USC Viterbi
School of Engineering

Compiler-Assisted Performance Tuning

Mary Hall

July 10, 2007

* This work has been sponsored by the NSF NGS and CSR programs, Intel Corporation, and the DOE SciDAC as part of the Performance Engineering Research Institute (PERI).

Collaborators

- Compiler group:
  - Jacqueline Chame (research scientist)
  - Chun Chen (postdoctoral researcher)
  - Spundun Bhatt (programmer)
  - Yoonju 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

Goal for this Workshop

- Discuss role of compiler technology in various tuning efforts
  - Application programmer assistant
  - Library developer assistant (e.g., intelligent code generation and search)
  - Fully automatic tuning (next talk)
- Our interests
  - New applications
  - New architectures
  - Other collaborations

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

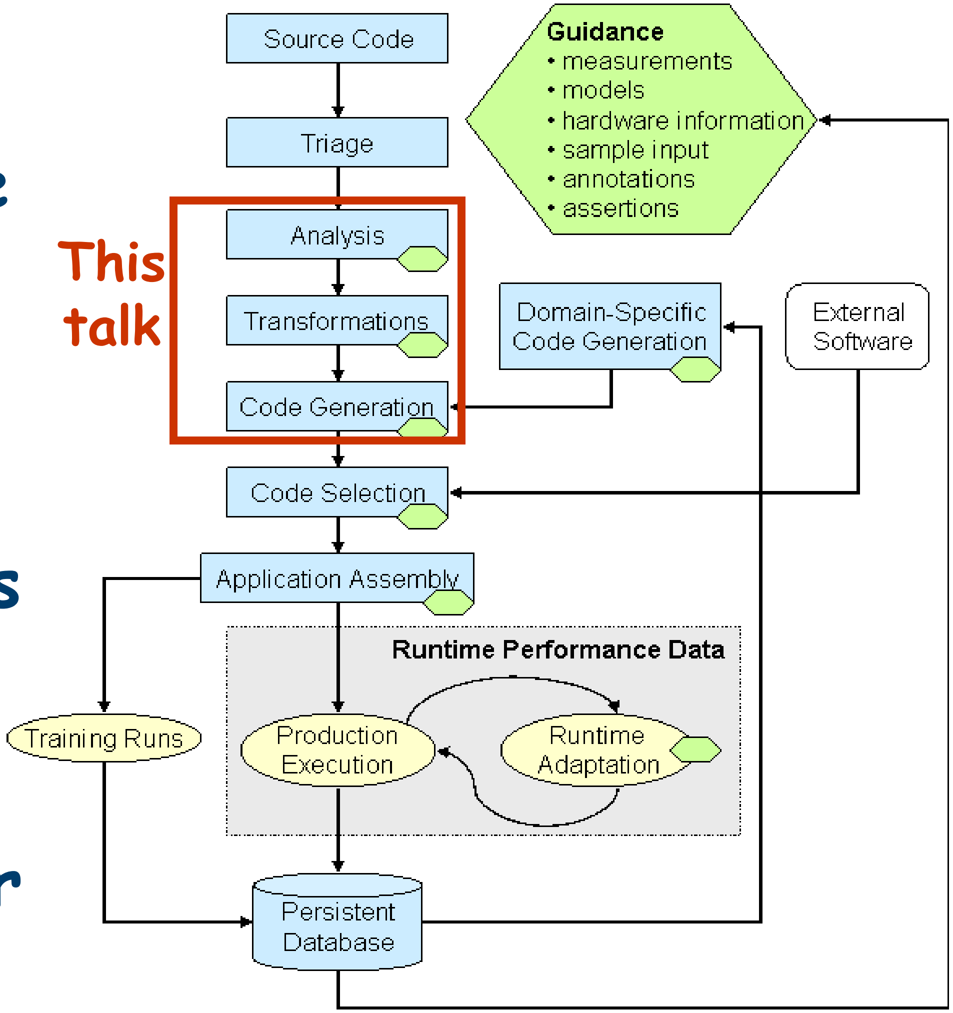
A Systematic, Principled Approach!

Outline

1. Motivation
2. Approach & potential of compiler-assisted tuning
   - New flexible and systematic compiler technology
   - Scenarios from application tuning
   - Automatic performance tuning
3. Overview of results (more in next talk)

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



This talk

Source Code

Triage

Analysis

Transformations

Code Generation

Code Selection

Application Assembly

Guidance
- measurements
- models
- hardware information
- sample input
- annotations
- assertions

