

GETTING THE MOST OUT OF RCPP

HIGH-PERFORMANCE C++ IN PRACTICE

Matt P. Dziubinski

R/Finance 2016

`matt@math.aau.dk // @matt_dz`

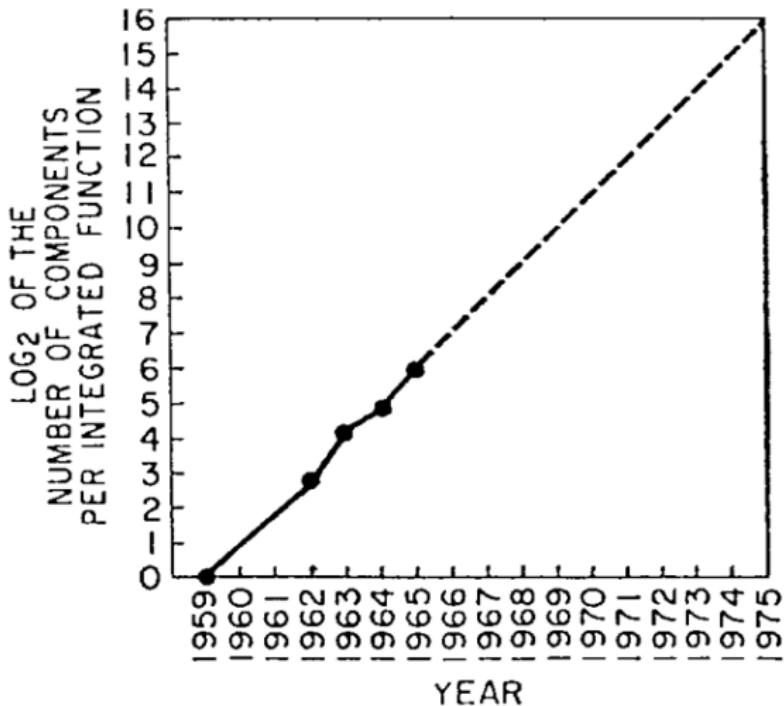
Department of Mathematical Sciences, Aalborg University
CREATES (Center for Research in Econometric Analysis of Time Series)

OUTLINE

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

WHY?

CRAMMING MORE COMPONENTS ONTO INTEGRATED CIRCUITS



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

SPENDING MOORE'S DIVIDEND

Spending Moore's Dividend

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

James Larus
larus@microsoft.com
Microsoft Research

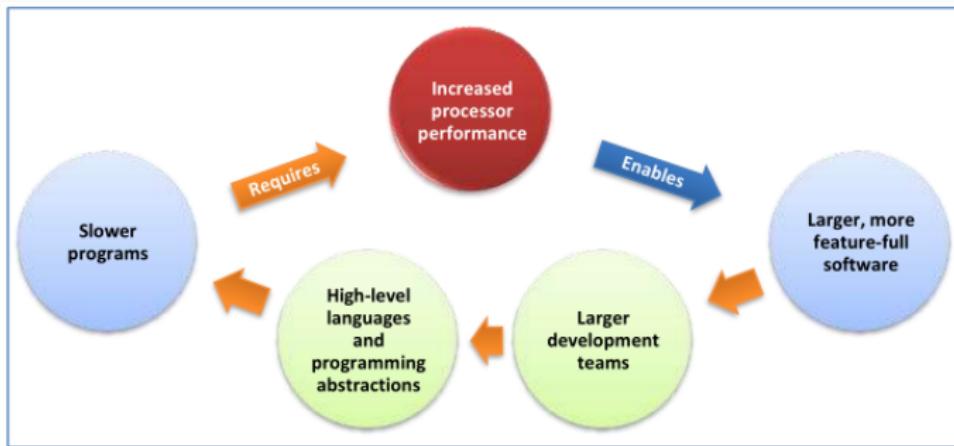


Figure 1. Cycle of innovation in computer industry.

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

TRANSFORMATION HIERARCHY

Problem

Algorithm

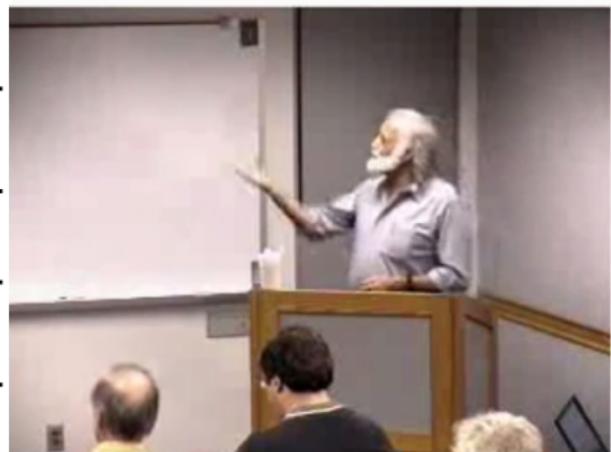
Program

ISA (Instruction Set Arch)

Microarchitecture

Circuits

Electrons



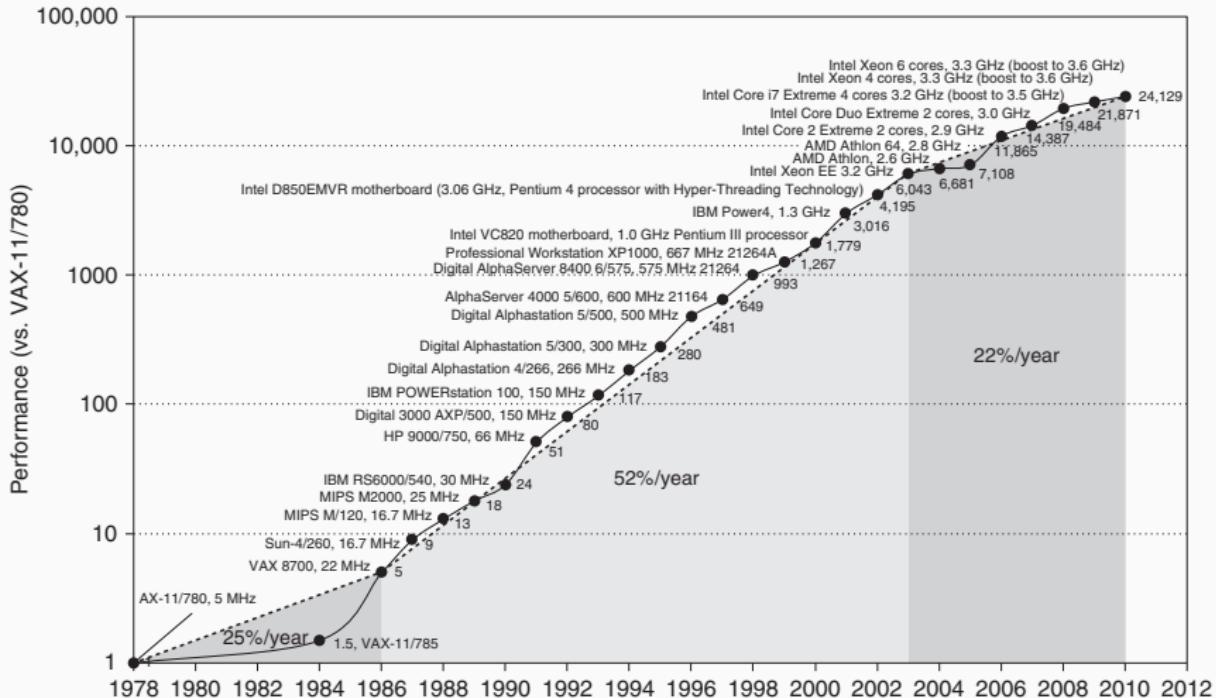
Yale N. Patt, *Microprocessor Performance, Phase 2: Can We Harness the Transformation Hierarchy*

<https://youtube.com/watch?v=0fLlDkC625Q>

Until Recently (Phase I)

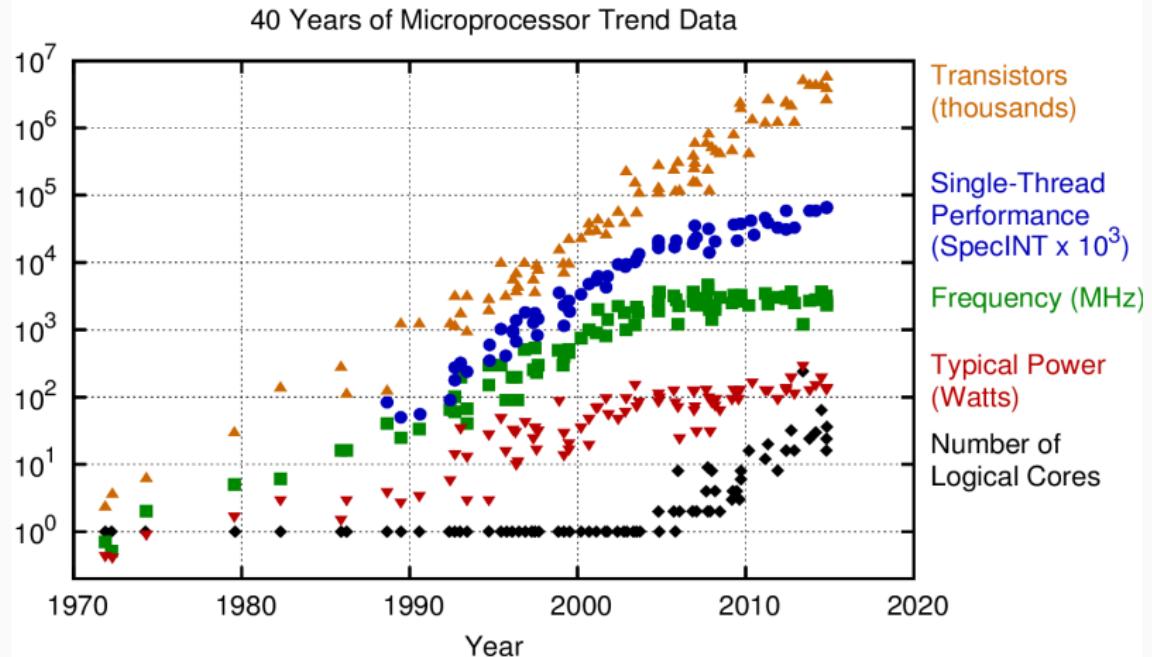
- *Maintain the artificial walls between the layers*
- *Keeps the abstraction layers secure*
 - *Makes for a better comfort zone*
- *(Mostly) Improves the Microarchitecture*
 - *Pipelining, Caches*
 - *Branch Prediction, Speculative Execution*
 - *Out-of-order Execution, Trace Cache*
- *Today, we have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

CPU PERFORMANCE TRENDS



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

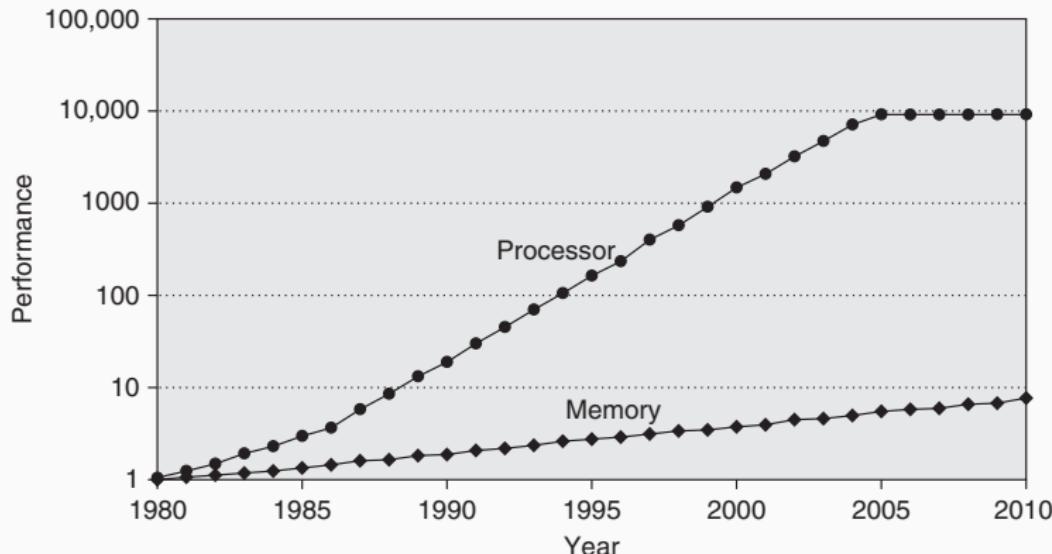
40 YEARS OF MICROPROCESSOR TREND DATA



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

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

PROCESSOR-MEMORY PERFORMANCE GAP

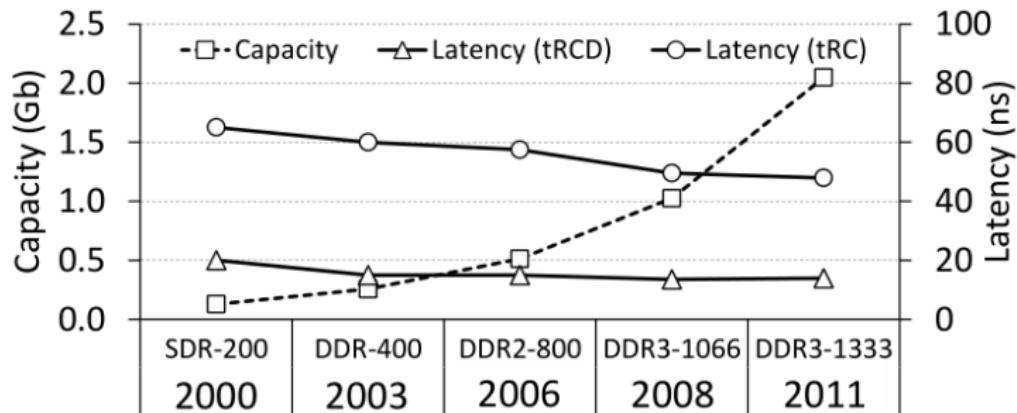


the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access

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>

DRAM PERFORMANCE TRENDS



† We refer to the dominant DRAM chips during the period of time [27, 111].

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

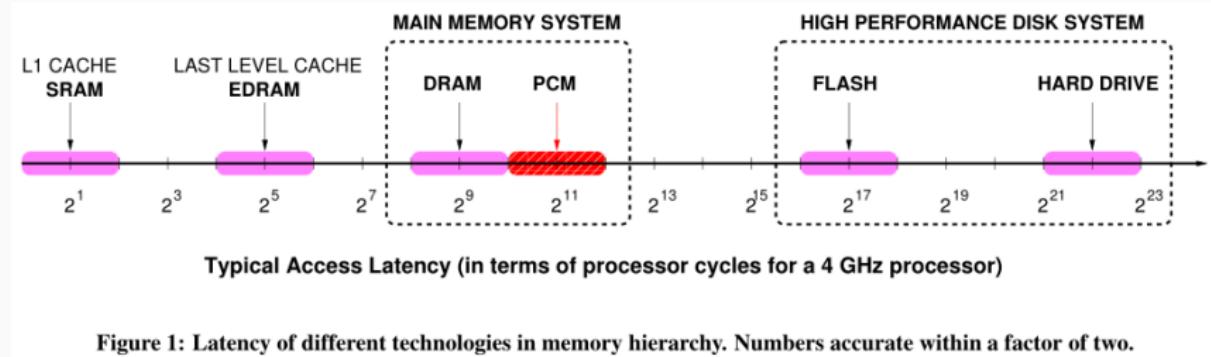


Figure 1: Latency of different technologies in memory hierarchy. Numbers accurate within a factor of two.

