Addressing Long-term Challenges in Compiler Construction
The PACE Project

Computer Science Affiliates Day, October 15, 2009

Keith D. Cooper
Doerr Professor in Computational Engineering
Department of Computer Science
Rice University
PACE Project Lead

This work is funded by the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915 with Rice University. The opinions and findings in this document do not necessarily reflect the views of either the United States Government or of Rice University.
The PACE Project

The PACE Project is a multi-institutional research project funded by the Defense Advanced Research Projects Agency through the Air Force Research Laboratory.

The PACE Project is developing an architecture aware compiler environment.

**Senior Personnel:**
Keith Cooper, John Mellor-Crummey, Krishna Palen, Vivek Sarkar, and Linda Torczon + Michael Burke, Philippe Charles, Erzsebet Merenyi, Ray Simar, Tim Harvey

Rishi Khan | Rice University
P. Sadayappan | ET International
Sanjiva Lele | Ohio State University
Reid Tatge | Stanford University

Texas Instruments, Inc.
The Challenge

It takes years to obtain a high-quality compiler for a new computer system
• Classic examples are three to five years in development and tuning
• Modern computer systems may be harder than the classic examples
  – Heterogeneity & ubiquitous parallelism

Why is porting an optimizing compiler hard?
• Compilers deal in details and interactions between the details
  – Cache parameters, ILP available in application, instruction fetch & issue
discipline, pipeline structure, thread creation cost, call overhead, ...
  – To achieve performance, compiler must discover rate limiters and relieve them
• Compilers must get all the details right to achieve performance
  – Simply takes time to understand the details, the problems, & the cures

How can we reduce the time to obtain high-quality compilers?

  Automate the process of tuning and porting the compiler

  Substitute computation for human effort

PACE
platform-aware compilation environment

Addressing Long Term Challenges in Compiler Construction 3
Meeting the Challenge

PACE will attack the problem of portable optimization

- Goal is to build a self-retargeting, source-to-source optimizer
  - PACE System will measure performance critical system properties
  - PACE System will adapt its behavior to those measured properties

- PACE will not attack portable code generation \((e.g., no \textit{back end research})\)
  - PACE will rely on existence of a C compiler for the target machine
  - PACE produces target-specific, compiler specific C code
    - Existing native C compiler produces executable code for the target
    - Might use more than one C compiler

Porting PACE to a new system should be quick and easy

- Compile & run the resource characterization software
- Compile PACE and start using it to compile & run your application
PACE Project Overview

The PACE System will embody four key themes

• Measure system performance, both hardware and software
  – View effective or available performance through the lens of system software

• Use system & application characterizations to drive optimization
  – Levels of optimization for platform-wide and target-specific concerns

• Adjust and adapt behavior at runtime
  – Shift strategies & parameters to improve observed performance

• Use offline learning to encode experience & observed behaviors
  – Learn about interplay of system, software, & applications

We will instantiate these themes in a sophisticated open-source compilation environment
PACE Project Overview

PACE decomposes the problem into four key subprojects

- **Resource Characterization Task**
  - Identify system parameters that should drive optimization
  - Develop software tools that can measure these parameters in a portable way

- **PACE Compiler Task**
  - Transform application code for system-wide performance
  - Tailor application code to specific processing elements

- **Runtime System Task**
  - Measure application behavior for characterization & performance debugging
  - Adjust code & parameters at runtime to improve performance

- **Machine Learning Activity**
  - Gather experience and observation and turn them into knowledge
  - Use statistical machine learning techniques to solve deep (noisy) problems
Resource Characterization

- System Characterization
  - Config File
  - Compiler Characterization
    - uBenchmarks

- Platform Aware Optimizer
  - IR
  - C or Fortran
  - Vendor compiler 1
  - Vendor compiler 2