Domain-Specific Code Generation

External Software

Runtime Performance Data

Training Runs

Production Execution

Runtime Adaptation

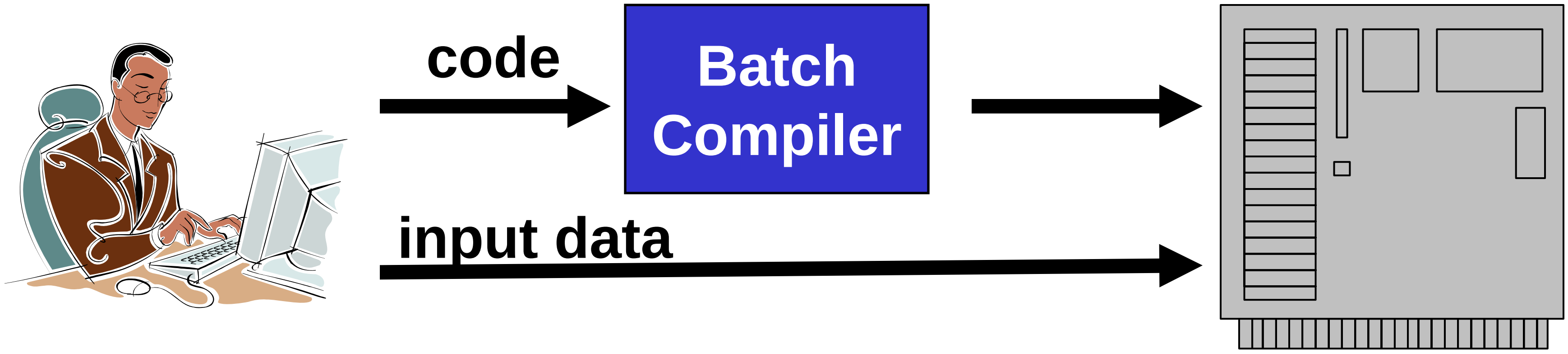
Persistent Database

Slide source: Bob Lucas and David Bailey

PERI automatic tuning framework

A New Kind of “Compiler”

Traditional view:

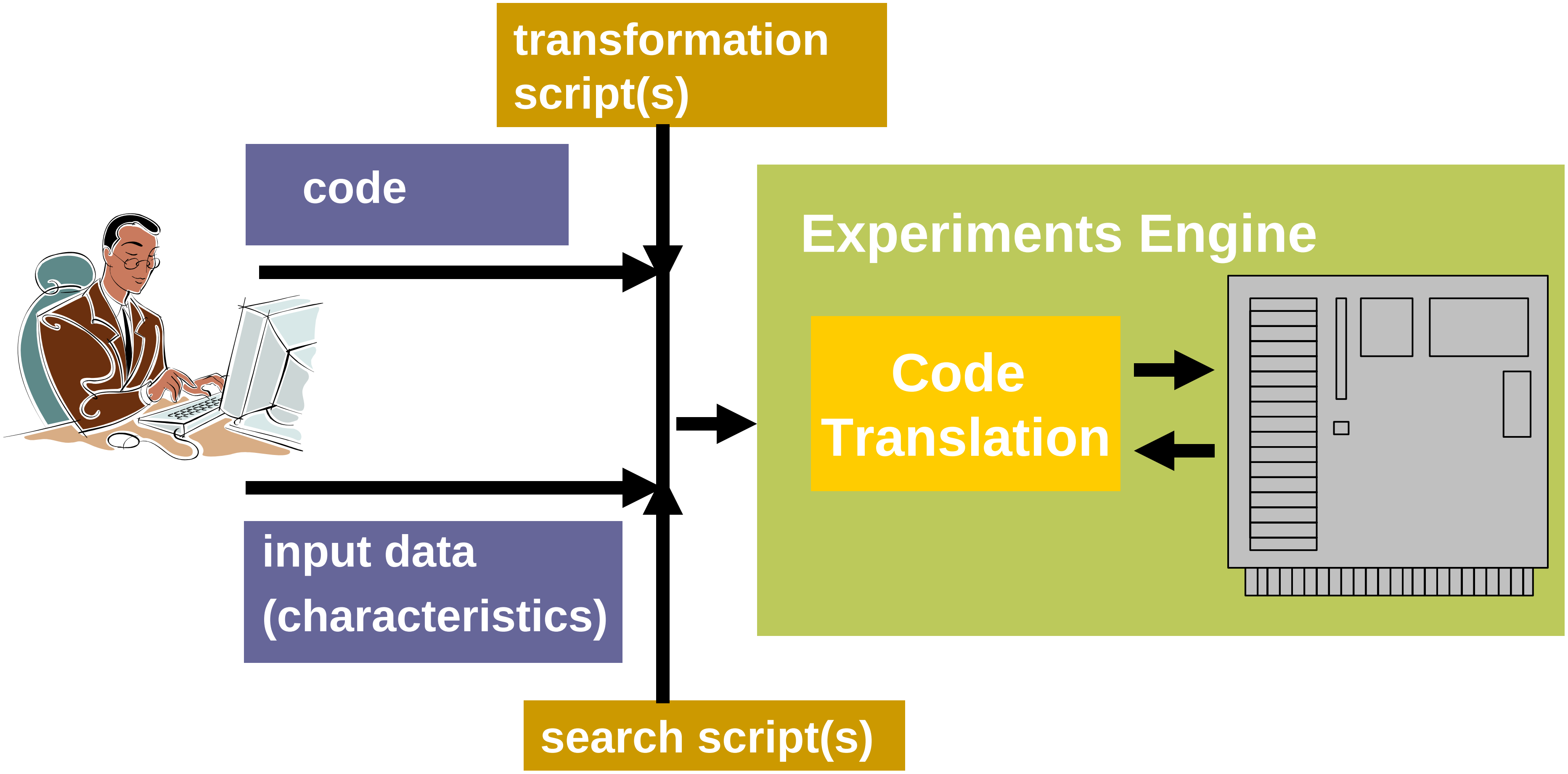


code

Batch Compiler

input data

Performance Tuning “Compiler”



transformation script(s)

