# High-level Synthesis from the Synchronous Language Esterel

Stephen A. Edwards*
Department of Computer Science
Columbia University, New York

**Abstract**

Producing efficient circuits from high-level language descriptions remains a problem. This paper proposes three techniques for improving the quality of circuits generated from high-level Esterel specifications, a synchronous, concurrent language designed to specify control-dominated systems.

Together, the three techniques aim to improve the quality of the input to logic synthesis to produce better results. The first uses control dependence information to synthesize small, fast circuits from control-flow graphs. The second involves solving the state assignment problem for Esterel-derived circuits at a much higher level than has previously been proposed, allowing many more optimization opportunities. The third technique extracts don't-care information from high-level representations to improve the quality of logic synthesis.

I believe these techniques will make Esterel a compelling alternative to existing RTL synthesis languages.

## 1 Introduction

I present three ideas for synthesizing circuits from Esterel [2] programs. While resembling the problem of generating circuits from Verilog or VHDL RTL descriptions the Esterel language differs in two main ways. First, Esterel supports implicit state machines through a *pause* statement that delays for a cycle. Thus, Esterel program counters hold their state between clock cycles, and in this sense resemble behavioral descriptions. Figures 1 and 2 illustrate this. Second, Esterel supports high-level control constructs such as concurrent composition, preemption, and exceptions. Both aspects make Esterel a more challenging language to translate into circuitry, but also enable aggressive optimizations because the compiler is able to gain a better understanding of the program's behavior. This paper focuses on this aspect of compiling Esterel.

Berry first outlined the translation of Esterel into circuitry in 1992 [1] and little has changed since. Analyzing and resynthesizing cyclic combinational circuits, which are easy to inadvertently write in Esterel, has been the major improvement. Starting with Malik's work [13], Berry et al. devised the costly but elegant symbolic techniques used in the V5 compiler [17]. This remains the best way known for dealing with this problem and could be applied in this work.

```
always @(posedge clk)
  HY = 0; FG = 0; FY = 0; HG = 0; RESTART = 0;
  case (state)
    0: if ( CAR && LONG ) begin
         HY = 1; RESTART = 1; state = 1; end
    1: if (SHORT) begin
         FG = 1; RESTART = 1; state = 2; end
    2: if (!CAR || LONG) begin
         FY = 1; RESTART = 1; state = 3; end
    3: if (SHORT) begin
         HG = 1; RESTART = 1; state = 0; end
  endcase
```

Figure 1: A fragment of register-transfer Verilog describing a traffic light controller. All state, including control state, must be explicitly saved in memory elements between clock cycles.

```
loop
  emit HG ; emit RESTART; await [CAR and LONG];
  emit HY ; emit RESTART; await SHORT;
  emit FG ; emit RESTART; await [not CAR or LONG];
  emit FY ; emit RESTART; await SHORT;
end
```

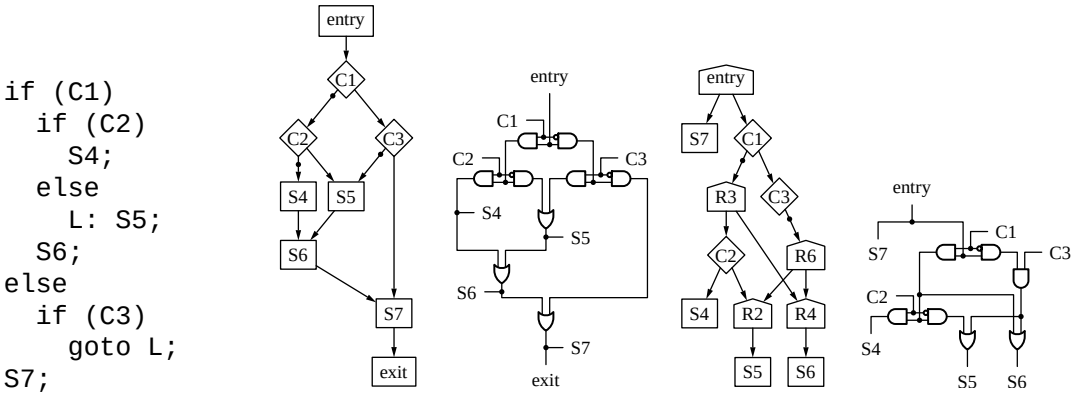
Figure 2: A fragment of Esterel code describing the same state machine as Figure 1. Control is now described using primitives such as *await* rather than being coded explicitly.

