Concurrency-Aware Compiler Optimizations for Hardware Description Languages

KALYAN SALADI, University of California, Santa Cruz
HARIKUMAR SOMAKUMAR, Google, Inc.
MAHADEVAN GANAPATHI, Independent Consultant

In this article, we discuss the application of compiler technology for eliminating redundant computation in hardware simulation. We discuss how concurrency in hardware description languages (HDLs) presents opportunities for expression reuse across different threads. While accounting for discrete event simulation semantics, we extend the data flow analysis framework to concurrent threads. In this process, we introduce a rewriting scheme named ∂VF and a graph representation to model sensitivity relationships among threads. An algorithm for identifying common subexpressions as applied to HDLs is presented. Related issues, such as scheduling correctness, are also considered.

Categories and Subject Descriptors: 1.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete event*

General Terms: Design, Performance

Additional Key Words and Phrases: Common sub-expression elimination, VHDL Verilog Hardware Verification.

ACM Reference Format:
Saladi, K., Somakumar, H., and Ganapathi, M. 2012. Concurrency-aware compiler optimizations for hardware description languages. ACM Trans. Embedd. Comput. Syst. 18, 1, Article 10 (December 2012), 16 pages.
DOI = 10.1145/2390191.2390201 http://doi.acm.org/10.1145/2390191.2390201

1. INTRODUCTION

Verification is one of the most expensive tasks in the semiconductor development cycle. Simulation continues to be the primary method for verification of large circuits specified using hardware description languages (HDLs) such as VHDL or Verilog [IEEE 1996]. These languages rely on a discrete event simulation framework to accurately model circuit behavior. Any reduction in simulation time directly leads to productivity improvement in the design verification cycle. To this end, we explore a new modeling framework and an optimization to speed up simulation. The proposed framework provides a way to represent the concurrent execution semantics of HDL assignments such that it enables data flow analysis across concurrent threads of execution.

HDLs allow for specification of time delay associated with an event to model delays in a hardware circuit. HDL simulators are designed to implement the semantics of the language statements and the progression of time, enabling the designer to model a hardware circuit before it is physically created. A typical compiled code event-driven simulator, [Hansen 1988; Krishnaswamy and Banerjee 1998] has the following core components.

Authors' addresses: K. Saladi, School of Engineering, University of California, Santa Cruz, CA; email: skalyan@soe.ucsc.edu; H. Somakumar, Google Inc., Mountain View, CA; email: harks@acm.org; M. Ganapathi; email: gana@pacbell.net

DOI 10.1145/2390191.2390201 http://doi.acm.org/10.1145/2390191.2390201

(i) *Parser*. This module parses the HDL description and does syntax checking as well as semantic analysis. An abstract syntax tree (AST) corresponding to the HDL input is generated as a result of parsing.
(ii) *Elaborator*. A full design typically comprises of many components picked up from different libraries. An elaborator resolves the design hierarchy and the connections between components by gathering all design components from various libraries and creating the full design tree.
(iii) *Code Generator*. This module generates code (machine-specific or independent) from the AST, following the steps taken by a traditional compiler, including transformations to intermediate forms and optimizations. This machine code in association with a runtime simulation kernel produces the simulation result. The optimizations in the generated code determine, to alarge extent the speed of simulation.
(iv) *Simulation Kernel Library*. This library provides highly optimized implementation for time wheel and event lists used by the generated code. The generated code is typically linked with this library to produce the simulation executable.

In this article, we present an analysis and optimization framework performed in the code generator component of a compiled event-driven simulator.

Traditional compiler optimizations based on data flow analysis [Aho et al. 2007; Rosen 1979], work on sequential blocks of code with extensions to perform interprocedural analysis using a call graph. They are not capable of dealing with the concurrent execution semantics which include the event-based triggering mechanism across sequential blocks of code—a core feature in HDLs. Edwards et al. [2003], methods for compiling Verilog on sequential processors. To our best knowledge, no method has so far been proposed to model the deferred assignment (i.e., value change in the future) semantics—in compiler intermediate representation—of variable, as defined by HDLs. In this article, we provide a framework that enables dataflow analysis on HDL programs and enables optimization. In VHDL, the concurrent threads of execution (each of them consisting of sequential blocks of code) are called processes. We will use the term *process* to mean a concurrent thread of execution in HDL for the rest of this article.

To illustrate of the applicability of the framework just mentioned, above we define the availability of an expression across different processes as well as multiple invocations of the same process at different time steps or in the same time step separated by delta (i.e., an infinitesimally small amount of time) delays. This concept helps in identifying and eliminating redundant computation across processes.

The rest of the article is organized as follows. In section 2, we explain the event-driven simulation model, VHDL semantics, to establish the background. Sections 3, 4, and 5 introduce the concepts of levelized ordering of processes, the process sensitivity graph, and novel representation form called Delta-Value Form, respectively. We introduce two auxiliary concepts—Event Vector and Sensitivity Vector—in section 6, before detailing the core optimization algorithm in section 7. Examples, results, and discussion follow, in sections 8 and 9. Section 10 and 11 discuss future work and conclusions, respectively.

2. BACKGROUND: EVENT-DRIVEN SIMULATION MODEL

In the event-driven simulation model, the execution starts with a set of processes executing at time zero. Execution of these processes may result in events, which are scheduled after a nonzero time delay or a delta delay. We look at the simulation algorithm used in VHDL [IEEE 1994] to illustrate the concept of delta delay. The delta delay concept is applicable also to other HDLs, like Verilog and System C.

```
1. Initialize: Execute all processes at time zero.
2. T = 0; /* Simulation time */
3. loop
4. loop   /* begin ∂-cycle */
5. for each signal s in signal update (SU) list of time T;
     a. remove s from signal update list;
     b. if s->newvalue != s->value then
          c. s->value := s->newvalue;
          d. for each process p which is sensitive to event on s
               i. add p to the process execution (PE) list of time T;
          e. end for;
     f. end if
6. end for;
7. for each process p in the process execution (PE) list of time T;
     a. remove p from PE list;
     b. execute p;
8. end for;
9. while SU list of time T is not empty; /*end ∂-cycle*/
10. T = the nearest time in future which has a non-empty SU or PE list;
11. until (no non-empty SU list or PE list in future)
        or (T >= required simulation time);
```

Fig. 1. Simulation algorithm.