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

FEATURE SCALING TRENDS

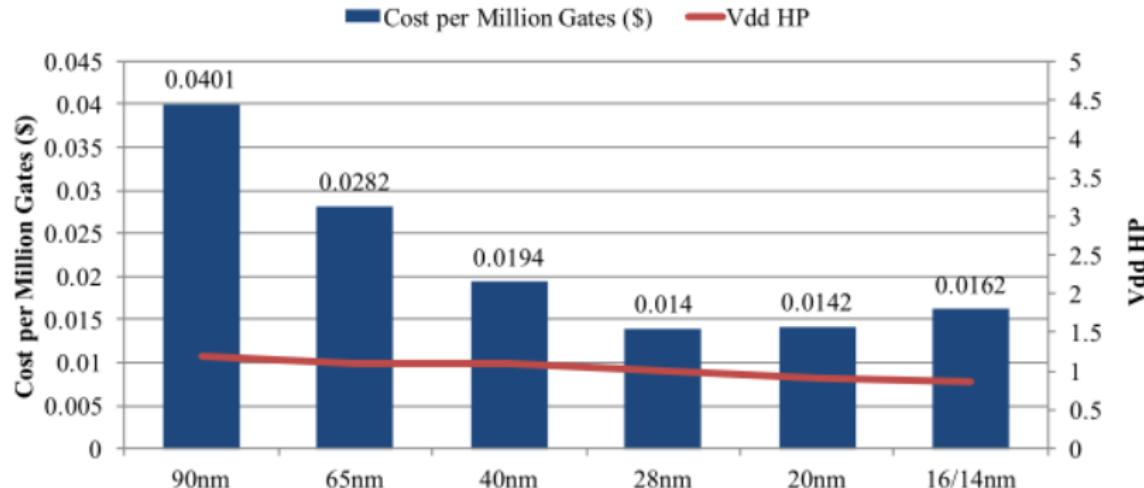


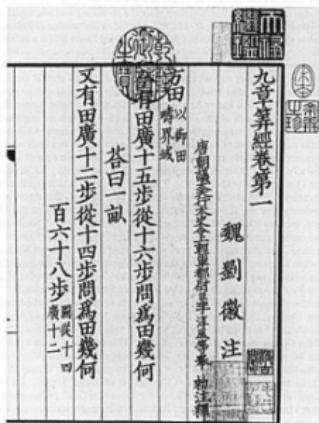
Figure 1.1: Trends in Cost per Gate and Nominal Vdd for Advanced HP (High-Performance) Process Nodes – Cost per gate data from [59] and nominal Vdd data from International Technology Roadmap of Semiconductors (ITRS).

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.

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.

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 \mathcal{T} such that \mathcal{T} computes the n th 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.

COMPLEXITY: ALGORITHMS & DATA STRUCTURES

$O(N)$

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

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

$O(N \cdot \log(N))$

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

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

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

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

$\log(N)$

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

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

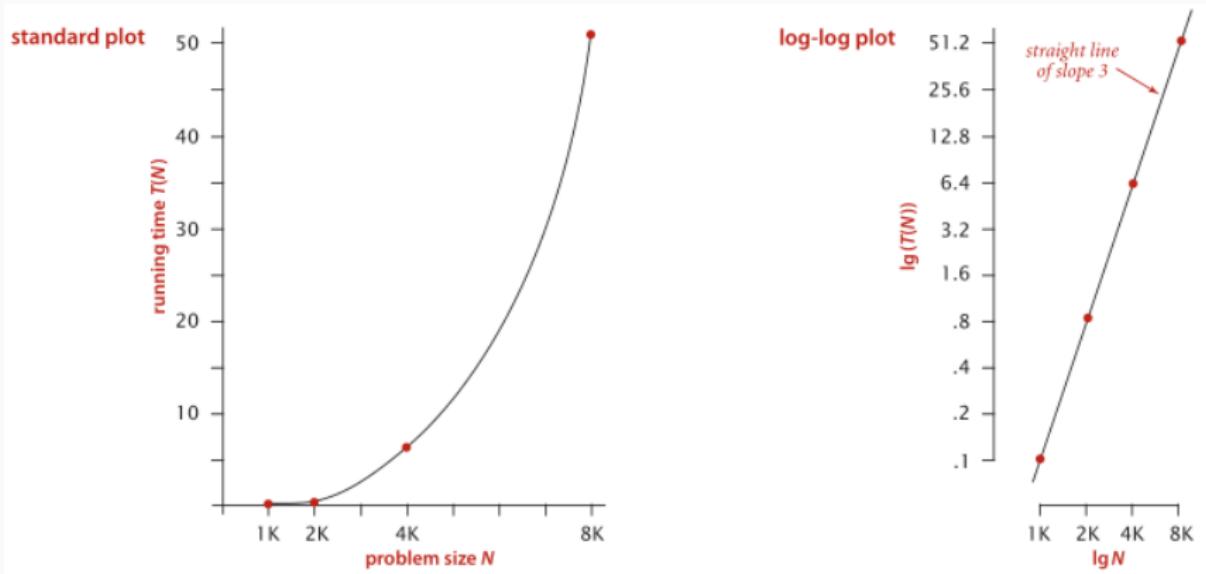
Scientific method The very same approach that scientists use to understand the natural world is effective for studying the running time of programs:

- *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 $\tilde{f}(N)$ 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 $T(N) \sim aN^b \lg N$ then $T(2N)/T(N) \sim 2^b$.

Proof: Immediate from the following calculation:

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

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

Rcpp EXAMPLE - PASS & SUM EXPERIMENT - C++ CODE |

```
// [[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

```
size = 100000000L
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"))
```

Rcpp Example - Pass & Sum Experiment - Results

		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

Rcpp EXAMPLE - PASS & SUM EXPERIMENT - NUMERICVECTOR

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/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++:

```
#include <vector>

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

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

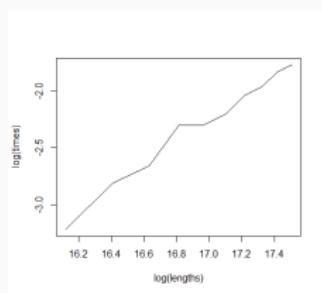
Rcpp Example - Timing Experiments - std::vector

```
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")
```



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

Coefficients:

(Intercept)	log(lengths)
-19.347	1.005

- 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

TRADE-OFFS - MULTIDIMENSIONAL - NUMERICAL OPTIMIZATION

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

Algorithm	Time per iteration	Error after T iterations	Error after N items
Newton	$O(d^2N + d^3)$	C_I^{2T}	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 $O(\text{Cost}(f))$ evaluations
 - central-difference: $O(\epsilon_M^{2/3})$ error, $2d$ $O(\text{Cost}(f))$ 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 $O(\text{Cost}(f))$
 - rough reverse-mode cost N $O(\text{Cost}(f))$

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.

- 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>

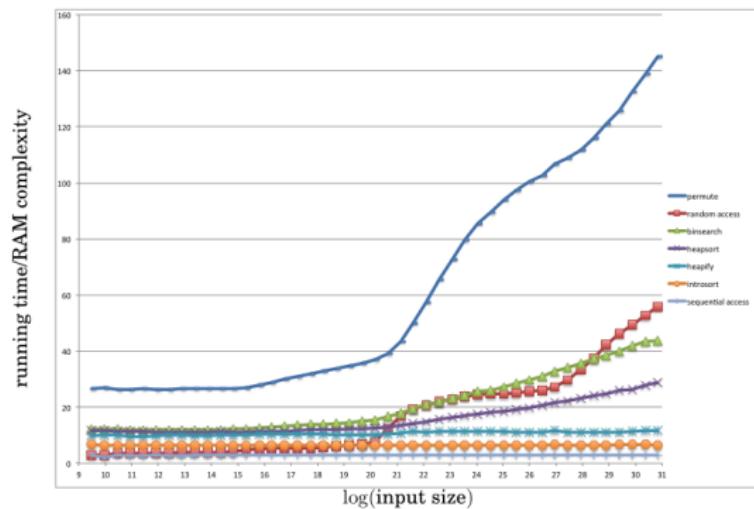


Fig. 1. The abscissa shows the logarithm of the input size. The ordinate shows the measured running time divided by the RAM-complexity (normalized operation time). The normalized operation times of sequential access, quicksort, and heapify are constant, the normalized operation times of the other programs 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

"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` implementation of red-black trees).

"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` implementation 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 microarchitectural 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

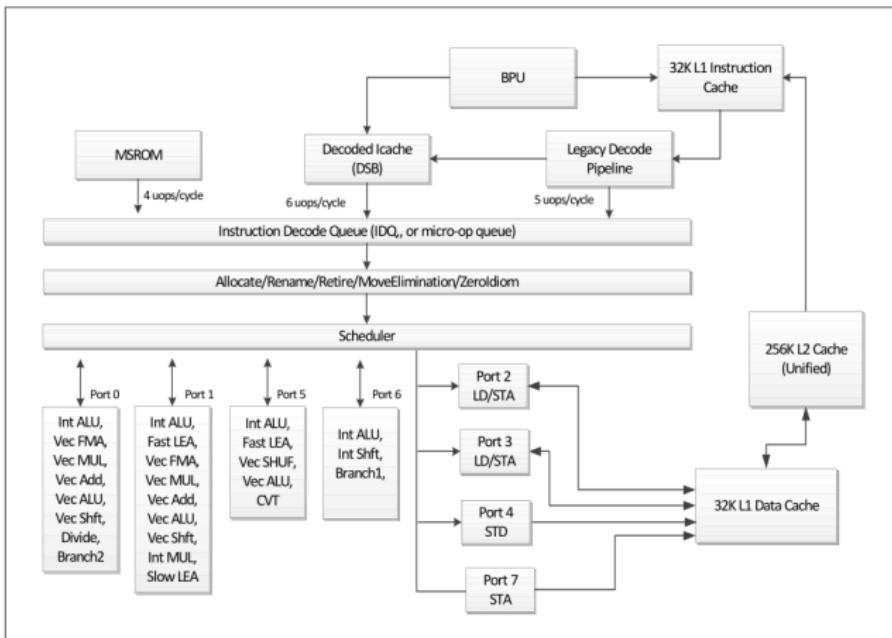


Figure 2-1. CPU Core Pipeline Functionality of the Skylake Microarchitecture

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>

- 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;
}
```

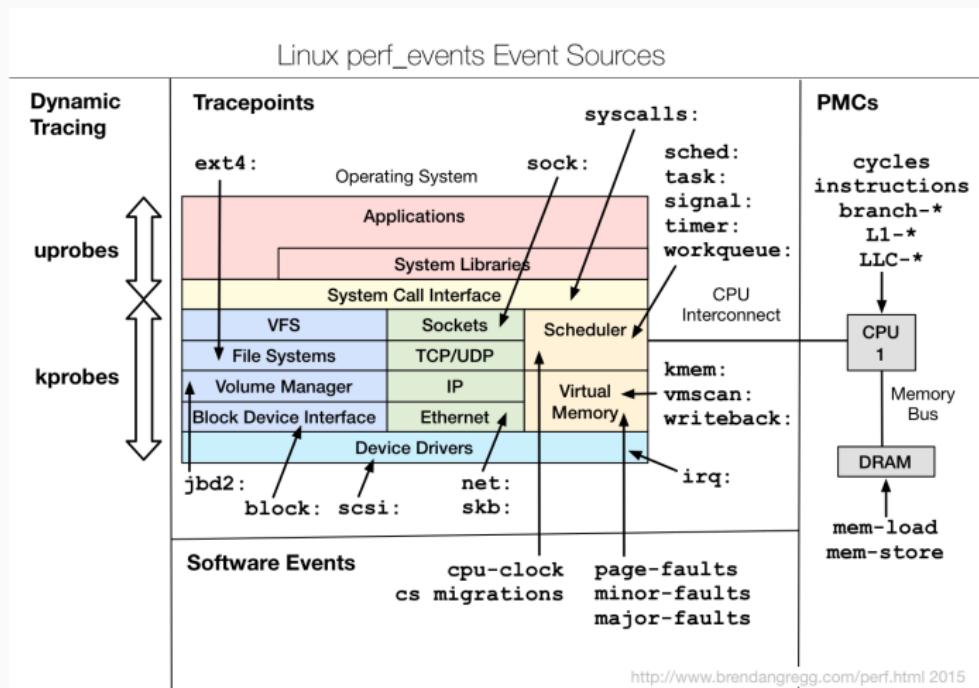
INSTRUCTION LEVEL PARALLELISM & LOOP UNROLLING - CODE III

```
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';
}
```

INSTRUCTION LEVEL PARALLELISM & LOOP UNROLLING - RESULTS

```
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%)
```

- <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/>

```
#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;
}
```

```
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/>

```
#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;
}
```

```
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)

```
#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 0xfffffffffffffe7

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

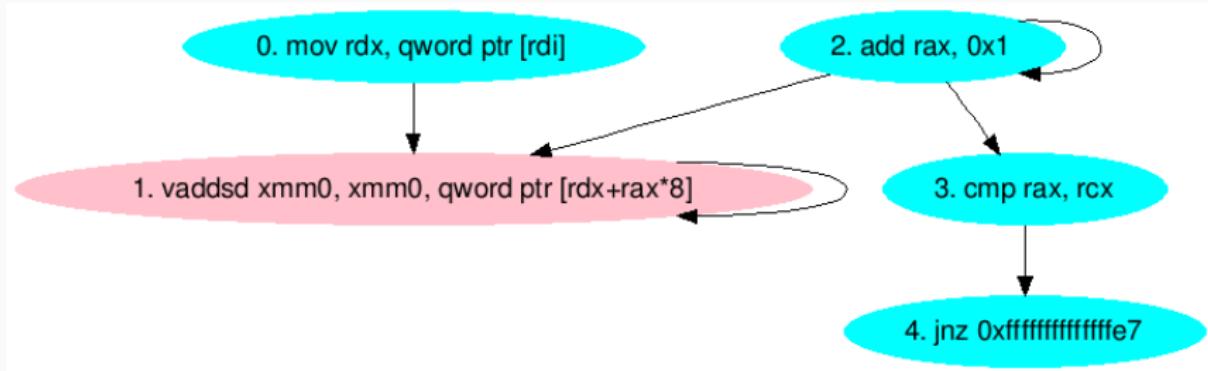
! - instruction not supported, was not accounted in Analysis