More relevant is Touati, Berry, Toma, and Sentovich's technique [20, 19, 16] for reducing the number of latches produced by Berry's mechanical translation procedure. They rely on computing the reachable state set implicitly using BDDs, then resynthesizing the circuit using this knowledge to remove sequential redundancies. They are able to improve the circuits because the group-hot encoding used by Berry's synthesis procedure is fairly inefficient.

Potop-Butucaru's work [14] is more closely related to this: it also reduces the number of latches in a circuit generated from Esterel, but does so much more cheaply. His technique eliminates latches that are identified as redundant by observing whether their thread can terminate independently of others in the group, an inexpensive computation already performed during the compilation process.

In this work, I propose three key ways to advance hardware synthesis from Esterel. First, I propose a new structural translation of Esterel based on the concept of control dependence. This representation of control flow, taken from the optimizing compiler literature [6], is ideally suited for hardware as it is inherently concurrent, compact, and generates wide, short graphs from tall, thin control flow specifications. Others,

*sedwards@cs.columbia.edu http://www.cs.columbia.edu/~sedwards




| Version | Init. Lit. | Opt. Lit. |
|---|---|---|
| CFG (c) | 18 | 11 |
| CDG (e) | 14 | 9 |

(f)

Figure 3: (a) A short procedure. (b) Its control-flow graph. Rectangles are actions; diamonds are decisions. Dots indicate true arcs. (c) A circuit generated from the CFG. (d) Its control dependence graph. Every successor of a region node (house-shaped) runs if the node does. (e) The circuit generated from the CDG. (f) The number of literals in each circuit before and after SIS optimization with script.rugged.

notably Chapman [4], have made this observation, but ironically have not applied it to a concurrent language such as Esterel, partially because control dependence has historically only been defined for sequential control flow.

The second advance is better state encoding. Berry's group-hot encoding is highly redundant (as evidenced by the success of Touati et al.), so there is room for improvement. However, rather than building a complicated gate-level representation and then fixing it (the current state-of-the-art), the technique proposed here choose states encodings at a high level, providing much greater flexibility and allowing drastically different encodings to be chosen.

The third advance is the extraction of don't-care information to improve the quality of logic synthesis. While in theory this information, which gives the logic synthesis tool more flexibility in choosing an implementation, can be inferred from the circuit, it is much less expensive to obtain from a high-level specification.

This paper describes these three techniques.

## 2 Control Dependence and Hardware

The synthesis procedure generates circuits from Esterel by first computing control dependence information, which has a natural, efficient translation into hardware. Ferrante et al. [10] invented the control-dependence graph as an intermediate representation for optimizing software compilers. It enables certain optimizations, especially those that require the reordering of statements. It is a concurrent representation of a sequential program represented with a control-flow graph.

A circuit generated from control dependence is smaller and faster than one translated directly from control flow. Figure 3 illustrates this for a small example from Ferrante et al. [10]. The sequential code in (a) has the control-flow graph in (b). Generating a circuit directly from this (using, for example, Berry's technique [1] produces the circuit in (c).

The circuit in Figure 3c is redundant because it contains many order dependencies that are not logically necessary. For example, S7 is always equal to the value of entry regardless of the values of C1, C2, and C3. Although such redundancy can be removed by carefully analyzing the circuit, it is much easier to remove by calculating control dependence.