- Target Aware Optimizer
  - IR
  - X86 Backend

- App-Aware Partitioner

- Performance Measurement

- Application Characterization

- Source Code

- Runtime System
  - Runtime opt.
The PACE Compiler

- Platform-aware compilation environment
- Addressing Long Term Challenges in Compiler Construction
- Performance Measurement
- C, C++, Fortran with MPI calls or OpenMP pragmas & calls

- System Characterization
- Config File
- Compiler Characterization
- uBenchmarks

- Learning Engine
- Knowledge Base
- App-Aware Partitioner
- Source Code
- Application Characterization
- HPCToolkit

- Platform Aware Optimizer
- Target Aware Optimizer
- Vendor compiler 1
- Vendor compiler 2
- X86 Backend

- IR
- C or Fortran

- Runtime System
- Runtime opt.
The PACE Runtime
The Role of Machine Learning

knowledge base

- App-Aware Partitioner
- Source Code
- Application Characterization
- Performance Measurement

- Target Aware Optimizer
- Platform Aware Optimizer
- IR
- C or Fortran
- X86 Backend
- Vendor compiler 1
- Vendor compiler 2

- uBenchmarks
- System Characterization
- Config File
- Compiler Characterization

- uBenchmarks
- Runtime System
- Runtime opt.
PACE Resource Characterization

The remainder of this talk will focus on resource characterization

• First part of PACE with a “due date”
• Piece of PACE where my efforts are currently directed

Present a little background, then a series of vignettes

The work I present is the product of a large and talented team

Senior Personnel  Keith Cooper, Tim Harvey, John Mellor-Crummey, Vivek Sarkar, Ray Simar, Reid Tatge, Linda Torczon

Junior Personnel  Heba Bevan, Arnold Schwaighofer, Jeff Sandoval, Anna Youseffi
PACE Approach to Resource Characterization

Select characteristics that the compiler or runtime system will actually use

• No point measuring numbers for their own sake
• Emphasis on discernible and effective numbers
• Each characteristic should be justified
PACE Approach to Resource Characterization

Select characteristics that the compiler or runtime system will actually use

- No point measuring numbers for their own sake
- Emphasis on discernible and effective numbers
- Each characteristic should be justified

Measure them in simple and direct ways (whenever possible)

- Multiple microbenchmarks for complex or tricky numbers
- Simple interfaces and simple code for portability
- Small set of software requirements (in the spirit of AACE)
PACE Approach to Resource Characterization

Select characteristics that the compiler or runtime system will actually use

• No point measuring numbers for their own sake
• Emphasis on discernible and effective numbers
• Each characteristic should be justified

Measure them in simple and direct ways (whenever possible)

• Multiple microbenchmarks for complex or tricky numbers
• Simple interfaces and simple code for portability
• Small set of software requirements (in the spirit of AACE)

Create a standalone tool that compiles, measures, and reports

• Should be robust across machines and compilers
• Perform extensive testing on variety of local machines
  – Add tests on remote machines when confident of behavior

Remember, the overarching goal is to build an optimizing compiler that retargets itself.
Software Resources Available to the PACE RC Software

Because PACE is to be portable, it has a small set of requirements.

• POSIX, including pthreads (will use POSIX clock() for timer support)
• C Compiler + standard libraries + math libraries
• An interface to specify thread affinity
• Syntax for compiling and executing microbenchmarks at appropriate optimization level
  – Must include compiler options to let microbenchmarks function correctly
• Standard set of tools, such as shell, make, autoconf, automake, ...

