1 = 0.0, sum2 = 0.0;
  for (std::size_t i = 0, n = input.size(); i != n; i += 2) {
    sum1 += input[i];
    sum2 += input[i + 1];
  }
  return sum1 + sum2;
}
```

```asm
1  sum_2(std::vector<double, std::allocator<double> > const&):
2          mov     rdx, QWORD PTR [rdi]
3          mov     rcx, QWORD PTR [rdi+8]
4          sub     rcx, rdx
5          sar     rcx, 3
6          je      .L4
7          vxorpd  xmm1, xmm1, xmm1
8          xor     eax, eax
9          vmovapd xmm0, xmm1
10 .L3:
11         vaddsd  xmm0, xmm0, QWORD PTR [rdx+rax*8]
12         vaddsd  xmm1, xmm1, QWORD PTR [rdx+8+rax*8]
13         add     rax, 2
14         cmp     rax, rcx
15         jne     .L3
16         vaddsd  xmm0, xmm0, xmm1
17         ret
18 .L4:
19         vxorpd  xmm0, xmm0, xmm0
20         ret
```

Intel Architecture Code Analyzer (IACA)

```cpp
#include <iacaMarks.h>

T sum_2(const std::vector<T> & input) {
    T sum1 = 0.0, sum2 = 0.0;
    for (std::size_t i = 0, n = input.size(); i != n; i += 2) {
        IACA_START
        sum1 += input[i];
        sum2 += input[i + 1];
    }
    IACA_END
    return sum1 + sum2;
}
```

```
$ g++ -std=c++14 -O2 -march=native vector_sums_2i.cpp
-o vector_sums_2i
$ iaca -64 -arch IVB -graph ./vector_sums_2i
```

- https://software.intel.com/en-us/articles/intel-architecture-code-analyzer
- https://stackoverflow.com/questions/26021337/what-is-iaca-and-how-do-i-use-it
- http://kylehegeman.com/blog/2013/12/28/introduction-to-iaca/

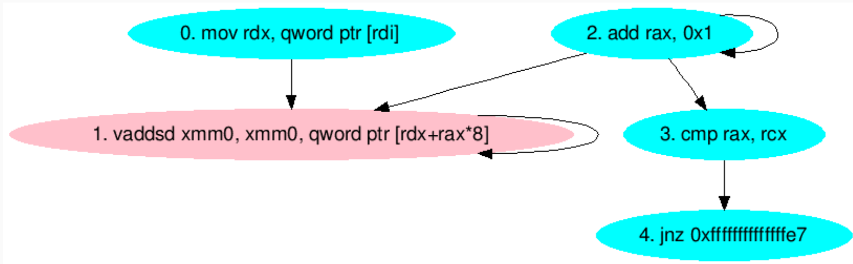
IACA Results - sum_1

```
$ iaca -64 -arch IVB -graph ./vector_sums_1i
Intel(R) Architecture Code Analyzer Version - 2.1
Analyzed File - ./vector_sums_1i
Binary Format - 64Bit
Architecture  - IVB
Analysis Type - Throughput

Throughput Analysis Report
--------------------------
Block Throughput: 3.00 Cycles       Throughput Bottleneck: InterIteration

Port Binding In Cycles Per Iteration:
-------------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |
-------------------------------------------------------------------------
| Cycles | 1.0    0.0  | 1.0  | 1.0    1.0  | 1.0    1.0  | 0.0  | 1.0  |
-------------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256 instruction, dozens of cycles penalty is expected
! - instruction not supported, was not accounted in Analysis

| Num Of |              Ports pressure in cycles               |    |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |    |
---------------------------------------------------------------------
|   1    |           |     | 1.0   1.0 |           |     |     |    | mov rdx, qword ptr [rdi]
|   2    |           | 1.0 |           | 1.0   1.0 |     |     | CP | vaddsd xmm0, xmm0, qword ptr [rdx+rax*8]
|   1    | 1.0       |     |           |           |     |     |    | add rax, 0x1
|   1    |           |     |           |           |     | 1.0 |    | cmp rax, rcx
|   0F   |           |     |           |           |     |     |    | jnz 0xffffffffffffffe7
Total Num Of Uops: 5
```

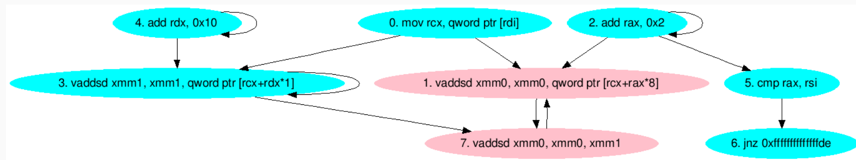
IACA Results - sum_2

```
$ iaca -64 -arch IVB -graph ./vector_sums_2i
Intel(R) Architecture Code Analyzer Version - 2.1
Analyzed File - ./vector_sums_2i
Binary Format - 64Bit
Architecture  - IVB
Analysis Type - Throughput

Throughput Analysis Report
--------------------------
Block Throughput: 6.00 Cycles       Throughput Bottleneck: InterIteration

Port Binding In Cycles Per Iteration:
-------------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |
-------------------------------------------------------------------------
| Cycles | 1.5    0.0  | 3.0  | 1.5    1.5  | 1.5    1.5  | 0.0  | 1.5  |
-------------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256 instruction, dozens of cycles penalty is expected
! - instruction not supported, was not accounted in Analysis

| Num Of |              Ports pressure in cycles               |    |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |    |
---------------------------------------------------------------------
|   1    |           |     | 0.5   0.5 | 0.5   0.5 |     |     |    | mov rcx, qword ptr [rdi]
|   2    |           | 1.0 | 0.5   0.5 | 0.5   0.5 |     |     | CP | vaddsd xmm0, xmm0, qword ptr [rcx+rax*8]
|   1    | 1.0       |     |           |           |     |     |    | add rax, 0x2
|   2    |           | 1.0 | 0.5   0.5 | 0.5   0.5 |     |     |    | vaddsd xmm1, xmm1, qword ptr [rcx+rdx*1]
|   1    | 0.5       |     |           |           |     | 0.5 |    | add rdx, 0x10
|   1    |           |     |           |           |     | 1.0 |    | cmp rax, rsi
|   0F   |           |     |           |           |     |     |    | jnz 0xffffffffffffffde
|   1    |           | 1.0 |           |           |     |     | CP | vaddsd xmm0, xmm0, xmm1
Total Num Of Uops: 9
```

IACA Data Dependency Graph - sum_1



IACA Data Dependency Graph - sum_2



Empty Issue Slots: Horizontal Waste & Vertical Waste

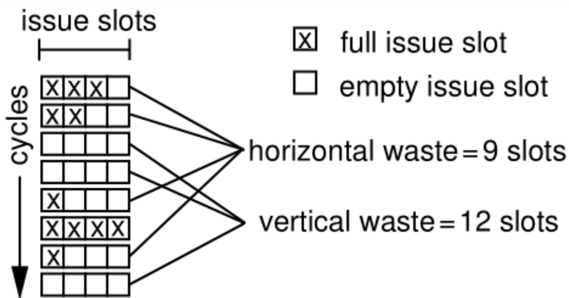


Figure 1: **Empty issue slots can be defined as either vertical waste or horizontal waste. Vertical waste is introduced when the processor issues no instructions in a cycle, horizontal waste when not all issue slots can be filled in a cycle. Superscalar execution (as opposed to single-issue execution) both introduces horizontal waste and increases the amount of vertical waste.**

D. M. Tullsen, S. J. Eggers and H. M. Levy, ”Simultaneous multithreading: Maximizing on-chip parallelism,” Proceedings, 22nd Annual International Symposium on Computer Architecture, 1995.

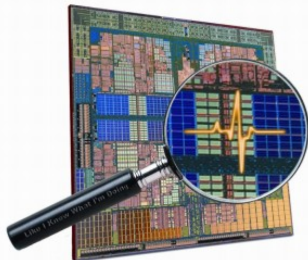
Wasted Slots: Causes

Source of Wasted Issue Slots	Possible Latency-Hiding or Latency-Reducing Technique
instruction tlb miss, data tlb miss	decrease the TLB miss rates (e.g., increase the TLB sizes); hardware instruction prefetching; hardware or software data prefetching; faster servicing of TLB misses
I cache miss	larger, more associative, or faster instruction cache hierarchy; hardware instruction prefetching
D cache miss	larger, more associative, or faster data cache hierarchy; hardware or software prefetching; improved instruction scheduling; more sophisticated dynamic execution
branch misprediction	improved branch prediction scheme; lower branch misprediction penalty
control hazard	speculative execution; more aggressive if-conversion
load delays (first-level cache hits)	shorter load latency; improved instruction scheduling; dynamic scheduling
short integer delay	improved instruction scheduling
long integer, short fp, long fp delays	(multiply is the only long integer operation, divide is the only long floating point operation) shorter latencies; improved instruction scheduling
memory conflict	(accesses to the same memory location in a single cycle) improved instruction scheduling

Table 3: **All possible causes of wasted issue slots, and latency-hiding or latency-reducing techniques that can reduce the number of cycles wasted by each cause.**

D. M. Tullsen, S. J. Eggers and H. M. Levy, ”Simultaneous multithreading: Maximizing on-chip parallelism,” Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on, Santa Margherita Ligure, Italy, 1995, pp. 392-403.

- https://github.com/RRZE-HPC/likwid
- https://github.com/RRZE-HPC/likwid/wiki
- https://github.com/RRZE-HPC/likwid/wiki/likwid-perfctr



LIKWID:

Lightweight performance tools

J. Treibig

RRZE, University Erlangen

likwid Results - sum_1: 489 Scalar MUOPS/s

```
$ likwid-perfctr -C S0:0 -g FLOPS_DP -f ./vector_sums 1000000000 1
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
CPU type:       Intel Core IvyBridge processor
CPU clock:      2.59 GHz
--------------------------------------------------------------------------------
n = 1000000000
f = 1
1e+09
 1.090122s wall, 0.880000s user + 0.000000s system = 0.880000s CPU (80.7%)
--------------------------------------------------------------------------------
Group 1: FLOPS_DP
+--------------------------------------+---------+------------+
|                 Event                | Counter |   Core 0   |
+--------------------------------------+---------+------------+
|           INSTR_RETIRED_ANY          |  FIXC0  | 8002493499 |
|         CPU_CLK_UNHALTED_CORE        |  FIXC1  | 4285189526 |
|         CPU_CLK_UNHALTED_REF         |  FIXC2  | 3258346806 |
| FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE |   PMC0  |      0     |
| FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE |   PMC1  | 1000155741 |
|       SIMD_FP_256_PACKED_DOUBLE      |   PMC2  |      0     |
+--------------------------------------+---------+------------+

