Compiler Technology for Scientific Computing

David A. Padua
University of Illinois at Urbana-Champaign
Compiler Technology and Performance

• Although clock rate gains have been impressive, architectural features have also contributed significantly to performance improvement.

• At the instruction level, compilers play a crucial role in exploiting the power of the target machine.
And, with SMT and SMP machines growing in popularity, automatic transformation and analysis at the thread level will become increasingly important.
Compiler Research

• Advances in compiler technology can have a profound impact
  – By improving performance.
  – By facilitating programming without hindering performance.

• For example, average performance improvement of advanced compiler techniques at the instruction level for the IA-64 is ~3. *This is beyond what can be achieved by traditional optimizations.*
Scientific Computing has had a very important influence on compiler technology.

- Fortran I (1957)
- Many classical optimizations
  - strength reduction
  - Common subexpression elimination
  - ...
- Dependence analysis
Compiler Research and Scientific Computing (Cont.)

• Relative commercial importance of scientific computing is declining.
  – End of Cold War
  – Increasing volume of “non-numerical” applications
Compiler Techniques

• Every optimizing compiler does:
  – Program analysis
  – Transformations

• The result of the analysis determines which transformations are possible, but the number of possible valid transformations is unlimited.

• Implicit or explicit heuristics are used to determine which transformations to apply.
## Compiler Techniques (Cont.)

- Program analysis is well understood for some classes of computations such as simple dense computations.
- However, compilers usually do a poor job in some cases, especially in the presence of irregular memory accesses.
- Program analysis and transformation is built on a solid mathematical foundation, but building the engine that drives the transformations is an obscure craft.
Outline of the Talk

• Compiler Techniques for Very High Level Languages
• Advanced Program Analysis Techniques
• Static Performance Prediction and Optimization Control
1. Compiling Interactive Array Languages (MATLAB)

Joint work with
George Almasi (Illinois), Luiz De Rose (IBM Research), Vijay Menon (Cornell), Keshav Pingali (Cornell)
Motivation

- Many computational science applications involve matrix manipulations.
- It is, therefore, appropriate to use array languages to program them.
- Furthermore, interactive array languages like APL and MATLAB are very popular today.
Motivation  (continued)

• Their environment includes easy to use facilities for input of data and display of results.
• Interactivity facilitates debugging and development.
  – Increased software productivity.
• But interactive languages are usually interpreted.
  – Performance suffers.
Long Term Project Goals

• Generation of high-performance code for serial and parallel architectures
• Efficient extraction of information from the high-level semantics of MATLAB
• Use of the semantic information in order to improve compilation
Problem Overview

- MATLAB does not require declarations
  - Static and dynamic type inference

- Variables can change characteristics during execution time
Type Inference

• Type inference needs to determine the following variable properties:
  – Intrinsic type: logical, integer, real, or complex
  – Rank: scalar, vector, or matrix
  – Shape: size of each dimension

• For more advanced transformations (not implemented) it also will be useful to determine:
  – Structure: lower triangular, diagonal, etc
# Test Programs

<table>
<thead>
<tr>
<th>Test programs</th>
<th>Problem size</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successive Overrelaxation (SOR)</td>
<td>420x420</td>
<td>a</td>
</tr>
<tr>
<td>Preconditioned CG (CG)</td>
<td>420x420</td>
<td>a</td>
</tr>
<tr>
<td>Generation of a 3D-Surface (3D)</td>
<td>51x31x21</td>
<td>d</td>
</tr>
<tr>
<td>Quasi-Minimal Residual (QMR)</td>
<td>420x420</td>
<td>a</td>
</tr>
<tr>
<td>Adaptive Quadrature (AQ)</td>
<td>1Dim (7)</td>
<td>b</td>
</tr>
<tr>
<td>Incomplete Cholesky Factorization (IC)</td>
<td>400x400</td>
<td>d</td>
</tr>
<tr>
<td>Galerkin (Poisson equation) (Ga)</td>
<td>40x40</td>
<td>c</td>
</tr>
<tr>
<td>Crank-Nicholson (heat equation) (CN)</td>
<td>321x321</td>
<td>b</td>
</tr>
<tr>
<td>Two body problem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4th order Runge-Kutta (RK)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Two body problem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Euler-Cromer method (EC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dirichlet (Laplace’s equation) (Di)</td>
<td>41x41</td>
<td>b</td>
</tr>
<tr>
<td>Finite Difference (wave equation) (FD)</td>
<td>451x451</td>
<td>b</td>
</tr>
</tbody>
</table>