Num Of	Ports pressure in cycles									
Uops	0	- DV	1	2	- D	3	- D	4	5	

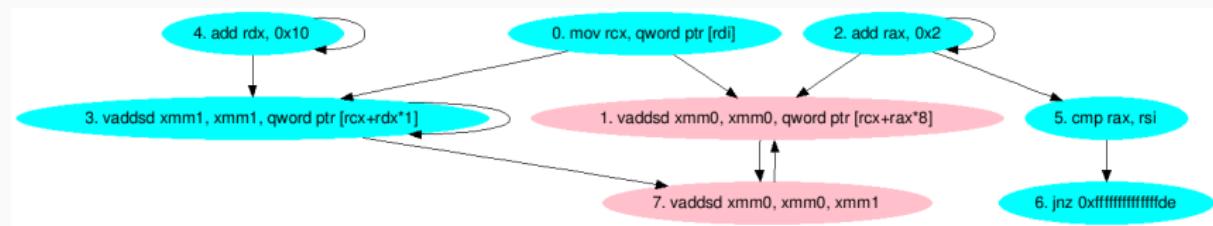
1			0.5	0.5	0.5	0.5				mov rcx, qword ptr [rdi]
2			1.0	0.5	0.5	0.5	0.5			vaddsd xmm0, xmm0, qword ptr [rcx+rax*8]
1	1.0									add rax, 0x2
2			1.0	0.5	0.5	0.5	0.5			vaddsd xmm1, xmm1, qword ptr [rcx+rdx*1]
1	0.5							0.5		add rdx, 0x10
1							1.0			cmp rax, rsi
0F										jnz 0xfffffffffffffde
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

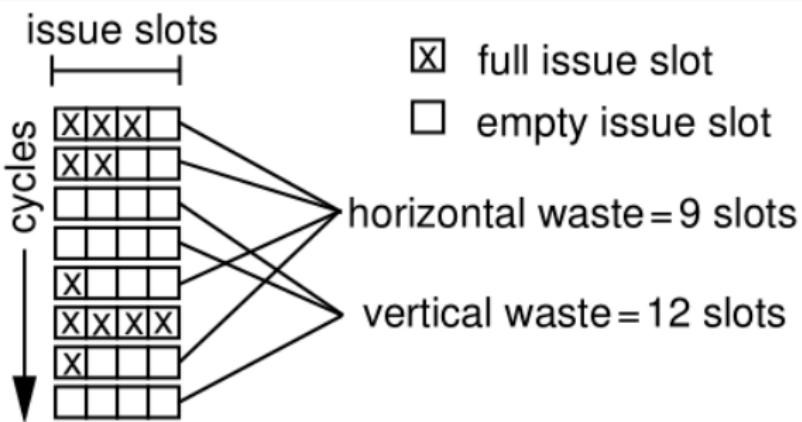


Figure 1: Empty issue slots can be defined as either vertical waste or horizontal waste. Vertical waste is introduced when the processor issues no instructions in a cycle, horizontal waste when not all issue slots can be filled in a cycle. Superscalar execution (as opposed to single-issue execution)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

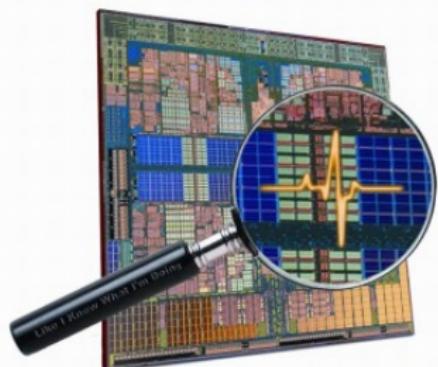
Source of Wasted Issue Slots	Possible Latency-Hiding or Latency-Reducing Technique
instruction tlb miss, data tlb miss	decrease the TLB miss rates (e.g., increase the TLB sizes); hardware instruction prefetching; hardware or software data prefetching; faster servicing of TLB misses
I cache miss	larger, 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 50:0 -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 50:0 -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

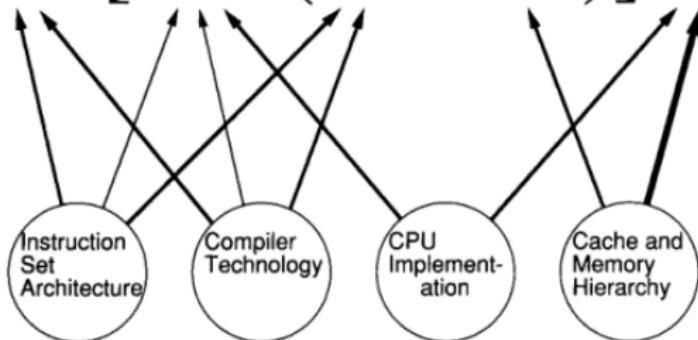
Metric	Core 0
Runtime (RDTSC) [s]	1.6809
Runtime unhalted [s]	1.1377
Clock [MHz]	3435.8987
CPI	0.4534
MFLOP/s	595.1079
AVX MFLOP/s	0
Packed MUOPS/s	0
Scalar MUOPS/s	595.1079

LIKWID RESULTS: SUM_VECTORIZED: 676 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

$$\begin{aligned} \text{Execution Time} &= \frac{\text{Instruction Count}}{\text{Cycles per Instruction}} \times \frac{\text{Cycle Time}}{\text{Instruction Count}} \\ &= \frac{\text{Instruction Count}}{\text{CPU Cycles per Instr.} + \frac{\text{Memory Cycles per Instr.}}{\text{CPU Cycles per Instr.}}} \times \frac{\text{Cycle Time}}{\text{Instruction Count}} \\ &= \frac{\text{Instruction Count}}{\left[\frac{\text{CPU Cycles per Instr.}}{\text{per Instr.}} + \left(\frac{\text{References per Instr.}}{\text{per Instr.}} \times \frac{\text{Cycles per Reference}}{\text{per Reference}} \right) \right]} \times \frac{\text{Cycle Time}}{\text{Instruction Count}} \end{aligned}$$



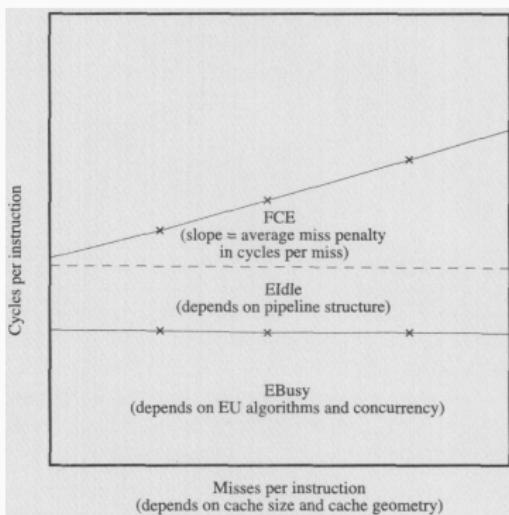
Steven K. Przybylski, "Cache and Memory Hierarchy Design – A Performance-Directed Approach," San Fransisco, Morgan-Kaufmann, 1990.

PERFORMANCE: SEPARABLE COMPONENTS OF A CPI

$$\text{CPI} = (\text{Infinite-cache CPI}) + \text{finite-cache effect (FCE)}$$

$$\text{Infinite-cache CPI} = \text{execute busy (EBusy)} + \text{execute idle (EIdle)}$$

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



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

PIPELINING

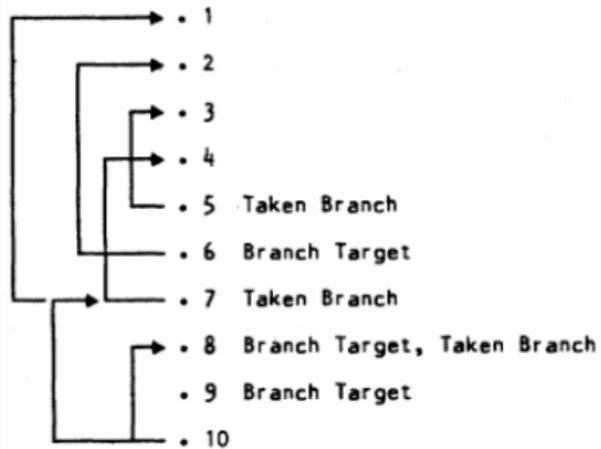


Fig. 1. A typical data-dependency graph.

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)

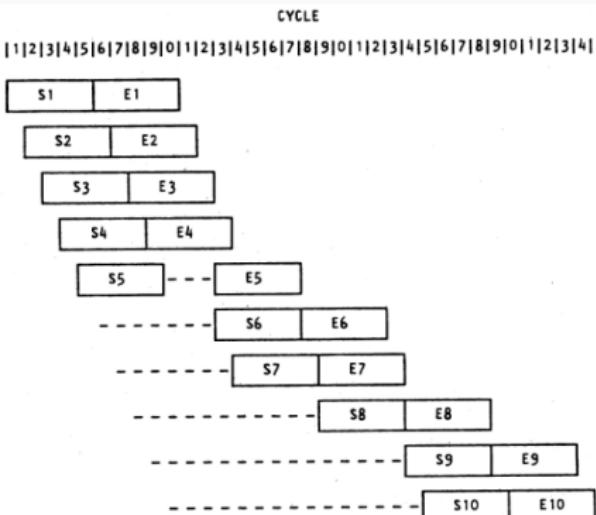


Fig. 2. Pipeline flow of ten instructions with $N_E = N_S = 5$.

BRANCH MISPREDICTION PENALTY

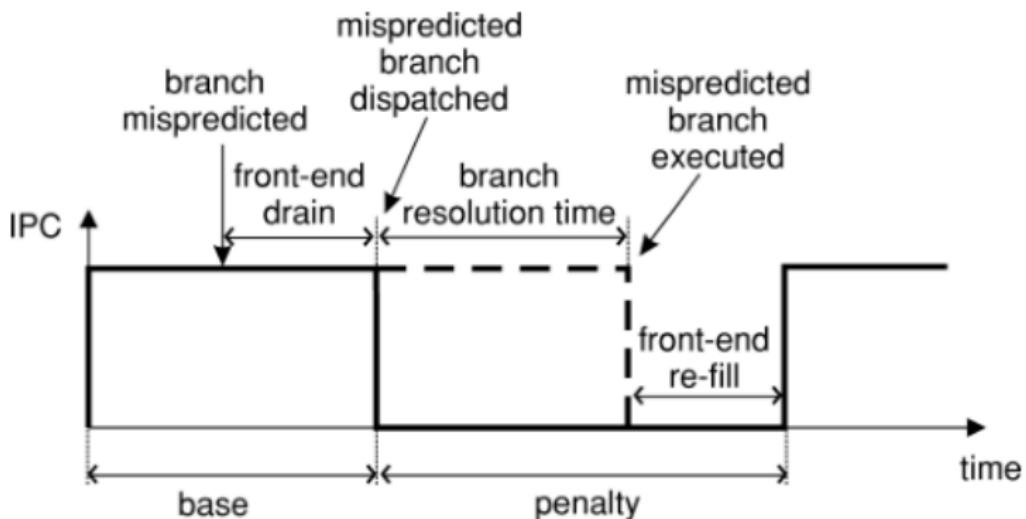


Fig. 6. Interval behavior for a branch misprediction.

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	PMCO	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% )

