USC Viterbi
School of Engineering

A Compiler-Based Strategy for Performance Tuning of Applications

Mary Hall

December 7, 2007

* This work has been partially sponsored by DOE SciDAC as part of the Performance Engineering Research Institute (PERI), the National Science Foundation and Intel Corporation.

Local Collaborators

- Compiler group:
  - Jacqueline Chame (research scientist)
  - Chun Chen (postdoctoral researcher)
  - Yoon-Ju Lee Nelson, Muhammad Murtaza, Melina Demertzi (Phd students)
- ISI collaborators:
  - Ewa Deelman, Yolanda Gil, Kristina Lerman, Robert Lucas
- Alumnus collaborator:
  - Jaewook Shin (Argonne)
- Other USC collaborators:
  - Rajiv Kalia, Aiichiro Nakano, Priya Vashishta

Prelude: Things Ken Taught Me

- **1983:** Team programming projects reflect the less-than-perfect real world.
- **1984:** Algorithms are more fascinating than accounting.
- **1985:** Compiler research combines elegant algorithms with building something tangible and important.
- **1986:** After three practices and rewrites, a talk is probably ready for IBM.
- **1992:** Follow the money.

- ~~**Throughout:** Always try to minimize compile time.~~

"Compile time should take as long as you need to get the performance you want." Keith Cooper, 11/27/07

USC Viterbi School of Engineering

Motivation: Petascale

1. Maximizing "node"* performance is crucial to getting to petascale performance.

   Fewer nodes, faster result, larger problem...

2. "Nodes" are getting increasingly complex.

   Deep memory hierarchies, SIMD engines, Double hummer, Multi/many core, Software-managed storage, ...

3. Application scientists have more worthwhile things to do than machine-dependent code rewrites and tedious performance-tuning experiments.

   Focus on science, application capability

4. Petascale resources can and should be harnessed to improve programmer productivity and application performance.

   Useful to tune long-running or frequently executed code.

   Tune in execution context.

***Node in this context may refer to a multi-core device.**

USC Viterbi School of Engineering

Motivation: Multicore Paradigm Shift

1. ~~Maximizing processor performance is crucial to getting to petascale performance.~~

   ~~Fewer nodes, faster result, larger problem…~~

2. "Nodes" are getting increasingly complex.

   Deep memory hierarchies, SIMD engines, Double hummer, multithreading, Multi/many core, Software-managed storage, …

3. Application **developers** have more worthwhile things to do than machine-dependent code rewrites and tedious performance-tuning experiments.

   Focus on ~~science,~~ application capability

4. **Multicore** resources can and should be harnessed to improve programmer productivity and application performance.

   Useful to tune long-running or frequently executed code.

   Tune in execution context.

***Node in this context may refer to a multi-core device.**

Driver for Research

- Overlap of requirements for petascale scientific computing and mainstream multi-core computing.
- Many new and "commodity" application domains are similar to scientific computing.
  - Communication, speech, graphics and games, some cognitive algorithms, biomedical informatics **(RMS applications)**
- Work with real applications **(client?)**.
  - Molecular dynamics simulation, Computational chemistry, Speech recognition, Biomedical image processing, …
- Start with development of libraries.

USC Viterbi
School of Engineering

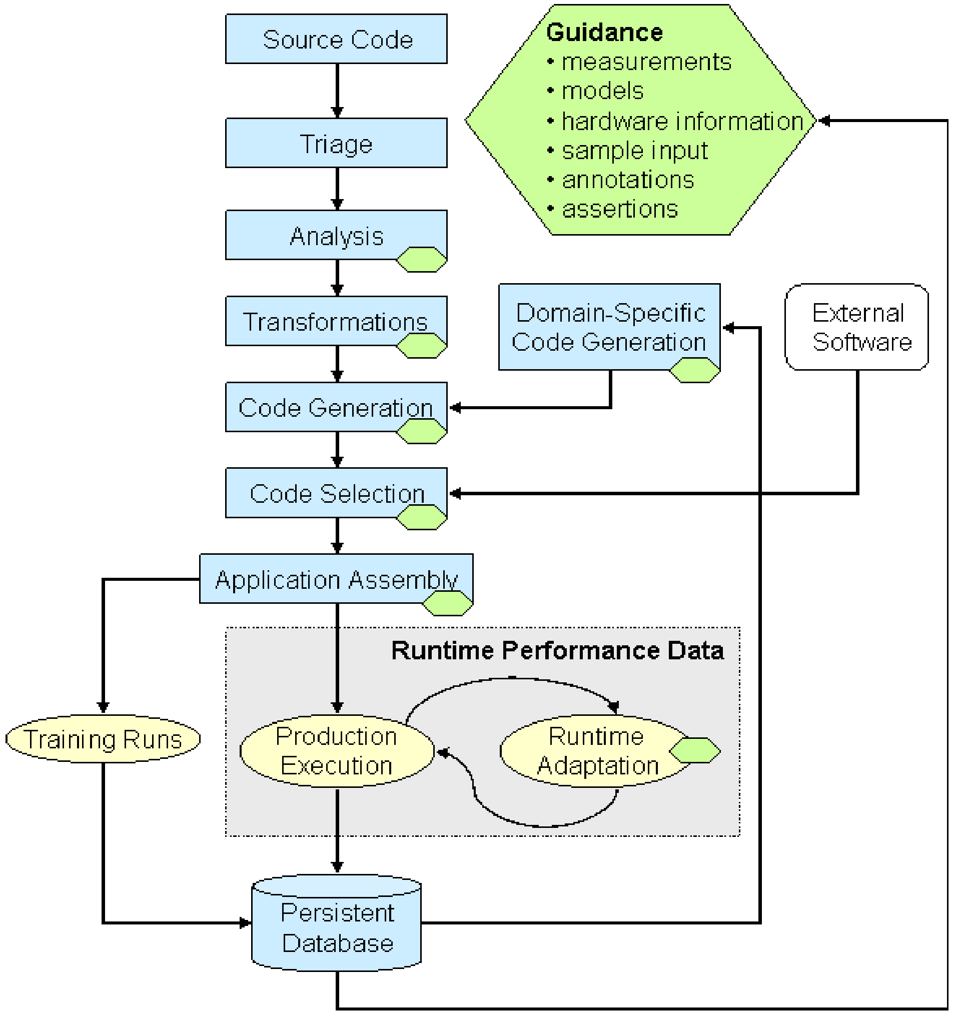
Outline

1. Motivation
2. Challenges & opportunities
3. Approach & potential of compiler-assisted tuning
    - New flexible and systematic compiler technology
    - Scenarios from application tuning
    - Automatic performance tuning
4. Future directions

Performance Engineering Research Institute (SciDAC-2)

- Long-term goal is to automate the process of tuning software to maximize its performance
- Reduces performance portability challenge for computational scientists.
- Addresses the problem that performance experts are in short supply.
- Builds on forty years of human experience and recent success with linear algebra libraries.