+----------------------+-----------+
|        Metric        |   Core 0  |
+----------------------+-----------+
|  Runtime (RDTSC) [s] |   2.0456  |
| Runtime unhalted [s] |   1.6536  |
|      Clock [MHz]     | 3408.2011 |
|          CPI         |   0.5355  |
|        MFLOP/s       |  488.9303 |
|      AVX MFLOP/s     |     0     |
|    Packed MUOPS/s    |     0     |
|    Scalar MUOPS/s    |  488.9303 |
+----------------------+-----------+
```

likwid Results - sum_2: 595 Scalar MUOPS/s

```
$ likwid-perfctr -C S0:0 -g FLOPS_DP -f ./vector_sums 1000000000 2
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
CPU type:       Intel Core IvyBridge processor
CPU clock:      2.59 GHz
--------------------------------------------------------------------------------
n = 1000000000
f = 2
1e+09
 0.620421s wall, 0.470000s user + 0.000000s system = 0.470000s CPU (75.8%)
--------------------------------------------------------------------------------
Group 1: FLOPS_DP
+--------------------------------------+---------+------------+
|                 Event                | Counter |   Core 0   |
+--------------------------------------+---------+------------+
|           INSTR_RETIRED_ANY          |  FIXC0  | 6502566958 |
|         CPU_CLK_UNHALTED_CORE        |  FIXC1  | 2948446599 |
|         CPU_CLK_UNHALTED_REF         |  FIXC2  | 2223894218 |
| FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE |   PMC0  |      0     |
| FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE |   PMC1  | 1000328727 |
|       SIMD_FP_256_PACKED_DOUBLE      |   PMC2  |      0     |
+--------------------------------------+---------+------------+

+----------------------+-----------+
|        Metric        |   Core 0  |
+----------------------+-----------+
|  Runtime (RDTSC) [s] |   1.6809  |
| Runtime unhalted [s] |   1.1377  |
|      Clock [MHz]     | 3435.8987 |
|          CPI         |   0.4534  |
|        MFLOP/s       |  595.1079 |
|      AVX MFLOP/s     |     0     |
|    Packed MUOPS/s    |     0     |
|    Scalar MUOPS/s    |  595.1079 |
+----------------------+-----------+
```

likwid Results: sum_vectorized: 676 AVX MFLOP/s

g++ -std=c++14 -O2 -ftree-vectorize -ffast-math -march=native -lboost_timer vector_sums.cpp -o vector_sums_vf

```
$ likwid-perfctr -C S0:0 -g FLOPS_DP -f ./vector_sums_vf 1000000000 1
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
CPU type:       Intel Core IvyBridge processor
CPU clock:      2.59 GHz
--------------------------------------------------------------------------------
n = 1000000000
f = 1
1e+09
 0.561288s wall, 0.390000s user + 0.000000s system = 0.390000s CPU (69.5%)
--------------------------------------------------------------------------------
Group 1: FLOPS_DP
+--------------------------------------+---------+------------+
|                 Event                | Counter |   Core 0   |
+--------------------------------------+---------+------------+
|           INSTR_RETIRED_ANY          |  FIXC0  | 3002491149 |
|         CPU_CLK_UNHALTED_CORE        |  FIXC1  | 2709364345 |
|         CPU_CLK_UNHALTED_REF         |  FIXC2  | 2043804906 |
| FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE |   PMC0  |      0     |
| FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE |   PMC1  |     91     |
|       SIMD_FP_256_PACKED_DOUBLE      |   PMC2  |  260258099 |
+--------------------------------------+---------+------------+

+----------------------+-----------+
|        Metric        |   Core 0  |
+----------------------+-----------+
|  Runtime (RDTSC) [s] |   1.5390  |
| Runtime unhalted [s] |   1.0454  |
|      Clock [MHz]     | 3435.5297 |
|          CPI         |   0.9024  |
|        MFLOP/s       |  676.4420 |
|      AVX MFLOP/s     |  676.4420 |
|    Packed MUOPS/s    |  169.1105 |
|    Scalar MUOPS/s    |   0.0001  |
+----------------------+-----------+
```

Performance: CPI

$$\text{Execution Time} = \text{Instruction Count} \times \text{Cycles per Instruction} \times \text{Cycle Time}$$

$$= \text{Instruction Count} \times \left(\text{CPU Cycles per Instr.} + \text{Memory Cycles per Instr.} \right) \times \text{Cycle Time}$$

$$= \text{Instruction Count} \times \left[\text{CPU Cycles per Instr.} + \left(\text{References per Instr.} \times \text{Cycles per Reference} \right) \right] \times \text{Cycle Time}$$

Instruction Set Architecture

Compiler Technology

CPU Implementation

Cache and Memory Hierarchy

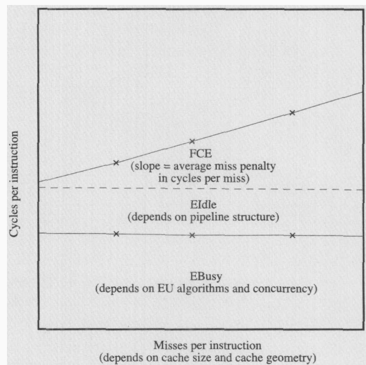
Steven K. Przybylski, "Cache and Memory Hierarchy Design – A Performance-Directed Approach," San Fransisco, Morgan-Kaufmann, 1990.

Performance: separable components of a CPI

CPI = (Infinite-cache CPI) + finite-cache effect (FCE)

Infinite-cache CPI = execute busy (EBusy) + execute idle (EIdle)

FCE = (cycles per miss) $\times$ (misses per instruction) = (miss penalty) $\times$ (miss rate)



P. G. Emma. Understanding some simple processor-performance limits. IBM Journal of Research and Development, 41(3):215–232, May 1997.

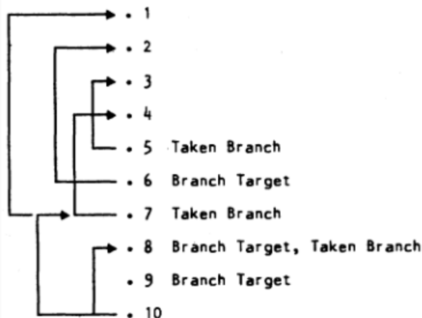


Fig. 1. A typical data-dependency graph.

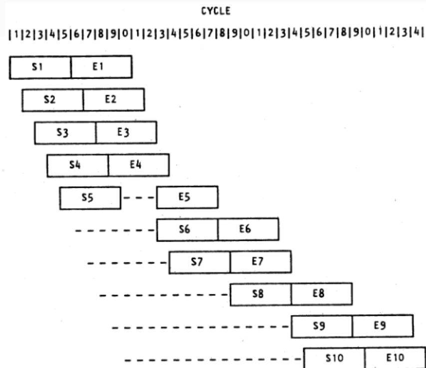


Fig. 2. Pipeline flow of ten instructions with $N_E = N_S = 5$.

P. Emma and E. Davidson, "Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance," IEEE Trans. Computers C-36, No. 7, 859-875 (July 1987)

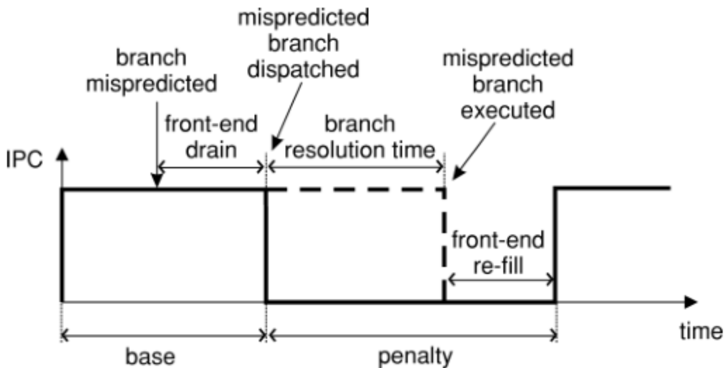


Fig. 6. Interval behavior for a branch misprediction.

Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2009. A mechanistic performance model for superscalar out-of-order processors. ACM Trans. Comput. Syst. 27, 2, Article 3.

Branch (Mis)Prediction Example I

```
#include <cmath>
#include <cstddef>
#include <cstdlib>
#include <future>
#include <iostream>
#include <random>
#include <vector>
#include <boost/timer/timer.hpp>

double sum1(const std::vector<double> & x, const std::vector<bool> & which)
{
    double sum = 0.0;
    for (std::size_t i = 0, n = which.size(); i < n; ++i)
    {
        sum += (which[i]) ? std::cos(x[i]) : std::sin(x[i]);
    }
    return sum;
}
```

Branch (Mis)Prediction Example II

```
double sum2(const std::vector<double> & x, const std::vector<bool> & which)
{
    double sum = 0.0;
    for (std::size_t i = 0, n = which.size(); i < n; ++i)
    {
        sum += (which[i]) ? std::sin(x[i]) : std::cos(x[i]);
    }
    return sum;
}

std::vector<bool> inclusion_random(std::size_t n)
{
    std::vector<bool> which;
    which.reserve(n);
    std::random_device rd;
    static std::mt19937 g(rd());
    std::uniform_int_distribution<int> u(1, 4);
    for (std::size_t i = 0; i < n; ++i)
```

Branch (Mis)Prediction Example III

```
        which.push_back(u(g) >= 3);
    return which;
}


int main(int argc, char * argv[])
{
    const std::size_t n = (argc > 1) ? std::atoll(argv[1]) : 1000;
    std::cout << "n = " << n << '\n';

    // branch takenness / predictability type
    // 0: never; 1: always; 2: random
    std::size_t type = (argc > 2) ? std::atoll(argv[2]) : 0;
    std::cout << "type = " << type << '\n';

    std::vector<bool> which;
    if (type == 0) which.resize(n, false);
    else if (type == 1) which.resize(n, true);
    else if (type == 2) which = inclusion_random(n);
```

Branch (Mis)Prediction Example IV

```
    std::vector<double> x(n, 1.1);

    boost::timer::auto_cpu_timer timer;
    std::cout << sum1(x, which) + sum2(x, which) << '\n';
}
```

Timing: Branch (Mis)Prediction Example