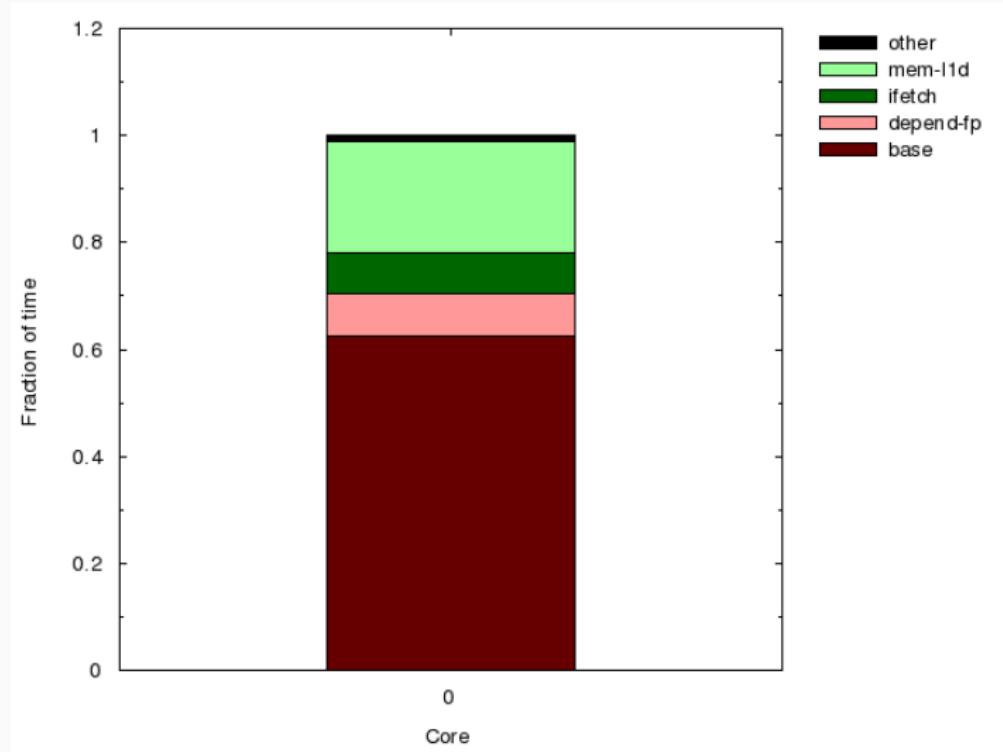
0.884858777 seconds time elapsed          ( +- 0.30% )
```

The Sniper Multi-Core Simulator



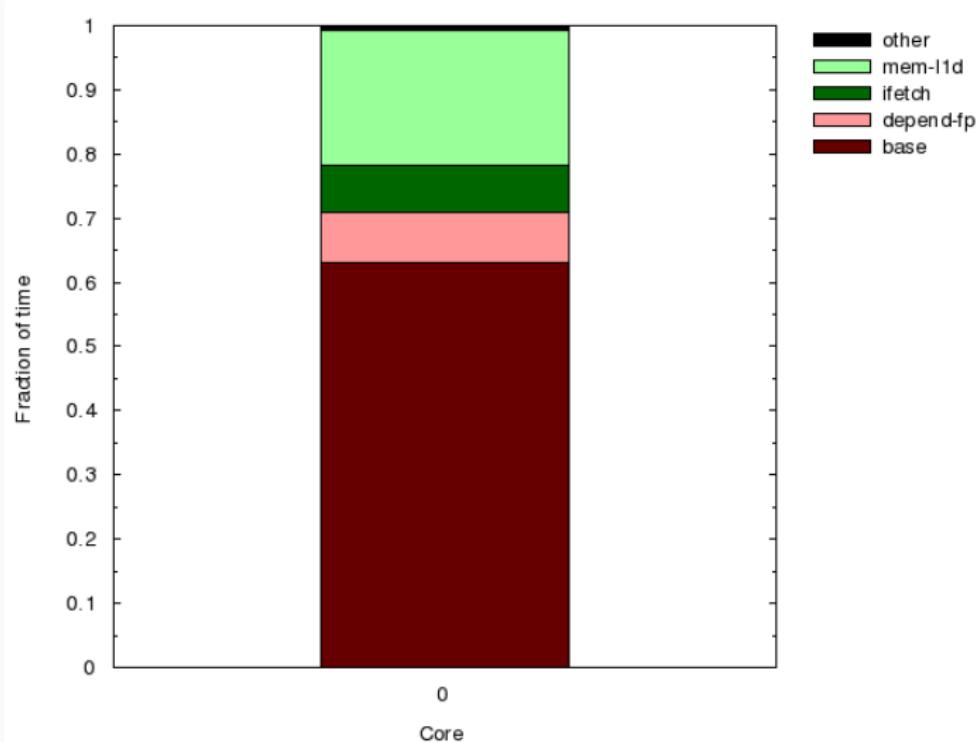
<http://snipersim.org/>

SNIPER: BRANCH (Mis)PREDICTION EXAMPLE



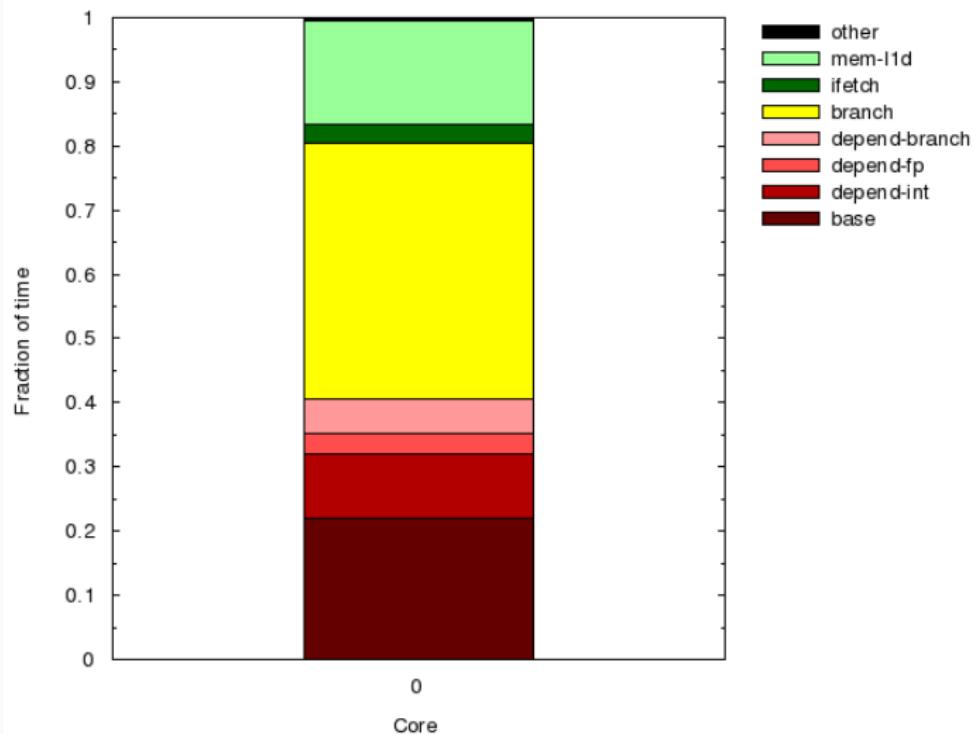
CPI stack: never taken

SNIPER: BRANCH (Mis)PREDICTION EXAMPLE



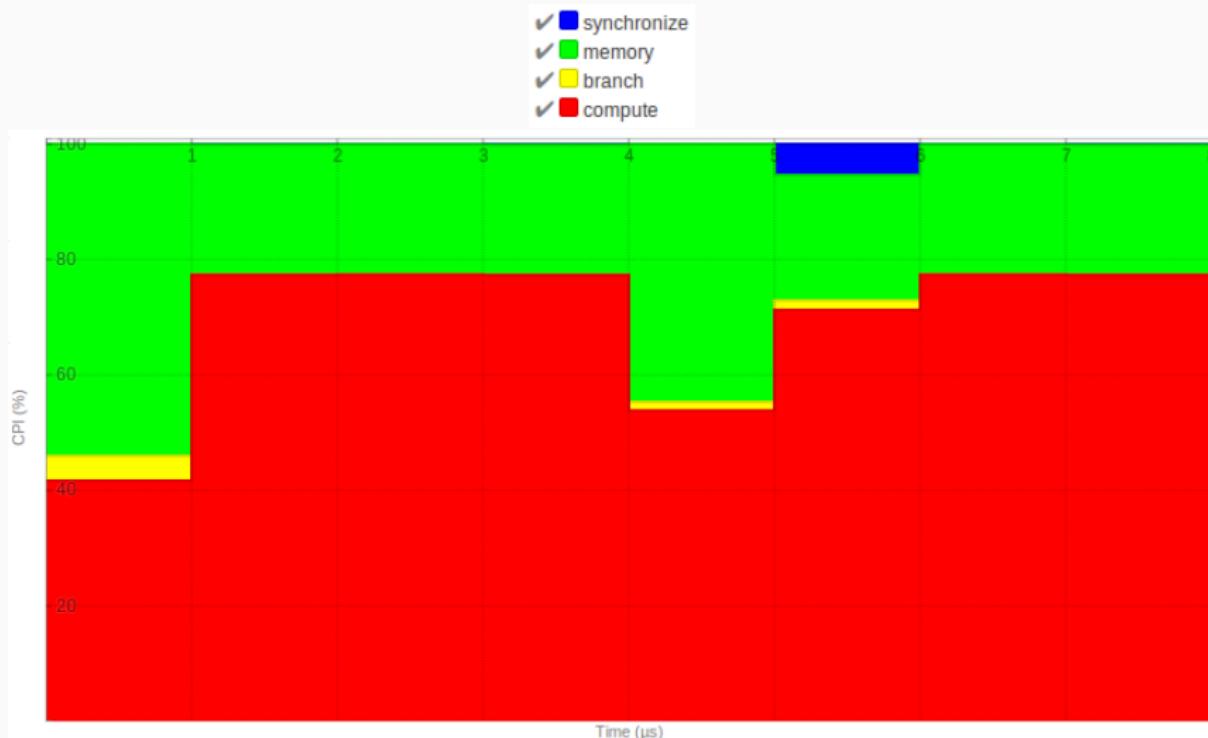
CPI stack: always taken

SNIPER: BRANCH (Mis)PREDICTION EXAMPLE



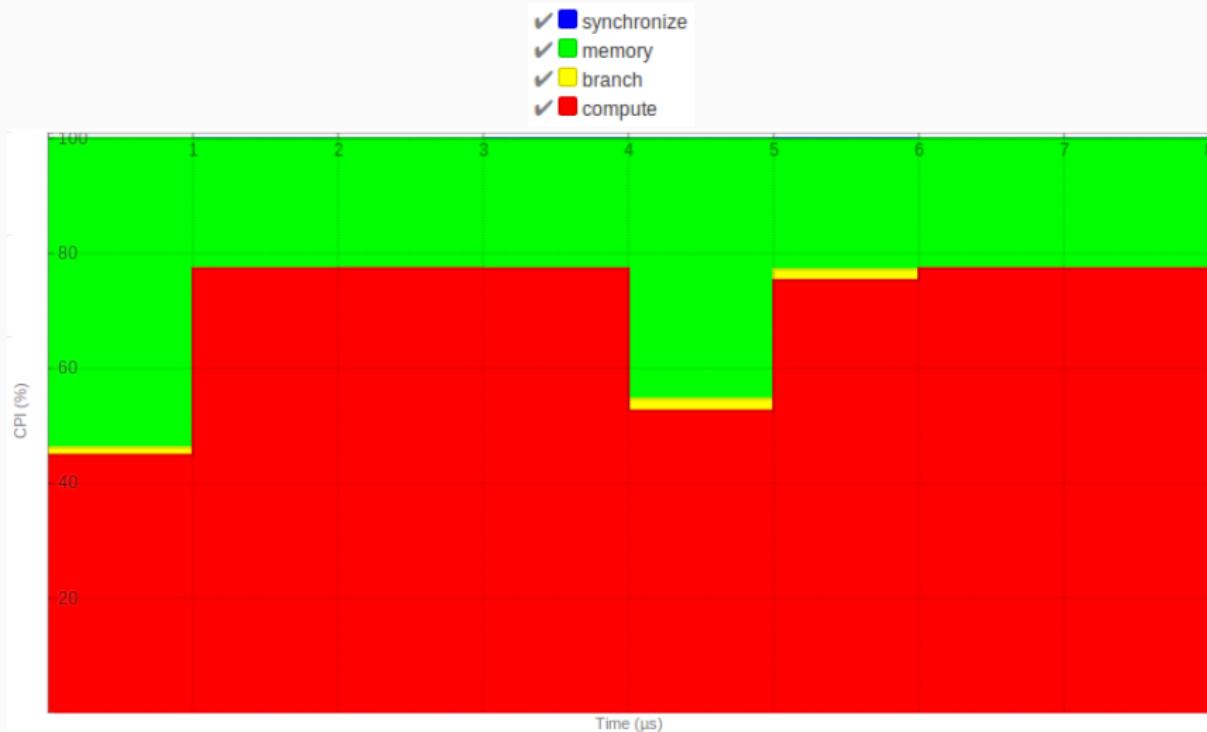
CPI stack: randomly taken

SNIPER: BRANCH (Mis)PREDICTION EXAMPLE



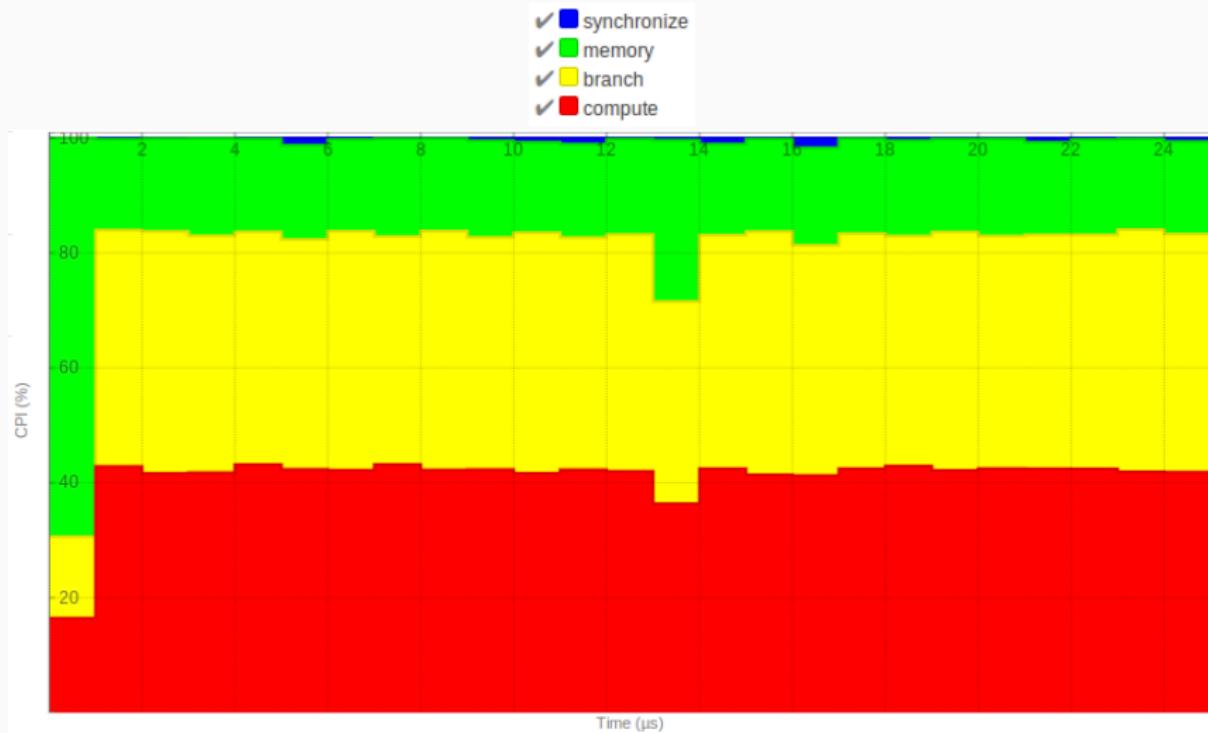
CPI graph: never taken

SNIPER: BRANCH (Mis)PREDICTION EXAMPLE



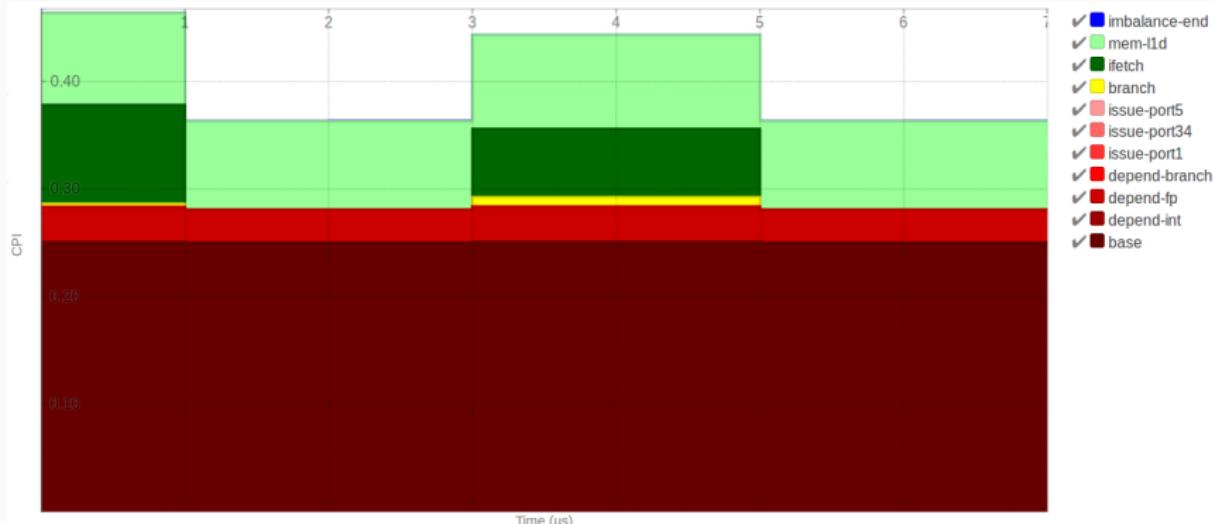
CPI graph: always taken

SNIPER: BRANCH (Mis)PREDICTION EXAMPLE



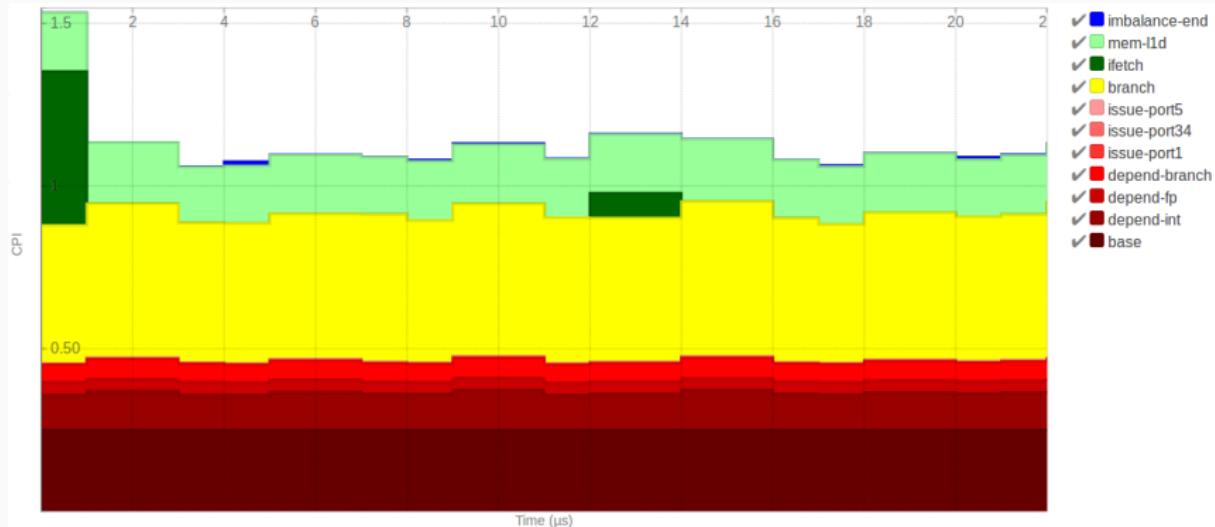
CPI graph: randomly taken

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=0x0000000000000002a
4031500: system.cpu T0 : @main+24.1 :	MOV_M_I : st t1, DS:[rdx] : MemWrite : D=0x0000000000000002a A=0x6dad58
4032500: system.cpu T0 : @main+31.0 :	ADD_R_I : limm t1, 0x8 : IntAlu : D=0x00000000000000008
4033000: system.cpu T0 : @main+31.1 :	ADD_R_I : add rdx, rdx, t1 : IntAlu : D=0x00000000000000000
4033500: system.cpu T0 : @main+35.0 :	CMP_R_R : sub t0, rdx, rcx : IntAlu : D=0x00000000000000001
4034000: system.cpu T0 : @main+38.0 :	JNZ_I : rdip t1, %ctrl153, : IntAlu : D=0x00000000000400d88
4034500: system.cpu T0 : @main+38.1 :	JNZ_I : limm t2, 0xfffffffffffff0 : IntAlu : D=0xfffffffffffff0
4035000: system.cpu T0 : @main+38.2 :	JNZ_I : wrpc , t1, t2 : IntAlu :

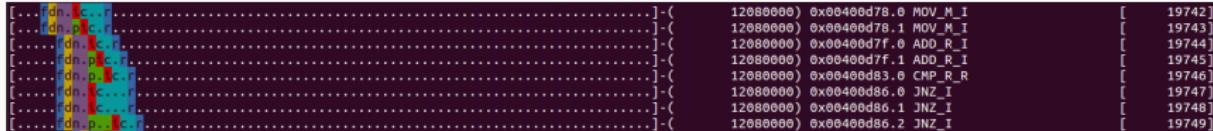
Assembly is Too High Level:

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

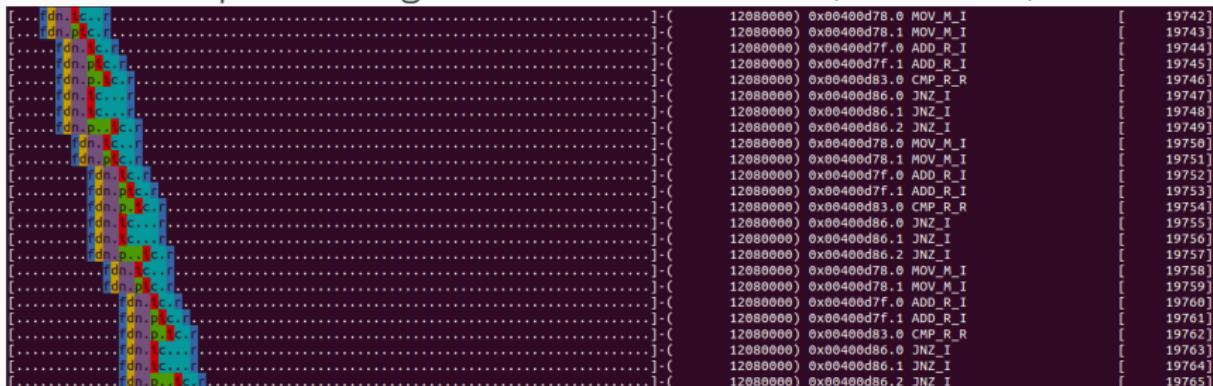
Micro-ops pipeline stages (std::vector)