Slide source: Bob Lucas and David Bailey



PERI automatic tuning framework

USC Viterbi School of Engineering

Key Research Themes

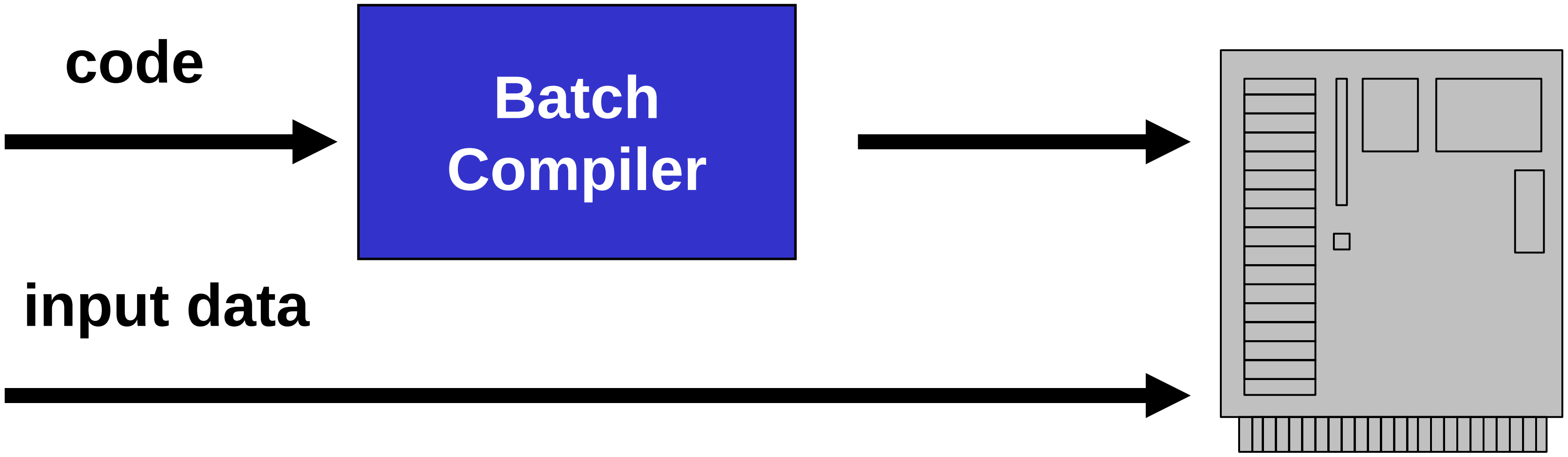
- Compiler-based performance tuning tools
  - Use vast resources of petascale systems
  - Enumerate options, generate code, try, measure, record (conceptually)
- Optimizing compilers built from modular, understandable chunks
  - Easier to bring up on new platforms
  - Facilitates collaboration, moving the community forward

A Systematic, Principled Approach!

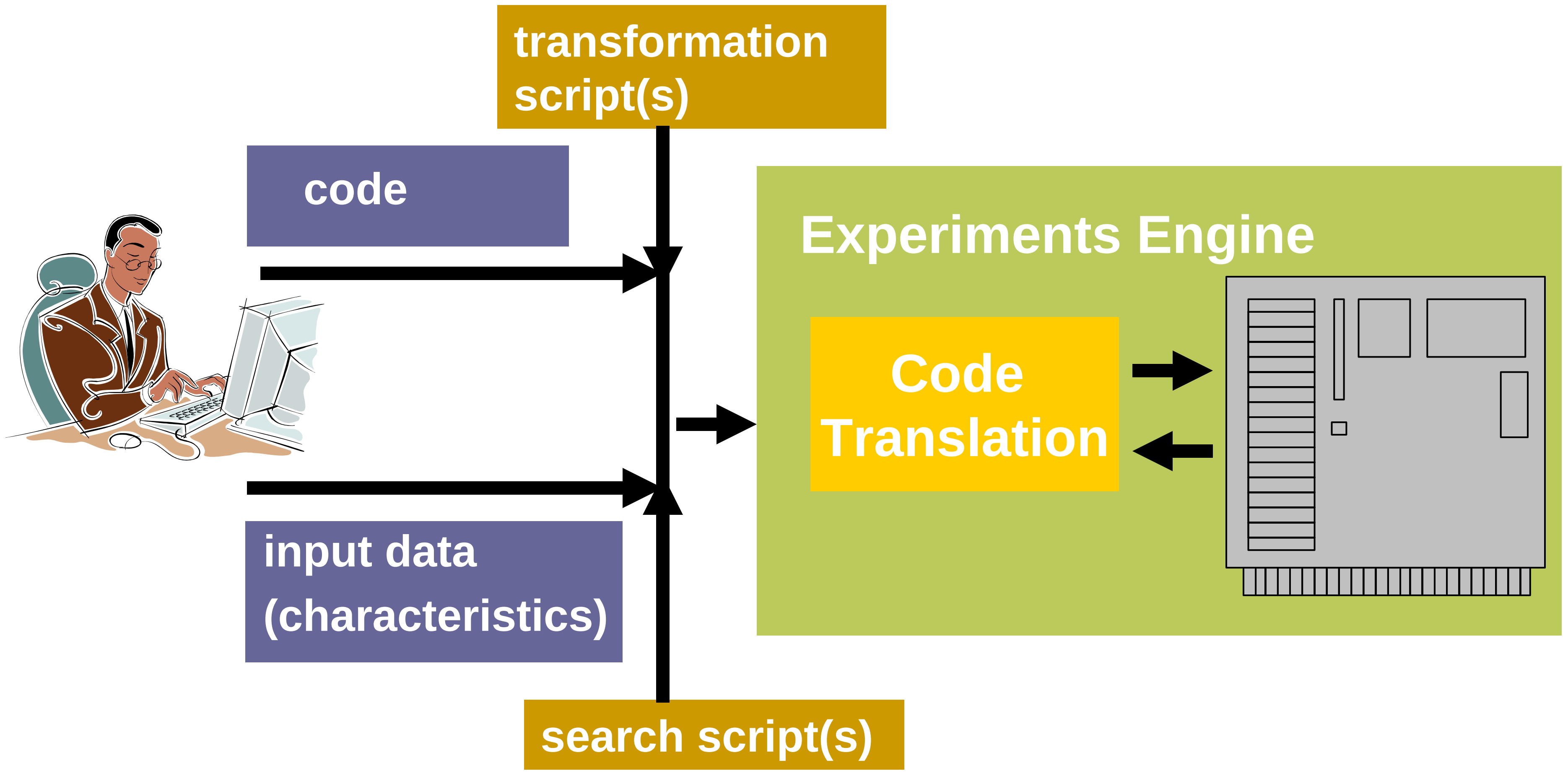
USC Viterbi School of Engineering

A New Kind of "Compiler"

Traditional view:

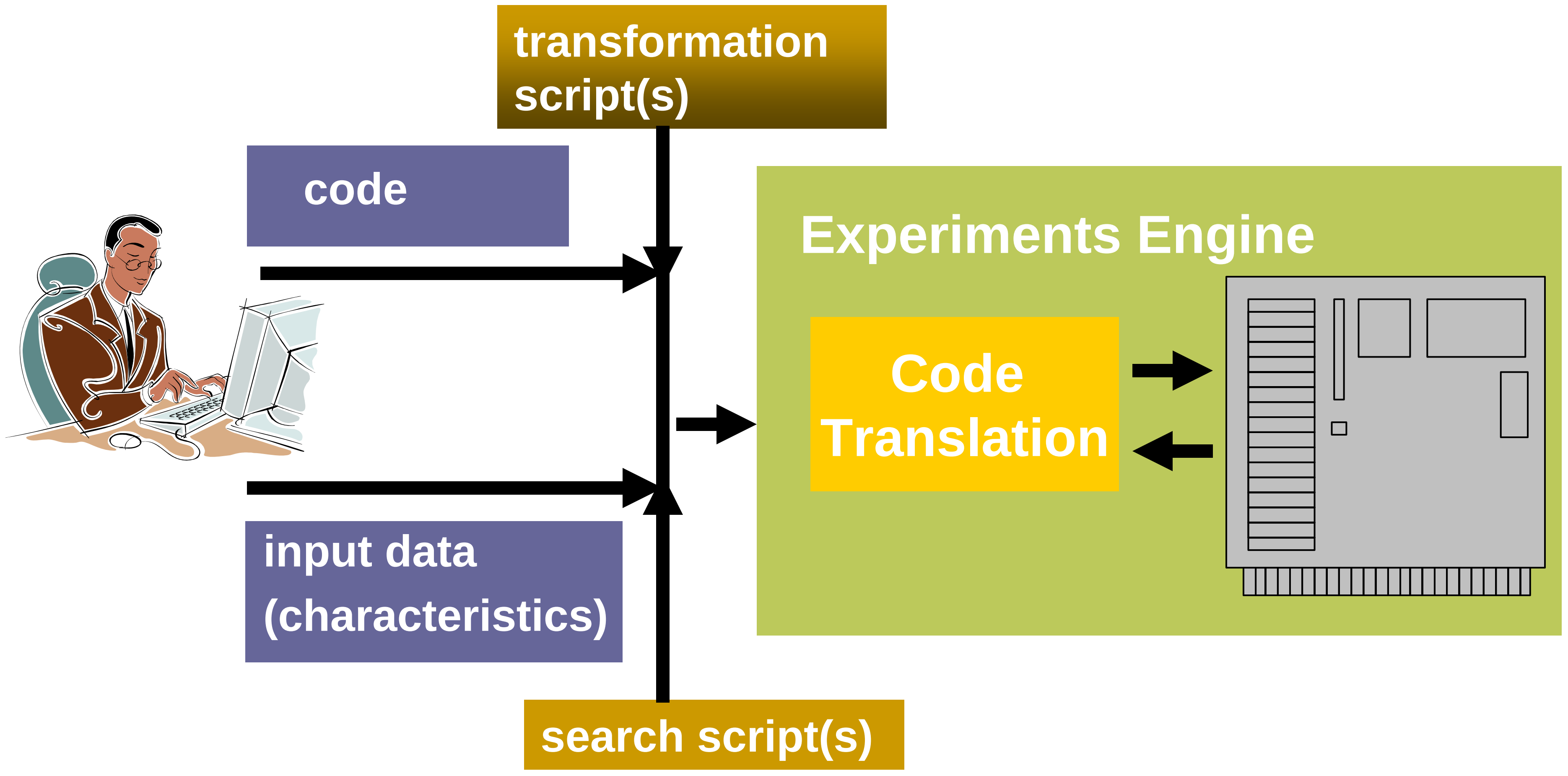


Performance Tuning "Compiler"



USC Viterbi School of Engineering

Performance Tuning “Compiler”



1. Programmer expresses application-level parameters and input data set properties. (ref. Active Harmony and Rose compiler)

USC Viterbi School of Engineering

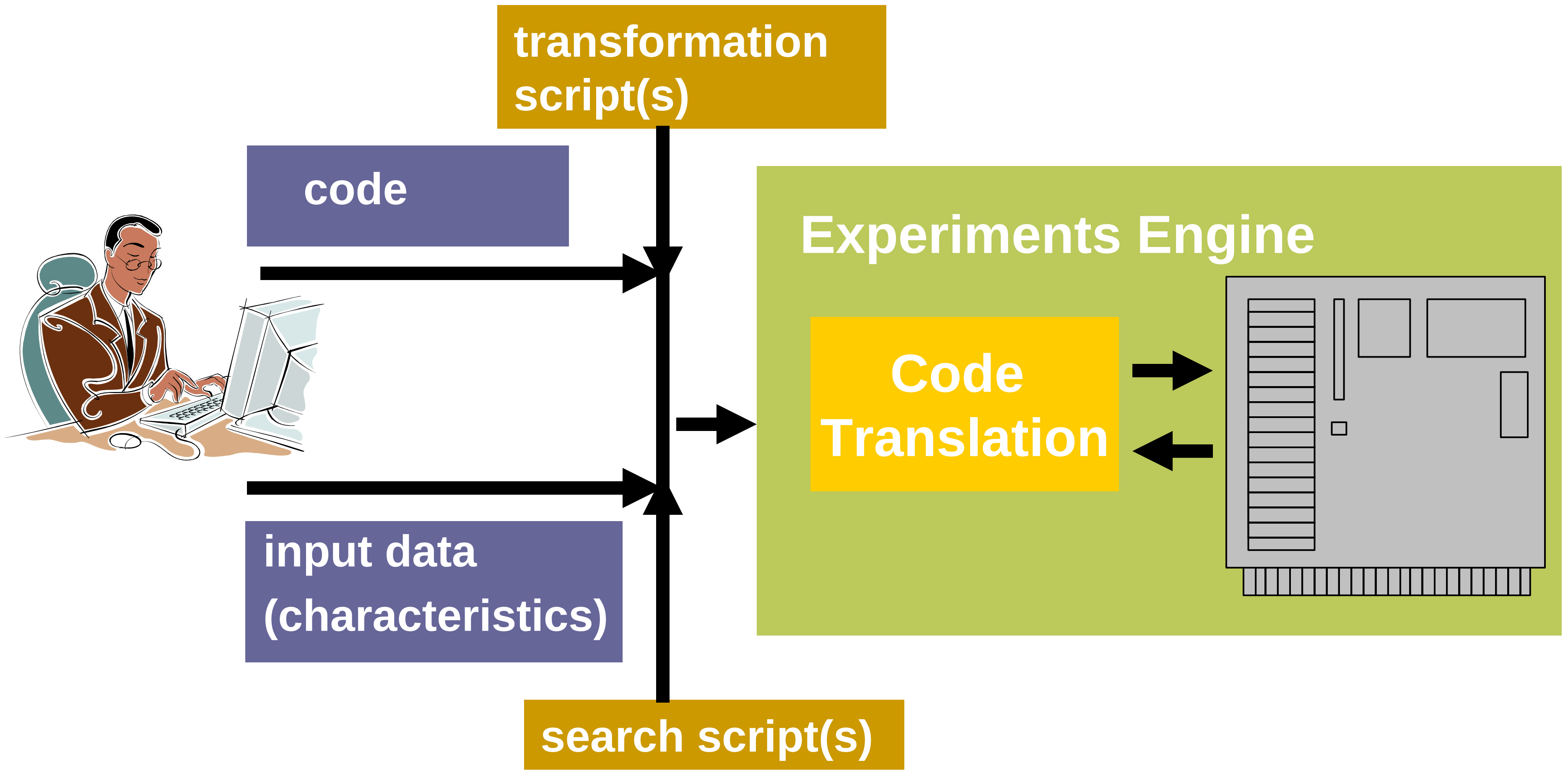
Scenario 1: Application-Level Parameters

- Programmer expresses parameters to be searched, input data set (*e.g.*, Visualization of MD Simulation)
- Tools automatically generate code and evaluate tradeoff space of application-level parameters

```
Parameter cellSize, range = 48:144, step 16

ncell = boxLength/cellSize

for i = 1, ncell
   /* perform computation */
```

```
Const cellSize = 48

ncell = boxLength/48

for i = 1, 48
   /* perform computation */
```

Performance Tuning “Compiler”



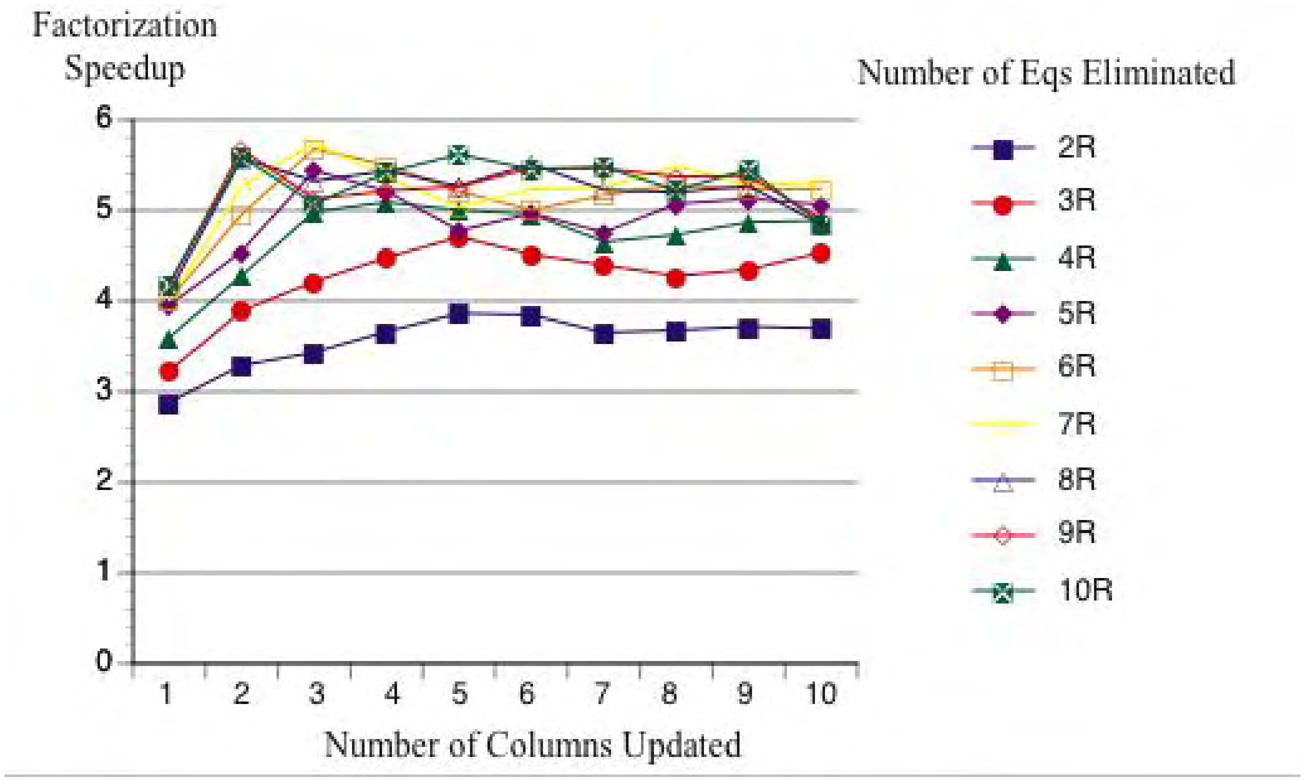
2. Application programmer interacts with compiler to guide optimization.