code

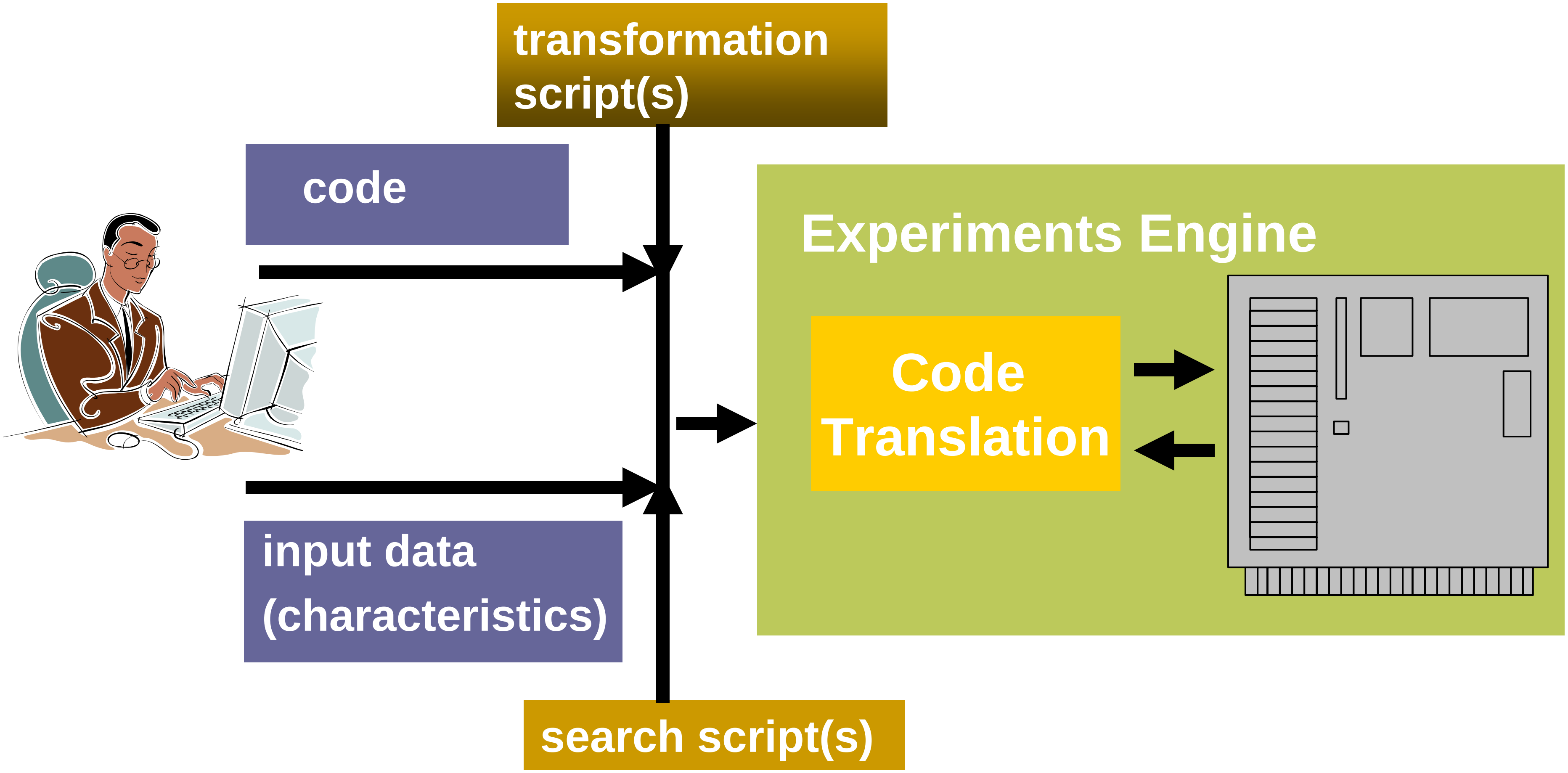
input data (characteristics)

search script(s)

Experiments Engine

Code Translation

Performance Tuning “Compiler”



transformation script(s)

code

Experiments Engine

Code Translation

input data (characteristics)

search script(s)