The processes are triggered by changes in value of special data elements called *signals*. Signals hold a new value as well as a current value. A write to a signal in VHDL updates the new value and posts the signal in a signal update queue for the current time or future time, [Willis and Siewiorek 1992]. The simulation kernel (described in the following) will copy the new value to the current value if they are different when it handles the signal update queue for the corresponding time. This phase is called the signal update, and the signal is said to produce an event if the new value is different from the current value. In simple terms, a change in the value of a signal creates an event at that time. It is useful to note that assignments to signal objects could be delayed by arbitrary amounts of time. Events on signals trigger processes, which are sensitive to them.

As assignment to a signal, even if it is a zero-delay assignment, does not immediately change the current value of the signal, since the new value is copied to the current value only when the kernel does a signal update. Any process execution preceding the next signal update phase will effectively read the relevant signal's current value. Thus, for zero-delay writes, we say that there is a delta delay between the signal write and signal update phases. The simulation algorithm is presented in Figure 1. Now, we introduce relevant simulation terms that are used throughout the article.

Event. A Boolean valued attribute of a signal object which is true if and only if the object's value changed in the current simulation cycle.

$$s.event \leftarrow true, \text{ iff } s.NewValue() \; != s.CurrValue().$$

Sensitivity. The set of signal objects which determines whether a process must be evaluated in the current simulation cycle. *For example, process P1 (s1, s2, s3),* implies that $P1$ must execute if at least one of *s1, s2,* and/or *s3* have an event occurrence in this cycle.

Driver. If a process P contains an assignment statement with a signal s on the LHS, then P is called a driver of s.

One execution of the inner loop (steps 4 to 9 in Figure 1) constitutes of one ∂-cycle. A process may be executed multiple times during a time step if the trigger signals become active across multiple ∂-cycles.

3. PROPOSED SOLUTION

As can be seen from the simulation algorithm, the wakeup semantics of processes combined with ∂-cycle-based evaluation may lead to multiple evaluations of an expression in different ∂-cycles for a given simulation time. Such a reevaluation, triggered by ∂-synchronization, need not necessarily involve an entirely new set of values for the constituent variables of a given expression. Our goal is to identify and eliminate redundant computations where the values of constituent variables are the same.

We propose a representation for a system of concurrently executing HDL processes such that analyses and optimizations can be built on top of it without being constrained by the simulation semantics. The deferred updating semantics and event-based triggering of the execution of the processes are captured in this representation. We detail the implementation of common subexpression elimination for a system of HDL processes based on the preceding representation.

In order to analyze a system of processes and identify (and eliminate) redundant computations, we highlight two requirements. The first is to predict a possible ordering among the processes achieving identical execution results [Barzilai et al. 1987; Ferrante et al. 1987; Willis and Siewiorek 1992] as that of the algorithm in Figure 1. In the next section, we define a partial order called levelized order (describing Section 4) for execution of processes in different ∂-cycles and an algorithm for computing the ordering. The second requirement is an ability to represent deferred updates of signal objects. Unlike variable assignments in languages like C and C++, an assignment does not immediately update the value of a signal object and is, by extension, not visible for subsequent expressions. We developed a representation called delta-value form (∂VF) to model this property, and it is presented in Section 5.

4. LEVELIZED ORDER

Definition. In levelized order if a process P_1 executes in an earlier ∂-cycle than the one in which another process P_2 executes, then level (P_1) < level (P_2).

To derive this partially ordered relation, [French et al. 1995; Wang and Maurer 1990] among the set of processes, we define two sets for each signal S_i as given next.

—*Trigger* (S_i) = $\{P_i |$ for each P_i, $S_i \partial$ *sensitivity* $(P_i)\}$.
—*Assign* (S_i) = Set of processes which have zero delay assignments to S_i.

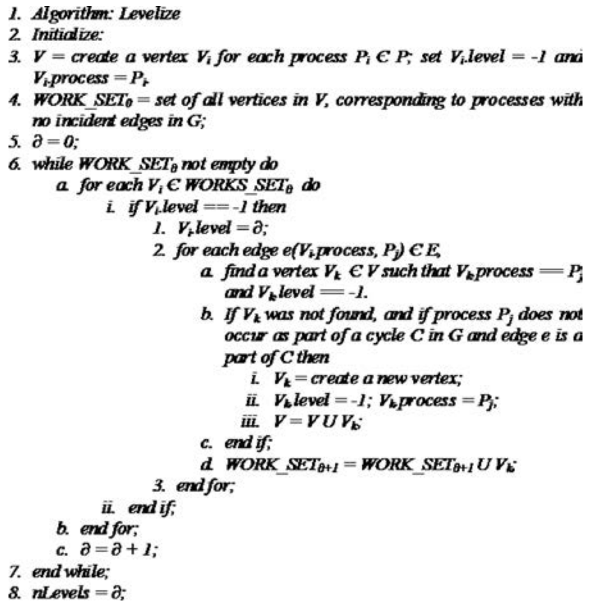
Assignments to each signal can cause a value change for the signal, which in turn will trigger processes sensitive to that signal for execution in the next delta cycle. Thus, processes in *Trigger* (S_i) will execute (if they are scheduled) one delta cycle later than processes in *Assign* (S_i).

To derive the levelized order, we construct a directed graph G = (P, E), where each vertex $P_i \partial$ P, and P is the set of all processes. For each signal S_i, we add an edge e_i to E, from P_i to P_j, where P_i is an element of *Assign*(S_i) and P_j is an element of *Trigger*(S_i).

After executing the levelization algorithm, we have a level associated with each vertex V_i. There is a process associated with each vertex V_i; in addition, a process P_i may be associated with multiple vertices in V. Combinational feedback loops are avoided by performing a cycle-check on G (see step *6(a)(i)(2)(b)* in Figure 2) before creating and adding a new vertex to V.

