Alef™ Formal Verification and Planning System

Samuel Luckenbill, James Ezick, Ph.D.,
Donald Nguyen, Peter Szilagyi, Richard Lethin, Ph.D.

Reservoir Labs, Inc.

11 May 2006
Presentation Outline

• Overview and Applications
• Alef Parallel SAT Solver
• Salt: Satisfiability Application Logic and Translator
• Alef Compiler
• Questions and Discussion
Alef Planning and Formal Verification System

Alef system: uses Reservoir’s R-Stream compiler technology, a parallel SAT engine, and HPC hardware to solve planning and verification problems.

- **Alef compiler**: accepts planning and formal verification problems and transforms them to the Salt language.

- **Salt tool**: translates Salt language into Conjunctive Normal Form (CNF) with partition annotations. Performs optimizations based on lazy-inference.

- **Parallel SAT solver**: runs on Cray XD1, incorporates complex parallel algorithms and solver heuristics to achieve significant speedup on some structured problems.
The Satisfiability Problem (SAT)

Definition: Given a Boolean formula \( E \), decide if there is some assignment to the variables in \( E \) such that \( E \) evaluates to true.

Example: \( E = (\neg a \lor b) \land (\neg a \lor \neg b \lor c) \land (\neg b \lor \neg c) \)

Solution: \( E \) evaluates to true (is satisfied) if \( a = 0, \ b = 0, \) and \( c = 0 \) or 1.

- **Alef Applications**
  - **Software Verification**: Verifying numerical precision and interprocedural control flow (Reservoir); assertion checking (Coverity); proving that an implementation meets an Alloy specification (MIT).
  - **Hardware Verification**: Bounded model checking (Cadence, Synopsys, etc.); test pattern generation (IBM, Intel, etc.).
  - **Planning**: Route planning (Lockheed); mission planning (DoD).
Alef Parallel SAT Solver Overview

- Parallel implementation allows multiple searches over different parts of the search space.
- Message passing approach reduces network load and round trip message delays.
- Multithreaded implementation increases latency tolerance, allowing multiple search threads per node.
- Dynamic load balancing ensures all nodes remain busy.
- Asynchronous sharing of learned information allows nodes to work together.

Alef Parallel SAT Solver
Boolean Constraint Propagation (BCP)

• Modern backtracking SAT algorithms spend more than 80% of their runtime performing Boolean Constraint Propagation (BCP) [Moskewicz 2001]
  – Choose a variable and assign it 0 or 1
  – Propagate the variable through the clauses, detecting implications
  – Iteratively propagate new implications, detecting any conflicts that may arise

• Example: \( E = (\neg a \lor b) \land (\neg a \lor \neg b \lor c) \land (\neg b \lor \neg c) \)
  – Assign \( a = 1 \)
    • \( b \) is implied to be 1 to satisfy first clause
    • Queue implication \( b = 1 \)
    • No implications from remaining clauses from \( a = 1 \)
  – Propagate new implication \( b = 1 \)
    • \( c \) is implied to be 1 to satisfy second clause
    • \( c \) is implied to be 0 to satisfy third clause (a conflict)
  – Backtrack and try \( a = 0 \), etc.
Alef Parallel BCP Algorithm

- Each node of the parallel machine runs a BCP worker thread, which performs BCP for one or more search threads.
- Each worker thread has a subset of the clauses (constraints).
- Search threads make decisions, resolve conflicts, and coordinate worker threads.
- After a decision, BCP requests are sent to nodes which contain the decision variable.
- Results from BCP are forwarded between worker threads until BCP terminates, then sent back to the search thread.

