Annotated Bibliography
Harpoon Project
C. Scott Ananian
June 10, 1999

References


of C++ code, but light on optimization results. We really want to peek at Cliff Click’s PhD thesis. [Ed note: I found it!].


[16] Vugranam C. Sreedhar, Guang R. Gao, and Yong-fong Lee. Incremental computation of dominator trees. ACM Transactions on Programming Languages and Systems, 19(2):239–252, March 1997. Seems to work well, but I don’t think we need incremental updates for Harpoon. Maybe I’m wrong. Very similar to their other papers on this topic. ¡Wry grin¿.


[18] Greg DeDouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL), pages 222–246, San Diego, California, January 1998. This basically tells us that class/type analysis will be very slow. We can make it faster, but we lose much precision. In hardware compilation we don’t care so much about speed, perhaps, so maybe we just suck up and deal. Return to this paper before implementing our type inference engine. [theory practice conflict: the idea I’ve got in my head doesn’t seem as slow as the fastest algorithm presented here. Reconcile this.] [Further thought: these algorithms look like they should be integrated with an (SSA?) constant-propagation/dead-code analysis to come up with the most precise class information possible. Will be slow, but worth it. This paper’s optimizations result in less class/type precision, which is unacceptable to us.].

Implementation (PLDI), pages 246–257, La Jolla, California, June 1995. Simple and works, but relies on heuristic to place, and seems overly wedded to basic blocks (which we want to eliminate). I like SSAPRE better, even though it is *much* more complicated. Reassessment: Maybe basic blocks aren’t so bad.


[22] Cliff Click. Combining Analyses, Combining Optimizations. PhD thesis, Rice University, February 1995. Lots of very useful optimization information, plus a very powerful optimistic analysis algorithm, which I plan to extend into our ‘extended type analysis’ pass.


NY 14853-7501, September 1990. interesting variation on ‘dependence-based program analysis’; defines a formal executable semantics for a DFG-like program representation, with special loop constructs.


[30] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. ACM Transactions on Programming Languages and Systems, 17(1):85–122, January 1995. Very powerful technique for induction variable analysis using a variant of the SSA form. Requires a symbolic math package for full power, but can accomplish classification easily. mu and nu nodes are used in addition to phi nodes; they might be worthwhile subclasses. Min/max bounds can often be computed. Useful useful stuff. Reference to paper on ‘loop distribution’ and ‘loop interchanging’ that might be worthwhile to track down. Implementable algorithms.


5
[34] David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, July 1997. Not useful at all. A waste of time. Although some of the ideas of mapping methods to events are similar to our approach.


[36] Valeria Bertacco and Maurizio Damiani. The disjunctive decomposition of logic functions. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 78–82, San Jose, California, November 1997. Computationally very expensive. Enables creation of a compact, canonical multi-level circuit directly from a BDD representation, which is a Good Thing. "We found the final netlist to be often close to the output of more complex dedicated optimization tools." Probably worth implementing, or at least researching further. BIG DRAWBACK: "For the largest benchmarks, the limited set of BDD transformations... do not compensate for the exceptional growth of the BDD representation with respect to the original representation." I wish I knew what the "original representation" was.


[39] Srinivasa R. Arikat and Ravi Varadarajan. A signature based approach to regularity extraction. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 542–545, San Jose, California, November 1997. The signature idea is a good one. Not enough info for an implementation, though. And our regularity information should come from a much higher level. Might be worthwhile referencing to prove that logic synthesis folk spend a lot of effort reconstructing the high-level patterns thrown away by the compiler. Not otherwise useful.
[40] Morgan Enos, Scott Hauck, and Majid Sarrafzadeh. Replication for logic bipartitioning. In Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD), pages 342–349, San Jose, California, November 1997. We can create a better logic partition if we are allowed to duplicate some logic. Good idea and good overview of partitioning methods. VERY USEFUL if we need to partition. We want to use the Strawman algorithm, apparently; we need to look up papers on it.

[41] Charles E. Leiserson, Flavio M. Rose, and James B. Saxe. Optimizing synchronous circuitry by retiming. In R. Bryant, editor, Third Caltech Conference on Very Large Scale Integration, pages 87–116, Pasadena, California, March 1983. Three good references on other optimization types. This paper describes how to optimize register placement to obtain the smallest possible clock cycle time. *DON’T HAVE COMPLETE PAPER*. Useful. Includes algorithms to compute the clock period of a circuit, among other things.

[42] Sharad Malik, Ellen M. Sentovich, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential networks with combinational techniques. IEEE Transactions on Computer-Aided Design, 10(1):74–84, January 1991. Useful extension of Leiserson’s Retiming paper. Perhaps more information can be found in malik93? Basic idea is that we can push registers to the periphery and disconnect cycles, optimize the resulting circuit, and then push all the registers back in and reconnect the feedback loops. Makes sense. We still have to find a valid combinational optimization technique to plug in the middle, though.


[44] Madhav Y. Chikodikar, Shridhar Laddha, and Ashish Sirasao. A technology mapper for Xilinx FPGAs. In Proceedings. Tenth International Conference on VLSI Design, pages 57–61, Hyderabad, India, January 1997. "This paper presents a method for area optimal technology mapping for Xilinx FPGAs." Maps logic onto CLBs; 10% better than the Xilinx method. If we ever need to do this, this is the method to use. [Hopefully we can use their code!]


[49] Scott Hauck. Asynchronous design methodologies: An overview. Technical Report TR-93-05-07, University of Washington, May 1993. Very good overview of state of the art in asynchronous design. As of 1993, there were significant problems with each async design methodology presented. Is there recent work overcoming some of these difficulties?


[68] Ron K. Cytron and Jeanne Ferrante. Efficiently computing $\phi$-nodes on-the-fly. *ACM Transactions on Programming Languages and Systems*, 17(3):487–506, May 1995. Contains: “Since one reason for introducing $\phi$-functions is to eliminate potentially quadratic behavior when solving actual data flow problems, such worst-case behavior during SEG or SSA construction could be problematic. Clearly, avoiding such behavior necessitates placing $\phi$-functions without computing or using dominance frontiers.”.