```
$ make BP CXXFLAGS="-std=c++14 -O3 -march=native" LDLIBS=-lboost_timer-mt

$ ./BP 10000000 0
n = 10000000
type = 0
1.3448e+007
 1.190391s wall, 1.187500s user + 0.000000s system = 1.187500s CPU (99.8%)

$ ./BP 10000000 1
n = 10000000
type = 1
1.3448e+007
 1.172734s wall, 1.156250s user + 0.000000s system = 1.156250s CPU (98.6%)

$ ./BP 10000000 2
n = 10000000
type = 2
1.3448e+007
 1.296455s wall, 1.296875s user + 0.000000s system = 1.296875s CPU (100.0%)
```

Likwid: Branch (Mis)Prediction Example

```
$ likwid-perfctr -C S0:1 -g BRANCH -f ./BP 10000000 0
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
CPU type:       Intel Core IvyBridge processor
CPU clock:      2.59 GHz
--------------------------------------------------------------------------------
n = 10000000
type = 0
1.3448e+07
 0.445464s wall, 0.440000s user + 0.000000s system = 0.440000s CPU (98.8%)
--------------------------------------------------------------------------------
Group 1: BRANCH
+------------------------------+---------+------------+
|            Event             | Counter |   Core 1   |
+------------------------------+---------+------------+
|      INSTR_RETIRED_ANY       |  FIXC0  | 2495177597 |
|    CPU_CLK_UNHALTED_CORE     |  FIXC1  | 1167613066 |
|     CPU_CLK_UNHALTED_REF     |  FIXC2  | 1167632206 |
| BR_INST_RETIRED_ALL_BRANCHES |   PMC0  |  372952380 |
| BR_MISP_RETIRED_ALL_BRANCHES |   PMC1  |    14796   |
+------------------------------+---------+------------+

+----------------------------+--------------+
|           Metric           |    Core 1    |
+----------------------------+--------------+
|    Runtime (RDTSC) [s]     |    0.4586    |
|   Runtime unhalted [s]     |    0.4505    |
|        Clock [MHz]         |   2591.5373  |
|            CPI             |    0.4679    |
|        Branch rate         |    0.1495    |
|  Branch misprediction rate | 5.929838e-06 |
| Branch misprediction ratio | 3.967263e-05 |
|   Instructions per branch  |    6.6903    |
+----------------------------+--------------+
```

Likwid: Branch (Mis)Prediction Example

```
$ likwid-perfctr -C S0:1 -g BRANCH -f ./BP 10000000 1
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
CPU type:       Intel Core IvyBridge processor
CPU clock:      2.59 GHz
--------------------------------------------------------------------------------
n = 10000000
type = 1
1.3448e+07
 0.445354s wall, 0.440000s user + 0.000000s system = 0.440000s CPU (98.8%)
--------------------------------------------------------------------------------
Group 1: BRANCH
+------------------------------+---------+------------+
|            Event             | Counter |   Core 1   |
+------------------------------+---------+------------+
|      INSTR_RETIRED_ANY       |  FIXC0  | 2495177490 |
|    CPU_CLK_UNHALTED_CORE     |  FIXC1  | 1167125701 |
|     CPU_CLK_UNHALTED_REF     |  FIXC2  | 1167146162 |
| BR_INST_RETIRED_ALL_BRANCHES |   PMC0  |  372952366 |
| BR_MISP_RETIRED_ALL_BRANCHES |   PMC1  |    14720   |
+------------------------------+---------+------------+

+----------------------------+--------------+
|           Metric           |    Core 1    |
+----------------------------+--------------+
|     Runtime (RDTSC) [s]    |    0.4584    |
|    Runtime unhalted [s]    |    0.4504    |
|         Clock [MHz]        |   2591.5345  |
|             CPI            |    0.4678    |
|         Branch rate        |    0.1495    |
|  Branch misprediction rate | 5.899380e-06 |
| Branch misprediction ratio | 3.946885e-05 |
|   Instructions per branch  |    6.6903    |
+----------------------------+--------------+
```

Likwid: Branch (Mis)Prediction Example

```
$ likwid-perfctr -C S0:1 -g BRANCH -f ./BP 10000000 2
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
CPU type:       Intel Core IvyBridge processor
CPU clock:      2.59 GHz
--------------------------------------------------------------------------------
n = 10000000
type = 2
1.3448e+07
 0.509917s wall, 0.510000s user + 0.000000s system = 0.510000s CPU (100.0%)
--------------------------------------------------------------------------------
Group 1: BRANCH
+------------------------------+---------+------------+
|             Event            | Counter |   Core 1   |
+------------------------------+---------+------------+
|       INSTR_RETIRED_ANY      |  FIXC0  | 3191479747 |
|     CPU_CLK_UNHALTED_CORE    |  FIXC1  | 2264945099 |
|     CPU_CLK_UNHALTED_REF     |  FIXC2  | 2264967068 |
| BR_INST_RETIRED_ALL_BRANCHES |   PMC0  |  468135649 |
| BR_MISP_RETIRED_ALL_BRANCHES |   PMC1  |  15326586  |
+------------------------------+---------+------------+

+----------------------------+-----------+
|           Metric           |   Core 1  |
+----------------------------+-----------+
|     Runtime (RDTSC) [s]    |   0.8822  |
|    Runtime unhalted [s]    |   0.8740  |
|         Clock [MHz]        | 2591.5589 |
|             CPI            |   0.7097  |
|         Branch rate        |   0.1467  |
|  Branch misprediction rate |   0.0048  |
| Branch misprediction ratio |   0.0327  |
|   Instructions per branch  |   6.8174  |
+----------------------------+-----------+
```

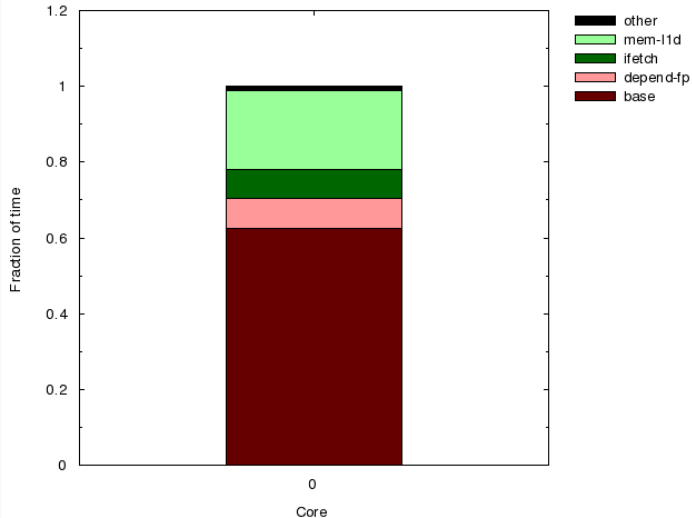
Perf: Branch (Mis)Prediction Example

```
$ perf stat -e branches,branch-misses -r 10 ./BP 10000000 0
Performance counter stats for './BP 10000000 0' (10 runs):
374,121,213 branches                                     ( +-  0.02% )
     23,260 branch-misses  # 0.01% of all branches ( +-  0.35% )
0.460392835 seconds time elapsed                         ( +-  0.50% )

$ perf stat -e branches,branch-misses -r 10 ./BP 10000000 1

Performance counter stats for './BP 10000000 1' (10 runs):

374,040,282 branches                                     ( +-  0.01% )
     23,124 branch-misses  # 0.01% of all branches ( +-  0.45% )

0.457583418 seconds time elapsed                         ( +-  0.04% )

$ perf stat -e branches,branch-misses -r 10 ./BP 10000000 2

Performance counter stats for './BP 10000000 2' (10 runs):

469,331,762 branches                                     ( +-  0.01% )
 15,326,501 branch-misses  # 3.27% of all branches ( +-  0.01% )

