Generating Hardware Designs by Source Code Transformation

Ashley W Brown, Wayne Luk and Paul H J Kelly
Department of Computing, Imperial College London
Email: {ashley.brown,p.kelly,w.luk}@imperial.ac.uk

***Abstract*— Current and potential users for field-programmable gate arrays (FPGAs) are increasingly looking to high-level languages as a means to widen applicability and cope with ever-increasing transistor counts.**

We present a transformation system for one such high-level language for electronic circuit design, which goes some way towards bridging the growing gulf between the domain and architecture experts. We demonstrate its effectiveness by realistic, albeit relatively small, case studies; performance improvements of up to 70% have been achieved.

I. Introduction

We present CML [1], a user-customisable transformation system for a high-level hardware description language.

CML provides three main contributions to high-level hardware design. Architecture experts, working separately from domain experts, can:

1) define transformations for the automatic derivation of efficient hardware designs from a naive C-based specification;
2) encapsulate knowledge of restructuring strategies as rewrite rules which can selectively be used to refine naive code;
3) automatically apply transformations in a number of combinations to find the solution with the best performance.

CML's core language design is based on Boekhold's CTT [2], using the hardware language Cobble [3], a Handel-C [4]-based language, as the subject for transformation.

Demonstration transformations for path shortening, code unwinding and automatic parallelisation with or without hardware duplication are shown to produce execution time improvements of 35-70%.

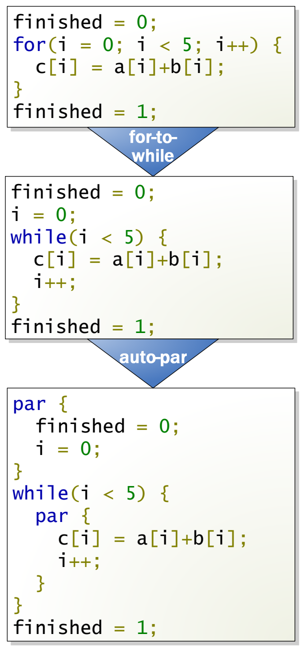
II. Application Area

High-level language to hardware design technology provides a faster development model than hand-crafting digital systems. The ultimate aim is to take software code and produce an efficient digital system design.

Celoxica's Handel-C [4] language is an example of this technology, allowing the specification of hardware designs in a C-like syntax. The language removes some features of ANSI C, such as side-effects, but adds constructs for explicit statement-level parallelism. Strict timing semantics constrain the optimisations available to the compiler; each statement is defined to take one cycle to execute.

Electronic circuit design requires fine control over parallelism to provide the best performance. "Best performance" need not necessarily be speed – power is an important performance measurement too. At the same time, users from diverse areas unfamiliar with low-level hardware details are moving into the FPGA arena, for example to accelerate scientific applications. Separating domain-specific knowledge from architecture-specific knowledge is especially useful under these conditions.

Figure 1 shows the application of two transformations to a naive piece of Handel-C code, illustating how a simple code fragment becomes more complicated in the pursuit of performance.


```
finished = 0;
for(i = 0; i < 5; i++) {
  c[i] = a[i]+b[i];
}
finished = 1;
```

```
finished = 0;
i = 0;
while(i < 5) {
  c[i] = a[i]+b[i];
  i++;
}
finished = 1;
```

```
par {
  finished = 0;
  i = 0;
}
while(i < 5) {
  par {
    c[i] = a[i]+b[i];
    i++;
  }
}
finished = 1;
```

Fig. 1. CML Transformation Steps: 1) Original Handel-C code; 2) After for loops converted to while loops; 3) After automatic parallelisation.

III. The CML Language: Example

A CML transformation consists of a rewrite rule defined using fragments of the Cobble language. The left-hand side of this rule, "pattern", matches an AST fragment and the right-hand side of the rewrite rule, "generate", provides a replacement code fragment which is inserted into the AST at the match point. An optional "conditions" block specifies when the rewrite rule is valid. The wildcards "cmlstmt" and "cmlstmtlist" match, respectively, a statement or list of statements, binding them to a name.