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

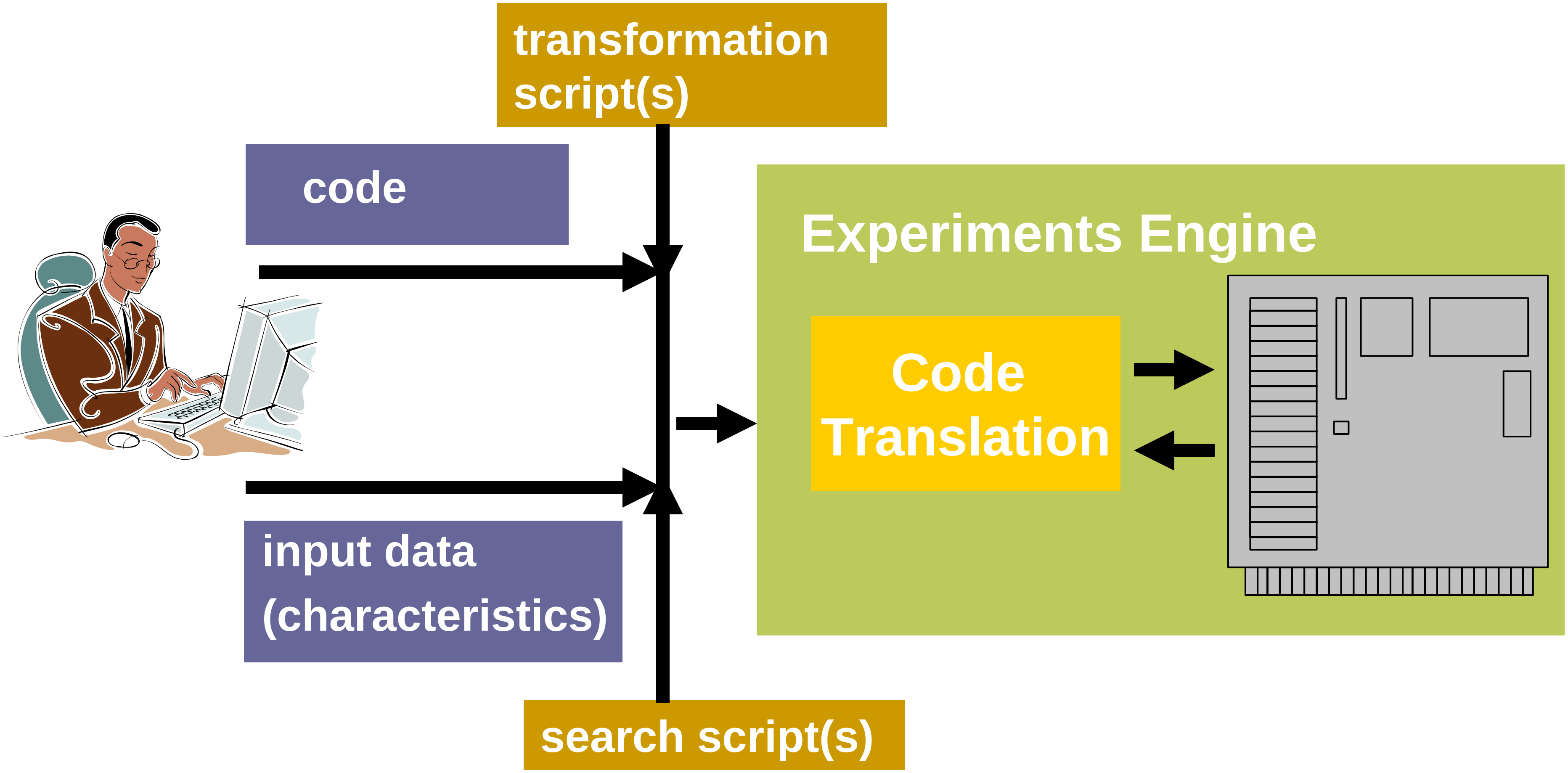
Scenario 1: Application-Level Parameters

- 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”



transformation script(s)

code

Experiments Engine

Code Translation

input data (characteristics)

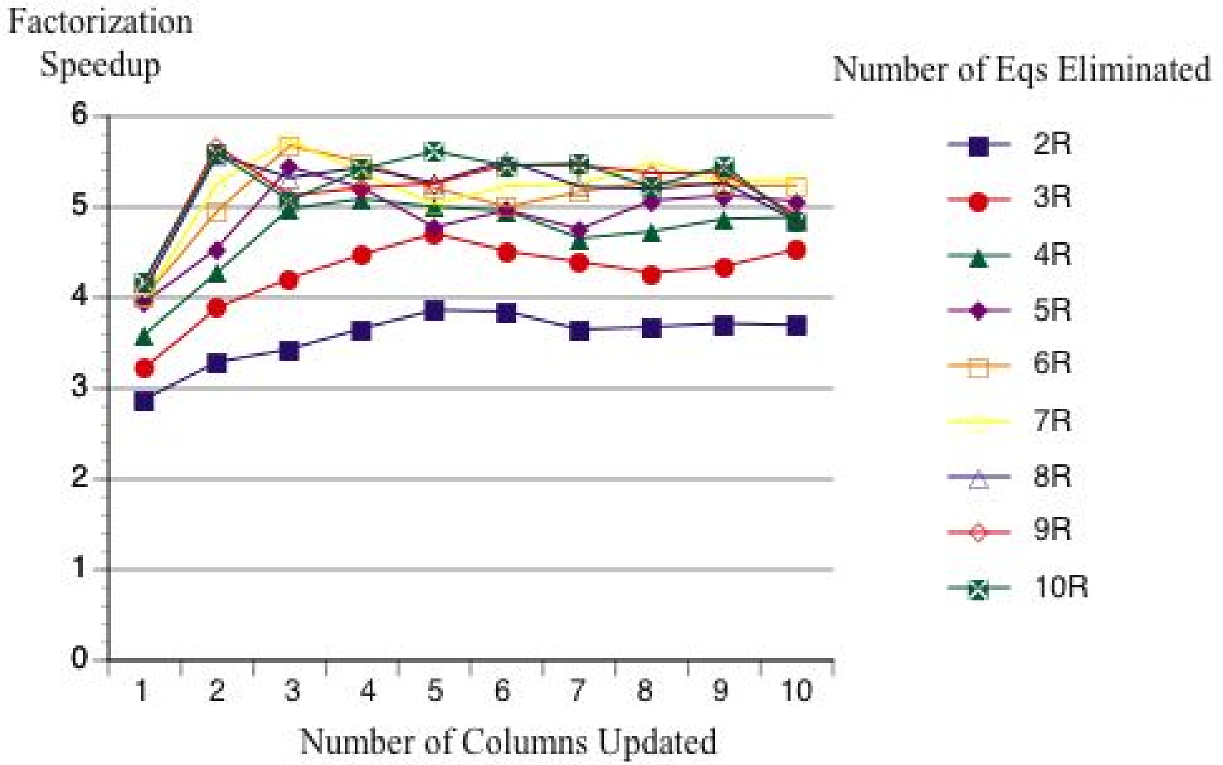
search script(s)