0.884858777 seconds time elapsed                         ( +-  0.30% )
```

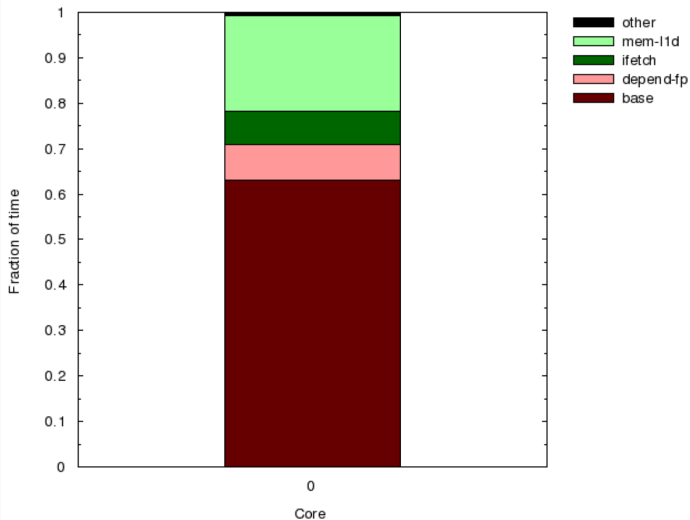
The Sniper Multi-Core Simulator

http://snipersim.org/

Sniper: Branch (Mis)Prediction Example

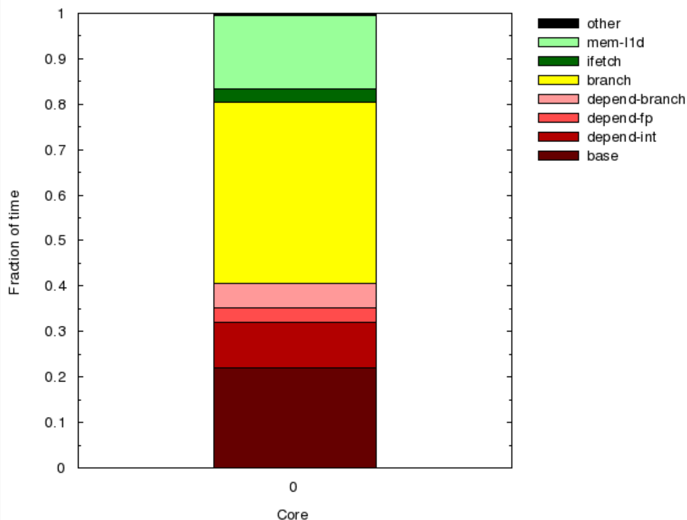


CPI stack: never taken



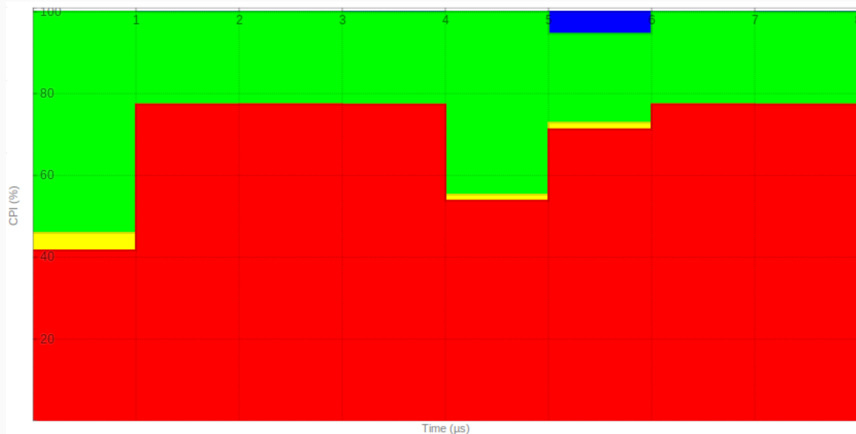
CPI stack: always taken

Sniper: Branch (Mis)Prediction Example

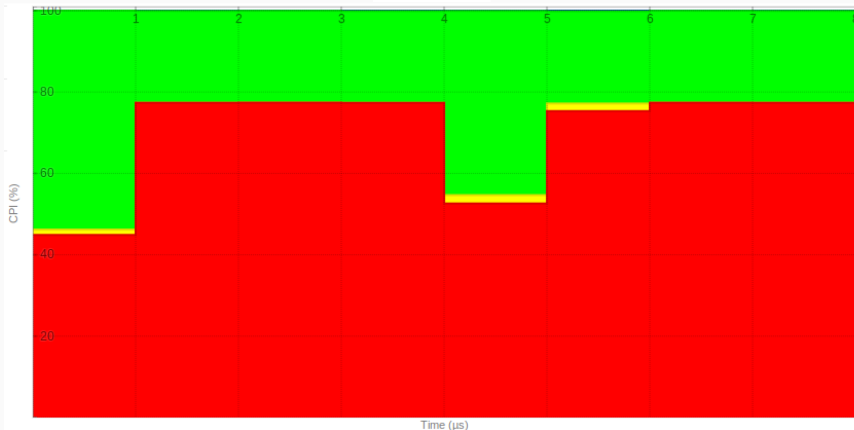


CPI stack: randomly taken

Sniper: Branch (Mis)Prediction Example

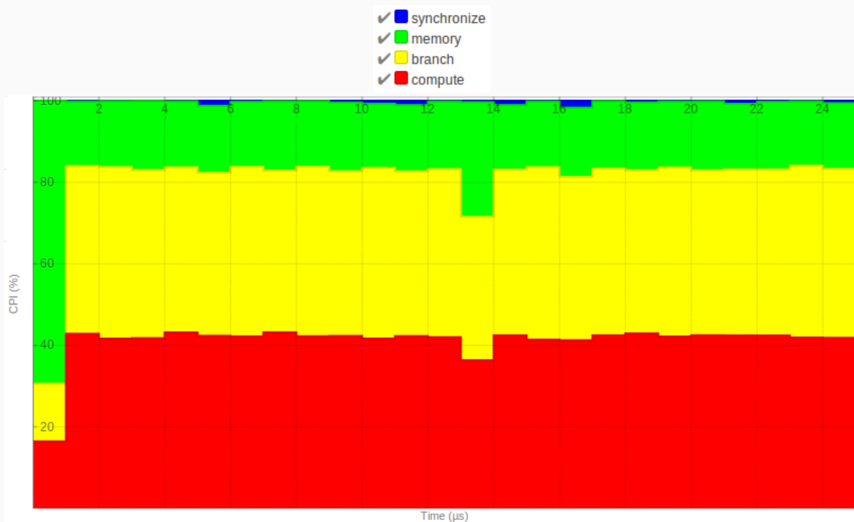


CPI graph: never taken



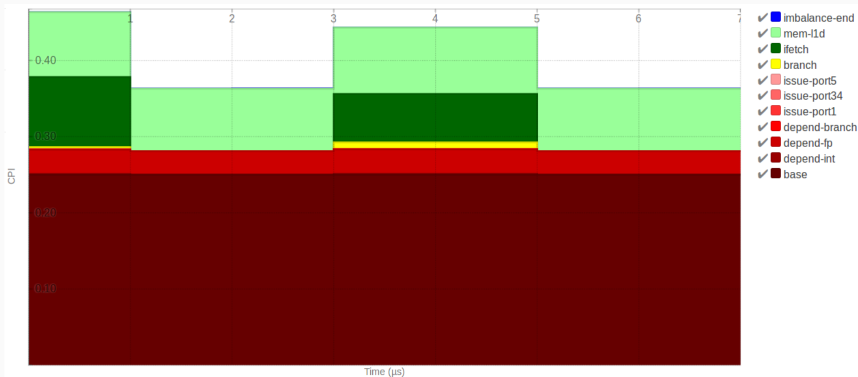
CPI graph: always taken

Sniper: Branch (Mis)Prediction Example



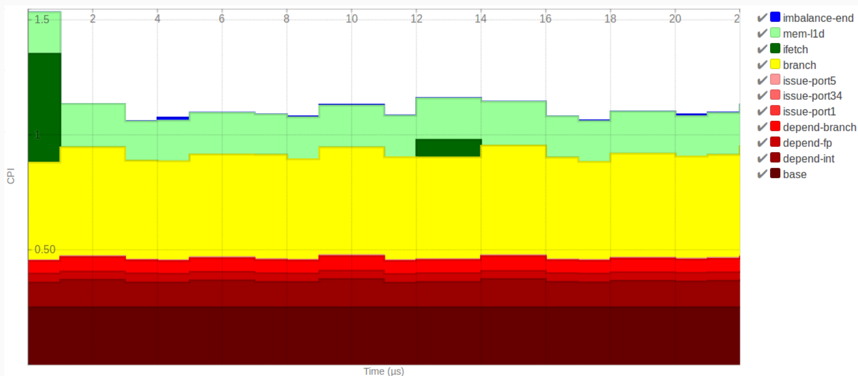
CPI graph: randomly taken

Sniper: Branch (Mis)Prediction Example



CPI graph (detailed): always taken

Sniper: Branch (Mis)Prediction Example



CPI graph (detailed): randomly taken

http://www.gem5.org/

Filling with numbers - std::vector vs. std::list

Machine code & assembly (std::vector)

```
400d78:     48 c7 02 2a 00 00 00     mov    QWORD PTR [rdx],0x2a
400d7f:     48 83 c2 08              add    rdx,0x8
400d83:     48 39 ca                 cmp    rdx,rcx
400d86:     75 f0                    jne    400d78 <main+0x18>
```

Micro-ops execution breakdown (std::vector)

```
4031000: system.cpu T0 : @main+24.0  :   MOV_M_I : limm   t1, 0x2a : IntAlu :  D=0x000000000000002a
4031500: system.cpu T0 : @main+24.1  :   MOV_M_I : st   t1, DS:[rdx] : MemWrite :  D=0x000000000000002a A=0x6dad58
4032500: system.cpu T0 : @main+31.0  :   ADD_R_I : limm   t1, 0x8 : IntAlu :  D=0x0000000000000008
4033000: system.cpu T0 : @main+31.1  :   ADD_R_I : add   rdx, rdx, t1 : IntAlu :  D=0x0000000000000000
4033500: system.cpu T0 : @main+35.0  :   CMP_R_R : sub   t0, rdx, rcx : IntAlu :  D=0x0000000000000001
4034000: system.cpu T0 : @main+38.0  :   JNZ_I : rdip   t1, %ctrl153,  : IntAlu :  D=0x0000000000400d88
4034500: system.cpu T0 : @main+38.1  :   JNZ_I : limm   t2, 0xfffffffffffffff0 : IntAlu :  D=0xfffffffffffffff0
4035000: system.cpu T0 : @main+38.2  :   JNZ_I : wrip   , t1, t2  : IntAlu :
```

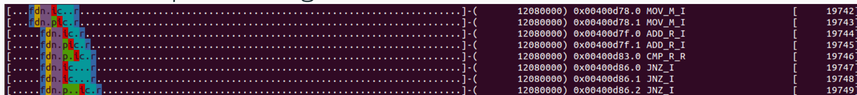
Assembly is Too High Level:

http://xlogicx.net/?p=369

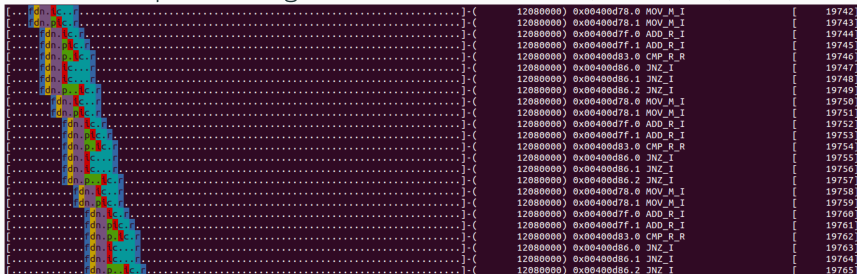
Micro-ops pipeline stages (std::vector)

```
O3PipeView:fetch:12081500:0x00400d78:0:19742:  MOV_M_I : limm   t1, 0x2a
O3PipeView:decode:12082000
O3PipeView:rename:12082500
O3PipeView:dispatch:12083500
O3PipeView:issue:12083500
O3PipeView:complete:12084000
O3PipeView:retire:12085500:store:0
O3PipeView:fetch:12080500:0x00400d83:0:19738:  CMP_R_R : sub   t0, rdx, rcx
O3PipeView:decode:12081000
O3PipeView:rename:12081500
O3PipeView:dispatch:12082500
O3PipeView:issue:12083500
O3PipeView:complete:12084000
O3PipeView:retire:12085000:store:0
O3PipeView:fetch:12080500:0x00400d86:2:19741:  JNZ_I : wrip   , t1, t2
O3PipeView:decode:12081000
O3PipeView:rename:12081500
O3PipeView:dispatch:12082500
O3PipeView:issue:12084000
O3PipeView:complete:12084500
O3PipeView:retire:12085500:store:0
```

Pipeline diagram - one iteration (std::vector)



Pipeline diagram - three iterations (std::vector)



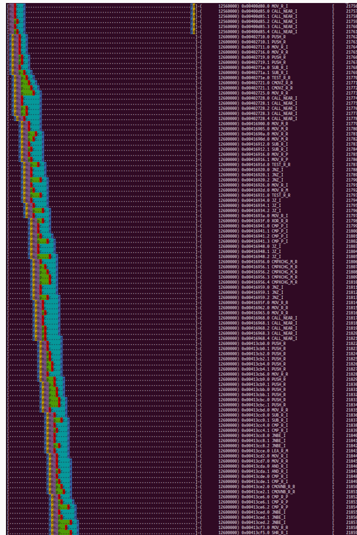
Machine code & assembly (std::list)