5. ∂-VALUE FORM

In this section, we introduce the concept of a ∂ operator to denote the value of an object or expression at the end of a given ∂-cycle. ∂_i(e) is the value of an expression ‘e’ at the ith ∂-cycle in the current simulation time. P_i(T) will be used to denote the instance of a process P in the ith delta cycle at time T. For convenience, we drop the time (T) and use



```
1. Algorithm: Levelize
2. Initialize:
3. $V$ = create a vertex $V_i$ for each process $P_i \in P$; set $V_i$.level = -1 an
   $V_i$.process = $P_i$
4. $WORK\_SET_\partial$ = set of all vertices in $V$, corresponding to processes with
   no incident edges in $G$;
5. $\partial = 0$;
6. while $WORK\_SET_\partial$ not empty do
     a. for each $V_i \in WORKS\_SET_\partial$ do
          i. if $V_i$.level == -1 then
               1. $V_i$.level = $\partial$;
               2. for each edge $e(V_i.process, P_j) \in E$,
                    a. find a vertex $V_k \in V$ such that $V_k$.process == $P_j$
                       and $V_k$.level == -1.
                    b. If $V_k$ was not found, and if process $P_j$ does not
                       occur as part of a cycle $C$ in $G$ and edge $e$ is a
                       part of $C$ then
                          i. $V_k$ = create a new vertex;
                         ii. $V_k$.level = -1; $V_k$.process = $P_j$;
                        iii. $V = V \cup V_k$;
                    c. end if;
                    d. $WORK\_SET_{\partial+1} = WORK\_SET_{\partial+1} \cup V_k$;
               3. end for;
         ii. end if;
     b. end for;
     c. $\partial = \partial + 1$;
7. end while;
8. nLevels = $\partial$;
```

Fig. 2. Levelization algorithm.

$P_{\partial(i)}$to denote the execution of a process P in the ith ∂-cycle of the current simulation time.

5.1. ∂VF

We define a naming scheme for signal/variable objects and related assignment statements in each $P_{\partial(i)}$, known as ∂VF. A simulation object can be visualized to attain a sequence of values (not necessarily distinct) in each of the ∂-cycles in a timestep. The *sequence of values* of an object S can be represented conceptually by an array $S_{\partial\text{values}}$, such that $S_{\partial\text{values}}[i]$ represents the value of S in delta cycle i. We define the ∂-qualified value of S in delta cycle i, represented by ∂i(S), to be $S_{\partial\text{values}}[i]$.[1]

In ∂VF, references to all simulation data objects are expressed in terms of their ∂-qualified values. ∂VF applies to expressions constructed from references to objects. Table I illustrates the translation of a set of HDL processes to the equivalent ∂VF.

In this table, the left column has the process code annotated with the ∂-cycle in which it executes. In the right column, the transformed assignments are shown. To model persistence of signal values across delta cycles, we introduce trivial copies at each delta boundary $\partial_{k+1}(S) = \partial_k(S)$; if $\partial_{k+1}(S)$ is not already defined. In a practical scenario, these trivial copies are not necessary, since a typical signal object implementation holds its value in memory across ∂-cycles. These trivial copies help make the availability of a variable be seen explicitly but don't have any corresponding generated code.

6. CODE OPTIMIZATION FRAMEWORK

Once a set of HDL processes is levelized and the code in each of the processes rewritten in ∂VF, we are closer to operating on sequential blocks of code without being limited by simulation semantics. We need to model the dependence between a process that drives

[1]A brief note on VHDL syntax. A signal assignment is written as *sig <= r.h.s*;. The assignment operator "<=" indicates a posting of the value of the r.h.s expression to the object *sig* in the next ∂-cycle.

Table I. Processes in ∂VF

Processes in Levelized order	Equivalent ∂VF
∂_1: P1 : Process(S1)	....P1
∂_1: S2 <= S1;	∂_2(S2) = ∂_1(S1);
∂_1: end;	* ∂_2(S1) = ∂_1(S1);
∂_2:P2: Process(S2)	....P2
∂_2: S3 <= S2;	∂_3(S3) = ∂_2(S2);
∂_2: end;	
∂_2:P3: Process(S2)	....P3
∂_2: S4 <= S3;	∂_3(S4) = ∂_2(S3);
∂_2: end;	* ∂_3(S2) = ∂_2(S2);
	* ∂_3(S1) = ∂_2(S1)
∂_3: P4: Process(S4)	....P4
∂_3: S5 <= S4;	∂_4(S5) = ∂_3(S4);
∂_3: S6 <= S2;	∂_4(S6) = ∂_3(S2);
∂_3: end;	

a signal *s* and the processes that are sensitive to the same signal *s* (processes which will execute due to an event on *s*). The sensitivity and driver relationships between processes encapsulate the wake-up/message-passing mechanism between the set of processes. Ferrante et al. [1987], duscuss a similar concept called program dependence graph and its applicability in optimization. To effectively capture these dependencies for the purposes of code optimization, we propose a data structure called the process sensitivity graph.

6.1. Process-Sensitivity Graph

A process sensitivity graph is defined as a directed graph G = <V, E>, with V being the set of vertices, such that vertex v ∈ V is a process instantiation for a given delta cycle, and E = E_r U E_c(defined next).

Let us define *SensDrivers* (P) to be the set of processes *SensDrivers* (P) = {P_i | P_i ∈ *Drivers* (S_i), for each S_i ∈ *Sensitivity* (P)}.

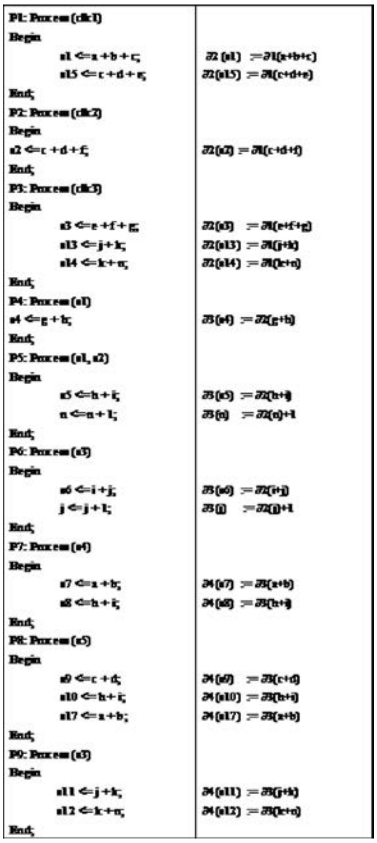
E_r (set of regular edges). The set of all direct sensitivity relationships between all pairs $P_j(\partial_k)$ and $P_{m}(\partial_{k+1})$. A regular edge e_r is created from P_i to P_j, if P_i ∈ *SensDrivers*(P_j).

