Algorithm Composition using Domain-Specific Libraries and Program Decomposition for Thread-Level Speculation

Troy A. Johnson
Purdue University
My Background

• ParaMount research group at Purdue ECE
  – http://cobweb.ecn.purdue.edu/ParaMount
  – high-performance computing, compilers, software tools, automatic parallelization

• Computational Science & Engineering Specialization
  – http://www.cse.purdue.edu
Compilers

Artificial Intelligence

Scientific & Parallel Computing

Architecture

Web, Grid, & Peer-to-Peer (P2P)

Kosha: P2P Enhancement for the Network File System

SC 04 JGC 06 JIE 01 JCAEE 02

OS3 – Online Student Survey System

Cetus – A Source-to-Source Transforming C Compiler

barrier synchronization

LCPC 03, 04

PPoPP 07

LCPC

International Workshop on Languages and Compilers for Parallel Computing

PLDI

ACM SIGPLAN Conf. on Programming Language Design & Implementation

PPoPP

ACM SIGPLAN Symp. on the Principles and Practice of Parallel Programming

WMPP

IEEE Workshop on Massively Parallel Processing

ICECS

IEEE International Conf. on Electronics, Circuits, and Systems

SC

ACM/IEEE Conference on Supercomputing

JGC

Journal of Grid Computing

JIE

ASEE/IEEE Frontiers in Education Conference

JCAEE

Journal of Computer Applications in Engineering Education
Outline

• Part I: Algorithm Composition using Domain-Specific Libraries
• Part II: Program Decomposition for Thread-Level Speculation
• Part III: Future Research, Funding, & Opportunities for Collaboration

PLDI 06
PLDI 04
PPoPP 07
Outline

• Part I: Algorithm Composition using Domain-Specific Libraries
• Part II: Program Decomposition for Thread-Level Speculation
• Part III: Future Research, Funding, & Opportunities for Collaboration
Motivation

• Increasing programmer productivity
• Typical language approach: increase abstraction
  – abstract further from machine; get closer to problem
  – do more using less code
  – reduce software development & maintenance costs
• Domain-specific languages / libraries (DSLs) provide a high level of abstraction
  – e.g., domains are biology, chemistry, physics, etc.
• But, library procedures are most useful when called in sequence
Example DSL: BioPerl

- http://www.bioperl.org
- DSL for Bioinformatics
- Written in the Perl language
- Popular, actively developed since 1995
- Used in the Dept. of Biological Sciences at Purdue
A Typical BioPerl Call Sequence

• Query a remote database and save the result to local storage:

```perl
Query q = bio_db_query_genbank_new("nucleotide",
    "arabidopsis[ORGN] AND topoisomerase[TITL] AND 0:3000[SLEN]");
DB db = bio_db_genbank_new();
Stream stream = get_stream_by_query(db, q);
SeqIO seqio = bio_seqio_new(">sequence.fasta", "fasta");
Seq seq = next_seq(stream);
write_seq(seqio, seq);
```

Example adapted from
http://www.bioperl.org/wiki/HOWTO:Beginners
A Library User’s Problem

- Novice users don’t know these call sequences
  - procedures documented independently
  - tutorials provide some example code fragments
    - not an exhaustive list
    - may need adjusted for calling context (no copy paste)

- User knows what they want to do, but not how to do it
As Observed by Others

• “most users lack the [programming] expertise to properly identify and compose the routines appropriate to their application”

• “a common scenario is that the programmer knows what type of object he needs, but does not know how to write the code to get the object”
My Solution

• Add an “abstract algorithm” (AA) construct to the programming language
  – An AA is named and defined by the programmer
    • definition is the programmer's goal
  – An AA is called like a procedure
    • compiler replaces the call with a sequence of library calls
• How does the compiler compose the sequence?
  – short answer: it uses a domain-independent planner
Defining and Calling the AA

- AA (goal) defined using some properties...

```algorithm
save_query_result_locally(db_name, query_string, filename, format)
=> { query_result(result, db_name, query_string),
contains(filename, result),
in_format(filename, format) }
```

Properties are not procedure calls. Their order does not matter. result is a named return value.