```
400d80:    bf 18 00 00 00              mov    edi,0x18
400d85:    e8 86 19 00 00              call   402710 <_Znwm>
400d8a:    48 85 c0                    test   rax,rax
400d8d:    74 17                       je     400da6 <main+0x46>
400d8f:    48 c7 00 00 00 00 00        mov    QWORD PTR [rax],0x0
400d96:    48 c7 40 08 00 00 00        mov    QWORD PTR [rax+0x8],0x0
400d9d:    00
400d9e:    48 c7 40 10 2a 00 00        mov    QWORD PTR [rax+0x10],0x2a
400da5:    00
400da6:    48 89 e6                    mov    rsi,rsp
400da9:    48 89 c7                    mov    rdi,rax
400dac:    e8 6f 05 00 00              call   401320 <_ZNSt8__detail15_List_node_base7_M_hookEPS0_>
400db1:    48 83 eb 01                 sub    rbx,0x1
400db5:    75 c9                       jne    400d80 <main+0x20>
```

heap allocation in the loop @ 400d85

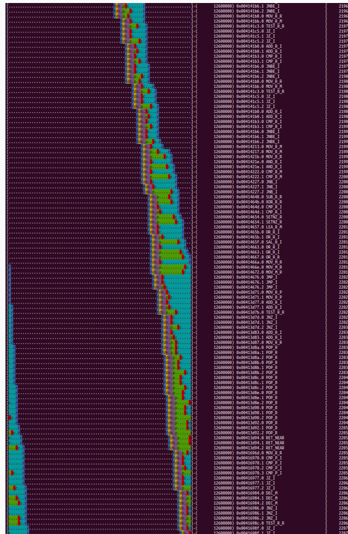
what could possibly go wrong?

STD::LIST - ONE ITERATION

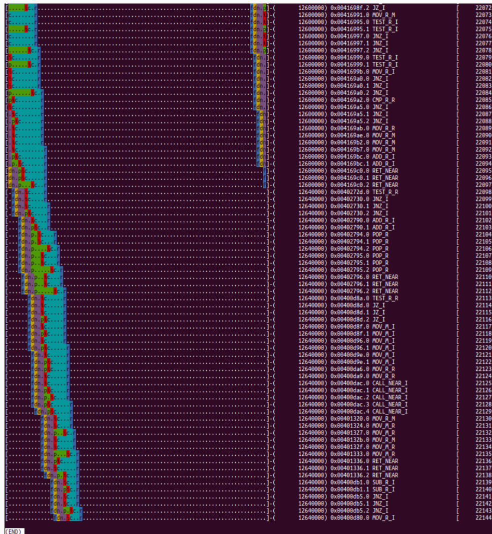


STD::LIST - ONE ITERATION (CONTINUED...)

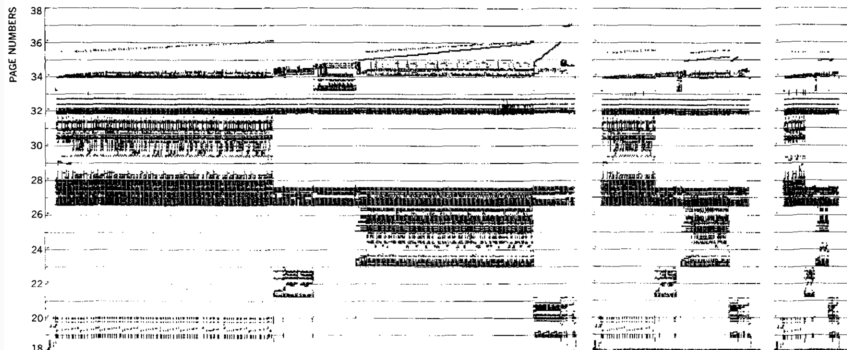
STD::LIST - ONE ITERATION (...CONTINUED STILL)



STD::LIST - ONE ITERATION (...DONE!)



Memory Access Patterns: Temporal & Spatial Locality



horizontal axis - time; vertical axis - address

D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. IBM Systems Journal, 10(3):168–192, 1971.

`0.429504s` (unfused) down to `0.287501s` (fused)

`g++ -Ofast -march=native (5.2.0)`

```
void unfused(double * a, double * b, double * c, double * d, size_t N)
{
  for (size_t i = 0; i != N; ++i)
    a[i] = b[i] * c[i];
  for (size_t i = 0; i != N; ++i)
    d[i] = a[i] * c[i];
}

void fused(double * a, double * b, double * c, double * d, size_t N)
{
  for (size_t i = 0; i != N; ++i)
  {
    a[i] = b[i] * c[i];
    d[i] = a[i] * c[i];
  }
}
```

Cache Miss Penalty: Different STC due to different MLP

MLP (memory-level parallelism) & STC (stall-time criticality)

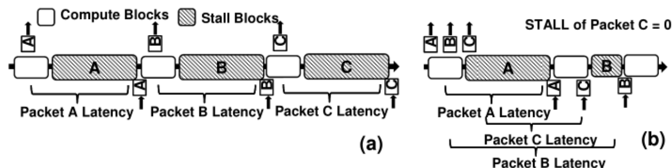


Figure 3: Execution timeline for: (a) an application with few packets, which do not overlap with each other, hence the packet latency is completely exposed to the processor as stall cycles, making each packet critical to the processor. (b) an application with packets that overlap with each other, hence some of the packet latency is hidden from the processor, accruing fewer stall cycles per packet. The packets are thus less critical.

R. Das, O. Mutlu, T. Moscibroda and C. R. Das, "Application-aware prioritization mechanisms for on-chip networks," 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), New York, NY, 2009, pp. 280-291.

Takeaway: Overlapping Latencies as a General Principle

Overlapping latencies also works on a ”macro” scale

- *load* as ”get the data from the Internet”
- *compute* as ”process the data”

Another example: Communication Avoiding and Overlapping for Numerical Linear Algebra

`https://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-65.html`

`http://www.cs.berkeley.edu/~egeor/sc12_slides_final.pdf`

Non-Overlapped Timings

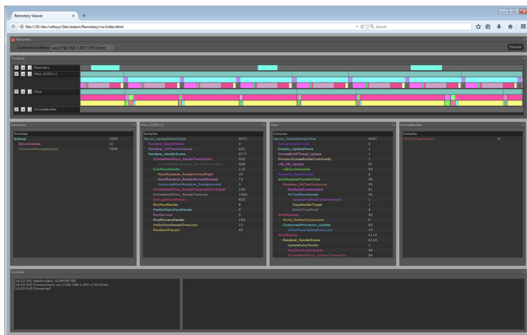
```
id,symbol,count,time
1,AAPL,565449,1.59043
2,AXP,731366,3.43745
3,BA,867366,5.40218
4,CAT,830327,7.08103
5,CSCO,400440,8.49192
6,CVX,687198,9.98761
7,DD,910932,12.2254
8,DIS,910430,14.058
9,GE,871676,15.8333
10,GS,280604,17.059
11,HD,556611,18.2738
12,IBM,860071,20.3876
13,INTC,559127,21.9856
14,JNJ,724724,25.5534
15,JPM,500473,26.576
16,KO,864903,28.5405
17,MCD,717021,30.087
18,MMM,698996,31.749
19,MRK,733948,33.2642
20,MSFT,475451,34.3134
21,NKE,556344,36.4545
```

Overlapped Timings