Informally, a node $Y$ in a control-flow graph is control dependent on a node $X$ if taking one of the outgoing arcs from $X$ guarantees $Y$ will be executed, but that taking another arc may mean $Y$ is not executed. For example, in Figure 3b, S6 is control-dependent on C3 and C1, but not C2. Ferrante et al. [10] define control dependence in terms of dominance. A node $V$ in a control-flow graph is post-dominated by a node $W$ if every path from $V$ to the exit node of the CFG passes through $W$. With this definition, a node $Y$ is control dependent on $X$ if there is a directed path from $X$ to $Y$ whose nodes are all post-dominated by $Y$ but $X$ is not post-dominated by $Y$. Cytron et al. [6] give an efficient ($O(E+N^2)$ worst-case, usually linear) algorithm for calculating control dependence. For each node in the control-flow graph, they compute its dominance frontier: a set of nodes whose predecessors are dominated by the node, but that are not themselves dominated by the node. Cytron et al. show this definition is equivalent to Ferrante et al.'s and give an algorithm for computing it.

Figure 3d shows control dependence relationships among the nodes in Figure 3b summarized in a form called the control dependence graph, which is a convenient starting point for hardware. The CDG consists of the nodes in the CFG augmented with region nodes (the house-shaped nodes in Figure 3d) such that there is a directed path from node $X$ to node $Y$ if $Y$ is control dependent on $X$. Ferrante et al. [10] give an efficient algorithm for generating such graphs from control dependence (or equivalently, dominance frontier) information, which involves looking for subset containment relationships among control dependencies.

Generating a circuit from the CDG representation is trivial, as shown in Figure 3e. A decision node becomes a pair of AND gates and OR gates collect multiple arcs incident on a node. Region nodes simply become fanout (wires).

```
abort pause; emit e; pause; emit f when b;
abort pause; emit g; pause; emit h; pause; emit i when c;
abort pause; emit j; pause; emit k when d
```

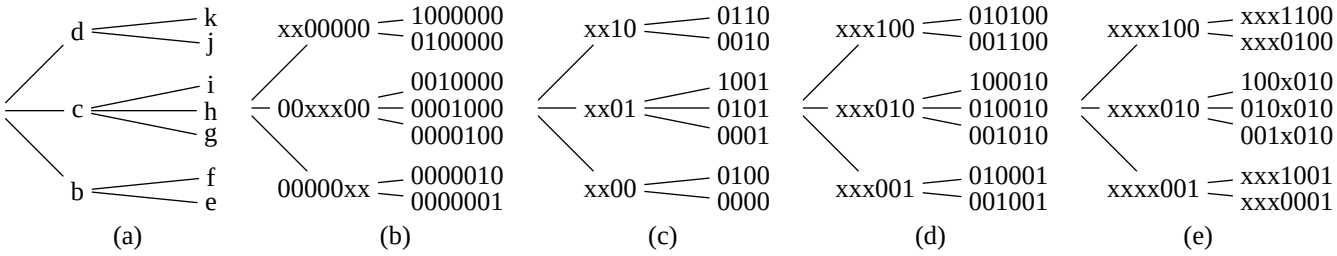



Figure 4: A fragment of Esterel code and (a) the tree to be encoded. (b) Berry's encoding. (c) The encoding used in my software Esterel compiler [7, 8]. The children at each branch are encoded in binary and shared maximally. (d) A variant: the branches are encoded with a one-hot code. (e) A redundant encoding: the b and d subtrees share a bit, but the c subtree has its own bits.

Figure 3f shows the literal counts for these two circuits before and after SIS optimization. Although this data set is hardly large enough to constitute proof, it does illustrate my hypothesis: *circuits generated from control dependence are a better starting point for logic synthesis.*

In theory, a logic synthesis system should produce a circuit of equal quality from either starting point, but in practice logic synthesis algorithms are always limited. It is unlikely that logic synthesis would use an algorithm as specific as control dependence, which relies on the control-flow-graph-like properties of the input circuit, since logic synthesis algorithms are designed to work well on general circuits. I therefore believe circuits generated from control dependence are a superior starting point that will produce smaller circuits after optimization.

The Esterel language has a natural translation into a control dependence graph because it combines sequential and concurrent constructs. This translation is not trivial; existing CDG computation algorithms have to be extended to accommodate some concurrency. Although there are some technical challenges, deriving a concurrent CFG (a starting point for calculating the CDG) is inexpensive and straightforward [8].

