Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs ⋆

Jaejin Lee[1], Samuel P. Midkiff[2], and David A. Padua[1]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign., {j-lee44, padua}@cs.uiuc.edu
[2] IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, midkiff@watson.ibm.com

Abstract. Static Single Assignment (SSA) form has shown its usefulness for powerful code optimization techniques such as constant propagation in sequential programs. We introduce the Concurrent Static Single Assignment (CSSA) form and the transformation algorithm for explicitly parallel programs with interleaving semantics and post-wait synchronization. The parallel construct considered in this paper is `cobegin/coend`. A new concept, π-assignment, which summarizes the information of interleaving statements among threads, is introduced. The Concurrent Control Flow Graph, which contains information about conflicting statements in addition to control flow and synchronization information, is used as an intermediate representation for the CSSA transformation. An extension of the Sparse Conditional Constant propagation algorithm based on the CSSA form makes it possible to apply the constant propagation optimization to explicitly parallel programs.

1 Introduction

Advances in integrated circuit technology that make multi-processor chips feasible and the wide availability of shared memory Unix, Linux and NT based workstations are making parallelism ubiquitous. One important consequence of this trend is that parallel programs, either written directly by the programmer or generated by a parallelizing compiler, are likely to become the norm. Parallel programming is already widespread in technical computing, and support for concurrent threads and non-determinism in languages such as **Java** will increase the spread of parallel applications. Despite of the importance of parallel programming, there are few studies on compiler optimization techniques for explicitly parallel programs.

It is not possible to apply classical optimization and analysis techniques directly to parallel programs since the classical methods do not account for

⋆ Supported in part by Army contract #DABT63-95-C-0097. This work is not necessarily representative of the positions or policies of the Army or the government. Supported in part by IBM. This work is not necessarily representative of the positions or policies of the IBM Corp.

updates to variables in threads other than the one being analyzed[9]. As shown in Figure 1, using classical constant propagation on a parallel program can result in an incorrect program in the presence of busy-waiting. The value of variable a should be 3 at the point just after the assignment `a = b` but by classical analysis appears to be 4 [Figure 1(B)]. This is because the interaction between threads was not considered. Additionally, incorrect dead code elimination is introduced [Figure 1(C)]. Thus, it is common for programmers to turn off optimizations when compiling parallel programs.

```
flag = 0
b = 4
cobegin
  while flag = 0 do
  endwhile
  a = b
  print a
//
  b = 3
  flag = 1
coend
```
(A)

```
flag = 0
b = 4
cobegin
  while flag = 0 do
  endwhile
  a = 4
  print 4
//
  b = 3
  flag = 1
coend
```
(B)

```
flag = 0
b = 4
cobegin
  while true do
  endwhile
//
  b = 3
  flag = 1
coend
```
(C)

Fig. 1. Incorrect constant propagation in the presence of busy-waiting. (A) Before constant propagation. (B) After simple constant propagation. (C) After conditional constant propagation.

```
a = 4
c = 5
read b
repeat
  if b > 0 then
    a = a + b
  else
    a = a - b
  endif
  c = c + 1
until c = 3
print a
```
(A)

```
a0 = 4
c0 = 5
read b0
repeat
  a1 = ϕ(a0,a4)
  c1 = ϕ(c0,c2)
  if b0 > 0 then
    a2 = a1 + b0
  else
    a3 = a1 - b0
  endif
  a4 = ϕ(a2,a3)
  c2 = c1 + 1
until c2 = 3
print a4
```
(B)

Fig. 2. An example of SSA transformation. (A) Original code. (B) SSA form.

To overcome these inabilities, it is important to develop optimization algorithms and the corresponding intermediate representations for parallel programs. During the last few years, the static single assignment (SSA) form has become popular as the underlying representation to compute data flow properties of sequential programs[2, 5]. The SSA form has shown its usefulness for powerful code optimization techniques such as constant propagation [15], common subexpression elimination[11], partial redundancy elimination[3], code motion, and induction variable analysis.

A parallel static single assignment form has been proposed by Srinivasan *et. al.*[13, 14]. However, it is restricted to the subset of parallel constructs in the PCF Parallel Fortran with copy-in/copy-out semantics in which the result of a parallel execution does not depend a particular interleaving of statements in the explicitly parallel program. In addition, they have not considered more general parallel constructs with interleaving semantics and the post-wait synchronization. Thus, their method is applicable to a restricted set of parallel programs.

In this paper, we introduce the concurrent static single assignment (CSSA) form for a language with interleaving semantics and a restricted form of **post/wait** synchronization. To convert an explicitly parallel program into our CSSA form, we introduce the concurrent control flow graph (CCFG), a parallel counterpart of the control flow graph of a sequential program. After giving

the algorithm for converting explicitly parallel programs into the CSSA form, we extend the sparse conditional constant propagation algorithm of Wegman and Zadeck[15] to operate on the concurrent static single assignment form. We call our algorithm *concurrent sparse conditional constant propagation.* To the best of our knowledge, this is the first discussion in the literature on constant propagation for explicitly parallel programs.