```
id,symbol,count,time
1,AAPL,565449,2.00713
2,AXP,731366,2.09158
3,BA,867366,2.13468
4,CAT,830327,2.19194
5,CSCO,400440,2.19197
6,CVX,687198,2.19198
7,DD,910932,2.51895
8,DIS,910430,2.51898
9,GE,871676,2.51899
10,GS,280604,2.519
11,HD,556611,2.51901
12,IBM,860071,2.51902
13,INTC,559127,2.51902
14,JNJ,724724,2.51903
15,JPM,500473,2.51904
16,KO,864903,2.51905
17,MCD,717021,2.51906
18,MMM,698996,2.51907
19,MRK,733948,2.51908
20,MSFT,475451,2.51908
21,NKE,556344,2.51909
```

https://github.com/Celtoys/Remotery

A realtime CPU/GPU profiler hosted in a single C file with a viewer that runs in a web browser.



Supported features:

- Lightweight instrumentation of multiple threads running on the CPU.
- Web viewer that runs in Chrome, Firefox and Safari. Custom WebSockets server transmits sample data to the browser on a latent thread.
- Profiles itself and shows how it's performing in the viewer.
- Can optionally sample CUDA/D3D11 GPU activity.
- Console output for logging text.
- Console input for sending commands to your game.

Timeline: Without Overlapping

Timeline

Remotery
V
GS
VZ
UNH
CSCO
AXP
MCD

Remotery

V

Console

[5:57:37 AM] Connecting to ws://127.0.0.1:17815/rmt
[5:57:37 AM] Connected
[6:00:55 AM] Connection Error
[6:00:55 AM] Disconnected
[6:00:56 AM] Connecting to ws://127.0.0.1:17815/rmt
[6:00:58 AM] Connection Error
[6:00:58 AM] Disconnected

[6:00:35 AM] PG
[6:00:35 AM] PFE
[6:00:35 AM] TRV
[6:00:35 AM] UNH
[6:00:35 AM] VZ
[6:00:35 AM] UTX
[6:00:35 AM] WMT
[6:00:35 AM] XOM
[6:00:35 AM] V
[6:00:52 AM] AAPL
[6:00:55 AM] AXP

Timeline: With Overlapping

Timeline

Remotery	
GS	
HD	LogText
TRV	LogText
CSCO	LogText
AAPL	LogText
UNH	LogText
MSFT	LogText

Remotery

Samples

GS

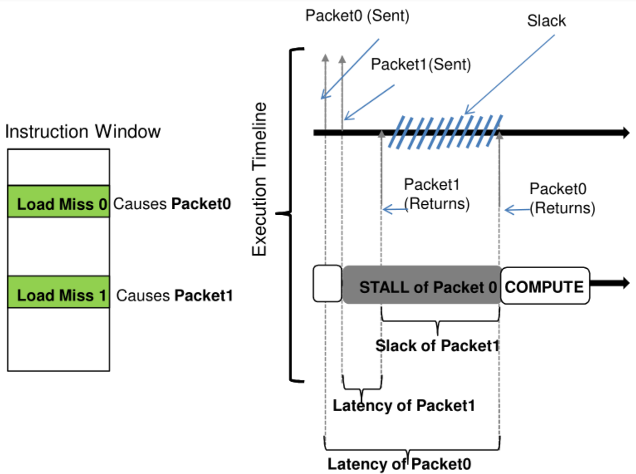
Samples

Console

[6:01:19 AM] Connecting to ws://127.0.0.1:17815/rmt
[6:01:23 AM] Connection Error
[6:01:23 AM] Disconnected
[6:01:25 AM] Connecting to ws://127.0.0.1:17815/rmt
[6:01:27 AM] Connected
[6:02:02 AM] Connection Error
[6:02:02 AM] Disconnected
[6:02:03 AM] Connecting to ws://127.0.0.1:17815/rmt
[6:02:04 AM] Connected

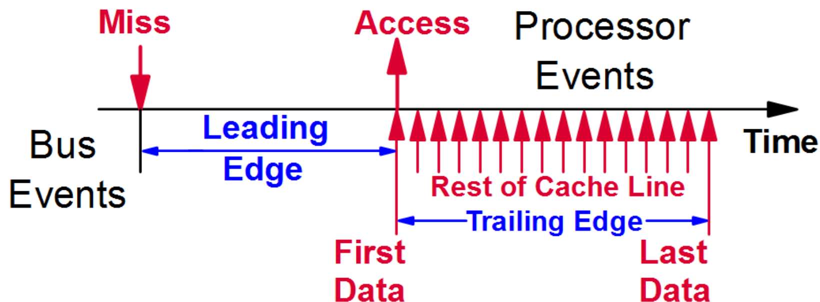
[6:02:12 AM] MSFT
[6:02:12 AM] NKE
[6:02:12 AM] PFE
[6:02:12 AM] PG
[6:02:12 AM] TRV
[6:02:12 AM] UNH
[6:02:12 AM] UTX
[6:02:12 AM] V
[6:02:12 AM] VZ
[6:02:12 AM] WMT
[6:02:12 AM] XOM

Cache Misses, MLP, and STC: Slack

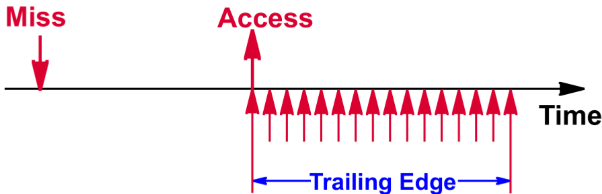


R. Das et al., "Aérgia: Exploiting Packet Latency Slack in On-Chip Networks," Proc. 37th Ann. Int'l Symp. Computer Architecture (ISCA 10), ACM Press, 2010.

Cache Miss Penalty: Leading Edge & Trailing Edge



"The End of Scaling? Revolutions in Technology and Microarchitecture as we pass the 90 Nanometer Node," Philip Emma, IBM T. J. Watson Research Center, 33rd International Symposium on Computer Architecture (ISCA 2006) Keynote Address `http://www.hpcaconf.org/hpca12/Phil_HPCA_06.pdf`



"The End of Scaling? Revolutions in Technology and Microarchitecture as we pass the 90 Nanometer Node," Philip Emma, IBM T. J. Watson Research Center, 33rd International Symposium on Computer Architecture (ISCA 2006) Keynote Address

`http://www.hpcaconf.org/hpca12/Phil_HPCA_06.pdf`

Memory Capacity & Multicore Processors

Memory utilization even more important - contention for capacity & bandwidth!

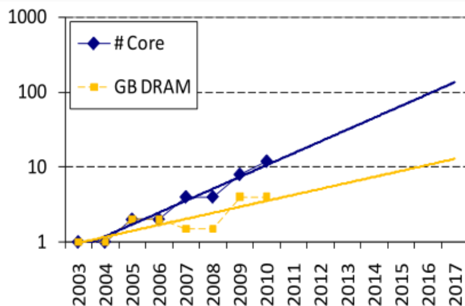


Figure 1: Projected annual growth in number of cores and memory capacity. *The expected number of cores per socket (blue line) is growing at a faster rate than the expected DRAM capacity (orange line). On average, memory capacity per processor core is extrapolated to decrease 30% every two years.*

"Disaggregated Memory Architectures for Blade Servers," Kevin Te-Ming Lim, Ph.D. Thesis, The University of Michigan, 2010.

Multicore: Sequential / Parallel Execution Model

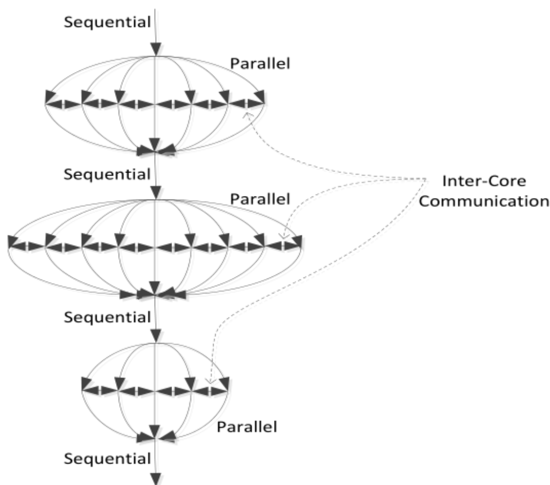


Figure 3: Sequential / Parallel Execution Model

L. Yavits, A. Morad, R. Ginosar, *The effect of communication and synchronization on Amdahl's law in multicore systems*, Parallel Computing, v. 40 n. 1, p. 1-16, January, 2014

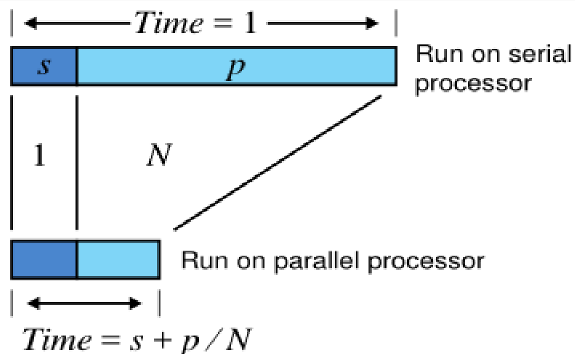
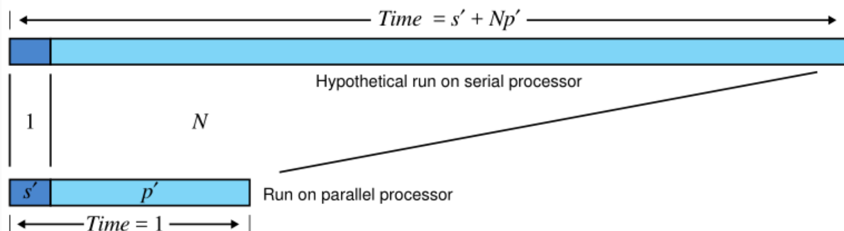


Figure 2a. Fixed-Size Model: *Speedup = 1 / (s + p / N)*

Reevaluating Amdahl's Law, John L. Gustafson, Communications of the ACM 31(5), 1988. pp. 532-533.

Multicore: Gustafson's Law, Weak Scaling

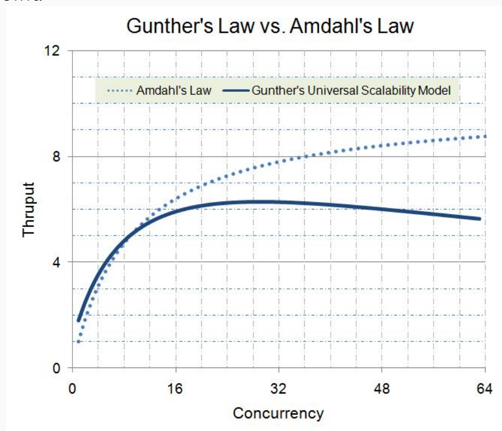


Reevaluating Amdahl's Law, John L. Gustafson, Communications of the ACM 31(5), 1988. pp. 532-533.

Amdahl's Law Optimistic

Assumes *perfect parallelism* of the *parallel portion*:
Only Serial Bottlenecks, No Parallel Bottlenecks

Counterpoint:



https://blogs.msdn.microsoft.com/ddperf/2009/04/29/parallel-scalability-isnt-childs-play-part-2-amdahls-law-vs-gunthers-law/

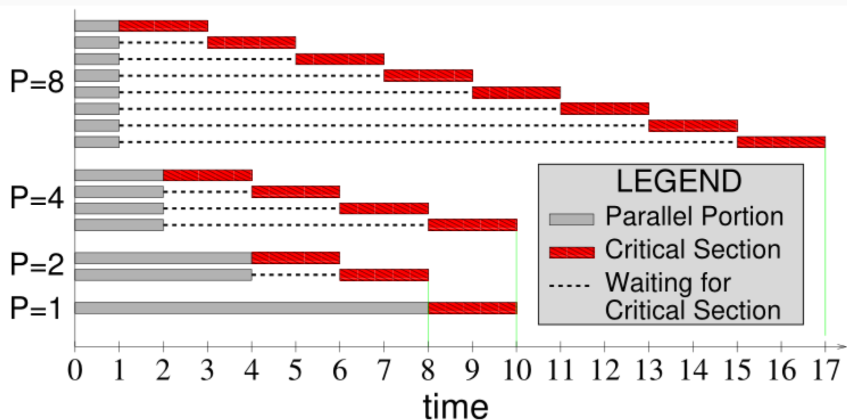


Figure 6. Example for analyzing impact of critical sections

M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs," in Proc. 13th Archit. Support Program. Lang. Oper. Syst., 2008, pp. 277–286.

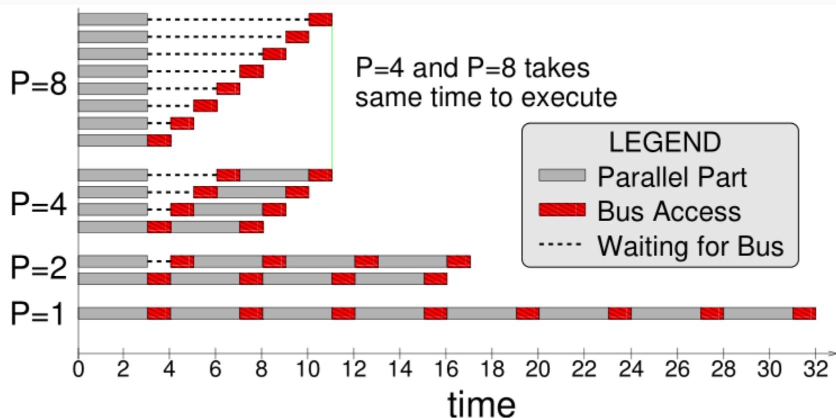


Figure 11. Example for analyzing bandwidth limited systems

M. A. Suleman, M. K. Qureshi, and Y. N. Patt, “Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs,” in Proc. 13th Archit. Support Program. Lang. Oper. Syst., 2008, pp. 277–286.

Multicore & DRAM: AoS I