## 3 State Encoding

Existing optimization techniques for Esterel-derived circuits begin from a low-level netlist generated by Berry's procedure. This makes it difficult to significantly restructure the circuit. For example, changing state encoding is problematic because it is best done with knowledge of the circuit's reachable states, an expensive computation at the netlist level. This is the approach taken by Touati et al. [20, 19, 16], who remove and share redundant latches due to Berry's encoding. To determine this, they compute the reachable states using a symbolic BDD-based technique. Instead, I propose to address the state encoding problem at a higher level, before generating the netlist and without having to calculate reachable states.

Broadly, the state encoding problem in Esterel amounts to assigning codes to leaves of a tree that make it somehow easy to compute the path from the root to the leaf. Each sequential thread of control has such a tree, whose leaves corresponds to *pause* statements (a single-cycle delay: a state). Preemption statements such as *abort* generate the branches in the tree, and each branch has a condition that must be checked before control can reach a leaf. Each state of a thread, therefore, describes a path from the root to a leaf, and the circuitry for each node in the tree needs to determine whether it is along the path from the current state.

Figure 4 illustrates this. The top program fragment can be in one of seven states (at any of the *pause* statements), and each of these emits a distinct signal so they must be distinguished. Furthermore, these states are divided into three groups, each preemptible by a different input signal (b, c, or d), and these groups must be distinguished. The tree in Figure 4a illustrates this structure. The generated circuit must not only distinguish which leaf is active in each state (at most one ever is), but also the branches along the path from the root to that leaf.

The obvious code of assigning leaves to integers, e.g., e=0, f=1, g=2, etc. works, but generally does not produce the best circuitry. The test at each leaf becomes a comparison: an AND of XORs, but the test at each branch is a costly integer range check, e.g., the test at node c would be $2 \leq s \leq 4$.

Berry's encoding (Figure 4b) uses seven state bits. The decoding at the leaves is trivial: the leaf is active if its state bit is, but each non-leaf node must OR together the statuses for all its children, which can get expensive.

Figure 4c is the encoding I used in my Esterel compiler for software [7, 8]. Each branch is encoded with a $\log n$ code, which is easily decoded in software. It's more costly in hardware, but saves state bits. There are still many unused codes in this approach so there are further optimization opportunities.

Figure 4d also encodes branch points, but uses a one-hot code at each branch. Figure 4e illustrates yet another choice in this framework. Here, subtrees b and d share state bits, but subtree c has a separate set. Such an encoding might be desirable if subtrees b and d already had a lot of shared circuitry and subtree c required a lot of additional circuitry. By not sharing bits, latches holding the state for subtree c would not have to communicate with circuitry for subtrees b and d, possibly eliminating a long wire that might slow the circuit.

Selecting the best choice for state encoding among these possibilities is an open problem, but one possibility is to sim-

```
present A then
   emit B
else
   emit C
end;
present C then
   present B else
     emit D
   end
else
   present B then
     emit D
   end
end
```