Source:
- c. “Numerical Methods for Physics”, A. Garcia
- d. Colleagues
# Times on the SGI Power Challenge

<table>
<thead>
<tr>
<th>Prog.</th>
<th>MATLAB</th>
<th>MCC</th>
<th>F 90</th>
<th>H. W.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR</td>
<td>18.12</td>
<td>18.14</td>
<td>2.733</td>
<td>0.641</td>
</tr>
<tr>
<td>CG</td>
<td>5.34</td>
<td>5.51</td>
<td>0.588</td>
<td>0.543</td>
</tr>
<tr>
<td>3D</td>
<td>34.95</td>
<td>11.14</td>
<td>3.163</td>
<td>3.158</td>
</tr>
<tr>
<td>QMR</td>
<td>7.58</td>
<td>6.24</td>
<td>0.611</td>
<td>0.562</td>
</tr>
<tr>
<td>AQ</td>
<td>19.95</td>
<td>2.30</td>
<td>1.477</td>
<td>0.877</td>
</tr>
<tr>
<td>IC</td>
<td>32.28</td>
<td>1.35</td>
<td>0.245</td>
<td>0.052</td>
</tr>
<tr>
<td>Ga</td>
<td>31.44</td>
<td>0.56</td>
<td>0.156</td>
<td>0.154</td>
</tr>
<tr>
<td>CN</td>
<td>44.10</td>
<td>0.70</td>
<td>0.098</td>
<td>0.097</td>
</tr>
<tr>
<td>RK</td>
<td>20.60</td>
<td>5.77</td>
<td>0.038</td>
<td>0.025</td>
</tr>
<tr>
<td>EC</td>
<td>8.34</td>
<td>3.38</td>
<td>0.012</td>
<td>0.007</td>
</tr>
<tr>
<td>Di</td>
<td>44.17</td>
<td>1.50</td>
<td>0.052</td>
<td>0.050</td>
</tr>
<tr>
<td>FD</td>
<td>34.80</td>
<td>0.37</td>
<td>0.031</td>
<td>0.031</td>
</tr>
</tbody>
</table>
Speedups on the SGI (1 Processor)

![Bar chart showing speedups on the SGI for various tasks. The x-axis represents different tasks with abbreviations SOR, CG, 3D, QMR, AQ, IC, Ga, CN, RK, EC, Di, FD. The y-axis represents speedup on a log scale. The chart is labeled with SGI Power Challenge. The tasks are color-coded with red for FALCON and blue for MCC.]
Hand-Written Code Comparison

Speedup of hand-written over FALCON

SGI Power Challenge

SOR, CG, 3D, QMR, AQ, IC, Ga, CN, RK, EC, Di, FD
MAJIC Overview

- MAJIC: (MAtrlab Just-In-Time Compiler) interactive and fast
  - combination interpreter/JIT compiler
  - builds on top of FALCON techniques
Compiler Techniques in MAJIC

Analysis
• Compile only code that takes time to execute (loops)
• type analysis and value/limit propagation
• recompile only when source has changed

Code Generation
• naïve (per AST node) JIT code generation
• uses built-in MATLAB functions where possible
• average compile time: 20ms per line of MATLAB source
Future Compiler Techniques

• JIT SMP parallelism: lightweight dependence analysis, execution on multiple Solaris LWPs

• speculative lookahead compilation: hiding compilation time from user

• exact shape/value propagation: allows precise unrolling for “small vector” operations
Compiler Techniques for Very High Level Languages