```
#include <cstddef>
#include <cstdlib>
#include <future>
#include <iostream>
#include <random>
#include <vector>
#include <boost/timer/timer.hpp>

struct contract
{
    double K;
    double T;
    double P;
};

using element = contract;
using container = std::vector<element>;
```

```
double sum_if(const container & a, const container & b,
              const std::vector<std::size_t> & index) {
    double sum = 0.0;
    for (std::size_t i = 0, n = index.size(); i < n; ++i)
    {
        std::size_t j = index[i];
        if (a[j].K == b[j].K) sum += a[j].K;
    }
    return sum;
}

template <typename F>
double average(F f, std::size_t m) {
    double average = 0.0;
    for (std::size_t i = 0; i < m; ++i)
        average += f() / m;
    return average;
}
```

Multicore & DRAM: AoS III

```
std::vector<std::size_t> index_stream(std::size_t n) {
    std::vector<std::size_t> index;
    index.reserve(n);
    for (std::size_t i = 0; i < n; ++i)
        index.push_back(i);
    return index;
}

std::vector<std::size_t> index_random(std::size_t n) {
    std::vector<std::size_t> index;
    index.reserve(n);
    std::random_device rd;
    static std::mt19937 g(rd());
    std::uniform_int_distribution<std::size_t> u(0, n - 1);
    for (std::size_t i = 0; i < n; ++i)
        index.push_back(u(g));
    return index;
}
```

Multicore & DRAM: AoS IV

```
int main(int argc, char * argv[])
{
    const std::size_t n = (argc >= 2) ? std::atoll(argv[1]) : 1000;
    const std::size_t m = (argc >= 3) ? std::atoll(argv[2]) : 10;
    std::cout << "n = " << n << '\n';
    std::cout << "m = " << m << '\n';
    const std::size_t threads_count = 4;

    // thread access locality type
    // 0: none (default); 1: stream; 2: random
    std::vector<std::size_t> thread_type(threads_count);
    for (std::size_t thread = 0; thread != threads_count; ++thread)
    {
        thread_type[thread] = (argc >= 4 + thread)
                            ? std::atoll(argv[3 + thread])
                            : 0;
        std::cout << "thread_type[" << thread << "] = "
                  << thread_type[thread] << '\n';
    }
```

```
endl(std::cout);

std::vector<std::vector<std::size_t>> index(threads_count);
for (std::size_t thread = 0; thread != threads_count; ++thread)
{
    index[thread].resize(n);
    if (thread_type[thread] == 1) index[thread] = index_stream(n);
    else if (thread_type[thread] == 2) index[thread] = index_random(n);
}

const container v1(n, {1.0, 0.5, 3.0});
const container v2(n, {1.0, 2.0, 1.0});

const auto thread_work = [m, &v1, &v2](const auto & thread_index)
{
    const auto f = [&v1, &v2, &thread_index] {
      return sum_if(v1, v2, thread_index); };
    return average(f, m);
};
```

Multicore & DRAM: AoS VI

```
    boost::timer::auto_cpu_timer timer;

    std::vector<std::future<double>> results;
    results.reserve(threads_count);
    for (std::size_t thread = 0; thread != threads_count; ++thread)
    {
        results.emplace_back(std::async(std::launch::async,
       [thread, &thread_work, &index] { return thread_work(index[thread]);
     );
    }
    for (auto && result : results) if (result.valid()) result.wait();
    for (auto && result : results) std::cout << result.get() << '\n';
}
```

1 thread, sequential access

```
$ ./DRAM_CMP 10000000 10 1
n = 10000000
m = 10
thread_type[0] = 1

1e+007
 0.395408s wall, 0.406250s user + 0.000000s system = 0.406250s CPU (102.7%)
```

Multicore & DRAM: AoS Timings

1 thread, random access

```
$ ./DRAM_CMP 10000000 10 2
n = 10000000
m = 10
thread_type[0] = 2

1e+007
 5.348314s wall, 5.343750s user + 0.000000s system = 5.343750s CPU (99.9%)
```

Multicore & DRAM: AoS Timings

4 threads, sequential access

```
$ ./DRAM_CMP 10000000 10 1 1 1 1
n = 10000000
m = 10
thread_type[0] = 1
thread_type[1] = 1
thread_type[2] = 1
thread_type[3] = 1

1e+007
1e+007
1e+007
1e+007
 0.508894s wall, 2.000000s user + 0.000000s system = 2.000000s CPU (393.0%)
```

4 threads: 3 sequential access + 1 random access

```
$ ./DRAM_CMP 10000000 10 1 1 1 2
n = 10000000
m = 10
thread_type[0] = 1
thread_type[1] = 1
thread_type[2] = 1
thread_type[3] = 2

1e+007
1e+007
1e+007
1e+007
 5.666049s wall, 7.265625s user + 0.000000s system = 7.265625s CPU (128.2%)
```

Memory Access Patterns & Multicore: Interactions Matter

Inter-thread Interference

Sharing - Contention - Interference - Slowdown

Threads using a shared resource (like on-chip/off-chip interconnects and memory) contend for it, interfering with each other's progress, resulting in slowdown (and thus negative returns to increased threads count).

cf. Thomas Moscibroda and Onur Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," Microsoft Research Technical Report, MSR-TR-2007-15, February 2007.

```
#include <cstddef>
#include <cstdlib>
#include <future>
#include <iostream>
#include <random>
#include <vector>
#include <boost/timer/timer.hpp>

// SoA (structure-of-arrays)
struct data
{
    std::vector<double> K;
    std::vector<double> T;
    std::vector<double> P;
};
```

Multicore & DRAM: SoA II

```
double sum_if(const data & a, const data & b, const std::vector<std::size_t
{
    double sum = 0.0;
    for (std::size_t i = 0, n = index.size(); i < n; ++i)
    {
        std::size_t j = index[i];
        if (a.K[j] == b.K[j]) sum += a.K[j];
    }
    return sum;
}

template <typename F>
double average(F f, std::size_t m)
{
    double average = 0.0;
    for (std::size_t i = 0; i < m; ++i)
    {
        average += f() / m;
    }
```

Multicore & DRAM: SoA III

```
    return average;
}

std::vector<std::size_t> index_stream(std::size_t n)
{
    std::vector<std::size_t> index;
    index.reserve(n);
    for (std::size_t i = 0; i < n; ++i)
        index.push_back(i);
    return index;
}

std::vector<std::size_t> index_random(std::size_t n)
{
    std::vector<std::size_t> index;
    index.reserve(n);
    std::random_device rd;
    static std::mt19937 g(rd());
    std::uniform_int_distribution<std::size_t> u(0, n - 1);
```

```
    for (std::size_t i = 0; i < n; ++i)
        index.push_back(u(g));
    return index;
}


int main(int argc, char * argv[])
{
    const std::size_t n = (argc >= 2) ? std::atoll(argv[1]) : 1000;
    const std::size_t m = (argc >= 3) ? std::atoll(argv[2]) : 10;
    std::cout << "n = " << n << '\n';
    std::cout << "m = " << m << '\n';
    const std::size_t threads_count = 4;

    // thread access locality type
    // 0: none (default); 1: stream; 2: random
    std::vector<std::size_t> thread_type(threads_count);
    for (std::size_t thread = 0; thread != threads_count; ++thread)
    {
```

```
        thread_type[thread] = (argc >= 4 + thread) ? std::atoll(argv[3 + th
        std::cout << "thread_type[" << thread << "] = " << thread_type[thre
    }
    endl(std::cout);

    std::vector<std::vector<std::size_t>> index(threads_count);
    for (std::size_t thread = 0; thread != threads_count; ++thread)
    {
        index[thread].resize(n);
        if (thread_type[thread] == 1) index[thread] = index_stream(n);
        else if (thread_type[thread] == 2) index[thread] = index_random(n);
        //for (auto e : index[thread]) std::cout << e; endl(std::cout);
    }

    data v1;
    v1.K.resize(n, 1.0);
    v1.T.resize(n, 0.5);
    v1.P.resize(n, 3.0);
```

Multicore & DRAM: SoA VI

```
data v2;
v2.K.resize(n, 1.0);
v2.T.resize(n, 2.0);
v2.P.resize(n, 1.0);

const auto thread_work = [m, &v1, &v2](const auto & thread_index)
{
    const auto f = [&v1, &v2, &thread_index] { return sum_if(v1, v2, th
    return average(f, m);
};

boost::timer::auto_cpu_timer timer;

std::vector<std::future<double>> results;
results.reserve(threads_count);
for (std::size_t thread = 0; thread != threads_count; ++thread)
{
    results.emplace_back(std::async(std::launch::async,
   [thread, &thread_work, &index] { return thread_work(index[thread]);
```

```
        );
    }
    for (auto && result : results) if (result.valid()) result.wait();
    for (auto && result : results) std::cout << result.get() << '\n';

}
```

1 thread, sequential access

```
$ ./DRAM_CMP.SoA 10000000 10 1 1 1 1
n = 10000000
m = 10
thread_type[0] = 1

1e+007
 0.211877s wall, 0.203125s user + 0.000000s system = 0.203125s CPU (95.9%)
```

Multicore & DRAM: SoA Timings

1 thread, random access