2. Application programmer interacts with compiler to guide optimization.

Scenario 2: Programmer-guided Transformations

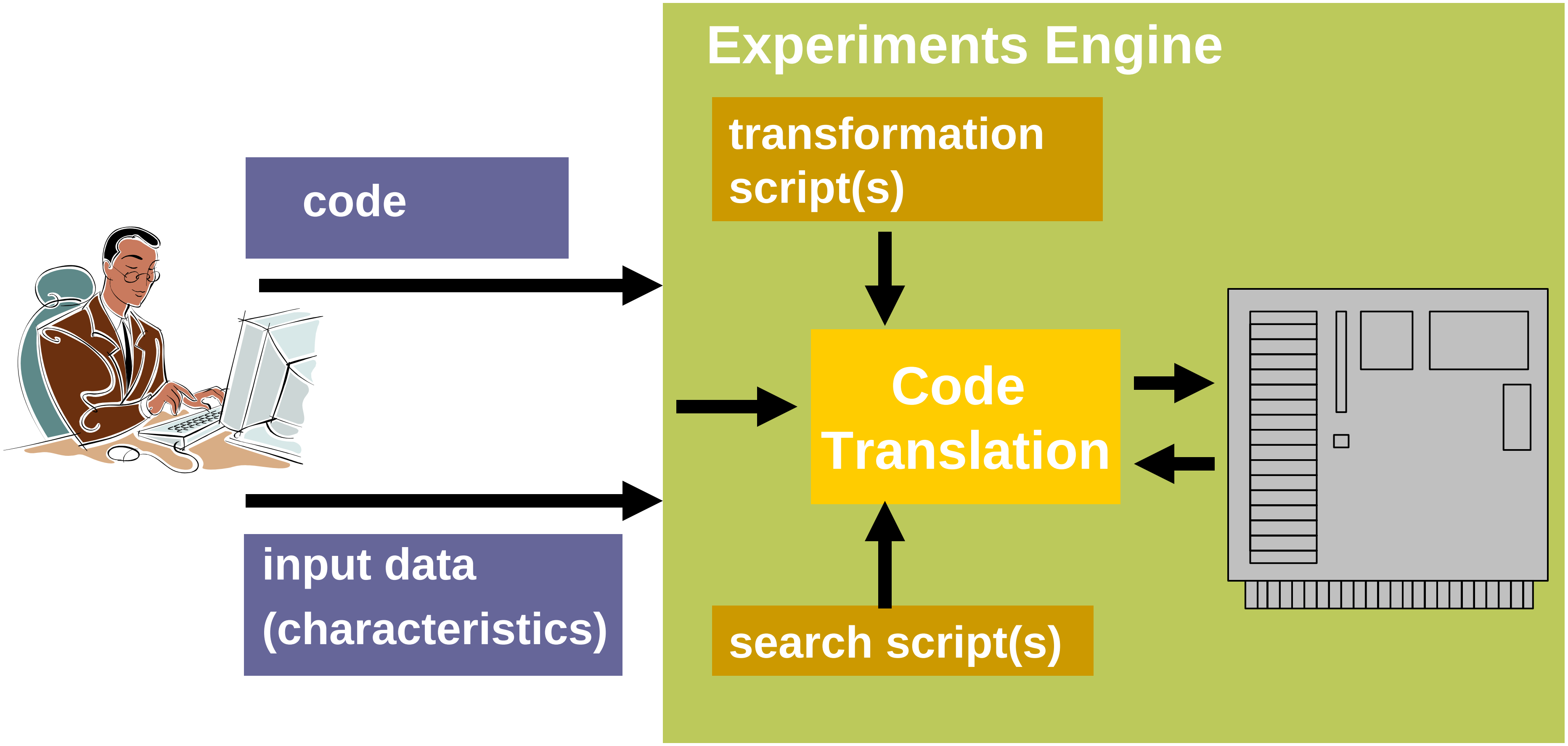
LS-DYNA Solver Performance Results

- 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”



Experiments Engine

code

transformation script(s)

Code Translation

input data (characteristics)

search script(s)