2. Compiling Tensor Product Language

Joint work with
Jeremy Johnson (Drexel), Robert Johnson (MathStar), Jose Moura (CMU), Viktor Prasanna (SC), Manuela Veloso (CMU)
Cooley-Tukey Theorem and the FFT

In 1964, Cooley and Tukey presented a divide and conquer algorithm for computing \( f = F_{nr}x \). Their algorithm is based on
Theorem. Let \( n = rs \), \( 0 \leq k_1, l_2 < r \), \( 0 \leq k_2, l_1 < s \), then
\[
y(l_1 + l_2s) = \sum_{k_1} w^{k_1l_2s} (w^{k_1l_1} (\sum_{k_2} x(k_1 + k_2r)w^{k_2l_1r}))
\]
Repeated application to \( n = 2^t \), for example, leads to an \( O(n \log n) \) algorithm, called the Fast Fourier Transform (FFT), for computing \( f = F_{nr}x \).
Tensor Product of Matrices

Let $A$ be an $m \times m$ matrix and $B$ an $n \times n$ matrix. Then $A \otimes B$ is the $mn \times mn$ matrix defined by the block matrix product

$$A \otimes B = (a_{i,j}B)_{1 \leq i, j \leq m}$$

$$= \begin{pmatrix}
a_{1,1}B & \cdots & a_{1,m}B \\
\vdots & \ddots & \vdots \\
a_{m,1}B & \cdots & a_{m,m}B
\end{pmatrix}$$

For example, if

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

then

$$A \otimes B = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{bmatrix}$$
Tensor Product Formulation of Cooley-Tuckey

**Theorem**

\[ F_{rs} = (F_r \otimes I_s) T_{s}^{rs} (I_r \otimes F_s) L_{r}^{rs} \]

- \( T_{s}^{rs} \) is a diagonal matrix
- \( L_{r}^{rs} \) is a stride permutation

**Example**

\[ F_4 = (F_2 \otimes I_2) T_{4}^{4} (I_2 \otimes F_2) L_{2}^{4} \]

\[
\begin{bmatrix}
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & -1 & 0 \\
0 & 1 & 0 & -1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 0 & 0 \\
1 & -1 & 0 & 0 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & -1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & -1 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
Properties of the Tensor Product

**Associativity**
\[(A \otimes B) \otimes C = A \otimes (B \otimes C)\]

**Multiplicative Property**
\[AC \otimes BD = (A \otimes B)(C \otimes D)\]
\[A \otimes B = (A \otimes I_n)(I_m \otimes B)\]

**Commutation Theorem**
\[A \otimes B = L_m^{mn} (B \otimes A)L_n^{mn}\]

**Stride Permutations**
\[L_s^{rst} L_t^{rst} = L_s^{rst}\]
TPL Program

#subname F4

( compose

  ( tensor (F 2) (I 2) )

  ( T 4 2 )

  ( tensor (I 2) (F 2) )

  ( L 4 2 )
)
Unrolled Program

subroutine F4(y,x)
implicit complex*16(f)
complex*16 y(4),x(4)

f0 = x(1) + x(3)
f1 = x(1) - x(3)
f2 = x(2) + x(4)
f3 = x(2) - x(4)
f4 = (0.0d0,-1.0d0) * f3
y(1) = f0 + f2
y(3) = f0 - f2
y(2) = f1 + f4
y(4) = f1 - f4
dend
subroutine F4L(y,x)
implicit complex*16(f)
complex*16 y(4),x(4),t1(4)

do i0 = 0, 1
    t1(2*i0+1) = x(i0+1) + x(i0+3)
    t1(2*i0+2) = x(i0+1) - x(i0+3)
end do

t1(4) = (0.0d0,-1.0d0) * t1(4)

do i0 = 0, 1
    y(i0+1) = t1(i0+1) + t1(i0+3)
    y(i0+3) = t1(i0+1) - t1(i0+3)
end do

end
Advanced Program Analysis Techniques

1. Analysis of Explicitly Parallel Programs

Joint work with
Samuel Midkiff (IBM Research and Jaejin Lee (Michigan State)

