Engineering Parallel Applications with Tunable Architectures

International Conference on Software Engineering, May 2010
Christoph A. Schaefer/Victor Pankratius/Walter Tichy

Institute for Program Structures and Data Organization
Where is Karlsruhe?
Where is Karlsruhe?

University of Karlsruhe - KIT, Germany

Faculty of Computer Science
Institute for Program Structures and Data Organization (IPD)

One of the leading CS departments in Europe
>40 faculty, >400 PhD students in CS
Challenges of Parallel Applications

- Parallelization is complex and error-prone
- Parallel programs contain a number of tuning parameters
- Manual optimization difficult and time-consuming
- Each target platform may require re-tuning

**Auto-tuning**: Let the computer do the tuning!

Examples for Tuning Parameters

- Number of pipeline stages
- Choice of best algorithm implementation
- Order of execution
- Size of data partitions
- Number of workers
- Load balancing strategy
Approach to Auto-Tuning

- **Atune-TA**: Approach for description of certain parallel, tunable architectures
  - Automatic implementation of architectures
  - Portability regarding performance

- **Atune-OPT**: Automatic search-based performance tuning on multi-core platforms (*auto-tuner*)
  - Not limited to specific application domain or numeric programs
  - Extension of search-based optimization to handle large parallel applications
Example: Parallel Desktop Search (Indexing)

1. Definition of Tasks

- Abstraction from threads and fine-grained parallelization
- Concept of tasks: definition of essential processing steps

Methods to implement

- Crawl():List<string>
- ParseAlgo1(string s):ParseResult
- ParseAlgo2(string s):ParseResult
- UpdateIndex(ParseResult p):Index
- CreateIndexFile(Index i):void

Save index datastructure on disk
Example continued

2. Design, Implementation and Optimization

TunablePipeline MyDesktopSearch
[source:AC_Crawl;sink:AC_CreateIndexFile]
{
    TunableAlternative
    {
        AC_ParseAlgo2
        AC_ParseAlgo1
    },
    AC_UpdateIndex
}

Parameterized architecture implementation

Automatic platform-specific optimization

Tunability in action:

- TunablePipeline MyDesktopSearch
- TunableAlternative
  - AC_ParseAlgo2
  - AC_ParseAlgo1
- AC_UpdateIndex

Diagram:

- Crawl
- ParseAlgo1
- ParseAlgo2
- Update Index
- Create IndexFile

Instances:

Instance 1, Instance 2, ..., Instance n

Variables:

lb1, num1, lb2, num2, alt
**Atune-TA**

*Tunable Architecture Description Language (TADL)*

- Description language for compact design of parallel tunable architectures
  - **Atomic components**
    - Represent essential sequential program tasks
    - Contain no internal parallelism, but allow replication
    - Implemented by program methods (AC methods)
  - **Connectors**
    - Connect atomic components and define processing and parallelization strategies
    - Support nesting
    - Implicitly expose predefined tuning parameters

```plaintext
TunablePipeline pipeline
{   [source: AC_InputMethod;
    sink: AC_OutputMethod]
    AC_Method1,
    AC_Method2,
    AC_Method3
}
```
### Sequential Composition
- General-purpose connector with sequential execution semantics

### Tunable Alternative
- Describes exclusive choice
- Auto-tuner tests alternatives during optimization process

### Tunable Fork/Join
- Introduces task parallelism
**Atune-TA**

**TADL Connectors (2)**

- **Tunable Pipeline**
  - Describes pipeline parallelism
  - Offers data stream semantics

- **Tunable Producer/Consumer**
  - Describes common synchronization pattern
  - Offers data stream semantics

- **Tunable Replication**
  - Introduces data parallelism
  - Creates instances of atomic component

![Diagram showing fusion of stages and tunable pipeline components](image-url)
**Atune-TA**

**Tunable Architecture Implementation: TADL Compiler**

- **TADL compiler** transforms TADL script into instrumented, parallel executable code
- **Result:** portable intermediate representation of parallel program, ready for optimization on target platform

---

Diagram:

1. **TADL script**
2. Transforms each TADL connector into wrapper class
3. Associates atomic component methods
4. Implement TADL connectors using **TALib**
5. Instrument wrapper classes using **Atune-IL**
6. Library containing configurable parallelization strategies
7. Tuning instrumentation language to specify tuning instructions
Atune-OPT
Overview and Process

- Context-based preprocessing steps to prepare search space
- Automatic search-based tuning of parallel architecture
  - Common search algorithms: *random sampling, hillclimbing, swarm optimization*