You can think of resource characterization as an adversarial game played with a few simple tools – C code, your understanding of architecture, and your colleagues ingenuity

C Compiler:
-- May need assembly output option on C compiler
-- May need assembly code “drop”
### Characteristics of Interest to PACE

<table>
<thead>
<tr>
<th>Category</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Hierarchy</td>
<td>Effective sizes, granularity, associativity, &amp; latency at each discernible level of the hierarchy (code &amp; data)</td>
</tr>
<tr>
<td></td>
<td>Memory bandwidth as measured by Stream benchmark</td>
</tr>
<tr>
<td></td>
<td>Number of concurrent memory streams before degradation</td>
</tr>
<tr>
<td></td>
<td>Outstanding memory operations before stall or interlock</td>
</tr>
<tr>
<td></td>
<td>Impact of stride on latency &amp; effective cache size</td>
</tr>
<tr>
<td>Processor</td>
<td>Issue width by typed operator</td>
</tr>
<tr>
<td></td>
<td>Relative operator costs</td>
</tr>
<tr>
<td></td>
<td>Branch costs</td>
</tr>
<tr>
<td>System</td>
<td>Effective number of threads before degradation</td>
</tr>
<tr>
<td></td>
<td>Cost of moving value between threads</td>
</tr>
<tr>
<td></td>
<td>Cost of thread creation, destruction, &amp; various p-thread and MPI primitives</td>
</tr>
</tbody>
</table>
Characteristics of Interest to PACE

Native Compiler

Performance on specific C language idioms (array vs. pointer, multiply-add, sequential vs. parallel loop, ...)

Number of concurrently live values before performance degradation

We are also interested in a class of questions that deal with resource sharing

• Not yet sure what the questions are
• Not sure how to measure useful answers
• Our Machine Learning folks are looking at this issue
Stories from Microbenchmark Development

• Some microbenchmarks are easy
• Some microbenchmarks are easy but have complex post-analysis
• Some microbenchmarks are simply hard
• Most of the microbenchmarks we have built incorporate subtle knowledge of interactions between compiler, architecture, and performance

We will see examples of each kind
Cost of Array Access versus Pointer Access

The TAO should make an intelligent decision about how to represent arrays

The Idea

• Use an array that fits in L1 cache
• Access it a significant number of times
  – Measure 1D array with pointer and array subscripts
  – Measure 2D array with pointer and array subscripts
  – Measure 3D array with pointer and array subscripts
• Microbenchmark produces a ratio of pointer cost to array cost
• On Intel T9600, with gcc 4.1 –O3

No effect, with this compiler and system, from row vs. column major order

<table>
<thead>
<tr>
<th>Intel T9600</th>
<th>1D</th>
<th>2D</th>
<th>3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rel. Cost of Ptr. vs. Array Access</td>
<td>1</td>
<td>0.62</td>
<td>0.28</td>
</tr>
</tbody>
</table>

TAO ≈ Target Aware Optimizer
Anomalies in Access Data by Stride

Changing the stride of array access can change the cost of those accesses

- The PAO should consider adjusting the dimension of an array if a nearby size will produce faster access

- Gathering the data is easy (Interest out to fairly large strides)
- Interpreting and summarizing the data is harder
- Summarization may reduce accuracy but increase usability!

- Both large and small sizes are of interest to PAO
- Polyhedral system wants to know 1, 2, and several sizes of large

PAO ≈ Platform Aware Optimizer
Measuring Memory Hierarchy Parameters

The PAO needs to know the effective size, associativity, granularity, & access cost of each discernible level in the memory hierarchy

- Used in polyhedral transformations (+ others) to improve memory behavior

- People assured us that membench already solved these problems
- Membench output requires expert interpretation
- It was not clear how to read this data with 75% accuracy
- It was not clear how various hardware features affect data

Graph generated locally with Yelick’s membench code
You Can Find Experts ...

Graphs & interpretation from Kathy Yelick @ Berkeley

But it may be hard to automate expert interpretation of this data
Measuring Memory Hierarchy Parameters

So, we set out to build our own microbenchmarks

• Focused on finding level size and granularity
  – Associativity is not too hard, once you know size

• Built a code that strides through an array
• Fix # of accesses & vary array size
• Time should rise when size exceeds cache size
• Here is the data from a Core 2 Duo
• What happened?
• All of the money that Intel spent on the design of prefetch units paid off
Measuring Memory Hierarchy Parameters

Strategy: to defeat the prefetch unit, use randomization

This curve is much better, but it still has noise and anomalies. It did not match what we expected.