USC Viterbi School of Engineering

Scenario 2: Programmer-guided Transformations

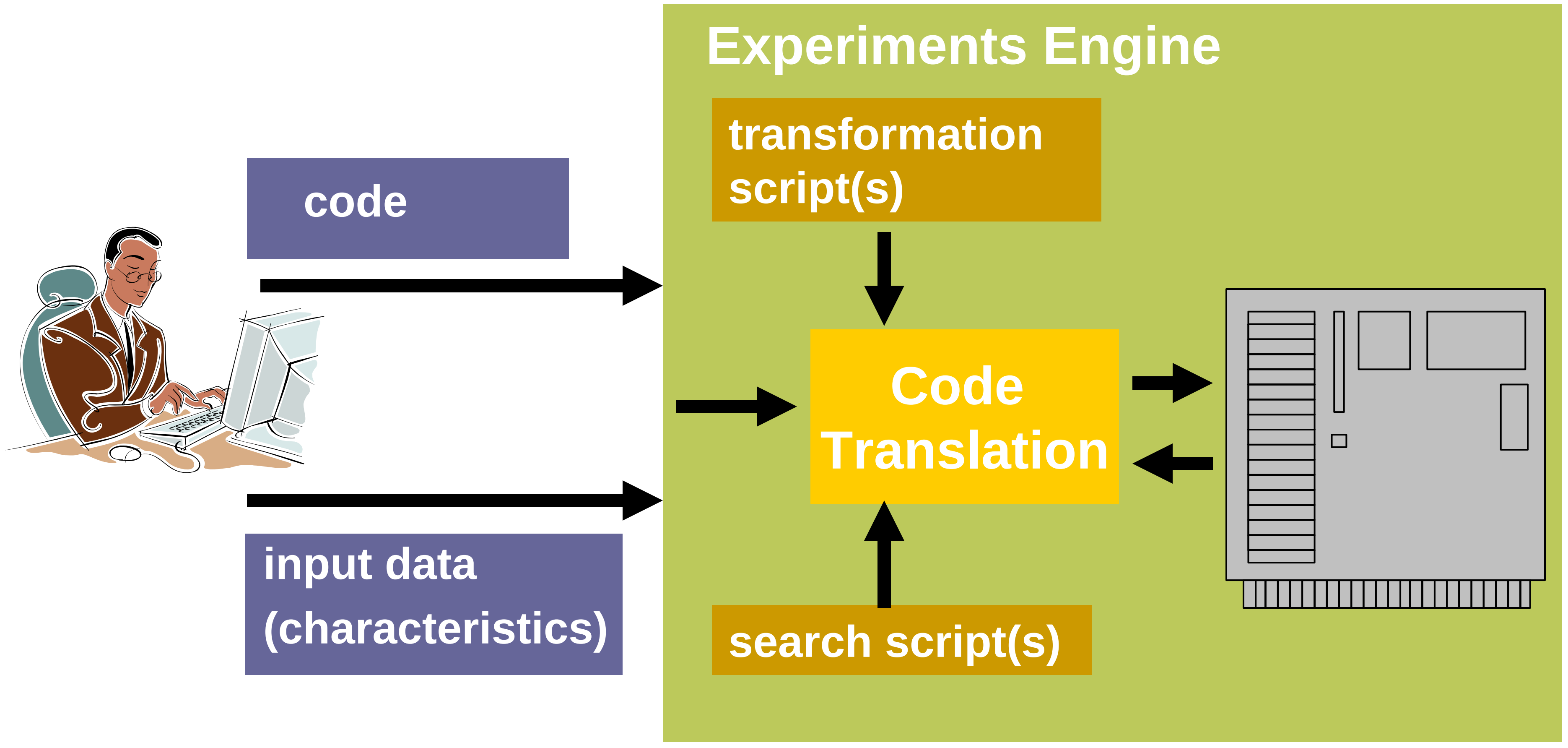
LS-DYNA Solver Performance Results

- Application programmer has written code variants for every possible unroll factor of two innermost loops
- Straightforward for compiler to generate this code and test for best version



Empirical Optimization for a Sparse Linear Solver: A Case Study, Y. Lee, P. Diniz, M. Hall and R. Lucas. *International Journal of Parallel Programming*, vol. 33, 2005.`

Performance Tuning "Compiler"



3. Compiler performs automatic performance tuning.

Scenario 3 (Simple): Expected Input Data Set Sizes are Known

- Application programmer can express to compiler expected data set size (MADNESS, NEK5K)
- Optimizations tuned for this (*e.g.*, loop unrolling for inner loop)

Input size **n, range = 2:6, step 2**

```
for i = 1, n
  s
```

```
Switch (n)
  case 2:
     s1; s2;
  case 4:
     s1; s2; s3; s4
  case 6:
     s1; s2; s3; s4; s5; s6
  default:
    for i = 1, n
         s
```