Preprocessing

1. Extract tuning instructions and context information
2. Context-based Search Space Partitioning
3. Context-based Search Space Reduction

Search-based Tuning

1. Compute new architecture configuration
2. Execute and monitor program
3. Apply configuration to architecture
Atune-OPT 🕵️
Context-based Search Space Partitioning

- **Goal**: Identification of program parts to tune independently → tuning entities
- Exploit semantics of
  - *Sequential Composition*
  - *Tunable Alternative*
- Sub trees can be tuned separately, as they never run concurrently
- Partitioning into tuning entities
- Separate optimization of the tuning entities → reduction of parameter configurations

\[
S = V_a \times V_b \times V_c \times V_d \times V_e \times V_f \\
R = V_a \times V_b \times V_c \cup V_d \times V_e \times V_f \\
R \subseteq S
\]
Atune-OPT
Context-based Search Space Reduction

- **Goal**: Search space reduction using guided search
- Exploit semantics of parallel TADL connectors

**Example**
- *Tunable Pipeline* with data-parallel stages
- Instead of “blind tuning” *we apply heuristics*:
  - Balancing the pipeline
  - Fuse groups of consecutive data-parallel stages

---

Non-balanced pipeline

Balanced pipeline

1. **Stage 1**
2. **Stage 2**
3. **Stage 3**
4. **Stage 4**
5. **Stage 5**

Processing time 1 element

---

**Balanced pipeline**

- Fused stages
- Time

---

**Non-balanced pipeline**

- Time

---

Fusion of consecutive stages
## Evaluation

### Case Studies

<table>
<thead>
<tr>
<th>Application</th>
<th>Purpose</th>
<th>Size (LOC)</th>
<th>Exec. time sequential</th>
<th>Parallelism Types$^1$</th>
<th>Input data / benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MetaboliteID (MID)</strong></td>
<td>Bio-chemical data analysis</td>
<td>~ 100,000</td>
<td>85 s</td>
<td>T / D</td>
<td>mass spectrograms (1 GB)</td>
</tr>
<tr>
<td><strong>GrGen.NET</strong></td>
<td>Graph rewriting</td>
<td>~ 80,000</td>
<td>45 s</td>
<td>T / D</td>
<td>simulation of biological gene expression (~ 9 mio. nodes)</td>
</tr>
<tr>
<td><strong>Desktop Search (DS)</strong></td>
<td>Indexing of documents</td>
<td>~ 5,500</td>
<td>14 h 35 m</td>
<td>P / D</td>
<td>10,700 text files (max. 613 KB)</td>
</tr>
<tr>
<td><strong>Video</strong></td>
<td>Video processing</td>
<td>~ 1,000</td>
<td>19 s</td>
<td>P / D</td>
<td>video (180 frames, 800x600 px.)</td>
</tr>
</tbody>
</table>

$^1$ P: pipeline parallelism, T: task parallelism, D: data parallelism
Evaluation
Experimental Results (1)

- Performance evaluation: achieved speedup after optimizing parallel programs

<table>
<thead>
<tr>
<th></th>
<th>MID</th>
<th>GrGen</th>
<th>DS</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst</td>
<td>1.6</td>
<td>1.8</td>
<td>1.7</td>
<td>2.6</td>
</tr>
<tr>
<td>Best</td>
<td>3.1</td>
<td>7.7</td>
<td>6.9</td>
<td>5.6</td>
</tr>
<tr>
<td>TPG</td>
<td>194%</td>
<td>428%</td>
<td>406%</td>
<td>215%</td>
</tr>
</tbody>
</table>

Metrics
- Worst speedup
- Best speedup after tuning
- Tuning Performance Gain (TPG)

Experiments performed on 8-core-machine (2x Intel Xeon QC @ 1.86 GHz/Core). Worst speedup results from testing most inappropriate parameter configuration.
Evaluation
Experimental Results (2)

- **Evaluation**: Reduction of implementation effort using *Atune-TA*
- **Comparison of manual and Atune-TA-based implementation**