Shared Memory Parallel Programming

- A global shared address space between threads
- Communication between threads:
  - reads and writes of shared variables

```plaintext
flag = 1

do while (flag==0)
    end do
```
Incorrect Constant Propagation

To avoid this incorrect propagation, we can conservatively assume `wait(g)` modifies variable `b`, but this cannot handle busy-wait synchronization.
Incorrect Loop Invariant Code Motion

- \( x = a + 1 \) is a loop invariant in classical sense, but moving it outside makes the loop infinite.
Incorrect Dead Code Elimination

- An instruction $I$ is dead if it computes values that are not used in any executable path starting at $I$.
- Eliminating $\texttt{flag = 1}$ in Thread 1 makes the while loop in Thread 2 infinite.
Incorrect Compilation

- The compiler makes the busy-wait loop infinite.
Concurrent Static Single Assignment Form (Cont.)

• A $\pi$-function of the form $\pi(v_1, \ldots, v_n)$ for a shared variable $v$ is placed where there is a use of the shared variable with $\delta^t$ conflict edges. $n$ is the number of reaching definitions to the use of $v$ through the incoming control flow edges and incoming conflict $\delta^t$ edges.

• The CSSA form has the following properties:
  – All uses of a variable are reached by exactly one (static) assignment to the variable.
  – For a variable, the definition dominates the uses if they are not arguments of $\phi$-, $\pi$-, and $\psi$-functions.
Concurrent Static Single Assignment Form (Cont.)

\[
\begin{align*}
\text{cobegin} & \quad b = 0 \\
& \quad f = 0 \\
& \quad g = 0 \\
& \quad s = 0 \\
& \quad \text{do while } (f == 0) \\
& \quad \text{end do} \\
& \quad b = 4 \\
& \quad \text{post } (g) \\
& \quad \text{coend} \\
& \quad \text{print } s
\end{align*}
\]
Synchronization Analysis

- We compute a set $\text{Prec}[n]$ of nodes $m$, which is guaranteed to precede $n$ during execution.

- Computing the exact execution ordering:
  - NP-hard problem
  - An iterative approximation algorithm
    - Common ancestor algorithm
Common Ancestor Algorithm

- Originally, a trace based algorithm, but we use it for static analysis.
- Data flow equations:

\[
Prec_{\text{cont}}[n] = \begin{cases} 
\bigcup_{(m,n) \in E_{\text{cont}}} (Prec[m] \cup \{m\}) & \text{if } n \text{ is a coend node} \\
\bigcap_{(m,n) \in E_{\text{cont}}} (Prec[m] \cup \{m\}) & \text{otherwise} 
\end{cases}
\]

\[
Prec_{\text{sync}}[n] = \bigcap_{(m,n) \in E_{\text{sync}}} (Prec[m] \cup \{m\})
\]

\[
Prec[n] = Prec_{\text{cont}}[n] \cup Prec_{\text{sync}}[n]
\]
Advanced Program Analysis Techniques

2. Analysis of Conventional Programs to Detect Parallelism

Joint work with
W. Blume (HP), J. Hoeflinger (Illinois), R. Eigenmann (Purdue), P. Petersen (KAI), and P. Tu
An Important Area

- Powerful high-level restructuring for parallelism and locality enhancement.
- Language constructs based on early research on detection of parallelism.
- Analysis and transformations important for instruction-level parallelism.
An Experiment on the Alliant FX/80

• In 1989-90, we did experiments on the effectiveness of automatic parallelization for an eight-processor Alliant FX/80 and the Cedar multiprocessor.

• We translated the Perfect Benchmarks using KAP-Cedar, a version of KAP modified at Illinois. The speedups obtained for the Alliant are shown next.


Automatically-obtained speedups on the Alliant FX/80
An experiment on the Alliant FX/80 (cont.)

- As can be seen, the speedups obtained were dismal.

- However, we found that there is much parallelism in the Perfect Benchmarks and, more importantly, that it can be detected automatically.