Distributed Boolean Constraint Propagation

Assignment: $a = 1$

Implication:
- $a = 1$
- $b = 1$
- $c = 1$

Conflict:
- $a = 1$
- $b = 1$
- $c = 1, 0$

Original BCP Request
Forwarded BCP Request
BCP Results
Message Forwarding and Termination Detection

- **Forwarding of messages**
  - Implications are forwarded on to other nodes to continue BCP
  - Reduces communication for BCP (over 80% of the runtime) by 50%

- **Fractional termination detection**
  - Incoming responses are matched to outgoing requests
  - Responses to messages are valued as fractions of the total response to a single request
  - Termination is detected when the sum of the values of the responses is 1
Communication Library

• **Implemented using MPI**
  – Native communication interface for Cray XD-1 (1.7µs message latency)
  – MPI calls hidden from application layer

• **Provides one-sided communication protocol**
  – Provides one-sided send, request, respond, and receive
  – Monitors outgoing messages until they are received, reclaims storage
  – Sorts incoming messages into priority queues for application threads
  – Tracks outstanding requests, allowing for invalidation of stale responses
  – Handles fractional termination detection

• **Selected (simplified) API Calls:**
  
  void communication_request(Message request) – Sends a request which is noted on a “scoreboard.” Responses are later matched to this request. Local storage is reclaimed when the request message is received.

  Message communication_receive(int thread_id) – Receives a waiting message for a thread sorted by priority then age.

  void communication_invalidate_responses(int thread_id) – Invalidates all pending responses to outstanding requests for a thread.

  int communication_is_complete(Message message) – Checks the “scoreboard” to see if all responses to a message have been received using fractional termination detection.
Alef and FPGAs

- **Potential uses for FPGAs on XD1:**
  - **Fast evaluation of unrolled time steps from bounded model checking (BMC):** Circuits derived from each time step of a BMC problem are similar and could be compiled once for an FPGA and used for fast propagation of signals. Back propagation circuits could be expensive.
  - **Compiled subcircuit representations:** As the solver works, it could select frequently-visited pieces of the problem and compile them to an FPGA for fast propagation.

- **Challenges**
  - **Long compilation times** and lack of dynamic compilation tools. Traditionally, compilation for FPGAs requires writing Verilog to describe a circuit, adjusting timing and layout, running the design through a compiler, and place and route. For real-time challenges such as route planning, the fast dynamic compilation is required.
  - **Time to reprogram the FPGA** is on the order of milliseconds to seconds, while Opterons have clock speeds up to 2.4GHz (0.42ns per clock cycle). This is 2.4 million to 2.4 billion clock cycles load a program.
  - **Latency to cross chip boundaries** limits ability to split an algorithm between two chips unless parts are relatively independent.
Alef Parallel Solver Performance Goals

• **Problem size**
  – Up to 100x through distribution of problem over HPC hardware
  – Requires partitioning using high-level problem structure

• **Data parallelism in Boolean Constraint Propagation**
  – Best case speedup on chip from data parallelism in BCP is 30x – 60x [Zhao]
    • Nanosecond-level message latency
  – Worst case speedup on cluster is 1/3x – 1.4x [Ganai, Gupta]
    • ~100µs MPI latency over gigabit Ethernet
  – Expected speedup on Cray XD1: 1x – 30x
    • 1.7 µs MPI latency within chassis

• **Search parallelism**
  – Threads work together sharing learned information
  – 1x – 20x speedup depending on benchmark [Blochinger]

• **Conclusion:**
  – Best case: 100x problem size, 30x speedup from data parallelism, 20x speedup from algorithmic parallelism
  – Worst case: no speedup, but we will likely be able to solve larger problems
Alef Parallel SAT Solver Status

- **Completed components**
  - Sequential SAT solver
  - Alpha-version parallel SAT solver
  - MPI-based communication library
  - Parallel BCP implementation
  - Conflict resolution
  - Unit test framework and library
  - ~15,500 lines of code

- **Incomplete components**
  - Verification and debugging
  - Load balancing through work stealing
  - Sharing of learned clauses
  - Experimentation and tuning
  - Optimization of code
  - Alef system integration

<table>
<thead>
<tr>
<th>ID</th>
<th>Task Name</th>
<th>Mar '05</th>
<th>May '05</th>
<th>Jul '05</th>
<th>Sep '05</th>
<th>Nov '05</th>
<th>Jan '06</th>
<th>Mar '06</th>
<th>May '06</th>
<th>Jul '06</th>
<th>Sep '06</th>
<th>Nov '06</th>
<th>Jan '07</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Phase II Start</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Solver</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Sequential Implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Parallel Data Structures</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>MPI Communication Library</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Parallel BCP Implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Unit Test Framework</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Mapper implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Conflict Resolution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Load Balancing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Algorithmic experimentation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Performance tuning</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>System Integration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Phase II Complete</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Salt: Satisfiability Application Logic and Translator