USC Viterbi
School of Engineering

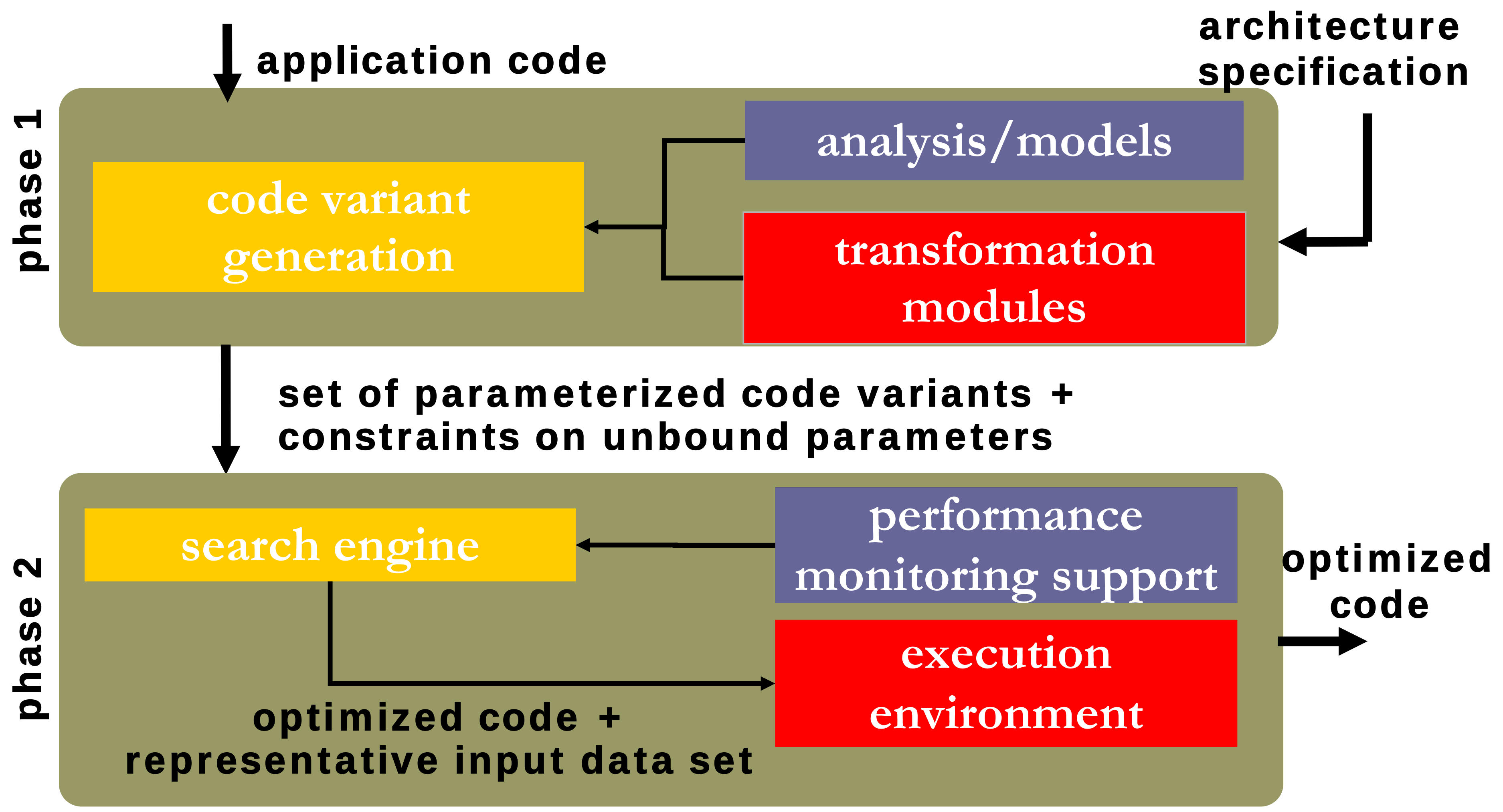
Model-guided empirical optimization

- **Model-guided optimization**
  - Static models of architecture, profitability
  - Models can get you pretty far [Yotov et al.,Qasem&Kennedy]
- **Empirical optimization**
  - Empirical data guide optimization decisions
  - ATLAS, PhiPAC, FFTW, SPIRAL etc.
- **Exploit complementary strengths of both approaches**
  - Compiler models prune unprofitable solutions
  - Empirical data provide accurate measure of optimization

Goal: Hand-tuned levels of performance from compiler-generated code for loop-based computation that is portable to new architectures.

USC Viterbi School of Engineering

Automatic Performance Tuning (Model-Guided Empirical Optimization)


application code
architecture specification
phase 1
analysis/models
code variant generation
transformation modules
set of parameterized code variants + constraints on unbound parameters
phase 2
search engine
performance monitoring support
execution environment
optimized code
optimized code + representative input data set

USC Viterbi School of Engineering

Transformation Framework

- Uniform polyhedral representation of transformations [Kelly][Lim&Lam][Cohen et al.]
- Direct mapping from transformation representation to generated code
- Mostly independent of compiler infrastructure

➔Straightforward to name alternative code variants and generate code, useful for search

Our contribution:

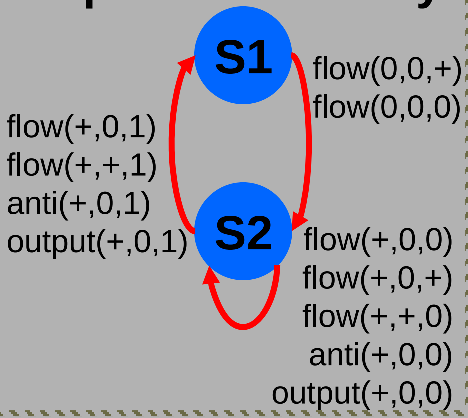
- Focus on high-level transformations robust for imperfect loop nests, high-quality code generation

```
do k=1,n-1
  do i=k+1,n
    a(i,k) = a(i,k)/a(k,k)
  do i=k+1,n
    do j=k+1,n
      a(i,j)=a(i,j)-a(i,k)*a(k,i)
```

```
foreach memory hierarchy level M
  select unmarked data structure D and loop L
       s.t. D has maximum reuse, carried by L
  if (level == register)
    make L innermost and unroll L
  else {
    permute & ti                    imension
    generate co
  }
  determine cons                    M
   (register/cach                   )
  mark D
```

```
permute([0,1,2])
tile(1,5,64,1)
split(1,3,[d3<=d1-2])
permute(2,[1,3,7,5])
permute(1,[1,5,7,3])
split(1,3,[d3>=d1-1])
tile(3,3,32,3)
split(3,5,[d9<=d3-1])
tile(3,9,32,5)
datacopy(3,7,2,1)
datacopy(3,7,3)
unroll(3,9,4)
tile(1,7,32,3)
tile(1,5,32,5)
datacopy(1,7,2,1)
datacopy(1,7,3,1)
unroll(1,9,4)
```

dependence analys



reuse analysis

register model

cache model

...

analysis and models

transformations

- original iteration space

s1 = {[k,i,j]: 1<=k<=n-1 ^ k+1<=i<=n ^ j=k+1}
s2 = {[k,i,j]: 1<=k<=n-1 ^ k+1<=i<=n ^ k+1<=j<=n}

- permute loops k and j

t1 := { [k,i,j] -> [0, j, 0, i, 0, k, 0] }
t2 := { [k,i,j] -> [0, j, 0, i, 1, k, 0] }

- tile loops

t1 := { [k,i,j] -> [0, jj, 0, kk, 0, j, 0, i, 0, k, 0] :
jj=2+16β && kk = 1+128α && i-15, 2 <= ii <=i && kk-127, 1 <= kk <= k}
t2 := { [k,i,j] -> [0, jj, 0, kk, 0, j, 0, i, 1, k, 0] :
jj=2+16β && kk = 1+128α && i-15, 2 <= ii <=i && kk-127, 1 <= kk <= k}

Transformed Code for LU (Automatically Generated)

```
REAL*8 P1(32,32),P2(32,64),P3(32,32),P4(32,64)
OVER1=0
OVER2=0
DO T2=2,N,64
  IF (66<=T2)
    DO T4=2,T2-32,32
      DO T6=1,T4-1,32
        DO T8=T6,MIN(T4-1,T6+31)
          DO T10=T4,MIN(T2-2,T4+31)
            P1(T8-T6+1,T10-T4+1)=A(T10,T8)
        DO T8=T2,MIN(T2+63,N)
          DO T10=T6,MIN(T6+31,T4-1)
            P2(T10-T6+1,T8-T2+1)=A(T10,T8)
        DO T8=T4,MIN(T2-2,T4+31)
          OVER1=MOD(-1+N,4)
          DO T10=T2,MIN(N-OVER1,T2+60),4
            DO T12=T6,MIN(T6+31,T4-1)
              A(T8,T10)=A(T8,T10)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10-T2+1)
              A(T8,T10+1)=A(T8,T10+1)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+1-T2+1)
              A(T8,T10+2)=A(T8,T10+2)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+2-T2+1)
              A(T8,T10+3)=A(T8,T10+3)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+3-T2+1)
          DO T10=MAX(N-OVER1+1,T2),MIN(T2+63,N)
            DO T12=T6,MIN(T4-1,T6+31)
              A(T8,T10)=A(T8,T10)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10-T2+1)
      DO T6=T4+1,MIN(T4+31,T2-2)
        DO T8=T2,MIN(N,T2+63)
          DO T10=T4,T6-1
            A(T6,T8)=A(T6,T8)-A(T6,T10)*A(T10,T8)
```

TRSM

data copy

unroll by 4

unroll cleanup

USC Viterbi School of Engineering

Transformed Code for LU (Cont.)