E_c(set of cross edges). The set of edges representing subset relationships between sensitivity lists of all pairs of processes $P_i(\partial_k)$ and $P_j(\partial_k)$. A cross edge $e_c$is created from $P_i$to P_j, if *Sensitivity* (P_j) is a subset of *Sensitivity* (P_i).

Since a process can be sensitive to multiple signals, each of which can be driven by different processes, a node in the process sensitivity graph (PSG) can have more than one predecessor. Since a process P will be scheduled for execution as a result of an event on one or more signals belonging to *Sensitivity* (P), there can be multiple execution paths leading to P in the process sensitivity graph. This multiple predecessor property (multiple paths of execution can be concurrently active) is significantly different from control flow predecessors in traditional programming languages, wherein only a single path of execution in the control flow graph is active during execution. In the HDL simulation domain, it is possible that a subset (not necessarily a proper subset) of the paths leading to a node in the PSG may be taken before control reaches that node. Table II has an example that illustrates the process sensitivity graph for a small group of VHDL processes. The corresponding graph is presented in Fig 3.

In order to understand the construction of the graph, let us examine a couple of scenarios. In Table II, process P2 writes to signal s2 and process P5 is sensitive to s2. Accordingly, there is an edge from P2 to P5 in Figure 3. The sensitivity list of process

Table II. A Set of Processes to Illustrate the PSG



P1: Process (clk1)	
Begin	
s1 <= a + b + c;	∂2 (s1) = ∂1(a+b+c)
s15 <= c + d + e;	∂2(s15) = ∂1(c+d+e)
End;	
P2: Process (clk2)	
Begin	
s2 <= c + d + f;	∂2(s2) = ∂1(c+d+f)
End;	
P3: Process (clk3)	
Begin	
s3 <= e + f + g;	∂2(s3) = ∂1(e+f+g)
s13 <= j + k;	∂2(s13) = ∂1(j+k)
s14 <= k + n;	∂2(s14) = ∂1(k+n)
End;	
P4: Process (s1)	
s4 <= g + h;	∂3(s4) = ∂2(g+h)
End;	
P5: Process (s1, s2)	
Begin	
s5 <= h + i;	∂3(s5) = ∂2(h+i)
n <= n + 1;	∂3(n) = ∂2(n)+1
End;	
P6: Process (s3)	
Begin	
s6 <= i + j;	∂3(s6) = ∂2(i+j)
j <= j + 1;	∂3(j) = ∂2(j)+1
End;	
P7: Process (s4)	
Begin	
s7 <= a + b;	∂4(s7) = ∂3(a+b)
s8 <= h + i;	∂4(s8) = ∂3(h+i)
End;	
P8: Process (s5)	
Begin	
s9 <= c + d;	∂4(s9) = ∂3(c+d)
s10 <= h + i;	∂4(s10) = ∂3(h+i)
s17 <= a + b;	∂4(s17) = ∂3(a+b)
End;	
P9: Process (s3)	
Begin	
s11 <= j + k;	∂4(s11) = ∂3(j+k)
s12 <= k + n;	∂4(s12) = ∂3(k+n)
End;	

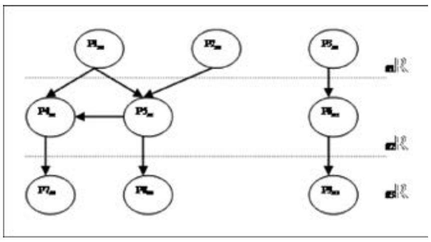
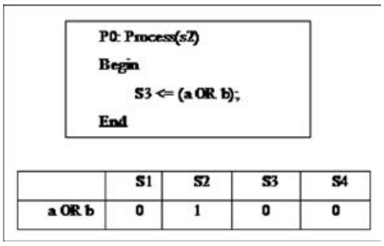


Fig. 3. Process sensitivity graph for the VHDL code in Table II.

P4 is {s1} and that of P8 is {s5}. Process P4 cannot generate any events on s5 and hence there is no edge between P4 and P8.

The task of identifying and propagating available expressions across multiple processes based on PSG is significantly different from traditional data-flow analysis-based approach due to the fact that multiple control paths may be simultaneously active and expressions from a disjoint control path (a different process) may be available for reuse.

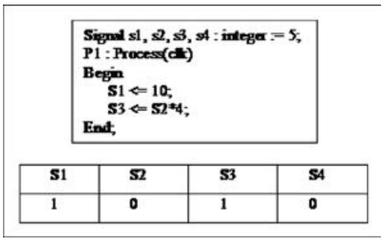
So far, we have introduced all the basic concepts needed to create a framework for performing optimizations on a group of concurrent threads based on static analysis. The framework still has to be able to capture the actual execution-time trace of signal activity so that we can expand the scope of optimizations to include redundancies that



```
P0: Process(s2)
Begin
    S3 <= (a OR b);
End
```

	S1	S2	S3	S4
a OR b	0	1	0	0

Fig. 4. Sensitivity vector for an expression in process P0.



```
Signal s1, s2, s3, s4 : integer := 5;
P1 : Process(clk)
Begin
  S1 <= 10;
  S3 <= S2*4;
End;
```

S1	S2	S3	S4
1	0	1	0

Fig. 5. Event vector for an expression in process P1.

can only be identified at runtime. We introduce the concepts of the sensitivity vector and event vector to aid in the process of identifying expressions available for reuse across ∂-cycles.

6.2. Sensitivity Vector

The sensitivity vector (SensVec) is a bit-vector with one bit for each signal in the set of sensitivity signals for the group of processes under consideration. For each subexpression E_j, available at the end of a particular process P_i, we define SensVec(E_j) to be SV_{Ej}, such that $SV_{Ej}[k] = 1$, for all $S_k \in$ *Sensitivity*(P_i). The sensitivity vector can be statically computed from sensitivity information.