3. Compiler performs automatic performance tuning.

USC Viterbi
School of Engineering

High-Level Concept: Exploit what compilers do well

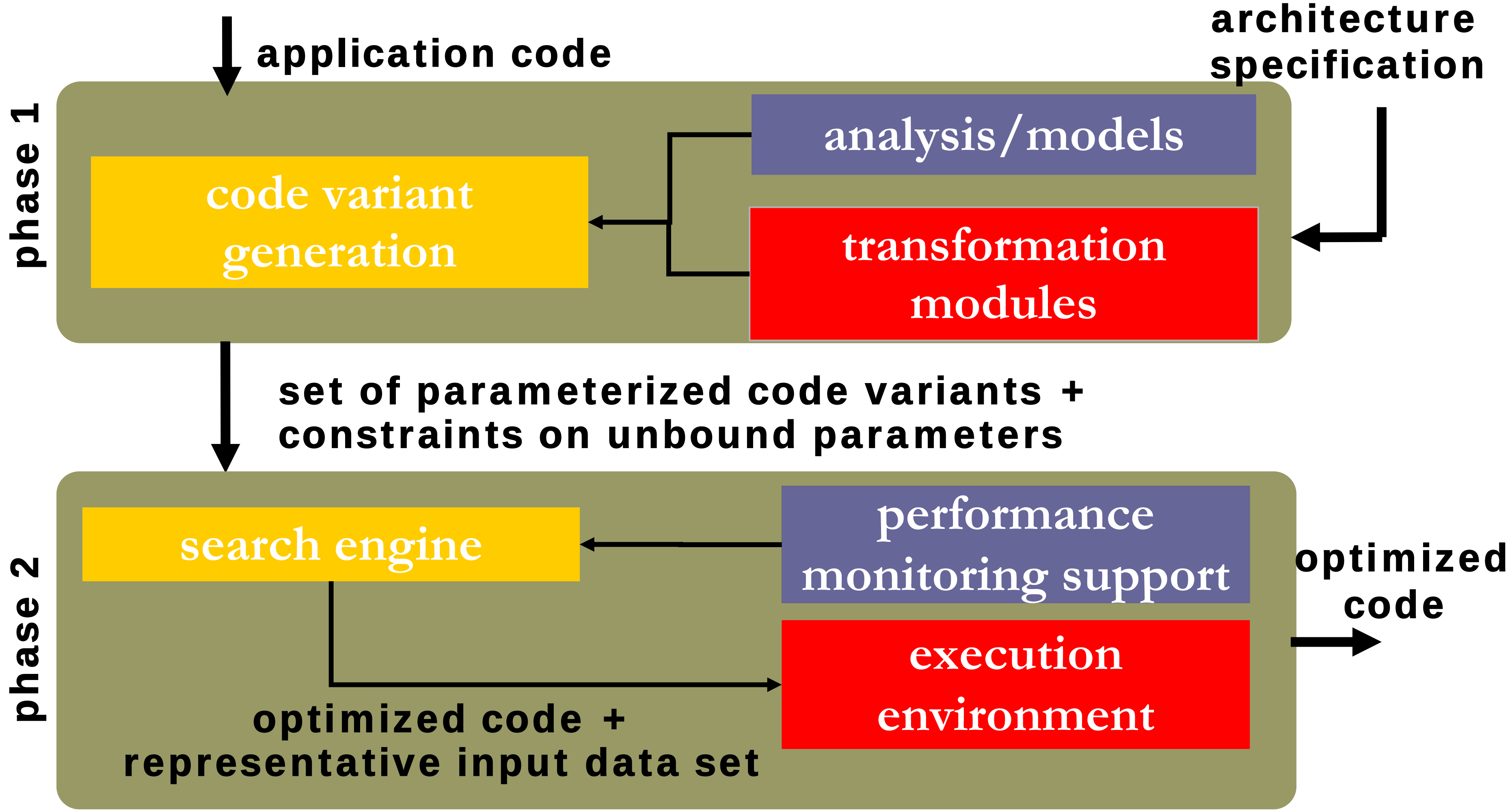
- Complex translation and transformation (rewriting rules)
- Domain knowledge of optimizations and optimization impact
- Analyze code to derive “features”
- Source-to-source
  - Rely on investment in backend native compilers to achieve ILP

Model-guided empirical optimization (our autotuning)

- **Model-guided optimization**
  - Static models of architecture, profitability
- **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 impact

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

Automatic Performance Tuning (Model-Guided Empirical Optimization)



application code

architecture specification

phase 1

code variant generation

analysis/models

transformation modules

set of parameterized code variants + constraints on unbound parameters

phase 2

search engine

performance monitoring support

execution environment

optimized code + representative input data set

optimized code

Transformation Framework

- Uniform representation of transformations
- Direct mapping from transformation representation to generated code
- Mostly independent of compiler infrastructure

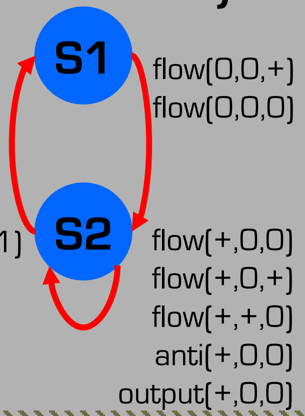
→Straightforward to name alternative code variants and generate code, useful for search

```
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 & til                          ension
     generate copy
  }
  determine const
    (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 analysis



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

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)
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)
```