...and called like a procedure

```Seq
seq = save_query_result_locally("nucleotide",
"arabidopsis[ORGN] AND topoisomerase[TITL] AND 0:3000[SLEN]",
"sequence.fasta", "fasta");
```

1 data type, 1 AA call
Describing the Programmer's Goal

• Programmer must indicate their goal somehow

• Library author provides a domain glossary
  – \texttt{query\_result}(\texttt{result}, \texttt{db}, \texttt{query}) – \texttt{result} is the outcome of sending \texttt{query} to the database \texttt{db}
  – \texttt{contains}(\texttt{filename}, \texttt{data}) – file \texttt{filename} contains \texttt{data}
  – \texttt{in\_format}(\texttt{filename}, \texttt{format}) – file \texttt{filename} is in format \texttt{format}

• Glossary terms are \texttt{properties} (facts), whereas \texttt{procedure} names are actions
Composing the Call Sequence

- AI planners solve a similar problem
- Given an initial state, a goal state, and a set of operators, a planner discovers a sequence (a plan) of instantiated operators (actions) that transforms the initial state to the goal state
- Operators define a state-transition system
  - planner finds a path from initial state to goal state
  - typically too many states to enumerate
  - planner searches intelligently
Traditional Planning Example

• Planners are not normally applied to software; they traditionally solve problems like this:

Initial State

- on(B, table)
- on(C, table)
- on(A, C)

These are properties. (Planner’s Input)

<table>
<thead>
<tr>
<th>These actions will move the plan towards the goal</th>
</tr>
</thead>
</table>

Plan

- move(A, table)
- move(B, C)
- move(A, B)

These are actions. (Planner’s Output)

Goal State

- on(C, table)
- on(B, C)
- on(A, B)

These are properties. (Planner’s Input)

*Actions in the plan modify a “world” of blocks*
To Solve Composition using Planning

- Initial state: calling context (from compiler analysis)
- Goal state: AA definition (from the library user)
- Operators: procedure specifications (from the library author)
  - Actions: procedure calls
- World: program state
My Composition System, DIPACS

DIPACS = Domain-Independent Planned Algorithm Composition and Selection

Troy A. Johnson - Purdue
My Composition System, DIPACS

1. Ontological Engineering – choosing a vocabulary for the domain

DIPACS = Domain-Independent Planned Algorithm Composition and Selection
Why High-Level Abstraction is OK

• Library author understands the properties
• Library user understands
  – via prior familiarity with the domain
  – via some communication from the author (glossary)
• Compiler propagates terms during analysis
  – meaning of properties does not matter
• Planner matches properties to goals
  – meaning of properties does not matter
My Composition System, DIPACS

1. Ontological Engineering – choosing a vocabulary for the domain
2. Determine Initial & Goal States – requires flow analysis; translation to a planning language
3. Object Creation – most planners assume a fixed set of objects
4. Merge the Plan into the Program – destructive vs. non-destructive plans

DIPACS = Domain-Independent Planned Algorithm Composition and Selection
Selection of a Call Sequence

• Multiple call sequences can be found
  – programmer or compiler can choose
• Incomplete specifications may cause undesirable sequences to be suggested
  – requiring complete specifications is not practical
  – permitting incompleteness is a strength
  – use programmer-compiler interaction for oversight
Related Work

• Languages and Compilers
  – Jungloids
  – Broadway
  – Speckle
Related Work (continued)

• **Automatic Programming**

• **Automated (AI) Planning**
  - Other work at NASA Ames Research Center
Conclusion

• A DSL compiler can use a planner to implement a useful language construct
• Gave an example using a real DSL (BioPerl)
• Identified implementation challenges and their general solutions in this talk
  – for detailed solutions see my PLDI 06 paper
Outline

• Part I: Algorithm Composition using Domain-Specific Libraries
• Part II: Program Decomposition for Thread-Level Speculation
• Part III: Future Research, Funding, & Opportunities for Collaboration
Motivation

• Multiple processors on a chip becoming common

• What should we do with them?
  – Run multiple programs simultaneously?
    • possibly more processor cores than programs
    • individual cores are generally simpler and slower
  – Run a single program using multiple cores?
    • program must be parallel
    • same problems as automatic parallelization in order to apply to a wide variety of existing programs
      – must prove independence of program sections
      – Is there a way to avoid these problems?
Thread-Level Speculation

- Decompose a sequential program into threads
- Run the threads in parallel on multiple cores
- Buffer program state and rollback execution to an earlier, correct state if not really parallel
- Inter-thread data dependences are OK, but slow execution due to the rollback overhead
Hardware Support