- In fact, after applying by hand a few automatable transformations, we obtained the following results.
Speedups on the Alliant FX/80
New compiler techniques

- Three classes of automatable transformations were found to be of importance in the previous study:
  - Recognition and translation of idioms
  - Dependence analysis
  - Array privatization
  - Transformations similar but weaker than those we applied by hand were applied by KAP.
Dependence Analysis

- Given the program:

```plaintext
do  i = 1, n
   do  j = 1, m
       do  j = 1, m
           S:   A(f(i,j),g(i,j)) = ...
           T:   ... = A(p(i,j),q(i,j)) + ...
       end do
   end do
end do
```
To determine if there is a flow dependence from S to T, we need to determine whether the system of equations:

\[ f(i,j) = p(i',j') \]
\[ g(i,j) = q(i',j') \]

has a solution under the constraints

\[ n \geq i, \quad i' \geq 1, \quad m \geq j, \quad j' \geq 1, \text{ and } (i',j') \geq (i,j) \]
Dependence Analysis (cont.)

• Loop-carried dependences preclude transformation into parallel form.
• They also preclude some important transformations.
• For example, statement S can be removed from the loop if f(i) is never 2
  
  
  do i=1,n
  S: A(f(i))=...
  T: Q=A(2)
  ... 
  end do
Dependence Analysis (cont.)

- Many techniques have been developed to answer this question efficiently.
- Banerjee's test suffices in most situations where the subscript expressions are a linear function of the loop indices.

Dependence Analysis (cont.)

• However, there are situations where traditional dependence analysis techniques, including Banerjee's test, fail. Two cases we found in our experiments are:

• Nonlinear subscript expressions. These could be present in the original source code or generated by the translator when replacing induction variables.

• Indexed arrays.
Dependence analysis (cont.)

• An example of nonlinear subscript expression from OCEAN:

```plaintext
do  j=0,n-1
   do  k=0,x(j)
      do  m=0,128
         ...
         p=258*n*k+128*j+m+1
         A(p)=A(p)−A(p+129*n)
         A(p+129*n)=...
         ...
```
Dependence analysis (cont.)

- Example of indexed array from TRACK

```fortran
    do i=1,n
        ...
        jok= ihits(1,i)
        nused(jok)=nused + 1
        ...
    end do
```
Speedups on SGI Challenge (8 processors)
Array privatization

• Each processor cooperating in the execution of a loop has a separate copy of all private variables.

• A variable can be privatized -- that is, replaced by a private variable -- if in every loop iteration the variable is always assigned before it is fetched.
Array privatization (cont.)

• Example of privatizable scalar:
  
  do k=
    s = A(k)+1
    B(k) = s**2
    C(k) = s-1
  end do

• Example of privatizable array:
  
  do k=
    do j=1,m
      s(j) = A(k, j)+1
    end do
    do j=1,m
      B(k, j) = s(j)**2
      C(k, j) = s(j)-1
    end do
  end do
Array privatization (cont.)

• Commercial compilers are effective at identifying privatizable scalars.
• However, we found a number of cases where array privatization is necessary for loop parallelization.
• New techniques were developed to deal effectively with array privatization.
Conclusions

• Significant progress is possible
• But few research groups are still focusing on this problem
• An experimental approach is crucial for progress.
  – Need good benchmarks (also desperately needed for research on compilers for explicitly parallel programs)
Advanced Program Analysis Techniques

3. Dependence Analysis
On-going Projects

• Analysis of non-affine subscript expressions.

• Compile-Time analysis of Index Arrays

• Analysis of Java Arrays
3a. Analysis of non-affine subscript expressions

*Joint work with*

*Y. Paek (KAIST) and J. Hoeflinger (Illinois)*

Y. Paek, J. Hoeflinger, and D. Padua. Simplification of array access patterns for compiler optimization. SIGPLAN’98 Conference on Programming Language Design and Implementation (PLDI)
Traditional Dependence Analysis

- Traditional data dependence analysis for arrays:
  - form dependence equation
  - solve equation, taking into account constraints

\[
\begin{align*}
\text{DO } & I = 1,N \\
\quad & A(I) = \ldots \\
\quad & \ldots = A(I+N) \\
\text{END DO}
\end{align*}
\]

\[
I = I' + N
\]

Given that $1 \leq I \leq N$
and $1 \leq I' \leq N$
What’s Wrong with That?