```
03PipeView:fetch:12081500:0x00400d78:0:19742: MOV_M_I : limm    t1, 0x2a
03PipeView:decode:12082000
03PipeView:rename:12082500
03PipeView:dispatch:12083500
03PipeView:issue:12083500
03PipeView:complete:12084000
03PipeView:retire:12085500:store:0
03PipeView:fetch:12080500:0x00400d83:0:19738: CMP_R_R : sub     t0, rdx, rcx
03PipeView:decode:12081000
03PipeView:rename:12081500
03PipeView:dispatch:12082500
03PipeView:issue:12083500
03PipeView:complete:12084000
03PipeView:retire:12085000:store:0
03PipeView:fetch:12080500:0x00400d86:2:19741: JNZ_I : wrip    , t1, t2
03PipeView:decode:12081000
03PipeView:rename:12081500
03PipeView:dispatch:12082500
03PipeView:issue:12084000
03PipeView:complete:12084500
03PipeView: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      mov    QWORD PTR [rax],0x0
400d96: 48 c7 40 08 00 00      mov    QWORD PTR [rax+0x8],0x0
400d9d: 00
400d9e: 48 c7 40 10 2a 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>
```

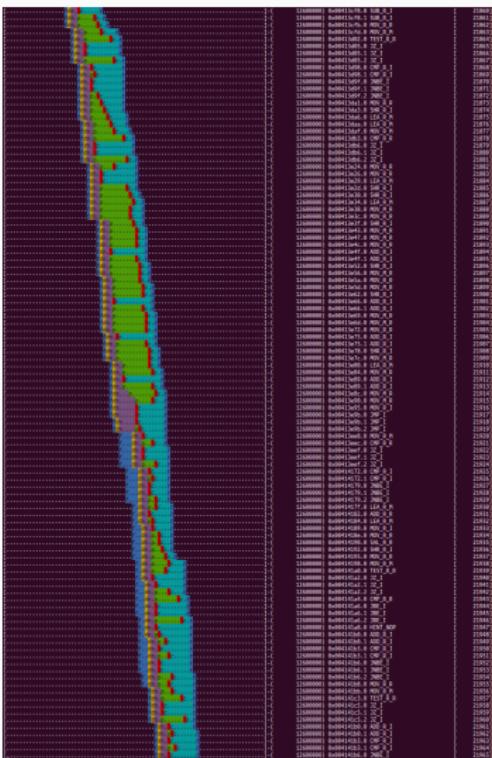
heap allocation in the loop @ 400d85

what could possibly go wrong?

STD::LIST - ONE ITERATION

112500000	0x00000000 3	MOV A, 1	11279
112500000	0x00000000 1	CALL MEM_1	11291
112500000	0x00000000 2	CALL MEM_2	11293
112500000	0x00000000 3	CALL MEM_3	11295
112500000	0x00000000 4	CALL MEM_4	11297
112500000	0x00000000 5	CALL MEM_5	11299
112500000	0x00000000 6	CALL MEM_6	11301
112500000	0x00000000 7	CALL MEM_7	11303
112500000	0x00000000 8	CALL MEM_8	11305
112500000	0x00000000 9	CALL MEM_9	11307
112500000	0x00000000 10	CALL MEM_10	11309
112500000	0x00000000 11	CALL MEM_11	11311
112500000	0x00000000 12	CALL MEM_12	11313
112500000	0x00000000 13	CALL MEM_13	11315
112500000	0x00000000 14	CALL MEM_14	11317
112500000	0x00000000 15	CALL MEM_15	11319
112500000	0x00000000 16	CALL MEM_16	11321
112500000	0x00000000 17	CALL MEM_17	11323
112500000	0x00000000 18	CALL MEM_18	11325
112500000	0x00000000 19	CALL MEM_19	11327
112500000	0x00000000 20	CALL MEM_20	11329
112500000	0x00000000 21	CALL MEM_21	11331
112500000	0x00000000 22	CALL MEM_22	11333
112500000	0x00000000 23	CALL MEM_23	11335
112500000	0x00000000 24	CALL MEM_24	11337
112500000	0x00000000 25	CALL MEM_25	11339
112500000	0x00000000 26	CALL MEM_26	11341
112500000	0x00000000 27	CALL MEM_27	11343
112500000	0x00000000 28	CALL MEM_28	11345
112500000	0x00000000 29	CALL MEM_29	11347
112500000	0x00000000 30	CALL MEM_30	11349
112500000	0x00000000 31	CALL MEM_31	11351
112500000	0x00000000 32	CALL MEM_32	11353
112500000	0x00000000 33	CALL MEM_33	11355
112500000	0x00000000 34	CALL MEM_34	11357
112500000	0x00000000 35	CALL MEM_35	11359
112500000	0x00000000 36	CALL MEM_36	11361
112500000	0x00000000 37	CALL MEM_37	11363
112500000	0x00000000 38	CALL MEM_38	11365
112500000	0x00000000 39	CALL MEM_39	11367
112500000	0x00000000 40	CALL MEM_40	11369
112500000	0x00000000 41	CALL MEM_41	11371
112500000	0x00000000 42	CALL MEM_42	11373
112500000	0x00000000 43	CALL MEM_43	11375
112500000	0x00000000 44	CALL MEM_44	11377
112500000	0x00000000 45	CALL MEM_45	11379
112500000	0x00000000 46	CALL MEM_46	11381
112500000	0x00000000 47	CALL MEM_47	11383
112500000	0x00000000 48	CALL MEM_48	11385
112500000	0x00000000 49	CALL MEM_49	11387
112500000	0x00000000 50	CALL MEM_50	11389
112500000	0x00000000 51	CALL MEM_51	11391
112500000	0x00000000 52	CALL MEM_52	11393
112500000	0x00000000 53	CALL MEM_53	11395
112500000	0x00000000 54	CALL MEM_54	11397
112500000	0x00000000 55	CALL MEM_55	11399
112500000	0x00000000 56	CALL MEM_56	11401
112500000	0x00000000 57	CALL MEM_57	11403
112500000	0x00000000 58	CALL MEM_58	11405
112500000	0x00000000 59	CALL MEM_59	11407
112500000	0x00000000 60	CALL MEM_60	11409
112500000	0x00000000 61	CALL MEM_61	11411
112500000	0x00000000 62	CALL MEM_62	11413
112500000	0x00000000 63	CALL MEM_63	11415
112500000	0x00000000 64	CALL MEM_64	11417
112500000	0x00000000 65	CALL MEM_65	11419
112500000	0x00000000 66	CALL MEM_66	11421
112500000	0x00000000 67	CALL MEM_67	11423
112500000	0x00000000 68	CALL MEM_68	11425
112500000	0x00000000 69	CALL MEM_69	11427
112500000	0x00000000 70	CALL MEM_70	11429
112500000	0x00000000 71	CALL MEM_71	11431
112500000	0x00000000 72	CALL MEM_72	11433
112500000	0x00000000 73	CALL MEM_73	11435
112500000	0x00000000 74	CALL MEM_74	11437
112500000	0x00000000 75	CALL MEM_75	11439
112500000	0x00000000 76	CALL MEM_76	11441
112500000	0x00000000 77	CALL MEM_77	11443
112500000	0x00000000 78	CALL MEM_78	11445
112500000	0x00000000 79	CALL MEM_79	11447
112500000	0x00000000 80	CALL MEM_80	11449
112500000	0x00000000 81	CALL MEM_81	11451
112500000	0x00000000 82	CALL MEM_82	11453
112500000	0x00000000 83	CALL MEM_83	11455
112500000	0x00000000 84	CALL MEM_84	11457
112500000	0x00000000 85	CALL MEM_85	11459
112500000	0x00000000 86	CALL MEM_86	11461
112500000	0x00000000 87	CALL MEM_87	11463
112500000	0x00000000 88	CALL MEM_88	11465
112500000	0x00000000 89	CALL MEM_89	11467
112500000	0x00000000 90	CALL MEM_90	11469
112500000	0x00000000 91	CALL MEM_91	11471
112500000	0x00000000 92	CALL MEM_92	11473
112500000	0x00000000 93	CALL MEM_93	11475
112500000	0x00000000 94	CALL MEM_94	11477
112500000	0x00000000 95	CALL MEM_95	11479
112500000	0x00000000 96	CALL MEM_96	11481
112500000	0x00000000 97	CALL MEM_97	11483
112500000	0x00000000 98	CALL MEM_98	11485
112500000	0x00000000 99	CALL MEM_99	11487
112500000	0x00000000 100	CALL MEM_100	11489

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



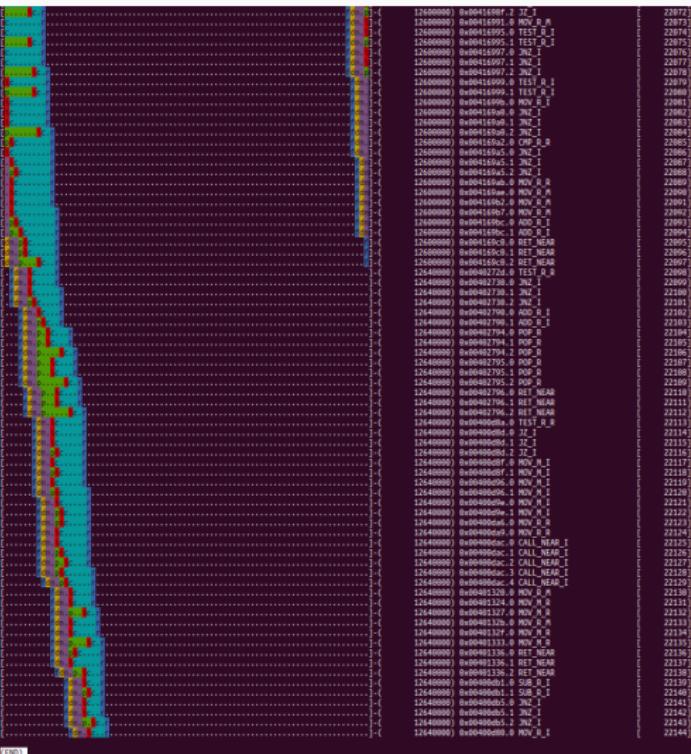
The image consists of a vertical stack of approximately 20 screenshots from a debugger's memory dump feature. Each screenshot shows a list of memory addresses and their corresponding values. The values are mostly zeros, with some non-zero data appearing in specific memory locations. The data is color-coded: green for positive values, red for negative values, and blue for zero values. The addresses are in hexadecimal format, such as 11000000, 11000001, etc. The non-zero data includes strings like "10000000", "10000001", "10000002", and "10000003". The stack of screenshots creates a visual effect of a list being iterated through.

Address	Value
11000000	10000000
11000001	10000001
11000002	10000002
11000003	10000003
11000004	10000004
11000005	10000005
11000006	10000006
11000007	10000007
11000008	10000008
11000009	10000009
1100000A	1000000A
1100000B	1000000B
1100000C	1000000C
1100000D	1000000D
1100000E	1000000E
1100000F	1000000F
11000010	10000010
11000011	10000011
11000012	10000012
11000013	10000013
11000014	10000014
11000015	10000015
11000016	10000016
11000017	10000017
11000018	10000018
11000019	10000019
1100001A	1000001A
1100001B	1000001B
1100001C	1000001C
1100001D	1000001D
1100001E	1000001E
1100001F	1000001F
11000020	10000020
11000021	10000021
11000022	10000022
11000023	10000023
11000024	10000024
11000025	10000025
11000026	10000026
11000027	10000027
11000028	10000028
11000029	10000029
1100002A	1000002A
1100002B	1000002B
1100002C	1000002C
1100002D	1000002D
1100002E	1000002E
1100002F	1000002F
11000030	10000030
11000031	10000031
11000032	10000032
11000033	10000033
11000034	10000034
11000035	10000035
11000036	10000036
11000037	10000037
11000038	10000038
11000039	10000039
1100003A	1000003A
1100003B	1000003B
1100003C	1000003C
1100003D	1000003D
1100003E	1000003E
1100003F	1000003F
11000040	10000040
11000041	10000041
11000042	10000042
11000043	10000043
11000044	10000044
11000045	10000045
11000046	10000046
11000047	10000047
11000048	10000048
11000049	10000049
1100004A	1000004A
1100004B	1000004B
1100004C	1000004C
1100004D	1000004D
1100004E	1000004E
1100004F	1000004F
11000050	10000050
11000051	10000051
11000052	10000052
11000053	10000053
11000054	10000054
11000055	10000055
11000056	10000056
11000057	10000057
11000058	10000058
11000059	10000059
1100005A	1000005A
1100005B	1000005B
1100005C	1000005C
1100005D	1000005D
1100005E	1000005E
1100005F	1000005F
11000060	10000060
11000061	10000061
11000062	10000062
11000063	10000063