```
$ ./DRAM_CMP.SoA 10000000 10 2
n = 10000000
m = 10
thread_type[0] = 2

1e+007
 4.534646s wall, 4.546875s user + 0.000000s system = 4.546875s CPU (100.3%)
```

4 threads, sequential access

```
$ ./DRAM_CMP.SoA 10000000 10 1 1 1 1
n = 10000000
m = 10
thread_type[0] = 1
thread_type[1] = 1
thread_type[2] = 1
thread_type[3] = 1

1e+007
1e+007
1e+007
1e+007
 0.256391s wall, 1.031250s user + 0.000000s system = 1.031250s CPU (402.2%)
```

Multicore & DRAM: SoA Timings

4 threads: 3 sequential access + 1 random access

```
$ ./DRAM_CMP.SoA 10000000 10 1 1 1 2
n = 10000000
m = 10
thread_type[0] = 1
thread_type[1] = 1
thread_type[2] = 1
thread_type[3] = 2

1e+007
1e+007
1e+007
1e+007
 4.581033s wall, 5.265625s user + 0.000000s system = 5.265625s CPU (114.9%)
```

Multicore & DRAM: SoA Timings

Better Access Patterns

yield

Better Single-core Performance

but also

Reduced Interference

and thus

Better Multi-core Performance

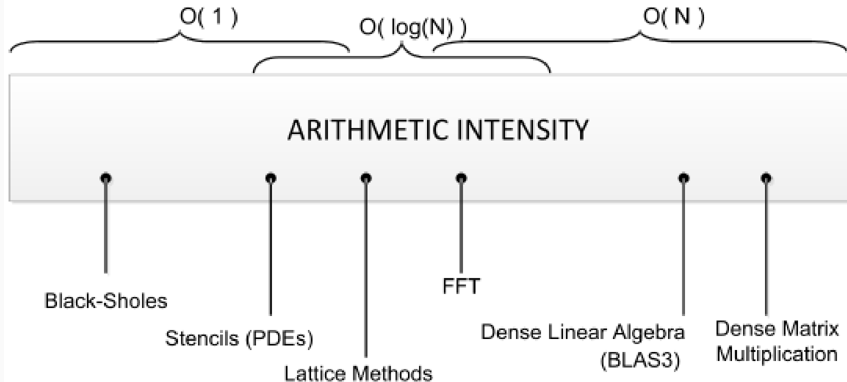


Figure 4: Arithmetic Intensity [26].

L. Yavits, A. Morad, R. Ginosar, *The effect of communication and synchronization on Amdahl's law in multicore systems*, Parallel Computing, v. 40 n. 1, p. 1-16, January, 2014

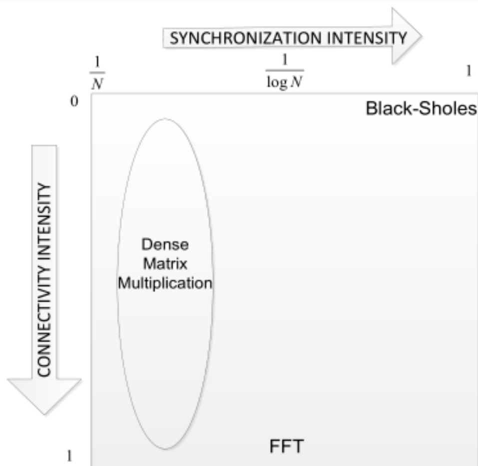


Figure 5: Synchronization and Connectivity Intensity

L. Yavits, A. Morad, R. Ginosar, *The effect of communication and synchronization on Amdahl's law in multicore systems*, Parallel Computing, v. 40 n. 1, p. 1-16, January, 2014

$$Speedup(f, n_c) = \frac{1}{1 - f + \frac{f}{n_c} + \frac{f_1(n_c)}{n_c} + f_2(n_c)}$$

f: parallelizable fraction

f_1: connectivity intensity

f_2: synchronization intensity

L. Yavits, A. Morad, R. Ginosar, *The effect of communication and synchronization on Amdahl's law in multicore systems*, Parallel Computing, v. 40 n. 1, p. 1-16, January, 2014

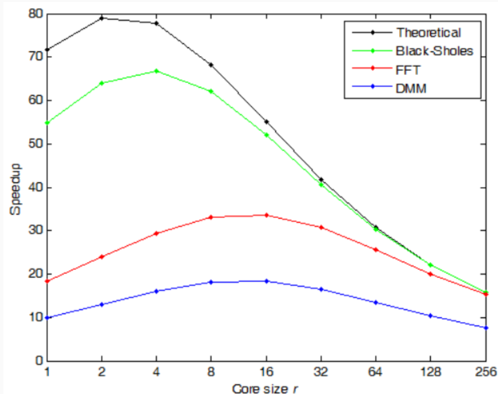
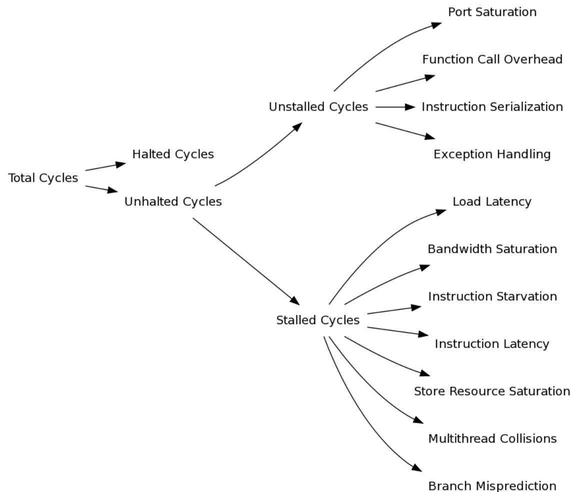


Figure 8. Simulation results: Speedup vs. core size

speedup affected by sequential-to-parallel data synchronization and inter-core communication

L. Yavits, A. Morad, R. Ginosar, *The effect of communication and synchronization on Amdahl's law in multicore systems*, Parallel Computing, v. 40 n. 1, p. 1-16, January, 2014

Hierarchical cycle accounting



Andrzej Nowak, David Levinthal, Willy Zwaenepoel: Hierarchical cycle accounting: a new method for application performance tuning. ISPASS 2015: 112-123
https://github.com/David-Levinthal/gooda

Top-down Microarchitecture Analysis Method (TMAM)

https://github.com/andikleen/pmu-tools/wiki/toplev-manual
https://sites.google.com/site/analysismethods/yasin-pubs

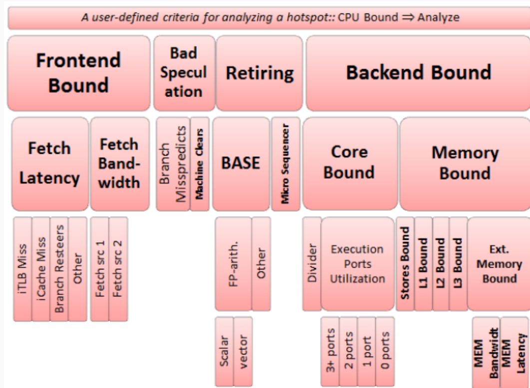


Figure 2: The Top-Down Analysis Hierarchy

A Top-Down Method for Performance Analysis and Counters Architecture, Ahmad Yasin. In IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014.

http://www.eecs.berkeley.edu/~waterman/papers/roofline.pdf
Roofline: an insightful visual performance model for multicore architectures Samuel Williams, Andrew Waterman and David Patterson Communications ACM 55(6): 121-130 (2012)

http://www.inesc-id.pt/ficheiros/publicacoes/9068.pdf
Cache-aware Roofline model: Upgrading the loft Aleksandar Ilic, Frederico Pratas, Leonel Sousa, IEEE Computer Architecture Letters, vol. 13, no. 1, pp. 1-1, Jan.-June, 2014

http://spiral.ece.cmu.edu:8080/pub-spiral/abstract.jsp?id=181
Extending the Roofline Model: Bottleneck Analysis with Microarchitectural Constraints Victoria Caparros and Markus Püschel Proc. IEEE International Symposium on Workload Characterization (IISWC), pp. 222-231, 2014

Principles

Data structures & data layout - fundamental part of design

CPUs & pervasive forms parallelism

- can support each other: PLP, ILP (MLP!), TLP, DLP

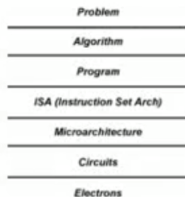
Balanced design vs. bottlenecks

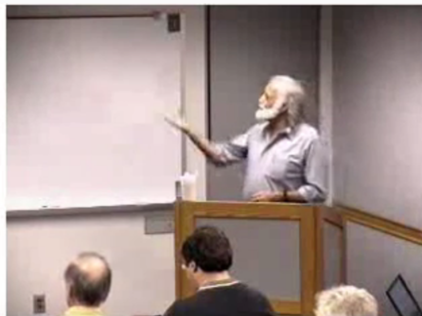
Overlapping latencies

Sharing-contention-interference-slowdown

Yale Patt's Phase 2: Break the layers:

- break through the hardware/software interface
- harness all levels of the transformation hierarchy





Yale N. Patt, *Microprocessor Performance, Phase 2: Can We Harness the Transformation Hierarchy*
https://youtube.com/watch?v=0fLlDkC625Q

Break the Layers

The Answer: Break the Layers

- *(We already have in limited cases)*

Yale N. Patt, *Microprocessor Performance, Phase 2: Can We Harness the Transformation Hierarchy*
https://youtube.com/watch?v=0fLlDkC625Q

Pigeonholing has to go

Yale N. Patt at Yale Patt 75 Visions of the Future Computer Architecture Workshop:

"Are you a software person or a hardware person?"
I'm a person
this pigeonholing has to go

We must break the layers

Abstractions are great
- AFTER you understand what's being abstracted

Yale N. Patt, 2013 IEEE CS Harry H. Goode Award Recipient Interview — https://youtu.be/S7wXivUy-tk

Yale N. Patt at Yale Patt 75 Visions of the Future Computer Architecture Workshop — https://youtu.be/x4LH1cJCvxs

Resources

http://www.agner.org/optimize/

https://users.ece.cmu.edu/~omutlu/lecture-videos.html

https://github.com/MattPD/cpplinks/

Slides

https://speakerdeck.com/mattpd

Thank You!
Questions?