The following code fragment defines a transformation to execute non-dependent statements in parallel:

```
transform auto_par {
  pattern { // find two consecutive statements
    cmlstmtlist(preamble);
    cmlstmt(par1);
    cmlstmt(par2);
    cmlstmtlist(postamble);
  }
  generate {
    cmlstmtlist(preamble);
    par {  // place statements in parallel
      cmlstmt(par1);
      cmlstmt(par2);
    }
    cmlstmtlist(postamble);
  }
  conditions {
    // don't assign to the same place
    defs(cmlstmt(par1)) & defs(cmlstmt(par2)) == {};
    // second statement not waiting on first
    defs(cmlstmt(par1)) & uses(cmlstmt(par2)) == {};
  }
}
```

IV. Results

We present an illustrative set of results showing the effect of transformations applied to a naive matrix-multiply implementation. At each stage an extra transformation from the following is added to the available pool:

- **autopar** - as defined above, run adjacent independent statements concurrently.
- **fortowhile** - for loops are converted to while loops, providing opportunities for parallelisation.
- **lttoeq** - for-loop conditions using "less-than" comparisons are converted to "equal-to" comparisons, where the loop counter allows this.
- **matrixpar** - a specific area of the matrix multiplication code is specifically targeted for pipelining – the transformation is not re-usable.

Figure 2 demonstrates the decrease in execution time as successive transformations are added to the transformation database. Generic transformations can produce 30-40% speedup, with application targeted ones adding an additional 30%.

V. Finding the Best Solution

A. Interactions

We were able to illustrate the effectiveness of small transformations applied in composition, instead of a larger complex transformation, which is harder to verify as correct.

Also of note was the behaviour of transformations on differing hardware platforms, with transformations which improve the maximum clock rate on one architecture harming it on another.

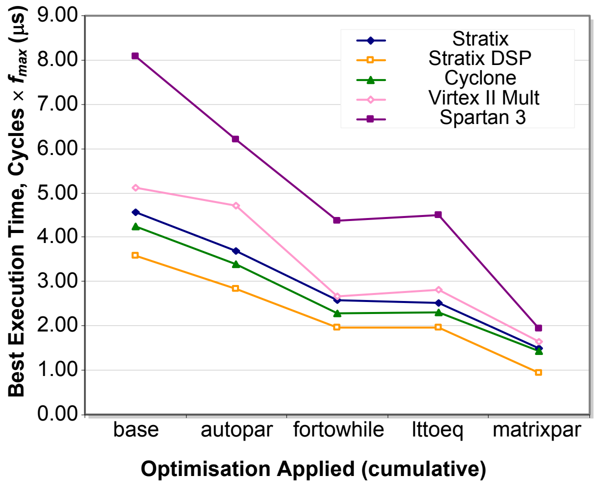


Fig. 2. Matrix Multiply Best Execution Time.

B. Design Space Exploration

Unpredictability of optimisations on FPGA performance has caused the adoption of "design-space exploration", the automatic generation and compilation of several circuit designs to find the fastest.

Cobble-CML supports a design-space exploration mode which generates a range of solutions, whilst taking steps to manage the combinatorial explosion of results. Transformations are effectively applied in all valid combinations to produce a range of solutions from which the best performing is found.

VI. Conclusion

This simple design provides performance improvements, though complex restructuring is difficult. Transformations can be prototyped in a development environment, then re-used in automatic design-space exploration.

More complex transformations involve sophisticated analysis, for example to select how arrays can be mapped onto registers, RAMs and shift registers. Integration of more sophisticated synthesis and optimisation algorithms is needed, for example for scheduling onto finite resources.

The issue of strategy is an important, but difficult one. Currently all possible orderings of transformations are tried using design space exploration. A strategy for the directed application of transformations is challenging due to the unpredictability of the optimisations across different platforms.

References

[1] A. Brown, "Optimising transformations for hardware compilation," Master's thesis, Department of Computing, Imperial College London, 2005.

[2] M. Boekhold, I. Karkowski, H. Corporaal, and A. G. M. Cilio, "A programmable ANSI C transformation engine," in *Proceedings of the 8th International Conference on Compiler Construction*. Springer-Verlag, 1999, pp. 292–295.

[3] T. Todman, "A customisable framework for hardware compilation," Ph.D. dissertation, Department of Computing, Imperial College London, 2002.

[4] Celoxica, "Handel-C language reference manual," http://www.celoxica.com/techlib/files/CEL-W0410251JJ4-60.pdf.