- **Current version:** 1.02 (full functionality)
- **Supports logic, sets, and arbitrary precision fixed point arithmetic**
  - Supports both unsigned and signed 2’s complement representations
  - Both truncation and rounding modes
  - Optional restrictions against overflow
- **60+ operators and ~40 translation directives**
  - Basic logical operations and compound operations (one_hot, at_least_n)
  - Union, intersection, cardinality operations
  - Basic arithmetic operations, shifts, integer roots
- **Design motivated by a CISC machine assembly language**
  - Macro operators translate directly to output expressions
  - Weak, implicit type system
  - Optimization based on lazy inference
- **Salt language does not express choice of how to encode**
- **Companion tool Shaker translates SAT results to readable form**
Salt-Shaker Data-Flow Pipeline

- Salt can target multiple domain-specific applications, each with distinct input formats tailored to a specific problem space.

- Salt and Shaker capture CNF conversion and solver solution extraction in a domain non-specific way.

- Salt can embody solver-specific optimizations and features, such as partition placement for the Alef parallel SAT solver.

Diagram:

1. Problem
2. Application
3. Salt Language
4. Salt Translator
5. Annotated CNF
6. Shaker
7. $x = 3.4, \ldots$
8. $1, -2, \ldots$
9. Alef Parallel SAT Solver
Salt Example: Sudoku

"Fill in the grid so that every row, every column, and every 3x3 box contains the digits 1 through 9."

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

- A single Salt file is generated that encodes each partial solution
- Sudoku encoding consists of four constraint groups:
  - Every square has at most one value
  - Every value occurs in every row
  - Every value occurs in every column
  - Every value occurs in every 3x3 box

Salt Output:
variables: 731 (+2)
clauses: 2261 (3.1:1)
solved: 272 (37.2%)
time: 0.010s

Alef Sequential Solver/Shaker Output:
RESULT: SAT
real 0m0.092s

9 6 3 1 7 4 2 5 8
1 7 8 3 2 5 6 4 9
2 5 4 6 8 9 7 3 1
8 2 1 4 3 7 5 9 6
4 9 6 8 5 2 3 1 7
7 3 5 9 6 1 8 2 4
5 8 9 7 1 3 4 6 2
3 1 7 2 4 6 9 8 5
6 4 2 5 9 8 1 7 3
Leveraging Reservoir’s R-Stream™ Compiler

- **Goal:** Augment R-Stream to generate Salt directly from C programs

- **EDG C front end**
  - Some verification conditions can be embedded as assertions
  - Other verification conditions can be expressed as stylized program annotations

- **Enhanced SSA internal representation**
  - Facilitates program simplifications in general
  - SSA conversion removes cycles from local dataflow graphs
  - Simplifies interpretation of statements as constraints

- **Large toolkit of compiler algorithms**
  - Confirm that the program has properties we require
    - Known loop bounds and functions
  - Transform program fragments into the form we require
    - Unroll loops and inline function calls
  - Perform program analysis to derive supplementary constraints
Software Bounded Model Checking with Alef

• **Model program and verification conditions using Salt constraints**
  – Program semantics, derived properties = P
  – Verification condition = Q
  – Refutation of correctness = P & ~Q

• **Verify using SAT solver**
  – SAT assignment provides counterexample to correctness
  – No assignment = correct program (but bounded)

• **Correctness guarantee limited by nature of SAT**
  – Finite program execution
  – Predicate statements on control dependencies
  - Can’t model unbounded executions in SAT
  - SSA renames local variables when they are modified so we have an acyclic DFG
  – Must be able to convert control dependence to data dependence
  – Ongoing work to characterize infinite execution properties as Salt constraints
Questions and Discussion