11000064	10000064
11000065	10000065
11000066	10000066
11000067	10000067
11000068	10000068
11000069	10000069
1100006A	1000006A
1100006B	1000006B
1100006C	1000006C
1100006D	1000006D
1100006E	1000006E
1100006F	1000006F
11000070	10000070
11000071	10000071
11000072	10000072
11000073	10000073
11000074	10000074
11000075	10000075
11000076	10000076
11000077	10000077
11000078	10000078
11000079	10000079
1100007A	1000007A
1100007B	1000007B
1100007C	1000007C
1100007D	1000007D
1100007E	1000007E
1100007F	1000007F
11000080	10000080
11000081	10000081
11000082	10000082
11000083	10000083
11000084	10000084
11000085	10000085
11000086	10000086
11000087	10000087
11000088	10000088
11000089	10000089
1100008A	1000008A
1100008B	1000008B
1100008C	1000008C
1100008D	1000008D
1100008E	1000008E
1100008F	1000008F
11000090	10000090
11000091	10000091
11000092	10000092
11000093	10000093
11000094	10000094
11000095	10000095
11000096	10000096
11000097	10000097
11000098	10000098
11000099	10000099
1100009A	1000009A
1100009B	1000009B
1100009C	1000009C
1100009D	1000009D
1100009E	1000009E
1100009F	1000009F
110000A0	100000A0
110000A1	100000A1
110000A2	100000A2
110000A3	100000A3
110000A4	100000A4
110000A5	100000A5
110000A6	100000A6
110000A7	100000A7
110000A8	100000A8
110000A9	100000A9
110000AA	100000AA
110000AB	100000AB
110000AC	100000AC
110000AD	100000AD
110000AE	100000AE
110000AF	100000AF
110000B0	100000B0
110000B1	100000B1
110000B2	100000B2
110000B3	100000B3
110000B4	100000B4
110000B5	100000B5
110000B6	100000B6
110000B7	100000B7
110000B8	100000B8
110000B9	100000B9
110000BA	100000BA
110000BB	100000BB
110000BC	100000BC
110000BD	100000BD
110000BE	100000BE
110000BF	100000BF
110000C0	100000C0
110000C1	100000C1
110000C2	100000C2
110000C3	100000C3
110000C4	100000C4
110000C5	100000C5
110000C6	100000C6
110000C7	100000C7
110000C8	100000C8
110000C9	100000C9
110000CA	100000CA
110000CB	100000CB
110000CC	100000CC
110000CD	100000CD
110000CE	100000CE
110000CF	100000CF
110000D0	100000D0
110000D1	100000D1
110000D2	100000D2
110000D3	100000D3
110000D4	100000D4
110000D5	100000D5
110000D6	100000D6
110000D7	100000D7
110000D8	100000D8
110000D9	100000D9
110000DA	100000DA
110000DB	100000DB
110000DC	100000DC
110000DD	100000DD
110000DE	100000DE
110000DF	100000DF
110000E0	100000E0
110000E1	100000E1
110000E2	100000E2
110000E3	100000E3
110000E4	100000E4
110000E5	100000E5
110000E6	100000E6
110000E7	100000E7
110000E8	100000E8
110000E9	100000E9
110000EA	100000EA
110000EB	100000EB
110000EC	100000EC
110000ED	100000ED
110000EE	100000EE
110000EF	100000EF
110000F0	100000F0
110000F1	100000F1
110000F2	100000F2
110000F3	100000F3
110000F4	100000F4
110000F5	100000F5
110000F6	100000F6
110000F7	100000F7
110000F8	100000F8
110000F9	100000F9
110000FA	100000FA
110000FB	100000FB
110000FC	100000FC
110000FD	100000FD
110000FE	100000FE
110000FF	100000FF

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

12000000	0x00111100.1	0001_1	21961
12000000	0x00111101.0	0001_0	21962
12000000	0x00111102.1	0001_1	21963
12000000	0x00111102.0	0001_0	21964
12000000	0x00111103.1	0001_1	21965
12000000	0x00111103.0	0001_0	21966
12000000	0x00111104.1	0001_1	21967
12000000	0x00111104.0	0001_0	21968
12000000	0x00111105.1	0001_1	21969
12000000	0x00111105.0	0001_0	21970
12000000	0x00111106.1	0001_1	21971
12000000	0x00111106.0	0001_0	21972
12000000	0x00111107.1	0001_1	21973
12000000	0x00111107.0	0001_0	21974
12000000	0x00111108.1	0001_1	21975
12000000	0x00111108.0	0001_0	21976
12000000	0x00111109.1	0001_1	21977
12000000	0x00111109.0	0001_0	21978
12000000	0x0011110A.1	0001_1	21979
12000000	0x0011110A.0	0001_0	21980
12000000	0x0011110B.1	0001_1	21981
12000000	0x0011110B.0	0001_0	21982
12000000	0x0011110C.1	0001_1	21983
12000000	0x0011110C.0	0001_0	21984
12000000	0x0011110D.1	0001_1	21985
12000000	0x0011110D.0	0001_0	21986
12000000	0x0011110E.1	0001_1	21987
12000000	0x0011110E.0	0001_0	21988
12000000	0x0011110F.1	0001_1	21989
12000000	0x0011110F.0	0001_0	21990
12000000	0x00111110.1	0001_1	21991
12000000	0x00111110.0	0001_0	21992
12000000	0x00111111.1	0001_1	21993
12000000	0x00111111.0	0001_0	21994
12000000	0x00111112.1	0001_1	21995
12000000	0x00111112.0	0001_0	21996
12000000	0x00111113.1	0001_1	21997
12000000	0x00111113.0	0001_0	21998
12000000	0x00111114.1	0001_1	21999
12000000	0x00111114.0	0001_0	22000
12000000	0x00111115.1	0001_1	22001
12000000	0x00111115.0	0001_0	22002
12000000	0x00111116.1	0001_1	22003
12000000	0x00111116.0	0001_0	22004
12000000	0x00111117.1	0001_1	22005
12000000	0x00111117.0	0001_0	22006
12000000	0x00111118.1	0001_1	22007
12000000	0x00111118.0	0001_0	22008
12000000	0x00111119.1	0001_1	22009
12000000	0x00111119.0	0001_0	22010
12000000	0x0011111A.1	0001_1	22011
12000000	0x0011111A.0	0001_0	22012
12000000	0x0011111B.1	0001_1	22013
12000000	0x0011111B.0	0001_0	22014
12000000	0x0011111C.1	0001_1	22015
12000000	0x0011111C.0	0001_0	22016
12000000	0x0011111D.1	0001_1	22017
12000000	0x0011111D.0	0001_0	22018
12000000	0x0011111E.1	0001_1	22019
12000000	0x0011111E.0	0001_0	22020
12000000	0x0011111F.1	0001_1	22021
12000000	0x0011111F.0	0001_0	22022
12000000	0x00111120.1	0001_1	22023
12000000	0x00111120.0	0001_0	22024
12000000	0x00111121.1	0001_1	22025
12000000	0x00111121.0	0001_0	22026
12000000	0x00111122.1	0001_1	22027
12000000	0x00111122.0	0001_0	22028
12000000	0x00111123.1	0001_1	22029
12000000	0x00111123.0	0001_0	22030
12000000	0x00111124.1	0001_1	22031
12000000	0x00111124.0	0001_0	22032
12000000	0x00111125.1	0001_1	22033
12000000	0x00111125.0	0001_0	22034
12000000	0x00111126.1	0001_1	22035
12000000	0x00111126.0	0001_0	22036
12000000	0x00111127.1	0001_1	22037
12000000	0x00111127.0	0001_0	22038
12000000	0x00111128.1	0001_1	22039
12000000	0x00111128.0	0001_0	22040
12000000	0x00111129.1	0001_1	22041
12000000	0x00111129.0	0001_0	22042
12000000	0x0011112A.1	0001_1	22043
12000000	0x0011112A.0	0001_0	22044
12000000	0x0011112B.1	0001_1	22045
12000000	0x0011112B.0	0001_0	22046
12000000	0x0011112C.1	0001_1	22047
12000000	0x0011112C.0	0001_0	22048
12000000	0x0011112D.1	0001_1	22049
12000000	0x0011112D.0	0001_0	22050
12000000	0x0011112E.1	0001_1	22051
12000000	0x0011112E.0	0001_0	22052
12000000	0x0011112F.1	0001_1	22053
12000000	0x0011112F.0	0001_0	22054
12000000	0x00111130.1	0001_1	22055
12000000	0x00111130.0	0001_0	22056
12000000	0x00111131.1	0001_1	22057
12000000	0x00111131.0	0001_0	22058
12000000	0x00111132.1	0001_1	22059
12000000	0x00111132.0	0001_0	22060
12000000	0x00111133.1	0001_1	22061
12000000	0x00111133.0	0001_0	22062
12000000	0x00111134.1	0001_1	22063
12000000	0x00111134.0	0001_0	22064
12000000	0x00111135.1	0001_1	22065
12000000	0x00111135.0	0001_0	22066
12000000	0x00111136.1	0001_1	22067
12000000	0x00111136.0	0001_0	22068
12000000	0x00111137.1	0001_1	22069
12000000	0x00111137.0	0001_0	22070
12000000	0x00111138.1	0001_1	22071
12000000	0x00111138.0	0001_0	22072

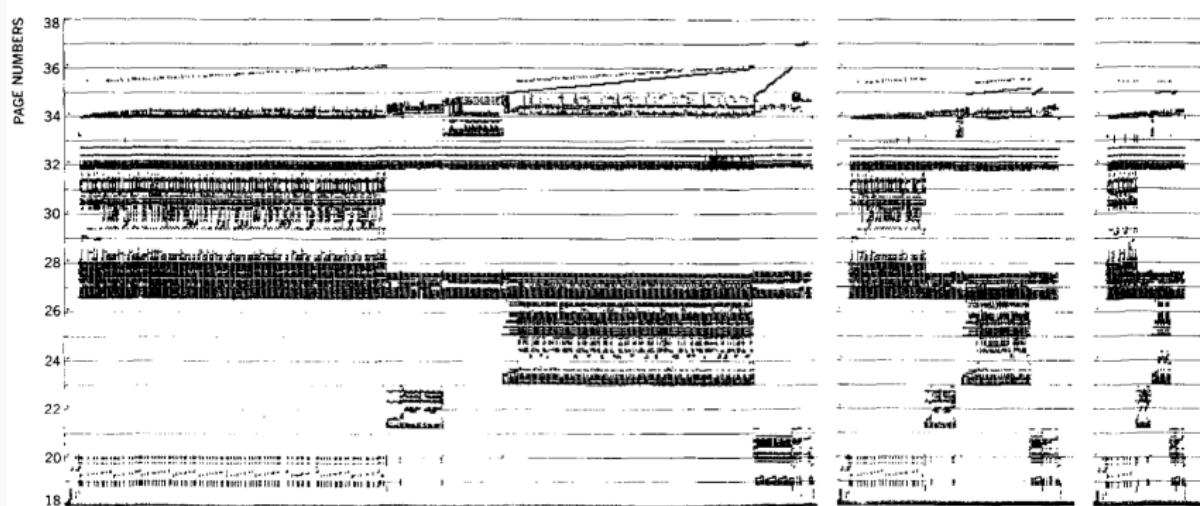
STD::LIST - ONE ITERATION (...DONE!)



(END)

MEMORY ACCESS PATTERNS: TEMPORAL & SPATIAL LOCALITY

Figure 2 Memory usage during separate compilations



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.

LOOP FUSION

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)

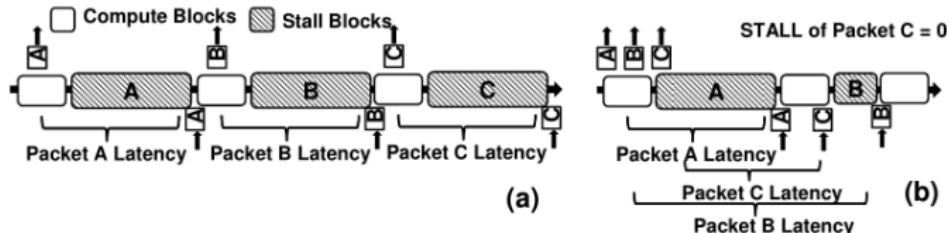


Figure 3: Execution timeline for: (a) an application with few packets, which do not overlap with each other, hence the packet latency is completely exposed to the processor as stall cycles, making each packet critical to the processor. (b) an application with packets that overlap with each other, hence 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.

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`

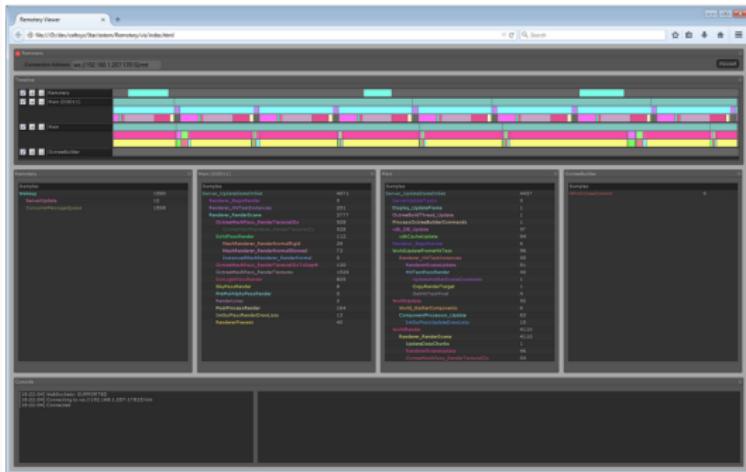
`20,MSFT,475451,2.51908`

`21,NKE,556344,2.51909`

VISUALIZING & MONITORING PERFORMANCE

<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

Sample	Value
Remotery	0
V	0
GS	0
VZ	0
UNH	0
CSCO	0
AXP	0
MCD	0

Remotery V GS VZ UNH CSCO AXP MCD

Samples Samples Samples

Console