Mini-LU

data copy

unroll by 4

unroll cleanup

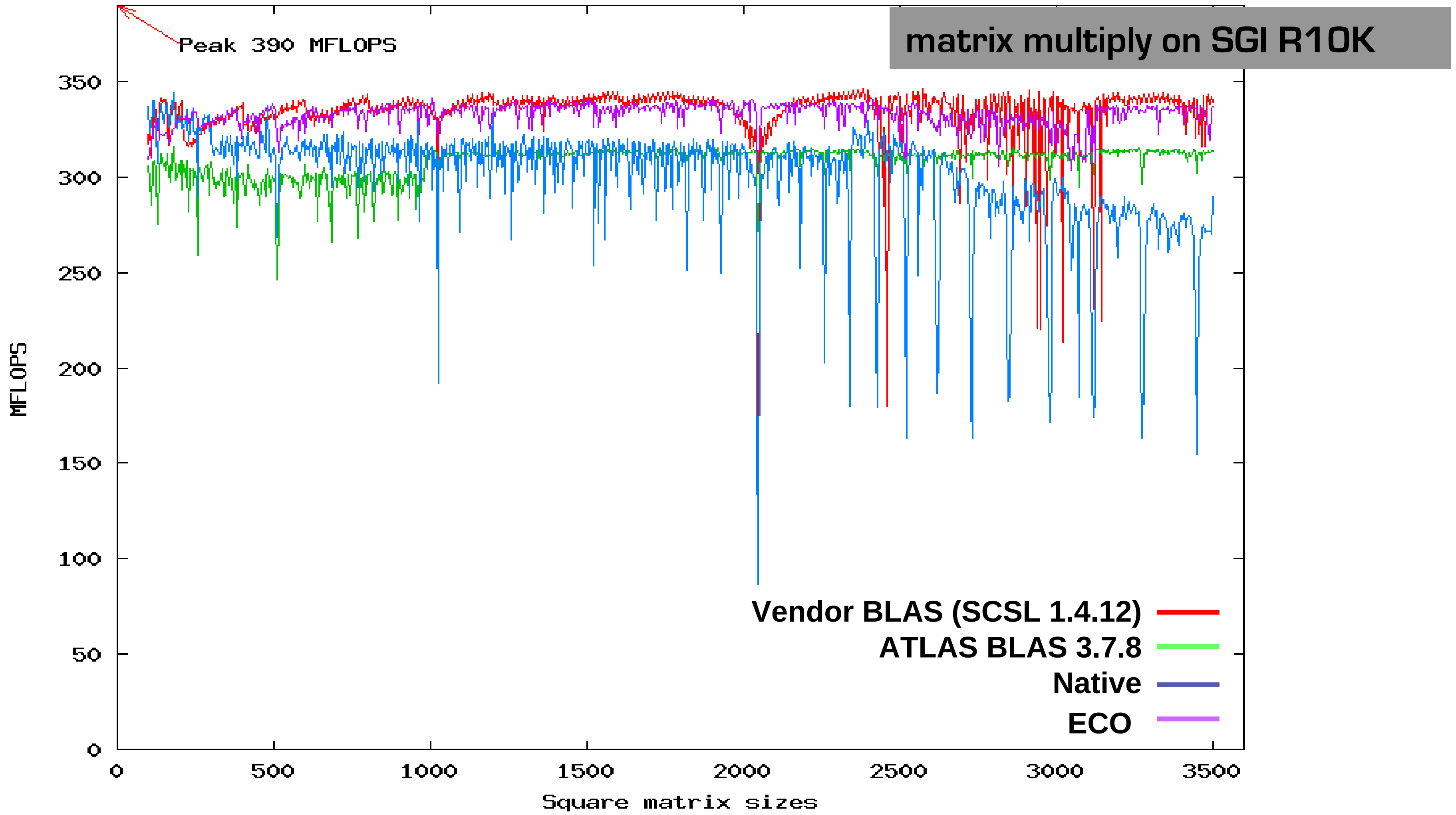
Search Space

- **Set of variants**
  - Different loop orders, copy yes or no, different loop splitting strategies, different prefetch strategies
  - Select variant with the best performance
- **Integer parameter values**
  - Unroll factors, tile sizes, prefetch distances
  - Each parameter has unique search properties
- **Constraints**
  - Limit unrolling amount by register capacity
  - Limit tiling parameters by cache/TLB capacity and set associativity

Comparison of Search Cost

Code	SGI R10K	Sun Ultrasparc IIe
MM (ATLAS)	35 min	14 min
MM (ECO)	8 min (60 pts)	6 min (44 pts)

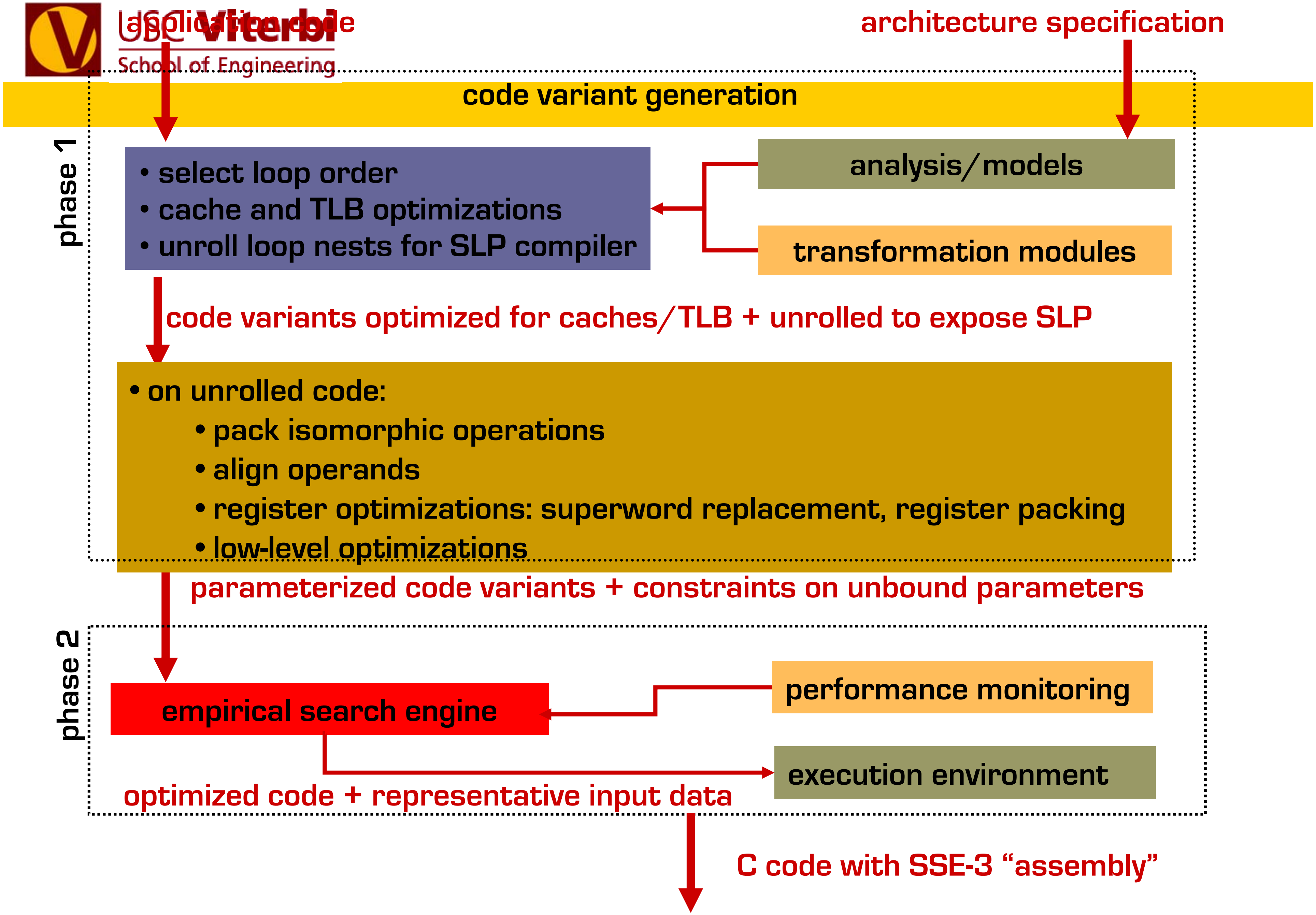
Matrix Multiply: Comparison with ATLAS, vendor BLAS and native compiler