This test uses a single permutation, stored in the accessed array.
Strategy: to defeat the prefetch unit, use randomization

This test uses a block cyclic permutation, stored in the accessed array. The data is much closer to what we expected.
Measuring Memory Hierarchy Parameters

Strategy: to defeat the prefetch unit, use randomization

Even though this graph appears easier to read, it still has minor anomalies.

Intel T9600
Block Cyclic Permutation
Blocksize = 1 KB
4 KB to 16 MB

Private TLB
Actual L1
Actual L2
Shared TLB
Anomalies
Measuring Memory Hierarchy Parameters

**New Strategy:** Now that we have fairly clean data, learn to read the data

- For level size: find points that we suspect are levels & zoom on on them
  - Three tests: log test, inflection test, and a concavity test
  - In a typical run on the T9600, five suspect points appear in the block cyclic data
Measuring Memory Hierarchy Parameters

New Strategy: Now that we have fairly clean data, learn to read the data

- Identify levels & zoom on on them
- Use inflection test, concavity test
- Three tests: log test, inflection test, and concavity test

In a typical run on the T9600, five suspect points appear in the block cyclic data

Closer look at 32 KB

Real cache transition

Actual L1

Actual L2

Actual TLB

Array Size Log Scale
Measuring Memory Hierarchy Parameters

**New Strategy:** Now that we have fairly clean data, learn to read the data
- Identify levels & zoom on them
  - Incongruent test
  - Concavity test
- Points appear in the block cyclic data

![Graph showing memory hierarchy parameters](#)

- **Closer look at 64 KB**
- **Real Transition, Hard to Find**
- **Actual L1**
- **Actual L2**
- **Actual TLB**

**Array Size**

Log Scale
New Strategy: Now that we have fairly clean data, learn to read the data
- Identify levels & zoom on them
- Use concavity test
- Points appear in the block cyclic data