```
[5:57:37 AM] Connecting to ws://127.0.0.1:17815/rmt [6:00:35 AM] PG
[5:57:37 AM] Connected [6:00:35 AM] PFE
[6:00:55 AM] Connection Error [6:00:35 AM] TRV
[6:00:55 AM] Disconnected [6:00:35 AM] UNH
[6:00:56 AM] Connecting to ws://127.0.0.1:17815/rmt [6:00:35 AM] VZ
[6:00:58 AM] Connection Error [6:00:35 AM] UTX
[6:00:58 AM] Disconnected [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

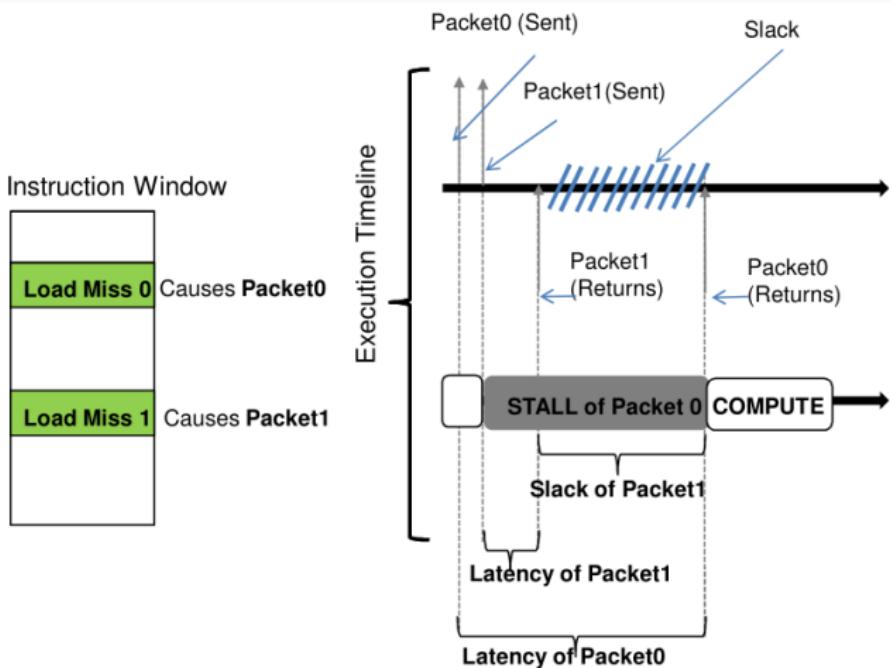
Sample	Label
✓ + - Remotery	LogText
✓ + - GS	LogText
✓ + - HD	LogText
✓ + - TRV	LogText
✓ + - CSCO	LogText
✓ + - AAPL	LogText
✓ + - UNH	LogText
✓ + - MSFT	LogText

Remotery G5

Console

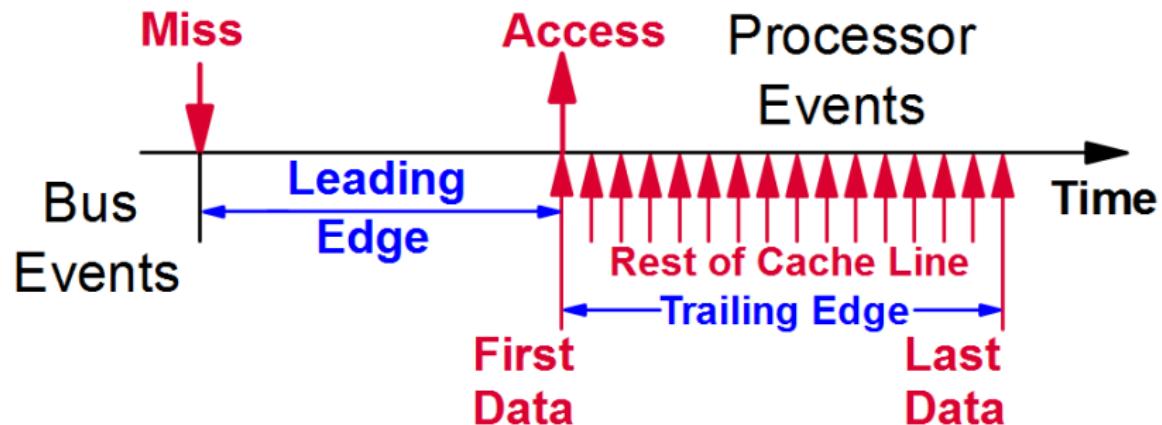
```
[6:01:19 AM] Connecting to ws://127.0.0.1:17815/rmt [6:02:12 AM] MSFT
[6:01:23 AM] Connection Error [6:02:12 AM] NKE
[6:01:23 AM] Disconnected [6:02:12 AM] PFE
[6:01:25 AM] Connecting to ws://127.0.0.1:17815/rmt [6:02:12 AM] PG
[6:01:27 AM] Connected [6:02:12 AM] TRV
[6:02:02 AM] Connection Error [6:02:12 AM] UNH
[6:02:02 AM] Disconnected [6:02:12 AM] UTX
[6:02:03 AM] Connecting to ws://127.0.0.1:17815/rmt [6:02:12 AM] V
[6:02:04 AM] Connected [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

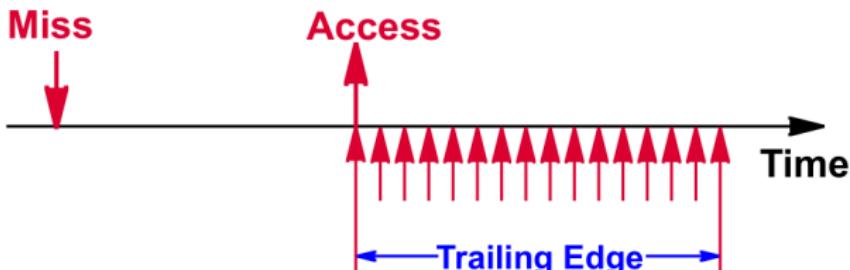


Miss Penalty = Leading Edge + Effects(Trailing Edge)

"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

How Are TE and BW Related?



$$TE = (\text{Line Size} / \text{Bus Width}) \times (\text{Proc. Freq.} / \text{Bus Freq.})$$

Bus Utilization = Trailing Edge / Intermiss Distance

"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.hpcacconf.org/hpcac12/Phil_HPCA_06.pdf

MEMORY CAPACITY & MULTICORE PROCESSORS

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

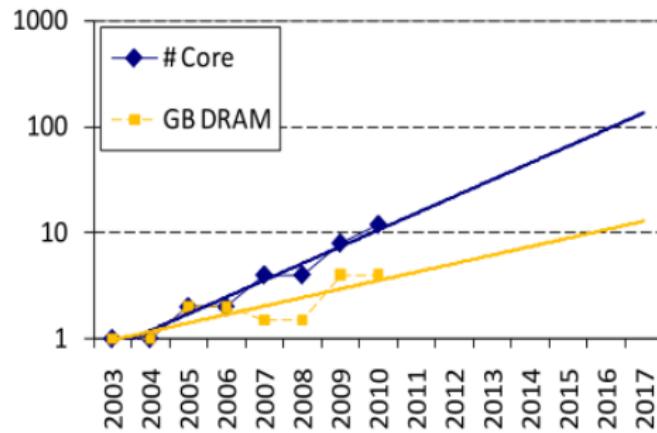


Figure 1: Projected annual growth in number of cores and memory capacity.

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

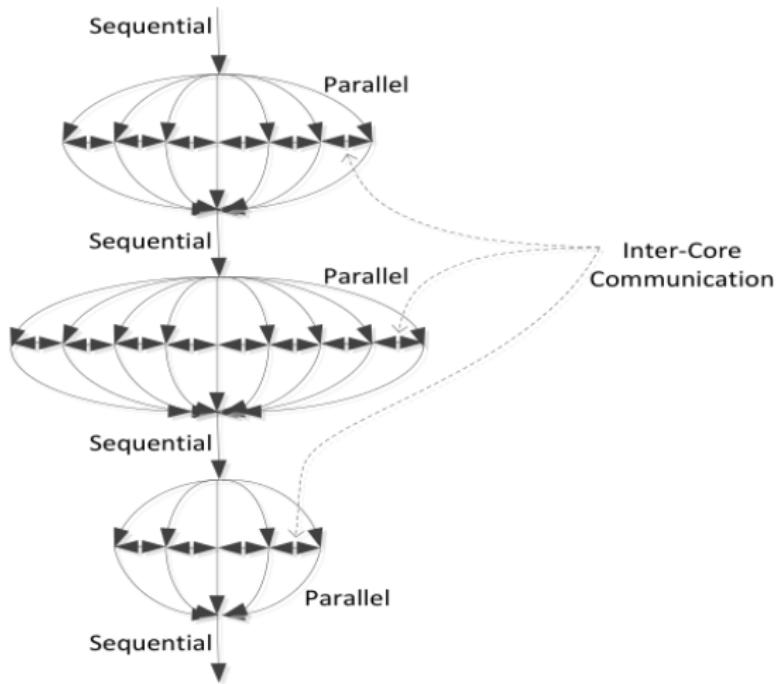


Figure 3: Sequential / Parallel Execution Model

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

MULTICORE: AMDAHL'S LAW, STRONG SCALING

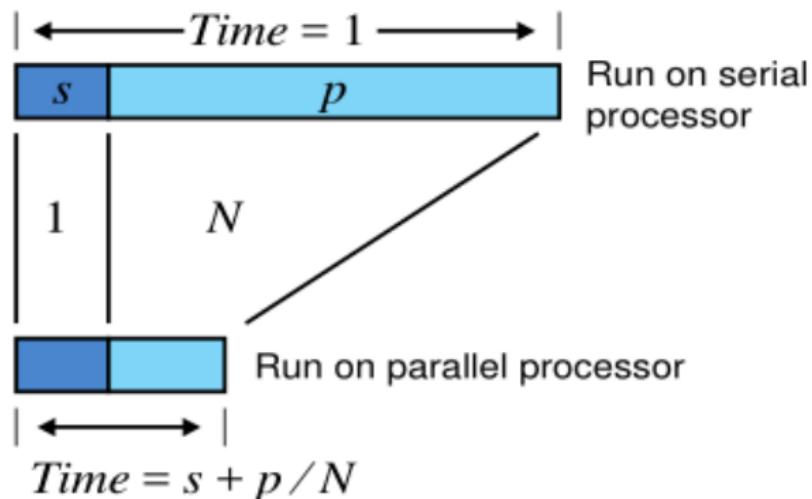


Figure 2a. Fixed-Size Model: $Speedup = 1 / (s + p / N)$

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

MULTICORE: GUSTAFSON'S LAW, WEAK SCALING

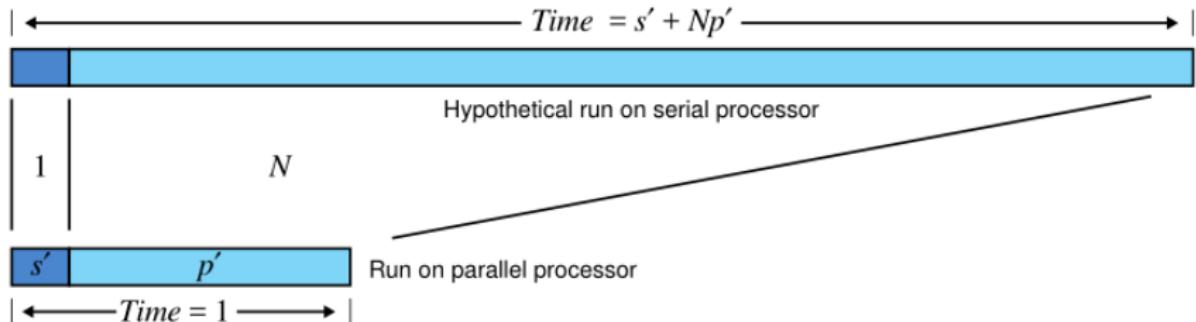


FIGURE 2b. Scaled-Size Model: $Speedup = s + Np$

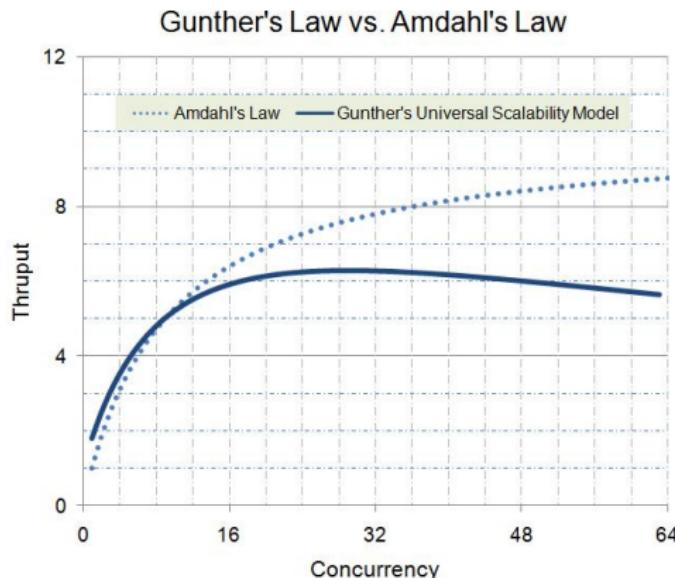
Reevaluating Amdahl's Law, John L. Gustafson, Communications of the ACM 31(5),
1988. pp. 532-533.

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-amdaahls-law-vs-gunthers-law/>

MULTICORE: SYNCHRONIZATION, ACTUAL SCALING

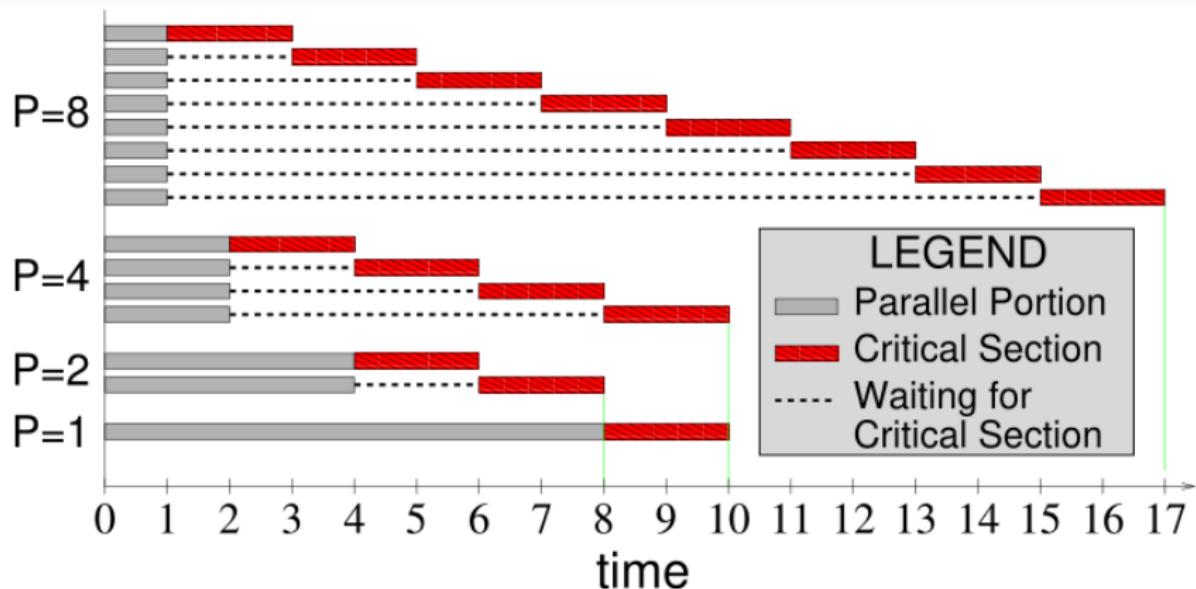


Figure 6. Example for analyzing impact of critical sections

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.

MULTICORE: COMMUNICATION, ACTUAL SCALING

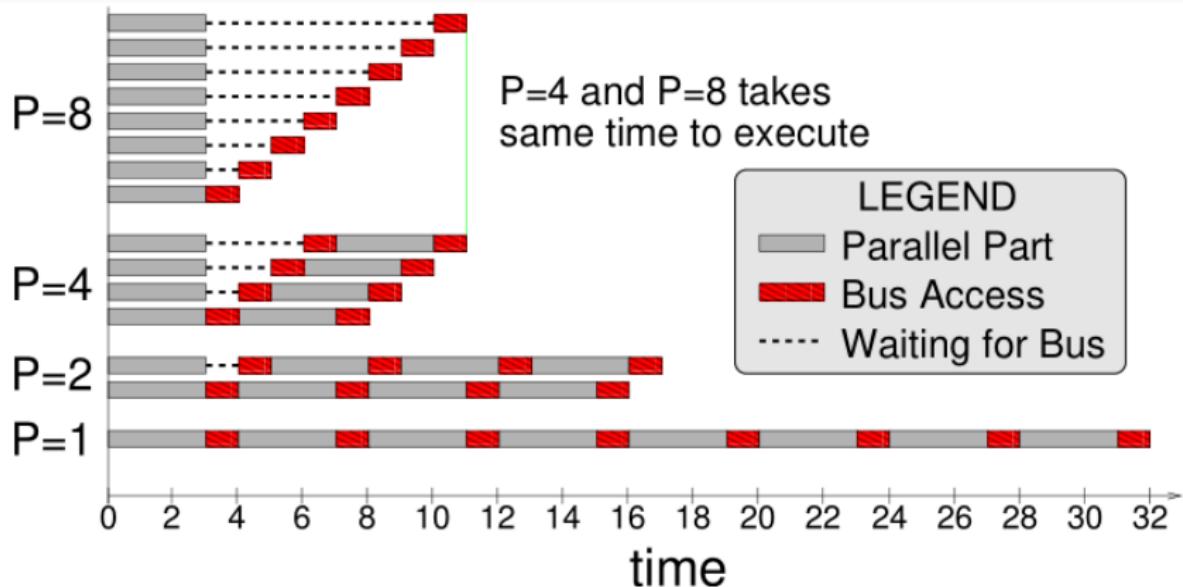


Figure 11. Example for analyzing bandwidth limited systems

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.

