The Liquid Metal Blokus Duo Design

Erik Altman Joshua S. Auerbach David F. Bacon Ioana Baldini
Perry Cheng Stephen J. Fink Rodric M. Rabbah
IBM T. J. Watson Research Center
Yorktown Heights, NY USA
{ealtman,josh,bacon,ioana,perry,sjfink,rabbah}@us.ibm.com

***Abstract*—This paper describes the Liquid Metal entry in the 2013 ICFPT Design Competition. The Liquid Metal system provides a high-level language called Lime and a toolchain targeting FPGAs. Lime allowed us to use standard software development processes for programming, debugging, and performance tuning our FPGA design. We believe such iteration and refinement are far more challenging with low-level languages and design tools commonly used for FPGA development.**

I. Introduction

The vast majority of hardware designers employ design languages such as VHDL and Verilog, which provide a lower level of abstraction than languages used for software development. Using such tools, FPGA design requires much more time, effort, and expertise than programming equivalent functionality in software.

To make hardware design easier, many projects have investigated hardware synthesis from higher-level languages; see [1] for a recent survey. To evaluate whether or not high-level synthesis is viable, we must answer a key question: *Using a high-level language, can a developer design hardware whose quality matches hardware designed with standard tools?*

A design competition provides an attractive laboratory to test the question for a particular challenge. To this end, we have developed an entry for the 2013 ICFPT Blokus Duo Design Competition [2], described in this paper, built with the Liquid Metal system from IBM Research.

The Liquid Metal system provides a high-level language and toolchain targeting heterogeneous systems which mix CPUs, GPUs, and FPGAs. Liquid Metal is based on Lime [3], a Java-like language enhanced with constructs that express parallelism and isolation.

II. Background

Many previous publications (*e.g.* [4], [3]) describe various aspects of Liquid Metal, which provides a language (Lime) and toolchain for programming heterogeneous systems. In this section, we briefly review aspects of Liquid Metal germane to FPGA design.

The Liquid Metal system provides the Lime programming language with its compiler and runtime system, an Eclipse-based integrated development environment (Figure 2), and a built-in integrated build system (Limeforge) that drives vendor synthesis tools remotely on a separately managed server.

Lime extends Java with constructs to express parallelism, isolation, bounded types, and stream dataflow. A Lime program

```
1  public interface Player {
2      /** Returns some descriptive name */
3      public string getName();
4
5      /** Resets the state of the game to g */
6      public void reset(Game g);
7
8      /** Computes the next move to play */
9      public Move nextMove(Move m);
10 }
```

Fig. 1. Blokus Player interface

can run on any Java Virtual Machine, so the programmer can develop and debug on a standard workstation with no specialized hardware. The compiler can translate all Lime code to JVM bytecode, and can additionally compile a subset of the language to Verilog for execution on an FPGA or simulator.

Lime encourages an incremental development path starting from Java:

1) Prototype the program in Java,
2) Add Lime constructs to express types and invariants which allow the compiler to generate Verilog for selected components,
3) Deploy the program with a JVM and a Verilog simulator for performance tuning, and
4) Synthesize a bitfile and deploy on an FPGA.

In the case of the FPGA design competition, the entire Lime Blokus program must run on the FPGA. Concretely, in Lime, the Blokus player is a stateful object that resides on the FPGA, and implements the `Player` interface shown in Figure 1.

For testing purposes, a Lime driver runs on the host JVM and calls the `Player` object, letting the Lime implementation transparently manage communication between host and device. For deployment in the contest, the design includes a small hand-written Verilog module which mediates communication between a UART interface and the generated Lime object, as per the contest communication protocol. Lime supports this via a foreign function interface called the Lime Native Interface.

While developing our Blokus entry, we spent almost all effort programming and debugging entirely on a laptop using a JVM. We tuned algorithms and parameters over several months, gradually improving the player strength by incorporating stronger ideas. Once we arrived at a candidate implementation, we spent some time tuning with a simulator to increase speed, and a little effort tuning data structures to reduce

Require: T is a tree with one node, the current game state
Require: B is an evaluation budget, an integer
Ensure: T is a tree of states, each representing a sequence of moves
  $level \leftarrow 0$
  $b \leftarrow 0$
  repeat
    $level \leftarrow level + 1$
    $j \leftarrow$ number of leaves in T
    $w \leftarrow width(level)$ (*)
    for each leaf l_j of T **do**
      $k \leftarrow$ number of legal moves in state l_j
      $M = \{m_1, \ldots, m_k\} \leftarrow$ each legal move from state l_j
      $b = b + k$
      $\{n_1, \ldots, n_w\} \leftarrow$ best w moves in M as per eval fn.
      for $i = 0$ to w **do**
        $s \leftarrow$ extend state l_j with move n_i
        Add s to T as a leaf of l_j
      end for
    end for
  until $b > B$

Fig. 3. Pseudo-code for search tree construction

resource constraints based on synthesis feedback. We tested extensively on the final hardware deployment, but needed no hardware debugging – by construction, the synthesized hardware behaves identically to compiled JVM bytecode.

