Relieving Capacity Limits on FPGA-Based SAT Solvers

Leopold Haller \(^1\) Satnam Singh \(^2\)

\(^1\)Oxford University Computing Laboratory
\(^2\)Microsoft Research Cambridge

FMCAD 2010
The quest for ever more efficient SAT solvers
Parallelising SAT

Software (ManySAT, Plingeling):

Solver 1  Solver 2  Solver 3

exchange of (unit) clauses

Coarse-grain parallelism

Hardware (e.g., [DTYZ08]):

Software solver

FPGA

Propagation

Clauses

···

Propagation

Clauses

Fine-grain parallelism
while true do
    if ¬decide() then
        return SAT;
    while BCP() = conflict do
        analyseAndLearn();
        if ¬backtrack() then
            return UNSAT;
while true do
  if \neg \text{decide()} then
    return SAT;
  while BCP() = conflict do
    analyseAndLearn();
    if \neg \text{backtrack()} then
      return UNSAT;

- Memory access in Boolean Constraint Propagation is hard to predict. Existing approaches rely on fast on-chip memory.
## Memory resources

<table>
<thead>
<tr>
<th></th>
<th><strong>on-chip</strong></th>
<th><strong>off-chip</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Block RAM</td>
<td>SDRAM</td>
</tr>
<tr>
<td></td>
<td>hundreds of small composable blocks</td>
<td>large atomic unit of memory</td>
</tr>
<tr>
<td></td>
<td>parallel access</td>
<td>sequential access</td>
</tr>
<tr>
<td></td>
<td>one-cycle access</td>
<td>non-deterministic delay</td>
</tr>
<tr>
<td></td>
<td>uniform access speed</td>
<td>streaming access faster</td>
</tr>
<tr>
<td></td>
<td>0.5-7MB (Virtex-5)</td>
<td>multiple GB</td>
</tr>
</tbody>
</table>
# Memory resources

## On-chip vs. Off-chip

<table>
<thead>
<tr>
<th>On-chip</th>
<th>Off-chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block RAM</td>
<td>SDRAM</td>
</tr>
<tr>
<td>Hundreds of small composable blocks</td>
<td>Large atomic unit of memory</td>
</tr>
<tr>
<td>Parallel access</td>
<td>Sequential access</td>
</tr>
<tr>
<td>One-cycle access</td>
<td>Non-deterministic delay</td>
</tr>
<tr>
<td>Uniform access speed</td>
<td>Streaming access faster</td>
</tr>
<tr>
<td>0.5-7MB (Virtex-5)</td>
<td>Multiple GB</td>
</tr>
</tbody>
</table>

Existing approaches store instance information on-chip

Capacity limits: e.g., [DTYZ08] - 64K variables/clauses
Relieving capacity limits

Previously suggested: Instance partitioning

We explore the feasibility of

- directly utilizing off-chip storage
- by building a custom memory hierarchy for SAT algorithms.

Our platform is the BEE3 (Berkeley Emulation Engine v3)

- 4 interconnected Virtex5 FPGAs
- 2 independent memory channels
- 8GB per channel (64GB in total)
- Ethernet, RS232, etc.
Relieving capacity limits

Previously suggested: Instance partitioning

We explore the feasibility of

- directly utilizing off-chip storage
- by building a custom memory hierarchy for SAT algorithms.
Relieving capacity limits

Previously suggested: Instance partitioning

We explore the feasibility of

- directly utilizing off-chip storage
- by building a custom memory hierarchy for SAT algorithms.

Our platform is the BEE3 (Berkeley Emulation Engine v3)

- 4 interconnected Virtex5 FPGAs
- 2 independent memory channels
- 8GB per channel (64GB in total)
- Ethernet, RS232, etc.
Xilinx Virtex-6 FPGA
XCVLX760
758,784 logic cells
864 DSP blocks
1,440 dual ported 18Kb RAMs

32-bit adder
(32/474,240 LUTs)
> 700MHz
procedure bcp(l: literal)
    wl ← readWatchlist(l);
    r ← ∅;
    for address ∈ wl do
        clause ← readClause(address);
        vals ← readVarValues(vals);
        r ← r ∪ propagate(clause, vals);
        //returns conflict, deduction or move WL
        processResults(r); writeResults(r);
    end for
end procedure
procedure bcp(\(l: \text{ literal}\))

\[\begin{align*}
wl & \leftarrow \text{readWatchlist}(l); \\
r & \leftarrow \emptyset; \\
\text{for } address \in wl & \text{ do} \\
& \quad \text{clause } \leftarrow \text{readClause}(address); \\
& \quad \text{vals } \leftarrow \text{readVarValues}(vals); \\
& \quad r \leftarrow r \cup \text{propagate}(\text{clause}, \text{vals}); \\
& \quad \text{//returns conflict, deduction or move WL} \\
& \quad \text{processResults}(r); \text{ writeResults}(r); \\
\end{align*}\]

end procedure

Read access pattern:

\[
\begin{array}{c|c|c}
\text{watch list} & \text{clause} & \text{vars} \\
\hline
\text{linear} & \text{linear} & \text{rand.} \\
\end{array}
\quad \begin{array}{c|c|c}
\text{clause} & \text{vars} \\
\hline
\text{linear} & \text{rand.} \\
\end{array}
\]
**BCP in CDCL algorithms**