```
#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';
    }
}
```

MULTICORE & DRAM: AoS V

```
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);
};
```

```
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%)
```

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%)
```

MULTICORE & DRAM: AoS TIMINGS

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.

MULTICORE & DRAM: SoA I

```
#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)
{
```

MULTICORE & DRAM: SoA V

```
thread_type[thread] = (argc >= 4 + thread) ? std::atoll(argv[3 + th
    std::cout << "thread_type[" << thread << "] = " << thread_type[thread
}
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 100000000 10 1 1 1 1  
n = 100000000  
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 100000000 10 2
```

```
n = 100000000
```

```
m = 10
```

```
thread_type[0] = 2
```

```
1e+007
```

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

MULTICORE & DRAM: SoA TIMINGS

4 threads, sequential access

```
$ ./DRAM_CMP.SoA 100000000 10 1 1 1 1
```

```
n = 100000000
```

```
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 100000000 10 1 1 1 2
```

```
n = 100000000
```

```
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%)
```

Better Access Patterns

yield

Better Single-core Performance

but also

Reduced Interference

and thus

Better Multi-core Performance

MULTICORE: ARITHMETIC INTENSITY

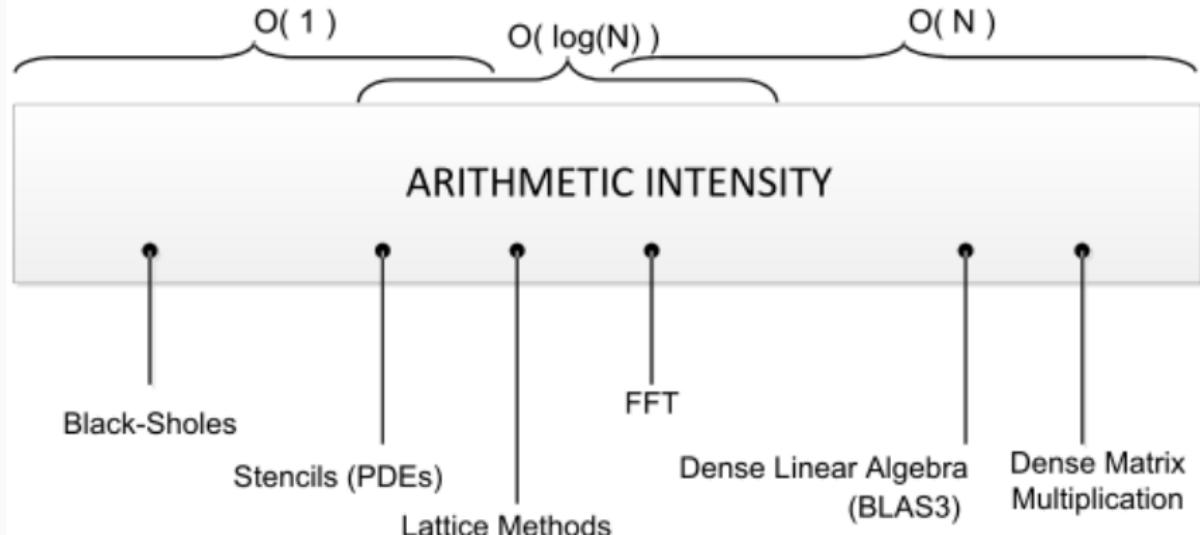


Figure 4: Arithmetic Intensity [26].

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

MULTICORE: SYNCHRONIZATION & CONNECTIVITY INTENSITY

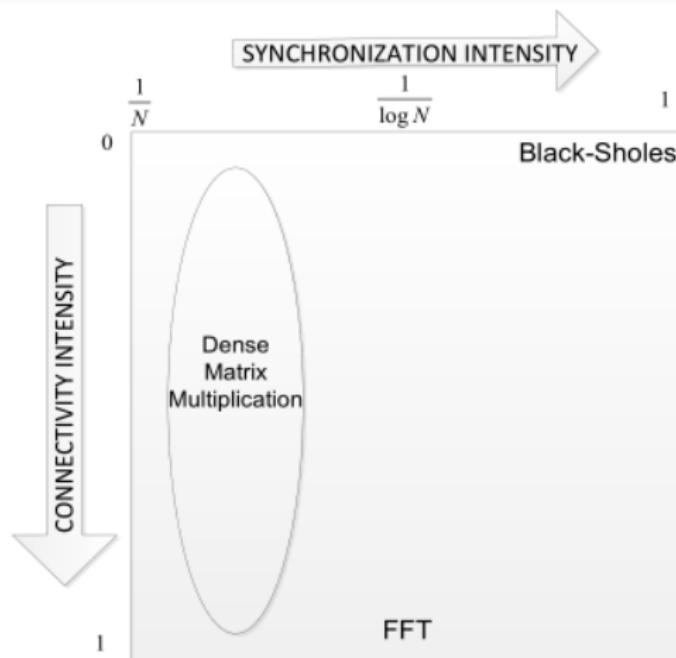


Figure 5: Synchronization and Connectivity Intensity

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

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

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

SPEEDUP: SYNCHRONIZATION & CONNECTIVITY BOTTLENECKS

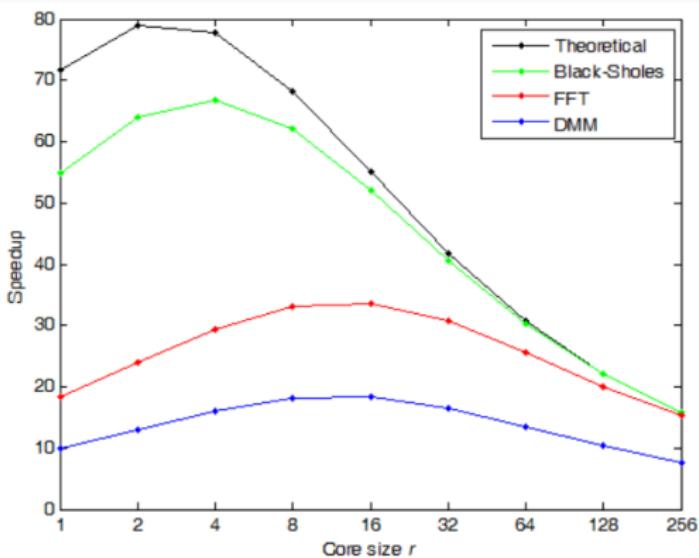
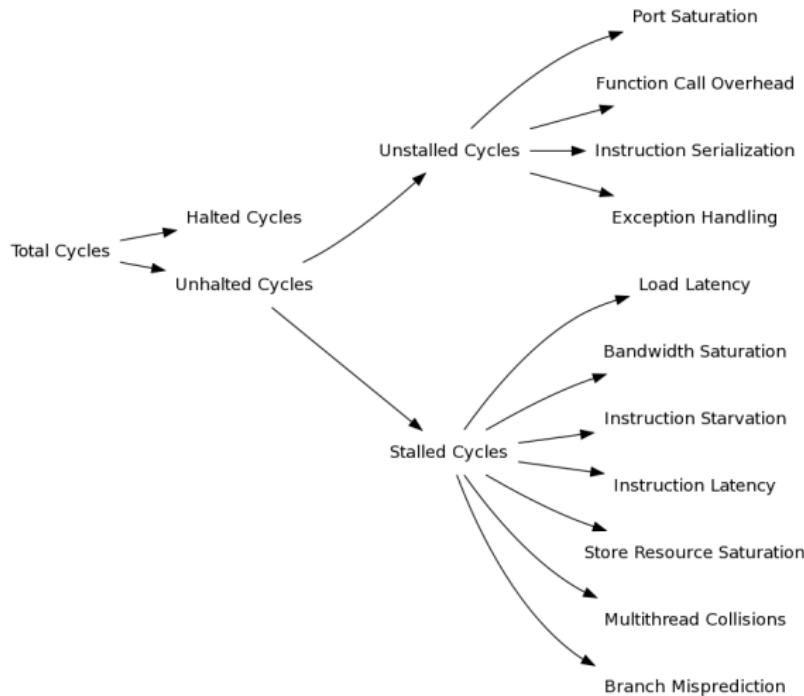


Figure 8. Simulation results: Speedup vs. core size

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>

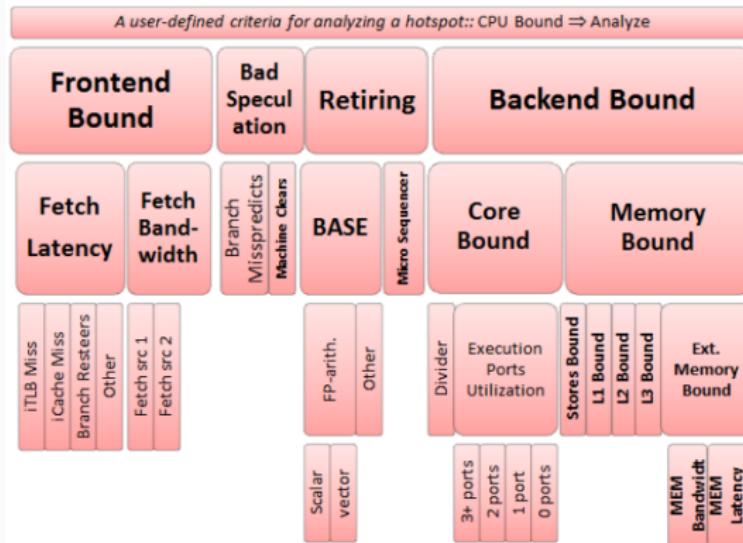


Figure 2: The Top-Down Analysis Hierarchy

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.

ROOFLINE MODEL(S)

<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

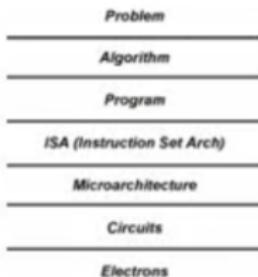
Overlapping latencies

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

PHASE 2: HARNESSING 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/>

<https://speakerdeck.com/mattpd>

THANK YOU!
QUESTIONS?