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Log Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,000</td>
<td>0.0002</td>
</tr>
<tr>
<td>4,000</td>
<td>0.0003</td>
</tr>
<tr>
<td>4,000</td>
<td>0.0004</td>
</tr>
<tr>
<td>4,000</td>
<td>0.0005</td>
</tr>
<tr>
<td>4,000</td>
<td>0.0006</td>
</tr>
<tr>
<td>4,000</td>
<td>0.0007</td>
</tr>
<tr>
<td>4,000</td>
<td>0.0008</td>
</tr>
<tr>
<td>4,000</td>
<td>0.0009</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Actual L1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20,000</td>
<td>0.0002</td>
</tr>
<tr>
<td>40,000</td>
<td>0.0003</td>
</tr>
<tr>
<td>60,000</td>
<td>0.0004</td>
</tr>
<tr>
<td>80,000</td>
<td>0.0005</td>
</tr>
<tr>
<td>100,000</td>
<td>0.0006</td>
</tr>
<tr>
<td>120,000</td>
<td>0.0007</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Actual L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20,000</td>
<td>0.0002</td>
</tr>
<tr>
<td>40,000</td>
<td>0.0003</td>
</tr>
<tr>
<td>60,000</td>
<td>0.0004</td>
</tr>
<tr>
<td>80,000</td>
<td>0.0005</td>
</tr>
<tr>
<td>100,000</td>
<td>0.0006</td>
</tr>
<tr>
<td>120,000</td>
<td>0.0007</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Actual TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20,000</td>
<td>0.0002</td>
</tr>
<tr>
<td>40,000</td>
<td>0.0003</td>
</tr>
<tr>
<td>60,000</td>
<td>0.0004</td>
</tr>
<tr>
<td>80,000</td>
<td>0.0005</td>
</tr>
<tr>
<td>100,000</td>
<td>0.0006</td>
</tr>
<tr>
<td>120,000</td>
<td>0.0007</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Private TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20,000</td>
<td>0.0002</td>
</tr>
<tr>
<td>40,000</td>
<td>0.0003</td>
</tr>
<tr>
<td>60,000</td>
<td>0.0004</td>
</tr>
<tr>
<td>80,000</td>
<td>0.0005</td>
</tr>
<tr>
<td>100,000</td>
<td>0.0006</td>
</tr>
<tr>
<td>120,000</td>
<td>0.0007</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Real Transition, Hard to Find</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20,000</td>
<td>0.0002</td>
</tr>
<tr>
<td>40,000</td>
<td>0.0003</td>
</tr>
<tr>
<td>60,000</td>
<td>0.0004</td>
</tr>
<tr>
<td>80,000</td>
<td>0.0005</td>
</tr>
<tr>
<td>100,000</td>
<td>0.0006</td>
</tr>
<tr>
<td>120,000</td>
<td>0.0007</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Clear Granularity at 1024 words or 4096 bytes (1 VM Page)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20,000</td>
<td>0.0002</td>
</tr>
<tr>
<td>40,000</td>
<td>0.0003</td>
</tr>
<tr>
<td>60,000</td>
<td>0.0004</td>
</tr>
<tr>
<td>80,000</td>
<td>0.0005</td>
</tr>
<tr>
<td>100,000</td>
<td>0.0006</td>
</tr>
<tr>
<td>120,000</td>
<td>0.0007</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seconds</th>
<th>L1 Line Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20,000</td>
<td>0.0002</td>
</tr>
<tr>
<td>40,000</td>
<td>0.0003</td>
</tr>
<tr>
<td>60,000</td>
<td>0.0004</td>
</tr>
<tr>
<td>80,000</td>
<td>0.0005</td>
</tr>
<tr>
<td>100,000</td>
<td>0.0006</td>
</tr>
<tr>
<td>120,000</td>
<td>0.0007</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Stride Test for Granularity, 1 MB array (Shows effects at 32KB &amp; 64 KB levels) Intel T9600</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20,000</td>
<td>0.0002</td>
</tr>
<tr>
<td>40,000</td>
<td>0.0003</td>
</tr>
<tr>
<td>60,000</td>
<td>0.0004</td>
</tr>
<tr>
<td>80,000</td>
<td>0.0005</td>
</tr>
<tr>
<td>100,000</td>
<td>0.0006</td>
</tr>
<tr>
<td>120,000</td>
<td>0.0007</td>
</tr>
</tbody>
</table>
Measuring Memory Hierarchy Parameters

New Strategy: Now that we have fairly clean data, learn to read the data
- Are levels & zoom in on them
- Concavity test
- Points appear in the block cyclic data

• For level size: find points that we suspect are levels & zoom on on them
- Three tests: log test, inflection test, and a concavity test
- In a typical run on the T9600, five suspect points appear in the block cyclic data

Closer look at 1 MB

Actual L1

Actual L2

Actual TLB

Real, if small, transition (TLB)
Measuring Memory Hierarchy Parameters

**New Strategy:** Now that we have fairly clean data, learn to read the data

- Find levels & zoom on on them
- Use a concavity test
- Three tests: log test, inflection test, and a concavity test

In a typical run on the T9600, five suspect points appear in the block cyclic data

To be useful, the methodology must be robust against anomalies

To have a closer look at 2 MB:

- Anomaly
- Actual L1
- Actual L2
- Actual TLB

Array Size
Log Scale
Measuring Memory Hierarchy Parameters

New Strategy: Now that we have fairly clean data, learn to read the data.