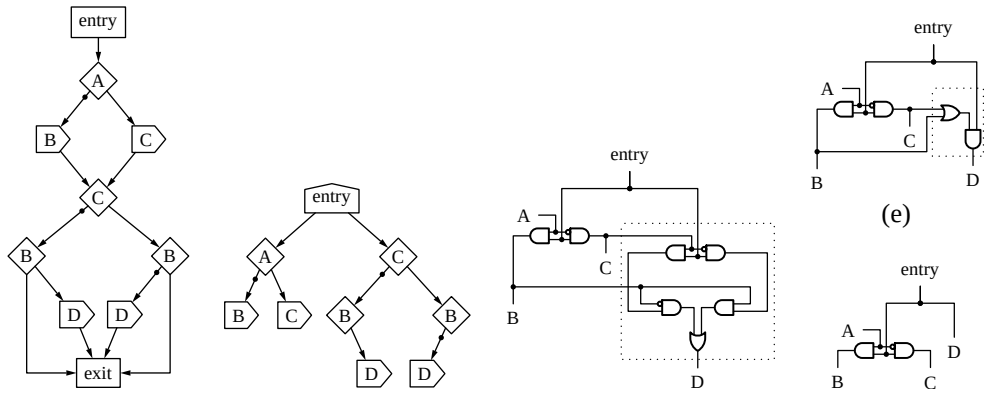



Figure 5: (a) An Esterel example. (b) Its control-flow graph. (c) Its control dependence graph. (d) The circuit generated from the CDG. (e) The circuit after simplifying the gates inside the dotted box using the observation that signals B and C are mutually exclusive, a "don't-care" derived from the control-flow graph. (f) Further simplification possible by observing the relationship between B, C, and the entry signal.

ply generate implementations and compare their sizes since the synthesis procedure will be fast. Estimating the quality of the optimized circuit from an unoptimized circuit may be difficult, but because the circuit will be reasonably good to start with (i.e., synthesized from control-dependence information), I expect this technique will be reasonable.

Other possibilities include doing more complete optimization on portions of the subtree, much like the technique of Seawright and Meyer [15]. Incorporating heuristics that consider communication dependencies like Crews and Brewer do [5] is another obvious possibility. I expect better results than theirs because the hierarchical state machines of Esterel are more structured than their starting point: state transition graphs.

## 4 Don't-Care Extraction

Allowing a logic synthesis system flexibility in its choice of implementation is key to producing high-quality circuits. This usually takes the form of "don't care" information that describes states the circuit can never enter. A logic synthesis system is therefore free to make the circuit's behavior in such a state match other, defined behavior, allowing the same circuitry to be reused.

Figure 5 illustrates how don't care information can help to simplify a circuit. Figure 5a shows an Esterel program fragment. This is translated into the control-flow graph of Figure 5b and the control-dependence graph of Figure 5c. Figure 5d shows a straightforward circuit translation from the PDG that can be improved by observing that signals B and C are mutually exclusive by virtue of their position in the control-flow graph (i.e., there is no directed path between emissions of B and C). This allows the gates in the dotted box to be replaced by a much simpler OR. Finally, the relationship between signals B, C, and entry can be exploited to further simplify the circuit (Figure 5f).

Don't-care information can be classified into two groups: internal don't-cares, which arise from the structure of the circuit; and external don't-cares, which arise from environmental constraints and expectations (I take this classification from Hong et al. [11], who also use don't-care information to simplify software synthesized from Esterel, but do so on programs generated from binary decision diagrams). For example, in Figure 5, the case when signals B and C are true simultaneously is an internal don't-care because the two AND gates than generate them guarantee this can never happen. How the rest of the circuit would react to this condition is irrelevant because it can never happen. There are few possibilities for external don't-cares in this circuit because it has so few primary inputs, but one might be that A is never true when entry is false.

There are two types of internal don't-cares: controllability and observability don't-cares. Signals B and C never true simultaneously is a controllability don't-care: the structure of the circuit ensures that this condition can never occur for any input pattern. That the C input to the dotted box in Figure 5d can be shorted to 1 without affecting the circuit's output is an example of an observability don't-care: internally the circuit behaves differently, but this difference cannot be observed at the outputs.

Deducing don't-care information from an arbitrary circuit is difficult and can be costly to do comprehensively, yet the optimization procedures of Touati et al. [20, 19, 16] demand it. Knowing how the circuit was constructed often makes certain properties true by construction. Furthermore, when establishing a property requires some analysis, that analysis is usually easier to carry out on the high-level representation that generated the circuit than on the circuit itself.

I propose two mechanisms for computing some don't-care inexpensively for these circuits. First, an overapproximation of the reachable states is available directly from the state encoding algorithms. For example, in Figure 4b, any state with two or more 1s is not reachable by construction. This supplies external don't-cares to the combinational portion of the circuit. Calculating this information directly from the circuit is theo-

retically possible but is often very expensive, requiring, e.g., symbolic state reachability analysis.