We now briefly describe the traditional SSA representation[5]. In SSA form, a program has the property that only one definition (assignment) of each variable in the program can reach its uses. The SSA form contains ϕ-functions that distinguish values of variables coming from different incoming control flow edges in the control flow graph[1] of the program. A ϕ-assignment has the form $V' = \phi(V_1, \ldots, V_n)$ where $V', V_1, \ldots, V_n$ are variables and n is the number of incoming control flow edges for the node where the ϕ-assignment is placed. We place ϕ-assignments for a variable at its join nodes[5]. Multiple definitions of the variable reach at the join node through its distinct incoming control flow edges. Figure 2 shows an example SSA transformation.

The remainder of the paper is organized as follows. We discuss the language model in Section 2. Section 3 describes concurrent control flow graphs. We develop the concurrent static single assignment form in Section 4. Section 5 describes the concurrent sparse conditional constant propagation. We conclude in Section 6.

2 Language Model

The parallel language used in this paper is a structured language (i.e. no `goto` or `exit` statement is provided) with the `cobegin/coend` construct used to express all parallelism. Synchronization is provided by `set/wait`. Figure 6(A) shows an example code which uses the language.

The parallel `cobegin/coend` construct consists of blocks of code, where each may contain another `cobegin/coend` construct. Blocks of statements are separated from one another by `//`. Threads are generated using the `cobegin` statement and threads are synchronized at the `coend` statement. The blocks share the same address space and execute concurrently with each other. A block not containing another `cobegin/coend` is called a *thread.* See Figure 1 for examples.

We place the restriction that loops may not contain a `cobegin/coend`.

An *event* is an integer variable with two states, *posted* and *cleared.* In this paper we allow only two operations, `set` and `wait` on an event variable `S`, and require that these operations not appear in a loop. The `set(S)` statement sets the event variable `S` to *posted*, if not already in *posted* state, and `wait(S)` suspends the calling thread until the event variable `S` is set to *posted.*

3 Concurrent Control Flow Graph

The Concurrent Control Flow Graph (CCFG) is an intermediate representation of explicitly parallel programs that has some similarities to the Parallel Program

Graphs by Sarkar and Simons [12] and the Parallel Control Flow Graphs and Parallel Precedence Graphs by Wolfe and Srinivasan [16]. It is different in that it contains *conflict* edges in addition to synchronization and control flow edges.

Definition 1 *Two memory references in different threads* **conflict** *if they reference the same memory location and at least one is a write. Two statements* **conflict** *if they contain mutually conflicting references.*

We modify the notion of a *basic block* for explicitly parallel programs. The definition of a concurrent basic block is as follows:

Definition 2 *A* **Concurrent Basic Block** *has the following properties.*

1. *A sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end (this is the definition of a basic block [1] in sequential setting).*
2. *Only the first statement can be a* `wait` *or contain a use of a conflicting variable.*
3. *Only the last statement can be a* `set` *or contain a definition of a conflicting variable.*
4. *If a concurrent basic block contains a* `cobegin` *or* `coend` *statement, then that statement is the only one in the concurrent basic block.*

The following is the definition of conflicts between two concurrent basic blocks.

Definition 3 *Two concurrent basic blocks conflict if they contain statements that conflict with each other.*

The Concurrent Control Flow Graph is defined as follows:

Definition 4 *A Concurrent Control Flow Graph (CCFG) is a directed graph* $G = (N, E, Ntype, Etype)$ *such that,*

1. N *is the set of nodes in* G. *Each node is a concurrent basic block.*
2. $E = E_{ct} \cup E_{sy} \cup E_{cf}$ *where:*
   - $E_{ct} = \{(m,n) \mid m,n \in N \wedge \; Etype(m,n) \in \{T,F,U\}\}$ *is the set of control flow edges.*
   - $E_{sy} = \{(m,n) \mid m,n \in N \wedge \; Etype(m,n) \in \{S\}\}$ *is the set of synchronization edges which show the order enforced by synchronization operations.*
   - $E_{cf} = \{(m,n) \mid m,n \in N \wedge \; Etype(m,n) \in \{DU, DD, UD\}\}$ *is the set of conflict edges. There is a conflict edge between any two concurrent basic blocks that conflict.*
3. $Ntype$ *is a function which tells the class of nodes, such that* $Ntype : N \mapsto T$, $T = \{$*Entry, Exit, Cobegin, Coend, Condition, Header, Compute, ThreadEntry, ThreadExit*$\}$.
4. $Etype$ *is a function that represents the type of edges in the graph.*

The *Entry* and *Exit* nodes are special nodes which have no predecessors and no successors in the *CCGF* respectively. Several threads, one per outgoing edge, are created at a *Cobegin* node. Threads are merged at a *Coend* node. *ThreadEntry* and *ThreadExit* nodes are special nodes that mark the beginning and the end of each thread between *Cobegin* and *Coend* nodes.

A *Condition* node is the same as the branch node in the Control Flow Graph[1] in sequential setting but if it is a loop header[1] node, it is called a *Header* node. Both *Condition* and *Header* nodes contain a condition for branching. *Compute* nodes are all the remaining nodes. Usually they contain a sequence of assignment statements.