Overall, using Lime allowed us to incrementally refine the algorithms and data structures over time using standard software development processes. Such iteration and refinement is much more difficult with low-level design tools.

III. Algorithms

Blokus is a turn-based strategy game, somewhat akin to Tetris and Go. Due to space constraints, we do not review the rules of Blokus here; we refer the reader to [2] for the contest rules.

Our Blokus player uses a minimax algorithm based on a board evaluation function. The evaluation function scores a board position based entirely on piece placement, without lookahead. The minimax search uses the evaluation function as a subroutine when exploring future moves.

As usual, the minimax algorithm operates over a search tree of candidate moves. Figure 3 shows pseudo-code for the search tree construction.

To build a search tree from a board state, the program examines every legal move, and computes a score for each move as per the evaluation function. Having evaluated each possible move, the algorithm picks the top w moves, and populates a search tree with w branches. Then the process iterates, expanding the search tree level-by-level, picking the "best" moves for each player in turn. The iteration proceeds until a global budget B is reached, where each board evaluation comprises one unit of work.

As just described, we limit the search based on a budget B measured in *evaluations*, which is a deterministic measure even when running the algorithm in software. For the final contest implementation, we will likely instead impose a search budget based on time and the one second response deadline. This change will allow a more aggressive search, but sacrifices the property that testing in software and hardware behave identically.

```
1  public typedef Row = int<16 * 1024>;
2  public local static boolean diffuse(Row[14] open,
       Row[14] grid)) {
3    boolean changed = false;
4    Row upper = 0;
5    Row mid = grid[0];
6    for (int<14> i){
7      Row lower = (i == 13) ? 0n : grid[i+1];
8      Row mark = (upper | lower | (mid << 1) | (mid
         >>> 1)) & ~mid & open[i];
9      changed |= mark != 0;
10     grid[i] |= mark;
11     upper = mid;
12     mid = lower;
13   }
14   return changed;
15 }
```

Fig. 4. Lime code for inner loop of board evaluation function.

Note that the search tree construction populates each level of the tree with a particular number of positions, indicated by the *width* function marked as (*) in Figure 3. The *width* function is a tunable parameter which determines the search tree breadth.

We have experimented with various parameters, and currently use a "decreasing width" heuristic – the search seeds the tree with a modest number of initial moves, and the number of children added per state decreases as the search deepens. After a few moves of lookahead, the width parameter decays to 1 – at subsequent levels, the tree only considers the *best* move from each position. Thus, the search tree reaches maximum width, then explores deeper without further search space expansion. This allows the implementation to statically allocate a data structure large enough to hold the maximum possible search tree, obviating the need for dynamic memory management.

Our current evaluation heuristic computes an approximation of the territory controlled by each player, similar in spirit to influence computations used for computer Go [5]. In the evaluation function, a player "influences" squares which it can immediately occupy with a one-square piece. Additionally, influence propagates transitively over a finite neighborhood, where influence can flow from one square to a Manhattan neighbor, subject to constraints that ensure the player might occupy the neighbor square sometime in the future.

The evaluation function thus performs a limited 5-point stencil relaxation over grids of bits which encode influence and future occupancy possibilities. The stencil calculation computes a 14x14 grid representing squares influenced by a player. The key operation in the implementation updates a single row of the grid using bit vector operations over 3 successive rows. Figure 4 shows the Lime code that implements the inner loop of the stencil.

IV. Data Structures

The Lime code follows a typical object-oriented design, with classes that correspond to concepts in Blokus such as *color*, *square*, *piece*, *move* and *board*. Figure 5 shows brief excerpts from the code, to give a flavor of the implementation.