<table>
<thead>
<tr>
<th>Metrics</th>
<th>MID</th>
<th>GrGen</th>
<th>DS</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manual</td>
<td>290</td>
<td>120</td>
<td>465</td>
<td>300</td>
</tr>
<tr>
<td>Atune-TA</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Reduction</td>
<td><strong>287</strong> (99%)</td>
<td><strong>117</strong> (98%)</td>
<td><strong>462</strong> (99%)</td>
<td><strong>297</strong> (99%)</td>
</tr>
<tr>
<td>Synchronization primitives&lt;sup&gt;1)&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manual</td>
<td>18</td>
<td>8</td>
<td>27</td>
<td>18</td>
</tr>
<tr>
<td>Atune-TA</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Reduction</td>
<td><strong>16</strong> (89%)</td>
<td><strong>8</strong> (100%)</td>
<td><strong>26</strong> (96%)</td>
<td><strong>18</strong> (100%)</td>
</tr>
<tr>
<td>Tuning instrumentation statements</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manual</td>
<td>39</td>
<td>16</td>
<td>30</td>
<td>16</td>
</tr>
<tr>
<td>Atune-TA</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Reduction</td>
<td><strong>39</strong> (100%)</td>
<td><strong>14</strong> (87%)</td>
<td><strong>30</strong> (100%)</td>
<td><strong>16</strong> (100%)</td>
</tr>
</tbody>
</table>

<sup>1)</sup> Includes all synchronization primitives, such as *lock, notify, wait, join,* etc.
Related Work

- **ATLAS/AEOS** (Whaley et al., 2000)
  - Auto-tuning system for algebraic operations and algorithms
  - Domain specific approach
  - No support for parallel programs

- **Active Harmony** (Tapus et al., 2002)
  - Search-based auto-tuning system for library optimization
  - Comprehensive analysis of search algorithms
  - Not applicable for parallel programs

- **MATE** (Morajko et al., 2007)
  - Model-based tuning system for distributed PVM programs
  - Provides good performance predictions
  - Limited to special program structures

- **Parallel Pattern Language** (Mattson et al., 2004)
  - Structured collection of parallel patterns
  - Provides guideline for parallel programming
  - Optimization is not considered
Future Work

- More tunable patterns!
- Language integration of patterns (XJava)
- Online tuning (instead of offline)
- Parameter prediction
  - Set good starting values for search, or eliminate search
    - Set replication depending on idle threads
    - Prefer tasks that have the most input waiting
    - Observe work stealing behavior for cutoff-value
  - First results: we achieve 90% of best configuration without search
Conclusion

- Multi-core systems force developers to exploit parallelism in programs
- Auto-tuning of parallel programs is indispensable to achieve good performance

- *Atune* provides automated approach to design, implement and optimize parallel tunable architectures
  - Combination of parallelization *and* optimization
  - High-level parallelization process of applications
  - Extension of search-based auto-tuning to handle entire architectures

- *Atune-TA*: Using tunable architectures results in reduction of implementation effort
- *Atune-OPT*: Novel tuning techniques provide efficient optimization and significant performance gain
THANK YOU!
QUESTIONS?

For details see:
Christoph A. Schaefer, Victor Pankratius, Walter F. Tichy:
*Engineering Parallel Applications with Tunable Architectures.*
In Proceedings of 32nd International Conference on Software Engineering (ICSE),
to appear May 2010
Atune-IL: Tuning Instrumentation Language (1)

- Declaration of Tuning Blocks
  
  ```plaintext
  #pragma atune STARTBLOCK myBlock type PIPELINE
  <source code statements>
  <other Atune-IL statements>
  ...
  #pragma atune ENDBLOCK
  ```

- Define scopes of tuning parameters

- Tuning Blocks support
  - Nesting (lexically or logically) to represent application structure
  - Types to specify context

- Declaration of Tuning Parameters

  ```plaintext
  #pragma atune SETVAR myParameter type int
  values 10-100 step 10, weight 3, inside myBlock
  ```
Atune-IL: Tuning Instrumentation Language (2)

- Further constructs to
  - declare measuring points (incl. metric)
  - declare permutation regions (to re-order statements in host language)

- Atune-IL’s design goals
  - Separation of program code and tuning instructions
  - Compact representation of performance-relevant variants of parallel architectures
  - Syntax suitable for automatic generation
### Evaluation

#### Assumptions

- Estimation of manual implementation effort to implement functionality of TADL connectors

<table>
<thead>
<tr>
<th>TADL connector</th>
<th>LOC</th>
<th># Syncs ¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tunable Alternative</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Tunable Fork/Join</td>
<td>170</td>
<td>10</td>
</tr>
<tr>
<td>Tunable Pipeline</td>
<td>180</td>
<td>10</td>
</tr>
<tr>
<td>Tunable Producer/Consumer</td>
<td>150</td>
<td>9</td>
</tr>
<tr>
<td>Tunable Replication</td>
<td>120</td>
<td>8</td>
</tr>
</tbody>
</table>

¹) Total number of synchronization-related statements in source code, such as `lock`, `notify`, `wait`, `join`, etc.