- Hardware uses a predictor to dispatch a sequence of threads
- Memory writes by threads are buffered
- All writes by a thread commit to main memory after they are known to be correct
- A misprediction or a data dependence violation causes a roll back and restart
Execution Model

- Data dependence and thread sequence misprediction contribute to overhead

Sequential program:

\[ x = \]
\[ = x \]

Time:

- \( T_0 \)
- \( T_1 \)
- \( T_2 \)
- \( T_3 \)

Processors:

- \( P_0 \)
- \( P_1 \)
- \( P_2 \)
- \( P_3 \)

Dependence violation or misprediction causes a roll back.
Optimizing for Speculation

- Any set of threads is valid, but amounts of overhead and parallelism will differ
  - thread decomposition is crucial to performance
- Decomposition can be done manually, statically (by a compiler), or dynamically (by the hardware or virtual machine)
  - I'll discuss two static approaches that use profiling
  - dynamic approaches have run-time overhead, do not know the program's high-level structure, & have difficulty performing trade-offs among overheads
Approach #1 (PLDI 04)

• Start with a control-flow graph
• Model data dependence and misprediction overheads as edge weight
  – use the min-cut algorithm to minimize overheads while partitioning the graph into threads
  – a new thread begins after a cut
• Load imbalance hard to model as edge weight
  – instead use balanced min-cut [Yang & Wong IEEE Trans. on CAD of ICS 96] to simultaneously minimize cut edge weight and help balance thread size
Classical Min-Cut

- Determines where to partition a graph such that the cost of the cut is minimal

\[
\begin{align*}
\text{thread 1} & \quad a \\
\downarrow & \\
\text{b} & \\
\uparrow & \\
\text{thread 2} & \\
\downarrow & \\
\text{f} & \quad \text{min-cut cost} = 5
\end{align*}
\]

\[
\begin{align*}
\text{c} & \quad 5 \\
\text{d} & \quad 5 \\
\text{e} & \quad 5 \\
\text{f} & \quad 5
\end{align*}
\]

thread dispatch overhead = 5 cycles
Min-Cut with Overhead Weights

- Edge weights cause the cut to avoid placing the producer and consumer in different threads.

Now suppose that there is a dependence between a and e, and the branch at b is 50% taken.
Not Quite That Simple

• Need to factor in parallelism
  – otherwise zero cuts (sequential program) looks “best”
  – estimate ideal execution time then add cut cost
• Edge weights depend on thread size
  – rollback penalty proportional to the amount of work thrown away
  – but when weights are assigned, the cut has not yet created the threads
  – solution: assign weights based on where threads will be, perform a balanced min-cut, repeat, …
Experimental Setup

• Simulator
  – Multiscalar architecture [Sohi ISCA 95] with 4 cores
  – core-to-core latency = 10 cycles
  – memory latency = 300 cycles
• Input
  – SPEC CPU2000 benchmark suite
  – compiler based on gcc
  – profiling data collected with \textit{train} input
    • dependences, branch-taken frequencies, etc.
  – performance data collected with \textit{ref} input
## SPEC CPU2000 Speedup Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Single Thread Insns Per Cycle</th>
<th>[Vijay Micro 98] Speedup</th>
<th>[Johnson PLDI 04] Min-Cut Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>0.48</td>
<td>1.98</td>
<td>2.11</td>
</tr>
<tr>
<td>art</td>
<td>0.19</td>
<td>2.06</td>
<td>2.43</td>
</tr>
<tr>
<td>equake</td>
<td>0.56</td>
<td>1.49</td>
<td>1.79</td>
</tr>
<tr>
<td>mgrid</td>
<td>0.35</td>
<td>5.41</td>
<td>5.54</td>
</tr>
<tr>
<td>swim</td>
<td>0.17</td>
<td>4.51</td>
<td>4.51</td>
</tr>
<tr>
<td>geometric mean</td>
<td></td>
<td>2.72</td>
<td>2.97</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.70</td>
<td>1.01</td>
<td>1.09</td>
</tr>
<tr>
<td>gzip</td>
<td>0.72</td>
<td>1.27</td>
<td>1.35</td>
</tr>
<tr>
<td>mcf</td>
<td>0.07</td>
<td>1.01</td>
<td>1.63</td>
</tr>
<tr>
<td>parser</td>
<td>0.51</td>
<td>0.87</td>
<td>1.24</td>
</tr>
<tr>
<td>vpr</td>
<td>0.63</td>
<td>1.38</td>
<td>1.09</td>
</tr>
<tr>
<td>geometric mean</td>
<td></td>
<td>1.09</td>
<td>1.27</td>
</tr>
</tbody>
</table>
Approach #2 (PPoPP 07)

• Position: static decomposition's effectiveness is limited due to the need for so much estimation.