**procedure** \( \text{bcp}(l: \text{literal}) \)

\[
\begin{align*}
\text{wl} & \leftarrow \text{readWatchlist}(l); \\
\text{r} & \leftarrow \emptyset; \\
\text{for } \text{address} & \in \text{wl} \text{ do} \\
& \quad \text{clause} \leftarrow \text{readClause}(\text{address}); \\
& \quad \text{vals} \leftarrow \text{readVarValues}(\text{vals}); \\
& \quad \text{r} \leftarrow \text{r} \cup \text{propagate}(\text{clause}, \text{vals}); \\
& \quad \text{//returns conflict, deduction or move WL} \\
& \quad \text{processResults}(\text{r}); \text{ writeResults}(\text{r}); \\
\end{align*}
\]

**end procedure**

Read access pattern:

- **watch list**
  - linear
  - off-chip
  - on-chip

- **clause**
  - linear
  - rand.

- **vars**
  - linear
  - rand.

- **clause**
  - linear
  - rand.
procedure bcp(l: literal)
    wl ← readWatchlist(l);
    r ← ∅;
    for address ∈ wl do
        clause ← readClause(address);
        vals ← readVarValues(vals);
        r ← r ∪ propagate(clause, vals);
    //returns conflict, deduction or move WL
        processResults(r); writeResults(r);
end procedure

Read access pattern:

watch list → clause → vars → clause → vars
linear ⇨ linear ⇨ rand. ⇨ linear ⇨ rand.
BCP in CDCL algorithms

**procedure** bcp(l: literal)

\[\begin{align*}
wl & \leftarrow \text{readWatchlist}(l); \\
r & \leftarrow \emptyset; \\
\textbf{for} \enspace address \in wl \enspace \textbf{do} & \\
\text{clause} & \leftarrow \text{readClause}(address); \\
\text{vals} & \leftarrow \text{readVarValues}(vals); \\
\text{r} & \leftarrow \text{r} \cup \text{propagate(clause, vals)}; \\
\text{//returns conflict, deduction or move WL} & \\
\text{processResults(r); writeResults(r); } & \\
\textbf{end for} & \\
\textbf{end procedure} & \\
\end{align*}\]

Read access pattern:

Parallelise?
Parallel Deduction

Possible overlap while processing clauses:

issue read  receive clause  read values  done

Overlap
Parallel Deduction

Possible overlap while processing clauses:

Use of multiple propagation units:
Parallel watched literals

Utilize the two independent memory channels.
Parallel watched literals

Utilize the two independent memory channels.

Store clauses redundantly

Memory channel A

<table>
<thead>
<tr>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>WL $1_A$</td>
</tr>
<tr>
<td>WL $-1_A$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>WL $-n_A$</td>
</tr>
</tbody>
</table>

Memory channel B

<table>
<thead>
<tr>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>WL $1_B$</td>
</tr>
<tr>
<td>WL $-1_B$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>WL $-n_B$</td>
</tr>
</tbody>
</table>

Split watch-list in two
Parallel watched literals

Utilize the two independent memory channels.

Memory channel A

\[ \text{Clauses} \]

\[ \begin{array}{c}
WL 1_A \\
WL -1_A \\
\vdots \\
WL -n_A
\end{array} \]

Memory channel B

\[ \text{Clauses} \]

\[ \begin{array}{c}
WL 1_B \\
WL -1_B \\
\vdots \\
WL -n_B
\end{array} \]

Store clauses redundantly

Split watch-list in two

\[ -1 \lor \ldots \lor -n \]
Parallel watched literals

Utilize the two independent memory channels.

Store clauses redundantly

Split watch-list in two

Local propagation
Architecture

BEE3 Board

FPGA

BCP_CORE

BCP_CORE controller

CDCL_MEM interface

memory controller channel A

DRAM channel A

BCP_CORE

propagator

propagator

var values

memory controller channel B

DRAM channel B

BCP_CORE

BCP_CORE controller

CDCL_MEM interface

propagator

propagator

BCP controller

BCP controller
Architecture

BCP_CORE
controller
propagator
CDCL_MEM
interface
BCP_CORE
controller
propagator
CDCL_MEM
interface
BCP_CORE
controller
propagator
memory
controller
channel A
memory
controller
channel B
DRAM
channel A
DRAM
channel B
FPGA
BCP_CORE
controller
propagator
BCP_CORE
controller
propagator
BEE3 Board
Progress

- Implementation of a BCP core on the BEE3 board.

- Validated in simulation, synthesized on the board
  - 15000 LUTs (20%)
  - 100MHz control logic, 250MHz memory

<table>
<thead>
<tr>
<th>Size limits (Virtex5 w. 16GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clauses (max. 24 literals)</td>
</tr>
<tr>
<td>variables</td>
</tr>
</tbody>
</table>

- Obtaining benchmark results is work in progress
Future work & Conclusion

Future work

- Application specific caches
- Parallelization through use of multiple FPGA chips
Future work & Conclusion

Future work

▶ Application specific caches
▶ Parallelization through use of multiple FPGA chips

Architecture for a hardware-based BCP core that:

▶ Achieves significantly *higher capacity* than existing approaches by directly accessing *large off-chip memory resources*.
▶ Employs fine-grain parallelisation strategies.
Thank you for your attention.

John D. Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang.
A practical reconfigurable hardware accelerator for boolean satisfiability solvers.

Sharad Malik and Lintao Zhang.
Boolean satisfiability from theoretical hardness to practical success.