In Figure 4, the process is only sensitive to signal S2, and hence the SensVec has only one bit on (for s2). The number of sensitivity signals for each expression can be pruned further by identifying the trigger condition controlling the basic block where this expression is generated.

6.3. Event Vector

Along with static sensitiving information, execution time event occurrence for each signal is necessary for determining actual process invocations. We introduce the event vector (EventVec) to denote signals which have had an event until the current delta cycle in a given time step. Like the sensitivity vector, event vector is a bit-vector having a bit for each signal in the set of sensitivity signals for the group of processes under consideration. The sensitivity vector is associated with each expression under consideration, while the event vector will be only one for the entire group of processes.

During simulation, at the beginning of each time step, the event vector is cleared. If a signal s_i has an assignment in the delta cycle which changes its value, then we set EventVec(i) = 1 in the delta cycle i+1.

In Figure 5, at the end of the execution of process P1, we can see that signals S1 and S3 have their values changed from what they started with (initial value of 5); thus, the event vector has the corresponding bits ON for signals s1 and s2, in the next delta cycle.

Thus armed with the sensitivity vector and event vector, we show how we can identify and reuse the set of available expressions correctly and conservatively.

7. ALGORITHM TO COMPUTE REUSABLE EXPRESSIONS

We can identify a set of expressions that are unconditionally available for reuse at each node in PSG. Also, with the help of execution time checks we can enable expression reuse in cases where reuse cannot be guaranteed by static analysis. We present both compile time and execution time expression reuse scenarios for a node in PSG.

7.1. Static Reuse

For a node in PSG having, the following.

—*Single Incident Edge*. All available expressions are forward propagated from the predecessor node.
—*Multiple Incident Edges*. The intersection of sets of available expressions corresponding to incoming edges is computed to form an available set of expressions for the node.
—*Incident Cross Edges*. All available expressions of the source node of the cross-edge are included in the available expression set of the destination node for static reuse in the next delta cycle.

7.2. Dynamic Reuse

—For a node in PSG having multiple incident edges (single incident edge and cross edges are handled by the static reuse case), a union of sets of available expressions corresponding to incoming edges is computed, and the expressions identified for static reuse are subtracted from the result to form the set of available expressions. To determine dynamic reusability of an expression EXP, we generate code to check if $(\text{SensVec}_{EXP} \cap \text{EventVec}) \neq \emptyset$ at each point of reuse.

7.3. Algorithm to Compute Reusable Expressions

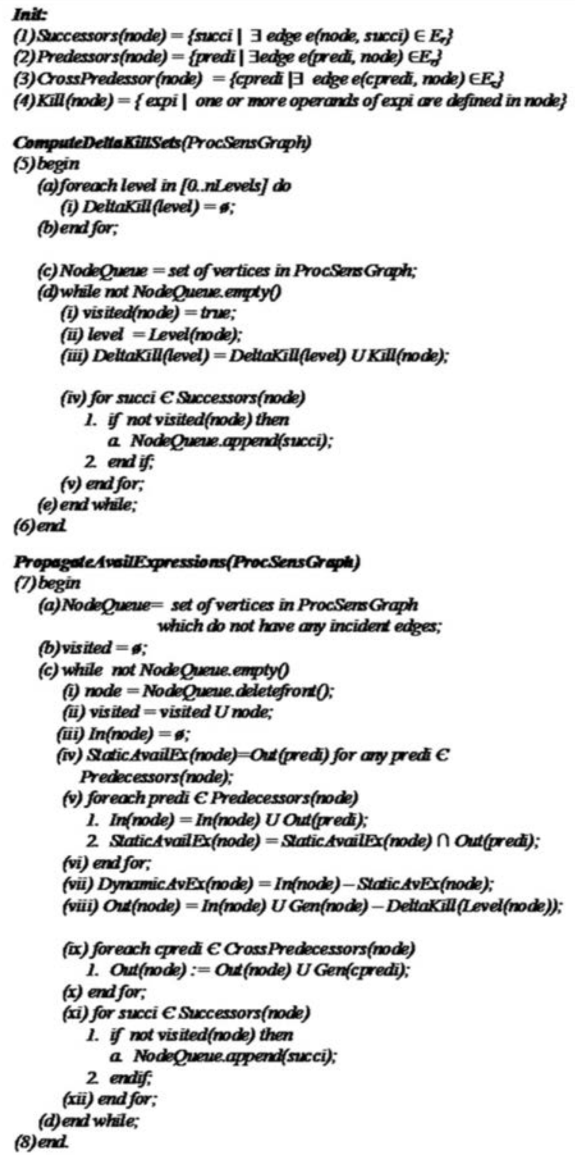
Using the PSG and the set of available expressions at each node in the PSG, we perform a one-pass forward propagation of available expressions, treating writes to signals in the processes as definitions of the corresponding signal objects. Any definition of object s kills all available expressions involving s, thereby making the expression invisible to successor nodes in the PSG.

If an expression becomes available at a node through all predecessors of that node in the PSG, then it is a candidate for static reuse. If it is available from a proper subset of the predecessors of that node, then it is a candidate for dynamic reuse. For dynamic reuse, we will need to check the EventVec and the SensVec of the expression to make sure that the expression was indeed computed at runtime, and we can then reuse it.

Using the process sensitivity graph, we present an algorithm to compute the set of reusable expressions. Computation of *Kill* set of a node used in the algorithm follows the standard technique used in the common subexpression elimination algorithm, [Aho et al. 2007], for a sequential control flow graph.

For the expressions marked as statically reusable, there is no need to generate additional code. On the other hand for dynamic cases of reuse, the candidate expressions are checked for availability at runtime, making use of the SensVec and EventVec for each occurrence. The test involving SensVec and EventVec provides the guarantee to avoid the re-computation.