• Instead, embed a search algorithm into a profile version of the program and have it try various decompositions while executing.

• Profile-time empirical optimization benefits from:
  – compiler-inserted instrumentation that guides the search based on high-level program structure
  – run-time system measuring performance
Candidate Threads

• Loop iterations
  – iterations are naturally balanced and predictable
  – dependence may cause rollback overhead

• Procedure calls
  – create larger threads in non-numerical applications

• Elsewhere
  – tends to make smaller threads; leads to imbalance
  – not the focus of this work
Decomposition Search Space

• Architecture provides three options for executing loops and calls
  – Option 0 (fine-grain)
    • all loop iterations or all threads in the callee are executed in parallel
  – Option 1 (sequential)
    • the loop or the callee is executed sequentially with the code before and after the loop or call
  – Option 2 (coarse-grain)
    • the loop or the callee is executed sequentially with the code before the loop or call but in parallel with what follows
Execution Options

<table>
<thead>
<tr>
<th>source code</th>
<th>threading options</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>(fine-grain)</td>
</tr>
<tr>
<td></td>
<td>(sequential)</td>
</tr>
<tr>
<td></td>
<td>(coarse-grain)</td>
</tr>
</tbody>
</table>

- call

- multiple new threads
- no new threads
- one new thread
Specifying a Decomposition

Each edge has a value \{0, 1, 2\}. Vertices are procedures or loops; we call this an extended call graph.

This program has \(3^5 = 243\) possible decompositions.
Speculation Locality

Potential to affect $T_0 \ldots T_{M-1}$

Locality shown for a 4-core system. Threads $T_0$ through $T_{M-1}$ are inside a call or loop.
Within loopA, the call to g runs as a single thread.

Because e₄ ≠ 0, any thread spawn points within g (and therefore e₅) are ignored by the hardware during that call to g.

The other call to g executes as multiple threads.
Overall Search Algorithm

• Bottom-up on extended call graph
  – measure performance of various options for leaf procedures & loops and keep the best options
  – when the search moves higher on the graph
    • either Option 1 (sequential) or 2 (coarse) is measured as best, such that the values of lower edges are ignored
    • or Option 0 (fine) is measured as best and the speculation locality property provides confidence that the values of lower edges are good and not influenced much by the decisions higher up
    • the values of lower edges are not reevaluated; once they are set, they remain fixed
Search Per Vertex

- Vertices with a small branching factor (number of outgoing edges) are searched exhaustively.
- We evaluate two strategies for the rest:
  - **Greedy:** tries $2n + 1$ solutions
    - starts by measuring performance of “00...0”
    - tries “10...0” and “20...0”, the best picks the first value (0, 1, or 2), then moves on to varying the next edge value.
  - **Hierarchical:** tries at most $0.5n^2 + 1.5n + 1$ solutions
    - starts by measuring performance of “22...2”
    - first pass tries changing each value to 1, keeping those 1s that improve performance
    - second pass tries changing each value to 0
### SPEC CFP2000 Speedup Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>0.48</td>
<td>1.98</td>
<td>2.11</td>
<td>3.37</td>
</tr>
<tr>
<td>art</td>
<td>0.19</td>
<td>2.06</td>
<td>2.43</td>
<td>3.00</td>
</tr>
<tr>
<td>equake</td>
<td>0.56</td>
<td>1.49</td>
<td>1.79</td>
<td>1.92</td>
</tr>
<tr>
<td>mgrid</td>
<td>0.35</td>
<td>5.41</td>
<td>5.54</td>
<td>6.09</td>
</tr>
<tr>
<td>swim</td>
<td>0.17</td>
<td>4.51</td>
<td>4.51</td>
<td>4.51</td>
</tr>
<tr>
<td>geometric mean</td>
<td></td>
<td>2.72</td>
<td>2.97</td>
<td>3.51</td>
</tr>
</tbody>
</table>