- For level size: find points that we suspect are levels & zoom on on them.
- For concavity test:
- Points appear in the block cyclic data.
- Real cache transition
- Shape indicates shared with other uses (other core, code cache)

- Log test finds this point
- Inflection test finds this point

Closer look at 6 MB

Array Size
Log Scale

Actual L1
Actual TLB
Actual L2
Measuring Memory Hierarchy Parameters

New Strategy: Now that we have fairly clean data, learn to read the data

• For level size: find points that we suspect are levels & zoom in on them
  – Three tests: log test, inflection test, and a concavity test
  – In a typical run on the T9600, five suspect points appear in the block cyclic data

• Run both straight permutation and block cyclic permutation
  – Apply same analysis to each and compare results
  – Consensus points are, in our experience, the real levels in the hierarchy

• Soft transitions, such as the Core 2 Duo L2 cache are real
  – We will report a range of numbers
  – Those numbers will depend on code cache activity & other cores’ activities

• We are launching a learning subproject to develop good estimates for shared resources

“Effective” comes into play with shared resources
Measuring Memory Hierarchy Parameters

What about the rest of the numbers?

• Associativity is mostly straightforward, given size
  – Concerns about ratio of cache to main memory
  – Same associativity at different levels can be hard to discern

Associativity may provide another check on level sizes
Measuring Memory Hierarchy Parameters

What about the rest of the numbers?

- Associativity is mostly straightforward, given size
  - Some number theory involved in isolating effects to a given level
  - Uniform associativity at different levels can be hard to discern

- Finding granularity resembles finding level sizes
- Use same analysis
- Compute slopes & find inflections

![Graph showing L1 Line Size Data](image-url)
Measuring Behavior of Multiplier

Need to know ratio of multiply cost to add cost, number of multiplies that can issue at each cycle, and presence of multiply-add  (Scheduling, unroll-and-jam)

The Strategy

• Generate independent streams of operations
• Use multiple multiply streams for issue slots
• Change ratio of operations to measure ratio of operation costs
• Careful mixing can measure ability to overlap add & multiply

<table>
<thead>
<tr>
<th>Stream 1</th>
<th>Stream 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_2 = a_0 \times a_1)</td>
<td>(b_2 = b_0 + b_1)</td>
</tr>
<tr>
<td>(a_3 = a_1 \times a_2)</td>
<td>(b_3 = b_1 + b_2)</td>
</tr>
<tr>
<td>(a_4 = a_2 \times a_3)</td>
<td>(b_4 = b_2 + b_3)</td>
</tr>
<tr>
<td>(a_5 = a_4 \times a_3)</td>
<td>(b_5 = b_4 + b_3)</td>
</tr>
<tr>
<td>(a_6 = \ldots)</td>
<td>(b_6 = \ldots)</td>
</tr>
</tbody>
</table>

The Challenges

• Compilers display sensitivity to naming regimes within straight-line code
• (Some) compilers have problems with long blocks of code
• Hardware timings can be value-sensitive

Conceptually clean microbenchmark
Myriad details to get right
Measuring Code Cache Size

The Strategy

• Hold number of operations executed constant while increasing the executing code
• Execute some benchmark function $k$ times
  – Use function pointers in C to call different (adjacent) copies of the code
  – Function must perform lots of computation on a small amount of data

The Lessons & the Challenges

• Must build blocks that can execute at prefetch speed or better
  – Otherwise, the i-cache does its job and you get flat response
• Must overwhelm prefetch & branch prediction hardware
  – Shuffle the instruction pointers
• Need to generate custom benchmark for each processor (today)
  – Need to know number and type of functional units
Summary & Conclusions

I focused on the Resource Characterization problems

• PACE has many other difficult technical challenges
• Remember, the overarching goal is to build an optimizer that retargets itself

Final Thoughts

• We are eight months into a 54 month project
• We are working on a broad agenda
  – I did not address the Compiler, Runtime, or Machine Learning subprojects
  – You will likely hear from them in future years
• We are having a great deal of fun and making a great deal of progress

Thank you for your attention