LEMMA 7.1. *The algorithm in Figure 6 identifies a subexpression efor static reuse in a process P if and only if e is guaranteed to be computed by a process Q in an earlier ∂-cycle and e reaches P alive (i.e., not killed by the DeltaKill set in step7(c)(viii) of the algorithm) through all possible paths to P.*



```
Init:
(1)Successors(node) = {succi | ∃ edge e(node, succi) ∈ E_r}
(2)Predessors(node) = {predi | ∃edge e(predi, node) ∈E_r}
(3)CrossPredessor(node)  = {cpredi |∃  edge e(cpredi, node) ∈E_c}
(4)Kill(node) = { expi | one or more operands of expi are defined in node}

ComputeDeltaKillSets(ProcSensGraph)
(5)begin
    (a)foreach level in [0..nLevels] do
        (i) DeltaKill(level) = ϕ;
    (b)end for;

    (c) NodeQueue = set of vertices in ProcSensGraph;
    (d)while not NodeQueue.empty()
        (i) visited(node) = true;
        (ii) level  = Level(node);
        (iii) DeltaKill(level) = DeltaKill(level) U Kill(node);

        (iv) for succi ∈ Successors(node)
            1. if not visited(node) then
                a. NodeQueue.append(succi);
            2. end if;
        (v) end for;
    (e) end while;
(6)end.

PropagateAvailExpressions(ProcSensGraph)
(7)begin
    (a)NodeQueue=  set of vertices in ProcSensGraph
                   which do not have any incident edges;
    (b)visited = ϕ;
    (c) while  not NodeQueue.empty()
        (i) node = NodeQueue.deletefront();
        (ii) visited = visited U node;
        (iii) In(node) = ϕ;
        (iv) StaticAvailEx(node)=Out(predi) for any predi ∈
             Predecessors(node);
        (v) foreach predi ∈ Predecessors(node)
            1. In(node) = In(node) U Out(predi);
            2. StaticAvailEx(node) = StaticAvailEx(node) ∩ Out(predi);
        (vi) end for;
        (vii) DynamicAvEx(node) = In(node) – StaticAvEx(node);
        (viii) Out(node) = In(node) U Gen(node) – DeltaKill(Level(node));

        (ix) foreach cpredi ∈ CrossPredecessors(node)
            1. Out(node) := Out(node) U Gen(cpredi);
        (x) end for;
        (xi) for succi ∈ Successors(node)
            1. if not visited(node) then
                a. NodeQueue.append(succi);
            2. endif;
        (xii) end for;
    (d)end while;
(8)end.
```

Fig. 6. Algorithm to compute reusable expressions.

Proof. Part (a). For ∂_1, StaticAvEx(P$_i(\partial_1)$) = C.

Assumption 1. We assume that the lemma holds true for all cycles ∂_1 to ∂_k.

Considering the case for cycle ∂_{k+1}, for any node P$_i(\partial_{k+1})$, according to the algorithm's step 7(c)(v)(2), StaticAvEx(P$_i(\partial_{k+1})$) = $\cap_{predi}$ Out($pred_i$).

Hence, an expression e $\in$ StaticAvEx, if and only if e $\in$ Out($pred_i$), for all $pred_i \in$ Predecessors(P). According to construction of the PSG, it is clear that any predecessor node $pred_i$ has to execute one delta cycle earlier than P (in ∂_k), if it executes at all.

(a) If $e \notin$ Gen($pred_i$), for any $pred_i \in$ Predecessors(P), e must have been computed in a delta earlier than ∂_k. e could have reached $pred_i$ through a regular edge or a cross edge.
   (i) If e reached $pred_i$ through a regular edge, e would have been computed in a node at least one delta earlier than $pred_i$, according to construction of PSG.
   (ii) Otherwise e would have reached $pred_i$ through a cross edge, which means that e was computed one delta cycle earlier than $pred_i$ (in ∂_{k-1}). This is ensured by the fact that cross edges are considered in step *7(c)(ix)* of the algorithm in Figure 4 after *StaticAvailEx(pred$_i$)* has been computed in step *7(c)(v)* of the same algorithm.

   Considering assumption (1), in both cases, e is a legitimate statically reusable expression.

(ii) If $e \in$ Gen($pred_i$), and $pred_i \in$ Predecessors(P), computation of the expression e has to happen in ∂_k according to construction of the PSG. $e \in$ DeltaKill(∂_k)according to step *7(c)(viii)* of the algorithm, since e is a reusable expression. Since e was computed in ∂_k and did not get killed (by *DeltaKill*), it is safe to reuse e.

Thus, in both cases we prove that it is safe to statically reuse e, and Lemma 1 holds for ∂_{k+1}, if it holds for ∂_k. Since the lemma trivially holds for $\partial_{1,}$by induction, it should be true for all ∂_k. □

If any or all of the $pred_i$ nodes are actually executed at runtime, the algorithm in Figure 6 guarantees the availability of an already identified set of expressions. If none of the $pred_i$ nodes actually execute, we will not have P executing, and expression e will not be used.

Expression reuse within the same ∂-cycle. The algorithm in Figure 6 addresses expression reuse in processes executing in different ∂-cycles. This can be extended to expressions being computed more than once in the same ∂-cycle, but by different processes. ∂VF helps simplify the expressions and object values by making a clear distinction between the current values and the scheduled value for each object. Once the processes are transformed into ∂VF, it is possible to apply data flow analysis seamlessly to eliminate redundant computations. Possible instances of intra-∂-cycle reuse are limited to the set of process nodes with cross edges.

Example. For the set of processes from Figure 3 and Table II, we enumerate the reusable expressions identified by our algorithm.

Static Reuse. Subexpression (c+d) computed by processes P1 and P2 in ∂_1 is reusable by process P8 in ∂_3. (c+d) is available to P5—since both of P5's predecessors compute it—propagates through to P8, and is available for reuse.

The subexpression (h+i) computed by process P5 in ∂_2 is available for reuse by process P7 in ∂_3, though there is no direct edge between P5 and P7. However, the expression propagates through the cross edge from P5 to P4 in $\partial_2$and then it propagates to P7 via a regular edge.

Dynamic Reuse. Subexpression (a+b) is computed in process P1 in ∂_1, but this cannot be guaranteed to be available for reuse in process P8 in ∂_3. Though there is a path from P1 to P8, (a+b) has not been identified for static reuse, since all of the predecessors of the intermediate node P5 don't compute the expression. It may still be possible to dynamically reuse (a+b) if the condition mentioned in Section 7.2 is met at execution time.