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.

Combining with SIMD Optimizations

- Motivation
  - Multimedia extension architectures (SSE3, AltiVec, …)
  - Node processors in high-end systems (*e.g.,* Intel and Opteron clusters)
- Developed SLP compiler
  - Initial approach by Larsen and Amarasinghe (PLDI '00)
  - Locality optimizations for superword registers, control flow support and other extensions, Shin, Chame and Hall, PACT '02, MSP '02, JILP '03, MSP '04, CGO '05
- Impact
  - Code variants generated anticipating SLP optimizations
  - Requires close integration with backend (in our case) or more search



application code

architecture specification

code variant generation

phase 1

- select loop order
- cache and TLB optimizations
- unroll loop nests for SLP compiler

analysis/models

transformation modules

code variants optimized for caches/TLB + unrolled to expose SLP

- on unrolled code:
  - pack isomorphic operations
  - align operands
  - register optimizations: superword replacement, register packing
  - low-level optimizations

parameterized code variants + constraints on unbound parameters

phase 2

empirical search engine

performance monitoring

execution environment

optimized code + representative input data

C code with SSE-3 “assembly”

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 8.0.2	ATLAS 3.7.14	Intel ifort compiler v9.1
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.

Full Set of Experiments to Date

	SGI R10000	Pentium M	UltraSparc IIe	Pentium D	PowerPC AltiVec
Matrix-Matrix	✓	✓	✓		✓
Matrix-Vector	✓	✓		✓	
Matrix-Vector (Transpose)	✓	✓		✓	
Triangular Solve	✓	✓			
LU Factorization	✓	✓			
Jacobi	✓		✓		
MADNESS		✓			

Performance Summary

Subset of results

Architecture	kernel	opt	ATLAS	vendor
Pentium M	mv	1.47x	1.33x	1.00x
	mvT	1.47x	1.34x	0.99x
	mm	3.35x	3.39x	3.04x
	lu	11.44x	-	12.88x
SGI R10000	mv	1.22x	1.02x	1.20x
	mvT	1.03x	0.87x	0.99x
	mm	1.72x	1.58x	1.73x
	lu	2.75x	-	3.53x

Baseline performance: best native compiler optimizations

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

What next?

- Where compilers can beat libraries
  - PERI: Auto-tuning of application code
  - Libraries used in unusual ways (e.g., MM on long, skinny matrices)
  - Composing library calls
- 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

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:** Express/derive parameters for search
- Lessons for other SciDAC projects
  - **PERI Outreach:** Working with applications informs tool development

Non-trivial Performance Tradeoffs

Version	TI	TJ	TK	Pref?	Loads	L1 misses	L2 misses	TLB misses	Cycles
mm1		32	64	N	4.20B	**142M**	21.6M	**231K**	10.2B
mm2		16	128	N	4.10B	210M	35.3M	105M	12.5B
mm3	8	256	256	N	**4.08B**	319M	**7.19M**	4.42M	9.70B
mm4	16	512	128	N	4.11B	182M	8.01M	2.78M	9.47B
mm5	16	512	128	Y	5.12B	188M	8.04M	2.78M	**9.18B**
j1				N	**25.5M**	8.78M	1.65M	7.52K	181M
j2				Y	34.0M	8.82M	1.64M	**7.49K**	137M
j3		16	8	N	28.0M	**6.10M**	1.32M	18.3K	155M
j4		16	8	Y	40.8M	7.62M	1.32M	18.6K	125M
j5	300	16		N	**25.5M**	8.79M	**1.18M**	9.99K	159M
j6	300	16		Y	34.0M	8.84M	1.19M	9.87K	**122M**

Observation: The best performance comes from balancing all optimization goals.