GEMM

```
IF (66<=T2)
  DO T4=1,T2-33,32
    DO T6=T2-1,N,32
      DO T8=T4,T4+31
        DO T10=T6,MIN(N,T6+31)
          P3(T8-T4+1,T10-T6+1)=A(T10,T8)
      DO T8=T2,MIN(T2+63,N)
        DO T10=T4,T4+31
          P4(T10-T4+1,T8-T2+1)=A(T10,T8)
      DO T8=T6,MIN(T6+31,N)
        OVER2=MOD(-1+N,4)
        DO T10=T2,MIN(N-OVER2,T2+60),4
          DO T12=T4,T4+31
            A(T8,T10)=A(T8,T10)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10-T2+1)
            A(T8,T10+1)=A(T8,T10+1)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10+1-T2+1)
            A(T8,T10+2)=A(T8,T10+2)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10+2-T2+1)
            A(T8,T10+3)=A(T8,T10+3)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10+3-T2+1)
        DO T10=MAX(T2,N-OVER2+1),MIN(T2+63,N)
          DO T12=T4,T4+31
            A(T8,T10)=A(T8,T10)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10-T2+1)
```

Mini-LU

```
DO T4=T2-1,MIN(N-1,T2+62)
  DO T8=T4+1,N
    A(T8,T4)=A(T8,T4)/A(T4,T4)
  DO T6=T4+1,MIN(T2+63,N)
    DO T8=T4+1,N
      A(T8,T6)=A(T8,T6)-A(T8,T4)*A(T4,T6)
```

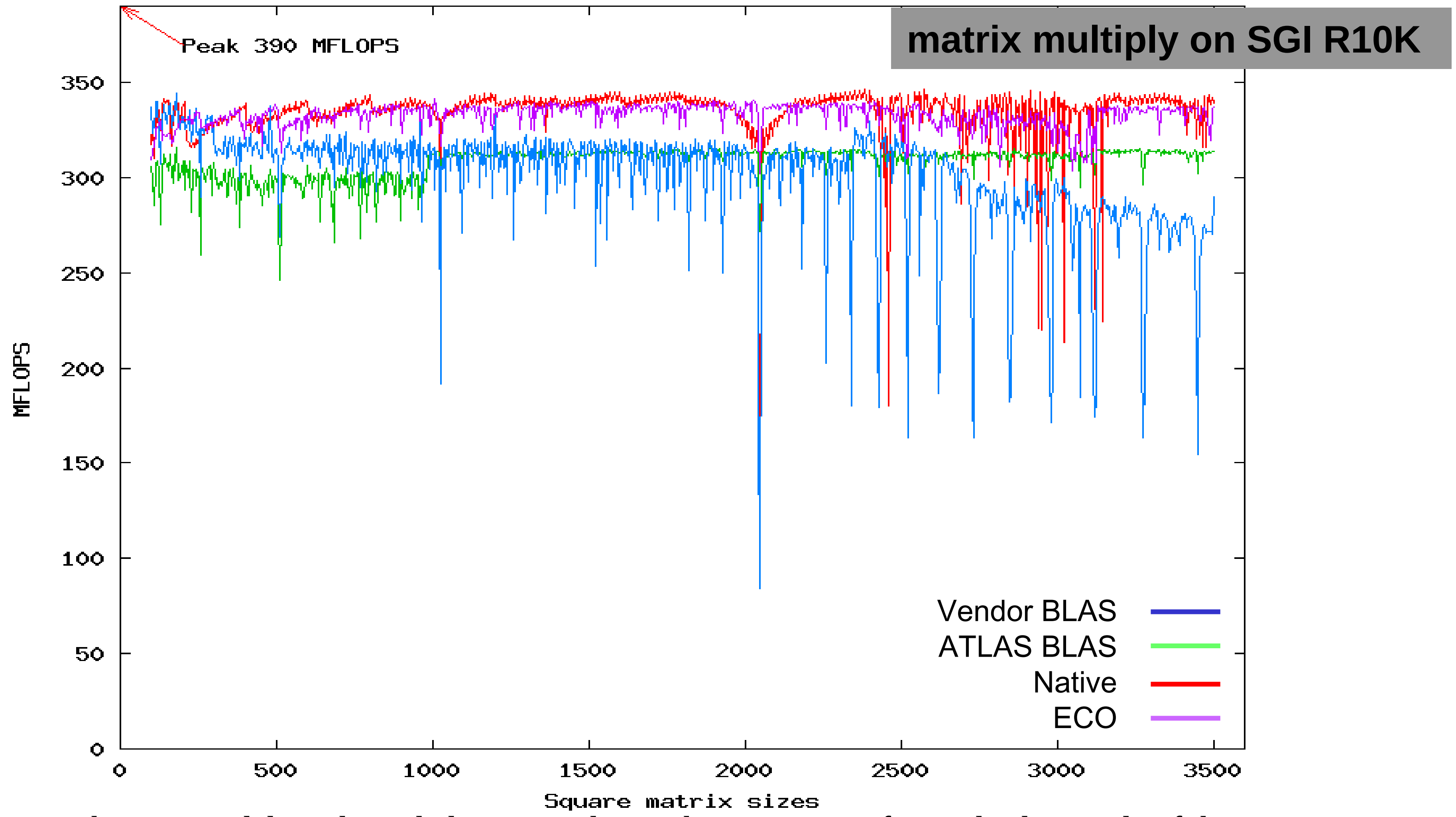
data copy

unroll by 4

unroll cleanup

USC Viterbi School of Engineering

Matrix Multiply: Comparison with ATLAS, vendor BLAS and native compiler



Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy, C. Chen, J. Chame and M. Hall. *Code Generation and Optimization*, March, 2005.

USC Viterbi School of Engineering

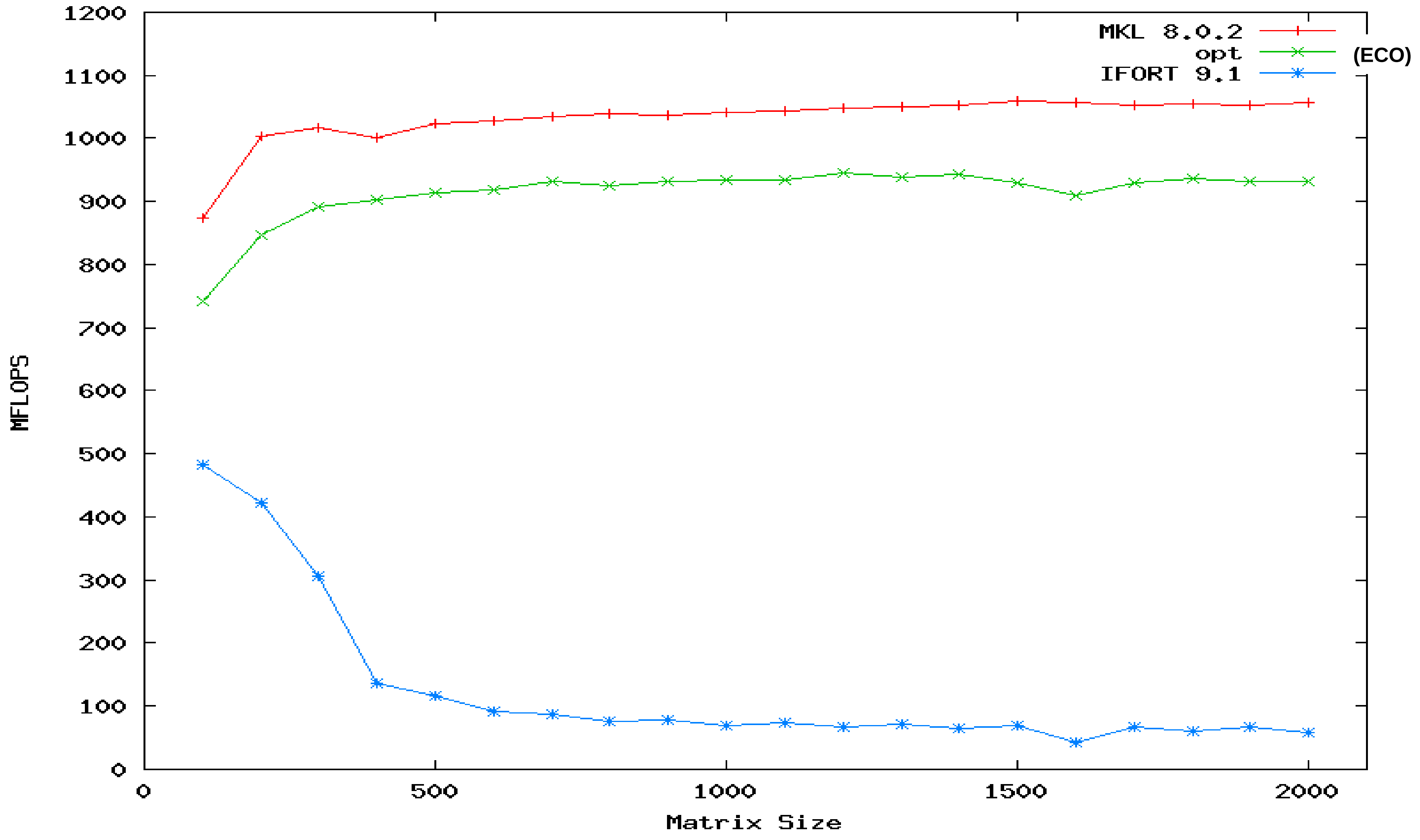
Pentium M: Combined Locality + SIMD Compiler

```
do i
  do j
    do k
        c(i,j) = c(i,j) + a(i,k)*b(k,j)
```