8. EXPERIMENTAL SYSTEM

As explained in Section 1, our optimization framework focuses on the codegenerator and the kernel of a compiled simulator. The process sensitivity graph was built from the elaborated design, after optimizations like module inlining were applied to expand

Table III. Execution Times before Applying Optimizations

Test Name	Compile Time (seconds)	Runtime (seconds)	Compile Memory (Mega bytes)	Runtime Memory (Mega bytes)
Processor 1	29.2	410.6	102.5	127.5
Processor 2	12.5	190.7	78.2	80.0
Processor 3	38.3	241.8	181.7	275.8
Processor 4	96.1	20,19.4	379.3	521.2
Processor 5	77.4	1,876.7	327.6	486.3
Graphics 1	415.8	3,623.1	1,216.8	2,637.5
Graphics 2	100.5	1,722.3	401.0	1,025.6
Network 1	110.9	1,291.6	425.2	1,291.9
Processor 6	178.6	2,101.5	506.1	1,776.4
Network 2	121.3	3,890.2	430.7	1,132.1

the scope of the process sensitivity graph. Our analysis was limited to a module, (architecture in the case of VHDL) at a time. This is not a limitation of the algorithm, but has to do with the ease of implementation.

Delta value form is constructed for all variables defined/used in a module. For each variable, we allocate an array of slots based on the number of deltacycles needed for the system of processes under consideration. Each subexpression computed in this system of processes is given an ID for ease of reference (we account for commutativity by means of a lexical ordering of the expression variables). With PSG and ∂VF ready, we run the algorithm described in Figure 6, to identify reusable expressions.

In order to eliminate redundant computations, we modified the code generator according to the conditions laid out in Sections 7.1 and 7.2 for static and dynamic reuse. We did not attempt to reuse the expression values within the same delta cycle across multiple processes, as it would require more runtime checking due to the unpredictable order of execution within a ∂-cycle. The case of reuse across multiple invocations of the same process in different ∂-cycles is a straightforward extension of the preceding algorithm, provided we clone the generated code for the process and use ∂VF.

We instrumented our compiler (code generator) to gather statistics on the total number of candidate expressions and those that were identified as statically reusable and as dynamically reusable. The instrumented compiler was run on a set of medium-to-large sized designs, and the results are presented next (Tables III and IV).

9. RESULTS

Static reuse and dynamic reuse algorithms share most of the compiletime analysis steps. Dynamic reuse results in more generated code to guard the reuse of available expressions, and as a result, has a slightly higher compile time impact than static reuse. The runtime benefit is also typically less for dynamic reuse. When we combine the two approaches, we didn't observe significant overhead at compile time compared to the static reuse alone. The runtime benefits are cumulative.

Table IV. Experimental Results on HDL Designs

Test name	Number of Expressions	Expressions Identified for Static Reuse	Expressions Identified for Dynamic Reuse	Total Percentage of Reused Expressions
Processor 1	1,2786	576	213	6.17
Processor 2	9874	413	156	5.76
Processor 3	1,5697	782	329	7.08
Processor 4	4,0311	1,872	541	5.99
Processor 5	36,235	2,379	768	8.69
Graphics 1	157, 247	20,218	1,015	13.5
Graphics 2	42, 586	2,661	891	8.34
Network 1	52, 871	310	1,093	2.6
Processor 6	71, 522	3,871	2,076	8.31
Network 2	47,349	2,517	1,291	8.04

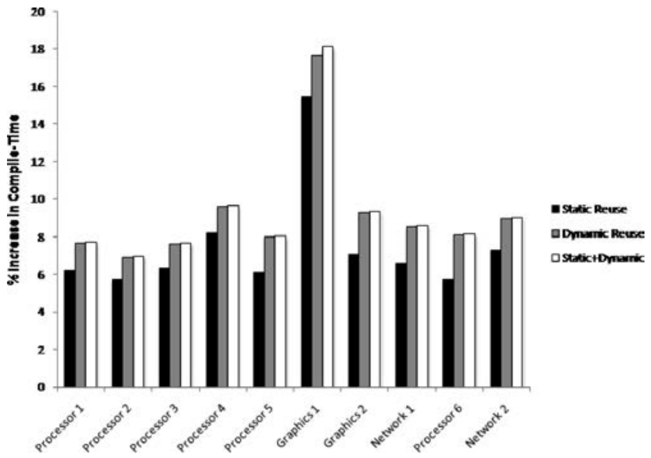


Fig. 7. Compile-time overhead caused by the optimizations.

Dynamic reuse of expressions necessitates changes to the runtime system to update the EventVec of each process and clear it at the end of a simulation time step. Due to the additional costs involved with dynamic reuse, the compiler engineer has to make a judicious decision, weighing the cost and potential benefit. The compile-time overheads caused by static, dynamic, and the combination of the two analyses are presented in Figure 7.

The runtime speedups are presented in Figure 8. As can be seen in Table IV, one particular design (Graphics1) stands out in terms of a higher percentage of reuse and corresponding runtime benefit. Upon further inspection we found that large portions of the design were machine generated and had a significant amount of redundant computation of subexpressions. In particular, modules with heavier emphasis on mathe-

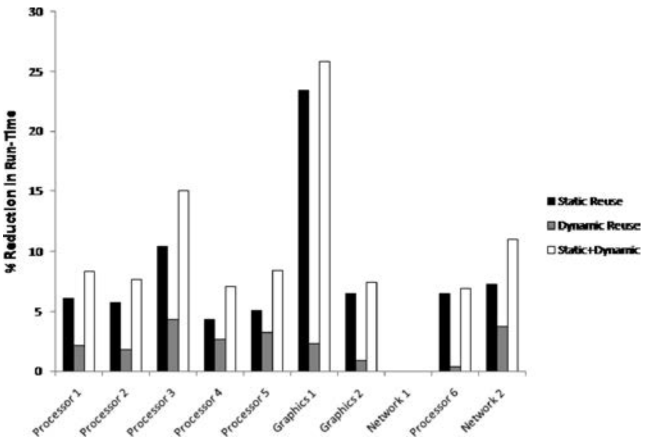


Fig. 8. Reduction in run-time due to the optimizations.