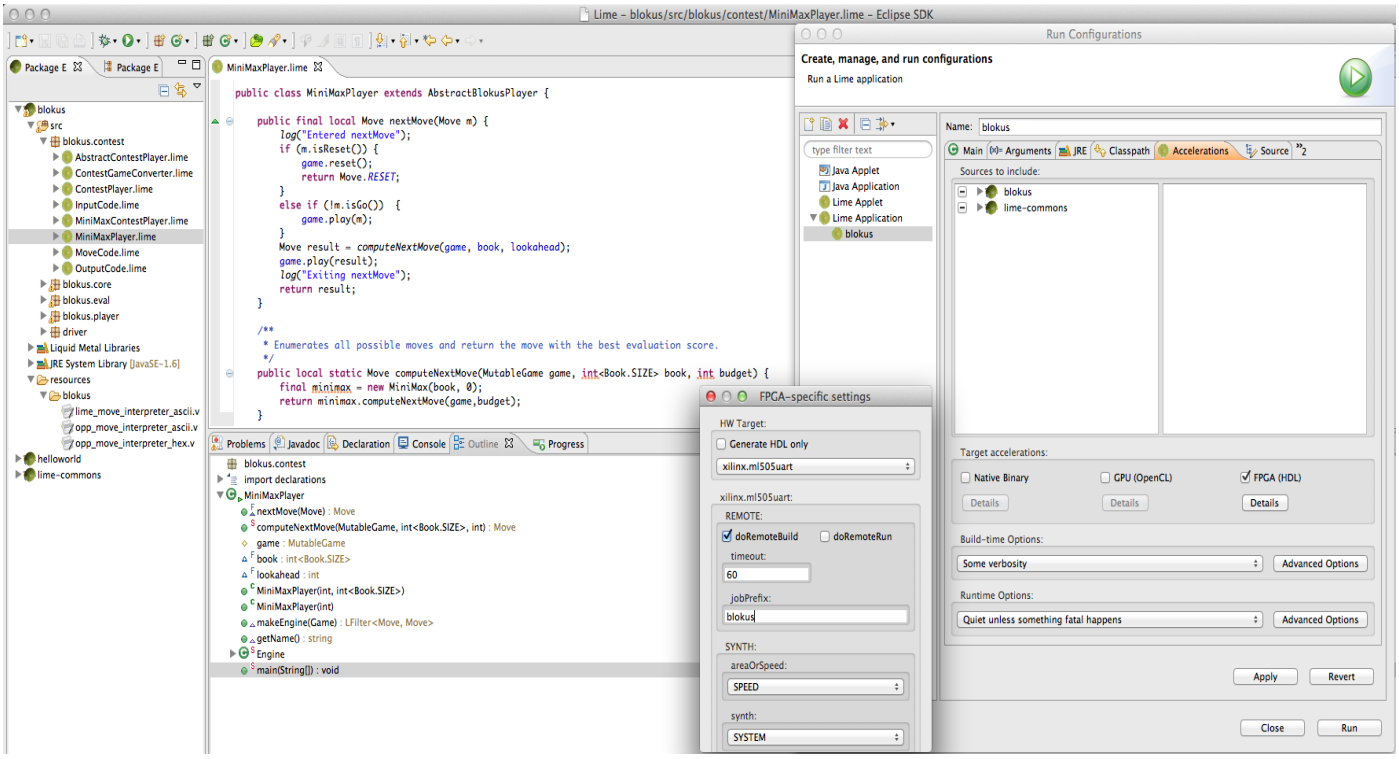

Fig. 2. Screenshot from the Liquid Metal IDE

The Lime `value` type qualifier indicates a class of immutable objects. In the Figure, the `Color` enumeration (line 1) and the `Square` class (line 2) are marked as values, which allows the compiler to aggressively optimize their implementation without aliasing complications.

A `Square` represents a square on the Blokus board, and consists of two values of type `int<14>` (lines 3,4); `int<14>` indicates an integer constrained to the range $[0, 13]$, inclusive. With this type constraint, the implementation represents a `Square` with 8 bits, four for each field. As typical in Java, the program defines instance methods on `Squares` (*e.g.* line 7), which allows a relatively high level of abstraction when programming application logic using the type (*e.g.* method `sharesCorner`, line 11), without any loss of efficiency in the implementation. For example, the compiler can translate the entire `sharesCorner` method into a (potentially, subject to timing constraints) single-cycle combinatorial expression in Verilog, and the expression is inlined into all calling contexts.

Although expressed in a superficially object-oriented style, the code does not rely on runtime polymorphism – the compiler can resolve all concrete types and dispatch targets statically.

We believe that most software developers would find this Java-like programming style more accessible and intuitive than standard alternatives such as Verilog and VHDL.

V. Implementation

Our contest entry runs on a ML505 board, with a Xilinx Virtex-5 LX50T FPGA consisting of 7200 slices, 48 DSP48E slices, and 2160 Kb of BRAM [6].

The Lime compiler compiles the entire Blokus player to Verilog, which is then synthesized using the Xilinx ISE 14.5 toolchain.

The compiler performs many aggressive optimizations, including inlining, classical dataflow optimizations, aggressive scalar replacement, scheduling and resource allocation, range and bitwidth analysis, strength reduction, if-conversion, loop optimizations, and branch optimizations. The bulk of the user logic is generated into a single large finite state machine, which mirrors the (interprocedural) control-flow graph after optimizations, adjusted for resource and critical path constraints. The compiler maps arrays to BRAMs or distributed RAMs; the design does not utilize on-board SRAM or DRAM.