Speedup factors using 4 cores
SPEC CFP2000 Overheads
### SPEC CFP2000 Mean Thread Size

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>[Vijay Micro 98]</th>
<th>[Johnson PLDI 04]</th>
<th>[Johnson PPoPP 07]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min-Cut</td>
<td>Greedy</td>
<td>Hierarchical</td>
</tr>
<tr>
<td>applu</td>
<td>36.7</td>
<td>46.2</td>
<td>296.0</td>
</tr>
<tr>
<td>art</td>
<td>15.7</td>
<td>19.3</td>
<td>27.4</td>
</tr>
<tr>
<td>equake</td>
<td>15.5</td>
<td>19.6</td>
<td>27.2</td>
</tr>
<tr>
<td>mgrid</td>
<td>113.9</td>
<td>115.4</td>
<td>160.6</td>
</tr>
<tr>
<td>swim</td>
<td>101.9</td>
<td>112.8</td>
<td>101.9</td>
</tr>
<tr>
<td>arithmetic mean</td>
<td>56.7</td>
<td>62.7</td>
<td>122.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>108.0</td>
</tr>
</tbody>
</table>

Mean instructions per thread (dynamic)
## SPEC CINT2000 Speedup Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Single Thread IPC</th>
<th>[Vijay Micro 98]</th>
<th>[Johnson PLDI 04]</th>
<th>[Johnson PPoPP 07]</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>0.70</td>
<td>1.01</td>
<td>1.09</td>
<td>1.07</td>
</tr>
<tr>
<td>gzip</td>
<td>0.72</td>
<td>1.27</td>
<td>1.35</td>
<td>1.11</td>
</tr>
<tr>
<td>mcf</td>
<td>0.07</td>
<td>1.01</td>
<td>1.63</td>
<td>1.07</td>
</tr>
<tr>
<td>parser</td>
<td>0.51</td>
<td>0.87</td>
<td>1.24</td>
<td>1.20</td>
</tr>
<tr>
<td>vpr</td>
<td>0.63</td>
<td>1.38</td>
<td>1.09</td>
<td>1.38</td>
</tr>
</tbody>
</table>

**geometric mean**

- Single Thread: 0.70
- [Vijay Micro 98]: 1.01
- [Johnson PLDI 04]: 1.09
- [Johnson PPoPP 07]: 1.07

---

speedup factors using 4 cores
Sampling Error in SPEC CINT2000

• Instrumentation varies a value in the solution string that corresponds to a call or loop that is not executed during the measurement

• Not significant for CFP2000
  – few branches, high coverage of loops and calls
  – speedup improvement is consistently good

• Significant for CINT2000
  – many branches, low coverage of loops and calls
  – speedup improvement is inconsistent
Related Work

• Manual Decomposition

• Static Decomposition
  – Liu et al., *POSH: A TLS Compiler that Exploits Program Structure*, PPoPP 2006
Conclusions

• Decomposition performed at profile time via empirical optimization shows significant speedup over previous static methods for SPEC CFP2000
• The technique does not rely on specific architecture parameters, so should apply widely
• Sampling error limits improvement for SPEC CINT2000
Outline

• Part I: Algorithm Composition using Domain-Specific Libraries
• Part II: Program Decomposition for Thread-Level Speculation
• Part III: Future Research, Funding, & Opportunities for Collaboration
Future Research

• Algorithm Composition (Part I of this talk)
  – show that it applies to more domains
    • collaborate with those doing research in the domains
  – investigate interaction with other language features and how much programmers benefit from using composition
    • collaborate with researchers in programming languages and software engineering
  – investigate alternative planning algorithms to see if they work better (faster, solve harder problems)
    • collaborate with researchers in artificial intelligence
  – examine the lower (instruction-level) and upper (application-level, grid-level) limits of composition
    • collaborate with researchers in architecture and grid computing
Possible Source of Funding

- National Science Foundation
  - Computer Systems Research Grant
    - Advanced Execution Systems
      - Application Composition Systems
  - “Designing methods for automatically selecting application components suitable to the application problem specifications”
Future Research (2)

• Multi-core Systems (Part II of this talk)
  – mitigate the sampling problem for empirical optimization of non-numerical applications
  – investigate parallelization strategies (speculation or otherwise) for these systems
    • collaborate with researchers in parallel & scientific computing, compilers, and architecture