MM Version (3200x3200)	Automatically-Generated	Intel MKL	ATLAS	Intel ifort compiler
Performance (Single precision)	**2.957 Gflops**	2.895 Gflops	3.076 Gflops	0.692 Gflops

Model-Guided Empirical Optimization for Multimedia Extension Architectures: A Case Study, C. Chen, J. Shin, S. Kintali, J. Chame and M. Hall., *Performance Optimization of High-Level Languages*, March, 2007.

LU Decomposition: Performance on Pentium M



Model-Guided Empirical Optimization for Memory Hierarchy, C. Chen, PhD Dissertation, University of Southern California, Dept. of Computer Science, May, 2007.

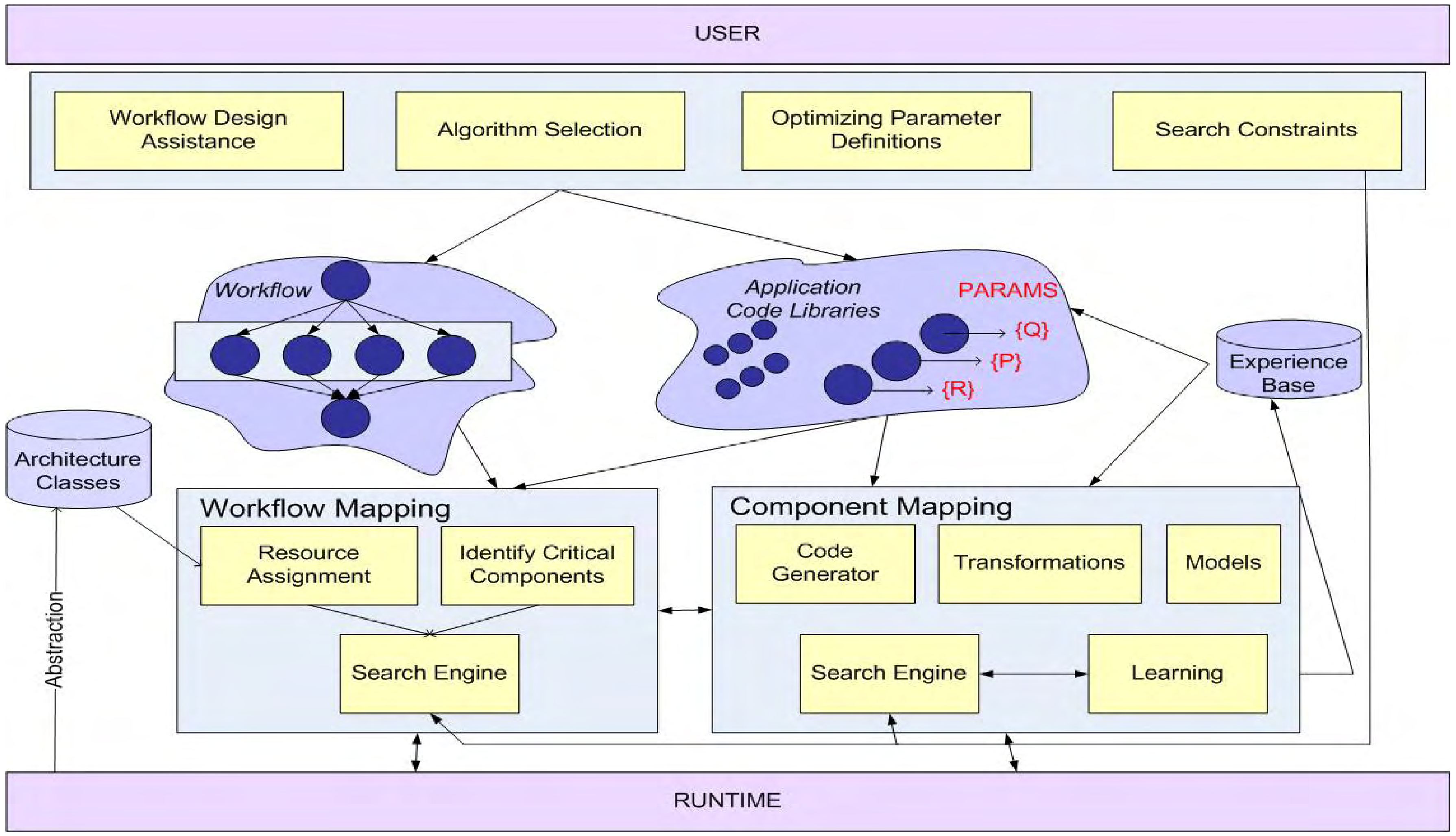
Implications on Real World

- Where **autotuners** can beat libraries
  - PERI: Auto-tuning of application code
  - Libraries used in unusual ways (e.g., MM on long, skinny matrices)
  - Composing library calls
- **Autotuners** can assist in building libraries
- Other ways compilers can make programmers more productive in tuning their code
  - Search for best values of application-level parameters
  - Apply user-directed code transformations
  - Tune for particular problem sizes

Where are we going?

- Multicore and porting to new architectures.
- Systematic approach to application composition and optimization.
  - Knowledge representation to compose applications from optimized components.
  - Efficient but systematic search techniques.
  - Start with distributed systems → move to heterogeneous chip architectures.
- Tools in the hands of users.