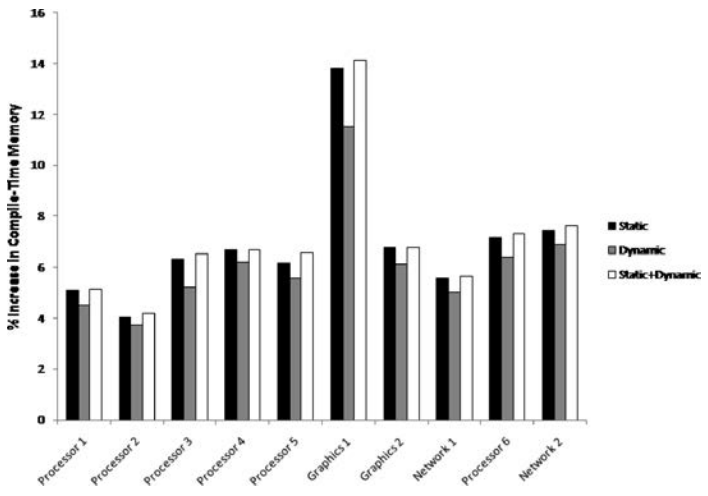


Fig. 9. Percentage in compile memory due to the optimizations.

matical computation, such as CRC, benefited more when compared to sequential/timed systems. On the flip side, two designs resulted in negligible benefits from our optimization due to lack of scope for reuse based on the coding style.

Dynamic reuse can get expensive in terms of runtime memory consumption (see Figure 10) if the implementation does not do optimizations, like keeping event bits only for the signals participating in reusable expressions and applying thresholds if the EventVec grows too big.

The test-bench stimulus may determine the runtime activity of various modules in the system. Even if a lot of expressions were identified for reuse but the part of the design that was optimized is not active at runtime, we may not see any benefits at all. Finally, we also present the memory overhead introduced by the optimizations at compile time and runtime in Figure 9 and Figure 10, respectively. With the potential for runtime reduction of longrunning simulations, the memory overheads observed do not seem large enough to prohibit deployment of this optimization in a production environment.

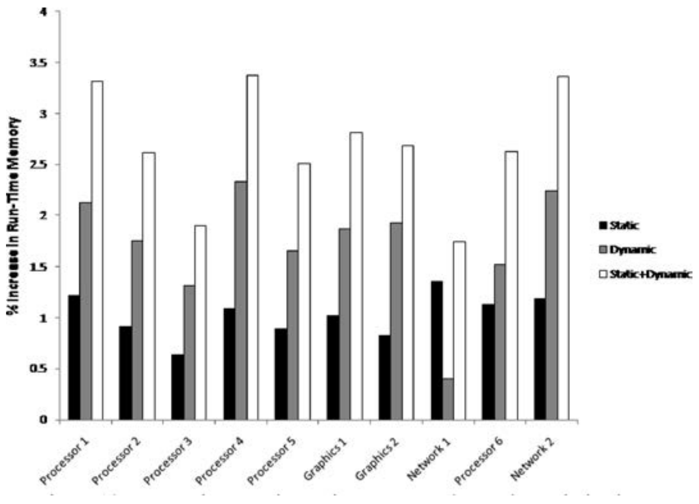


Fig. 10. Percent increase in runtime memory due to the optimizations.

10. FUTURE WORK

In this article, we presented a framework that enables us to analyze a given system of HDL processes without worrying about the special semantics associated to variables. In particular, we focused on identifying reusable expressions and reducing redundant computation across ∂-cycles in a single time step. It is possible to extend this framework to analyze repetitive execution of a set of processes at different simulation times and identify redundancies.

A future direction could be to explore more compiler optimizations that take advantage of the framework presented here. Some related ideas include the elimination of unnecessary resolution function execution in the case of multiply driven wires and elimination of expensive simulation-kernel calls based on the PSG.

11. CONCLUSION

In this article, we discussed the inadequacy of traditional data flow analysis [Rosen 1979], in the presence of HDL semantics and concurrency. To apply compiler optimizations across concurrent threads, we have introduced a transformation from HDL assignments/expressions to a form named ∂VF. Thereafter, we presented the PSG (process sensitivity graph)—a way to model process sensitivity and the resulting relationships among processes. Along with these novel concepts, we have introduced two auxiliary data structures for extending the reuse of expressions to dynamic cases. Utilizing all of these concepts, we presented an algorithm to compute the sets of statically and dynamically reusable available expressions for each process. The results shown indicate the potential of this optimization in discrete event simulation of real HDL designs.

REFERENCES

AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques and Tools*. Pearson Addison-Wesley, Reading, MA.

BARZILAI, Z., CARTIER, J. L., ROSEN, B. K., AND RUTLEDGE, J. D. 1987. HSS—a high-speed simulator. *IEEE Trans Comput. Aided Des. Interg. Circuits syst.*, 6, 4, 601–616.

EDWARDS, S. A. 2003. Tutorial: Compiling concurrent languages for sequential processors. *ACM Trans Des Autom. Electro. Syst., 8*, 2, 141–187.

FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3, 319–349.

French, R. S., Lam, M. S., Levitt, J. R., and Olukotun, K. 1995. A general method for compiling event-driven simulations. In *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference,* 151–156.

Hansen, C. 1988. Hardware logic simulation by compilation. In *Proceedings of the 25th Annul ACM/IEEE Design Automation Conference*. 712–716.

IEEE. 1996. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. IEEE Computer Society, New York, NY. 1364–1995.

IEEE. 1994. *IEEE Standard VHDL Language Reference Manual: IEEE Std 1076-1993*. Institute of Electrical and Electronic Engineers, New York, N Y.

Krishnaswamy, V. and Banerjee, P. 1998. Parallel compiled event driven VHDL simulation. In *Proceedings of the 12th International Conference on Supercomputing*. 297–304,.

Rosen, B. K. 1979. Data flow analysis for procedural languages. *J. ACM 26*, 2, 322–344.

Wang, Z. and Maurer, P. M. 1990. LECSIM: A levelized event driven compiled logic simulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 491–496.

Willis, J. C. and Siewiorek, D. P. 1992. Optimizing VHDL compilation for parallel simulation. *IEEE Des. Test Comput. 9*, 3, 42–53.

Received August 2010; revised July 2011, January, March 2012; accepted October 2012