• Tradition is successful, but limited.
  – *coupled subscripts* and *non-affine subscripts* cause problems
  – non-loop parallelization has become important
  – interprocedural parallelization has become important

• Specifically, the system-solving paradigm is too limiting.
Representing an Integer Sequence

• We can precisely represent any regular integer sequence by:
  – its starting value,
  – an expression for the difference between successive elements of the sequence,
  – the total number of elements.

\[
\text{do } I=1,N \\
\quad A(2^{**I}) \\
\text{end do}
\]

\[
\begin{align*}
\text{Start: } & 2 \\
S_{I+1} - S_I &= 2^I \\
\text{\# elements: } & N
\end{align*}
\]

S.O.S. = 2, 4, 8, 16, ... 2^N
LMADs

\[ A_{\text{span}_1}, \ldots, A_{\text{span}_d} \text{ stride } l, \ldots, \text{ stride } d + \text{ base offset} \]
Summarizing Loops and CALLs

REAL A(N)
DO I=1,N
A(I) . . .
END DO

\[ A_0^{0} + I - 1 \rightarrow A_{N-1}^{1} + 0 \]

expand

REAL A(N)
CALL X(A(I))

\[ A_0^{0} + I - 1 \rightarrow A_{M-1, T^*M}^{1, M} + I - 1 \]

REAL Z(*)

\[ Z_{M-1, T^*M}^{1, M} + 0 \]
Overlap in an LMAD

If a given loop index causes the subscripting offset sequence to produce the same element more than once, then the LMAD is said to have an *overlap* due to that loop index.

This corresponds to a *loop-carried dependence*.

```plaintext
do I = 1,N
    do J = 1, M
        A(J) = . . .
    end do
end do

Real A(25)
do I = 0,4
    do J = 0, 5
        A(I+3*J) =
    end do
end do
```
Advanced Program Analysis Techniques

3c. Analysis of Java Arrays

Joint work with
Paul Feautrier (U. Versailles) and Peng Wu (Illinois)
Analyzing Java Arrays

• Traditional loop-level optimizations are not directly applicable to Java arrays

• Multi-dimensional Java arrays may have irregular shapes
  – combness analysis

• Common use of reference ("pointers"),
  – a pointer-based dependence test
Fortran style optimizations

blocking,
loop unrolling,
loop interchange,
loop fusion,
parallelization,
...

Index-based
DD test
...

alias/shape

exception

Fortran

Java
Combness Analysis
Combness analysis: analyze the shape of an array

level-1, level-2, level-3 comb
level-1, level-2 comb
An Example

- If a is a comb of level-1 only, loop-i is parallelizable
- If a is a comb of level-2 only, loop-j is parallelizable
- If a is a comb of level-1 and level-2, the loop-nest-i-j is parallelizable

```cpp
... int a[n][n][n];
...
for (int i = 0; i<n; i++)
    for (int j = 0; j<n; j++)
        for (int k = 0; k<n; k++)
            a[i][j][k] = a[i][j][k] + 1;
```
Static Performance Prediction and Optimization Control

Joint work with
D. Reed (Illinois) and C. Cascaval (Illinois)
Compile-time Performance Prediction Goals

- Provide the compiler with information to enable performance related optimizations
- Predict the execution time of scientific Fortran programs, with reasonable accuracy, using architecture independent models
- Rapid performance tuning tool
- Architecture evaluation
Compile-time Performance Prediction

- Fast performance estimation results with reasonable accuracy
  - light load on the compiler
  - rapid feedback to the user
- Abstract models (symbolic expressions), as a function of
  - program constructs
  - input data set
  - architecture parameters
Performance Prediction Model

- Model different parts of the system independently
- Each model generates one or more terms in a symbolic expression
- Advantages
  - simplicity
  - modularity
  - extensibility
Experimental Results
Splib Execution time
SPECfp95 - Mips R100000