The other source of don't-care information comes directly from the structure of the CFG, as in Figure 5. By definition, two nodes within the CFG with no directed path between them cannot run simultaneously. This produces internal controllability don't-cares. Again, this information could be derived from the circuit, but doing so would be much more costly.

## 5 Conclusions

I have presented three ideas to advancing the state of the art in synthesizing hardware from high-level Esterel descriptions. First, I showed how control dependence information can lead to an efficient combinational circuit starting from a control-flow graph description. Second, I described the state encoding problem for Esterel and some ways to make improvements over existing techniques. And third, I showed how don't-care information can easily be derived from these high-level descriptions to improve the quality of the generated circuit after logic synthesis.

All these techniques are applicable to traditional Esterel [2], the new Hardware Esterel dialect [3], as well as proposed hardware dialects of the Esterel-derived ECL language [12]. Work is underway on implementing these in the ESUIF synchronous language compiler framework under development at Columbia University [9]. I am implementing ESUIF in Lam et al.'s SUIF system [21]. Smith et al.'s MachSUIF system [18] is being used for control-flow and control-dependence analysis.

### References


[1] Gérard Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London. Series A*, 339:87–103, April 1992. Issue 1652, Mechanized Reasoning and Hardware Design.

[2] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[3] Gérard Berry and M. Kishinevsky. Hardware esterel language extension proposal. Technical report, Esterel Technologies, August 2000.

[4] Richard Oliver Chapman. *Formally Verified High Level Synthesis*. PhD thesis, Cornell, 1994.

[5] Andrew Crews and Forrest Brewer. Shape-based sequential machine analysis. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, volume 1, pages 395–399, Orlando, Florida, May 1999.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[7] Stephen A. Edwards. Compiling Esterel into sequential code. In *Proceedings of the 37th Design Automation Conference*, pages 322–327, Los Angeles, California, June 2000.

[8] Stephen A. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2):169–183, February 2002.

[9] Stephen A. Edwards. ESUIF: An open Esterel compiler. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, volume 65.5 of *Electronic Notes in Theoretical Computer Science*, Grenoble, France, April 2002. Elsevier Science.

[10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[11] Youpyo Hong, Peter A. Beerel, Luciano Lavagno, and Ellen M. Sentovich. Don't care-based BDD minimization for embedded software. In *Proceedings of the 35th Design Automation Conference*, pages 506–509, San Francisco, California, June 1998.

[12] Luciano Lavagno and Ellen Sentovich. ECL: A specification environment for system-level design. In *Proceedings of the 36th Design Automation Conference*, pages 511–516, New Orleans, Louisana, June 1999.

[13] Sharad Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):950–956, July 1994.

[14] Dumitru Potop-Butucaru. Fast redundancy elimination using high-level structural information from Esterel. Technical Report RR-4330, INRIA, Sophia Antipolis, France, November 2001.

[15] Andrew Seawright and Wolfgang Meyer. Partitioning and optimizing controllers synthesized from hierarchical high-level descriptions. In *Proceedings of the 35th Design Automation Conference*, pages 770–775, San Francisco, California, June 1998.

[16] Ellen M. Sentovich, Horia Toma, and Gérard Berry. Efficient latch optimization using exclusive sets. In *Proceedings of the 34th Design Automation Conference*, pages 8–11, Anaheim, California, June 1997.

[17] Thomas R. Shiple, Gérard Berry, and Hervé Touati. Constructive analysis of cyclic circuits. In *Proceedings of the European Design and Test Conference*, pages 328–333, Paris, France, March 1996.

[18] Michael D. Smith. Extending SUIF for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Stanford, California, January 1996.

[19] Horia Toma, Ellen Sentovich, and Gérard Berry. Latch optimization in circuits generated from high-level descriptions. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 428–435, San Jose, California, November 1996.

[20] Hervé Touati and Gérard Berry. Optimized controller synthesis using Esterel. In *Proceedings of the International Workshop on Logic Synthesis (IWLS)*, Tahoe City, California, May 1993.

[21] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.