Systematic Composition and Optimization of Workflows

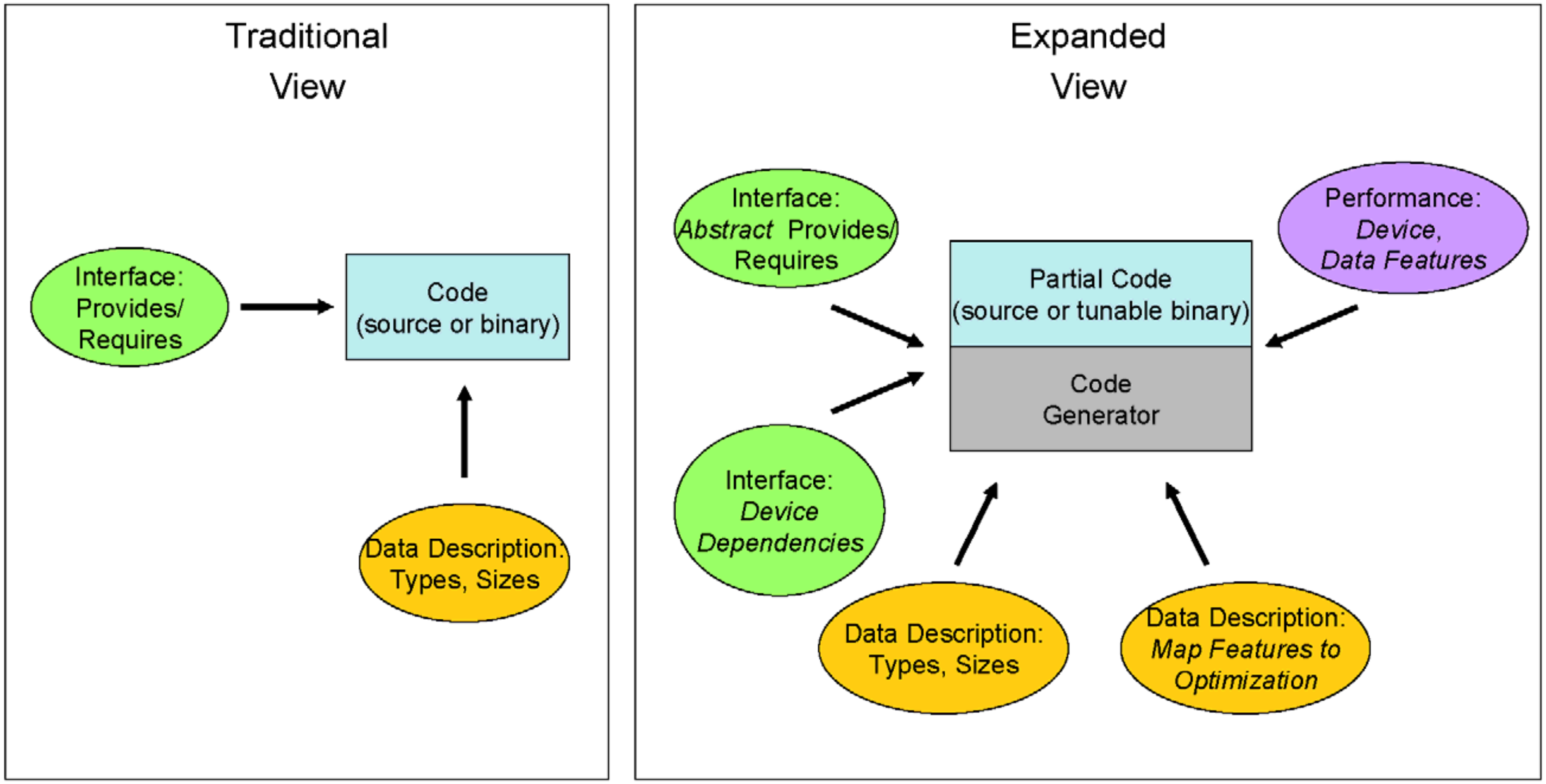


"Intelligent Optimization of Parallel and Distributed Applications," B. Bansal, U. Catalyurek, J. Chame, C. Chen, E. Deelman, Y. Gil, M. Hall, V. Kumar, T. Kurc, K. Lerman, A. Nakano, Y. Nelson, J. Saltz, A. Sharma, P. Vashishta, Workshop on Next Generation Software, March, 2007.

USC Viterbi School of Engineering

Components Optimized in Execution Context

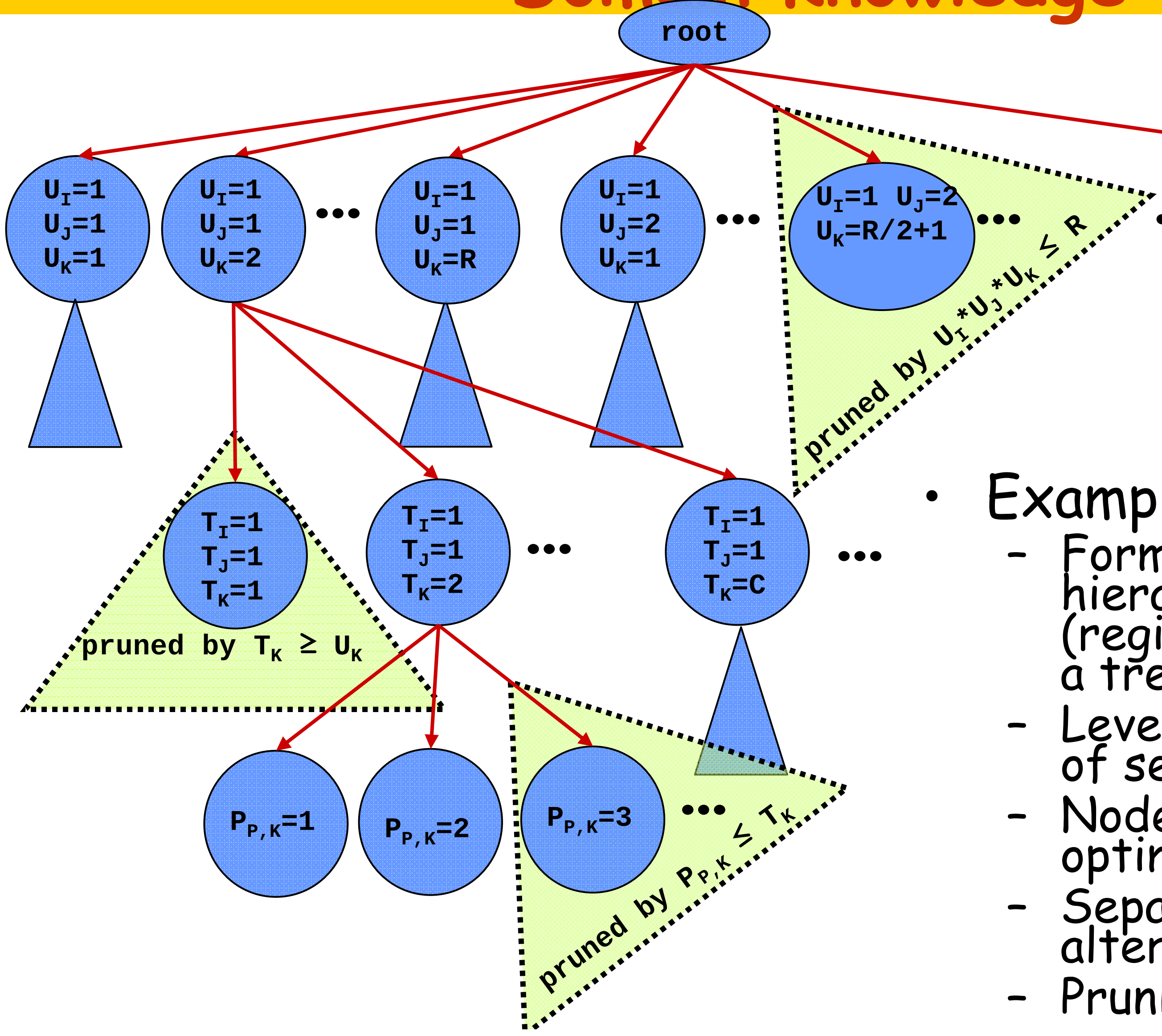
- (Heterogeneous) multi-core will lead us in a similar direction to distributed computing.
- Component technology will become critical.


Traditional View
Interface: Provides/ Requires
Code (source or binary)
Data Description: Types, Sizes
Expanded View
Interface: *Abstract* Provides/ Requires
Partial Code (source or tunable binary)
Code Generator
Performance: *Device, Data Features*
Interface: *Device Dependencies*
Data Description: Types, Sizes
Data Description: *Map Features to Optimization*

"Programming for Edge Computing: Using Cognitive Techniques to Manage Heterogeneous Resources," M. Hall, Y. Gil and R. Lucas, To appear, Proceedings of the IEEE, Feb. 2008.

Limiting Search with Domain Knowledge

Parameters:
unroll factors, tile sizes, prefetch distances



- Example
  - Formulate memory hierarchy optimization (registers, caches, TLB) as a tree search
  - Levels represent ordering of search
  - Nodes represent optimization parameters
  - Separate trees used for alternative components
  - Pruning, backtracking easy

"A Systematic Approach to Model-Guided Empirical Search for Memory Hierarchy Optimization," Chen, Chame, Hall, Lerman, LCPC 2005.

USC Viterbi School of Engineering

Concluding Remarks

- Three core technical ideas
  - **Compiler technology:** Modular compilers, systematic approach to optimization, empirical search, ***hand-tuned performance.***
  - **User Tools:** Access to transformation system, express parameters for automatic search, express expected problem size
  - **Systematic:** Compose optimized applications in context. Express/derive parameters for search