Previous publications [4], [7] have described Lime constructs to express coarse-grain parallel structures, such as stream graphs and data-parallel loops. Currently, the Blokus code does not use explicitly parallel constructs – the code embodies a single, monolithic thread of control. The compiler extracts and implements fine-grain parallelism within a basic block based on dependency analysis, and endeavors to form large basic blocks with techniques similar to if-conversion [8].

We currently set the compiler options to generate a design for 85 MHz. Table I lists some statistics from the synthesized design as of this writing.

We believe the maximum frequency is currently limited by routing delays involving BRAM signals corresponding to the search tree, and multiplexers arising from the large finite state machine.

As described earlier, the player is constrained to a fixed budget of board evaluations, chosen as a parameter to fit within

```
public value enum Color { NONE, B, W; }
public value class Square {
  public final int<14> x;
  public final int<14> y;
  public Square(int<14> x, int<14> y) { this.x = x; this.y = y; }
  ....
  public boolean onNorthBorder() { return x == 0; }
  /**
   * Do two Squares share a corner?
   */
  public static boolean sharesCorner(Square sq1, Square y) {
    if (!sq1.onNorthBorder() && !sq1.onEastBorder() && sq1.northeast() == y) { return true; }
    if (!sq1.onNorthBorder() && !sq1.onWestBorder() && sq1.northwest() == y) { return true; }
    if (!sq1.onSouthBorder() && !sq1.onEastBorder() && sq1.southeast() == y) { return true; }
    if (!sq1.onSouthBorder() && !sq1.onWestBorder() && sq1.southwest() == y) { return true; }
    return false;
  }
}
public final value class PieceInfo  {
  typedef Block = '(int<5>,int<5>);
  ...
  // piece names are standard:  http://c2strategy.wordpress.com/2011/04/10/piece-names/
  // but the precise cells are dictated by: http://lut.eee.u-ryukyu.ac.jp/dc13/rules.html.
  final static PieceInfo ONE = new PieceInfo(new Block[[4]] { C,     C,     C,  C });
  final static PieceInfo TWO = new PieceInfo(new Block[[4]] { SOUTH, C,     C,  C });
  final static PieceInfo I3  = new PieceInfo(new Block[[4]] { NORTH, SOUTH, C,  C });
  final static PieceInfo V3  = new PieceInfo(new Block[[4]] { NORTH, EAST,  C,  C });
  final static PieceInfo I4  = new PieceInfo(new Block[[4]] { NORTH, SOUTH, SS, C });
  final static PieceInfo L4  = new PieceInfo(new Block[[4]] { NORTH, SOUTH, SW, C });
  ....
}
/**
 * Represents the placement of a piece on the board
 */
public final value class Move {
  public Color color;
  public Piece piece;
  public Transform transform;
  public Square center;
  ....
}
```

Fig. 5. Lime code excerpts

Lime Lines of Code	4,231
Lines of Verilog generated	24,657
Lines of Hand-Written Verilog	186
Frequency	85 MHz
LUTs	15917 (55.3% of device)
Flip Flops	11050 (38.4% of device)
18Kb BRAMs	65 (54.2% of device)
DSPs	14 (29.2% of device)

TABLE I. STATISTICS DESCRIBING THE SYNTHESIZED DESIGN

the one second maximum response time as per the contest rules. Currently a single board evaluation takes roughly 8000 cycles including associated bookkeeping, and we budget 5000 evaluations per move. These parameters may change as we tune the algorithms and compiler.

VI. RESULTS

We look forward to the competition to learn the results of this experiment. We thank the contest organizers for their hard work, and for the opportunity to participate.

REFERENCES

[1] D. F. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Commun. ACM*, vol. 56, no. 4, pp. 56–63, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2436256.2436271

[2] "ICFPT 2013 design competition." [Online]. Available: http://lut.eee.u-ryukyu.ac.jp/dc13/

[3] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a Java-compatible and synthesizable language for heterogeneous architectures," in *OOPSLA*, Oct. 2010, pp. 89–108. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869469

[4] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla, "A compiler and runtime for heterogeneous computing," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 271–276. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228411

[5] B. Bouzy and T. Cazenave, "Computer Go: An AI-oriented survey," *Artificial Intelligence*, vol. 132, no. 1, pp. 39 – 103, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370201001278

[6] Xilinx, "Virtex-5 family overview." [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[7] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a high-level language for GPUs: (via language support for architectures and compilers)," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254066

[8] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1983, pp. 177–189. [Online]. Available: http://doi.acm.org/10.1145/567067.567085