A control flow edges is labeled T (if a **true** branch), F (if a **false** branch) or U (if an unconditional branch.) There always exists a control flow edge from *Entry* node to *Exit* node. The direction of a synchronization edge is from the node which contains the `set` to the node which has a `wait` for the same event variable. The label of a synchronization edge is S. Conflicting statements are joined by two edges with opposite directions. The function *Etype* for conflict edges is defined by Figure 3.

The leftmost column is the type of conflict at the tail of the edges and the uppermost row is the type of conflict at the head of the edges. For example, if the tail of the conflict edge contains the definition of the conflicting variable, its status is *def.* If it contains a use, the status is *use.* If it contains both of them (the definition and use do not have to be for the same variable.), it is *def&use.* In the case of head, the rule is similar to the case of tail. An example of labeling conflict edges is shown in Figure 4.

`Do/enddo` and `repeat/until` loops are first converted to `while/endwhile` loops (see Figure 5), and then to CCFG form. If the loop is `while/endwhile`-like loop, it is easy to separate the loop header from the nodes in the loop body because the exit from the loop is from the header. Figure 6(B) shows the CCFG for the explicitly parallel program in Figure 6(A).

4 Concurrent Static Single Assignment Form

The meaning of ϕ-functions in the parallel and sequential settings are the same, i.e. they distinguish values of variables coming from distinct incoming control flow edges. In this paper, we extend the placement of ϕ-functions to cover the case of the *Coend* node where threads are merged.

Definition 5 *A ϕ-function has the form $\phi(V_1, V_2, \ldots, V_n)$, where n is the number of incoming control flow edges of the node where it is placed. The value of $\phi(V_1, V_2, \ldots, V_n)$ is one of the V_i's and the selection depends on the control flow. At Coend node, the selection depends on the interleavings of statements in different threads in execution.*

A new node, the π-function, summarizes the interleaving information of conflicting variables where the π-function is placed. Similar to ϕ-functions, the π-function distinguishes values of variables coming from an incoming control flow

	def&use	def	use
def&use	*DU*	*DD*	*DU*
def	*DU*	*DD*	*DU*
use	*UD*	*UD*	

Fig. 3. The labeling rule for conflict edges.

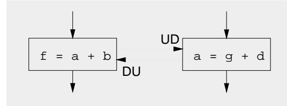

Fig. 4. An example of labeling conflict edges.

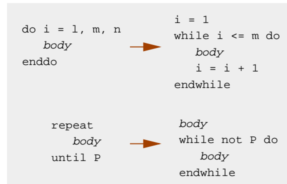

Fig. 5. Converting `do` and `repeat/until` loop into `while` loop.

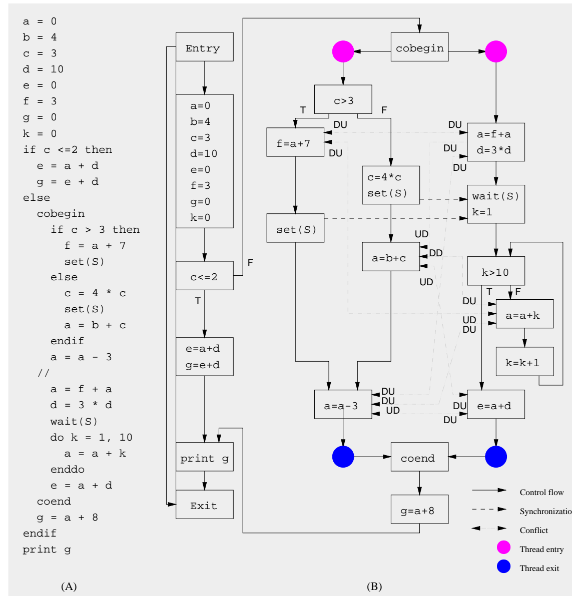

Fig. 6. An example code and its Concurrent Control Flow Graph. (A) An example code. (B) The CCFG for the code in (A).

edge (the number of incoming control flow edges into the node where π-functions are placed is always 1 by the construction of CCFG) and distinct conflict edges labeled *DU*.

Definition 6 *A π-function has the form $\pi(V_1, V_2, \ldots, V_n)$, where n is the number of incoming control flow edges plus the number of incoming distinct conflict edges labeled DU. The value of the $\pi(V_1, V_2, \ldots, V_n)$ is one of the V_i's and determined nondeterministically because of the concurrency between threads.*

The basic idea of the CSSA form is that it summarizes the interleaving information for conflicting variables in an explicitly parallel program by using π-functions. The values of all conflicting variables are well defined by the π-function at the point where the π-function is placed. Like the SSA form, CSSA form has the property that all uses of a variable are reached by exactly one assignment to the variable.

Translating the CCFG of an explicitly parallel program into its CSSA form is a three-step process:

1. Get the partial ordering of conflicting statements. [Section 4.1]
2. Place ϕ-functions (ϕ-assignments). [Section 4.2]
3. Place π-functions (π-assignments). [Section 4.3]

4.1 Partial ordering of conflicting statements

In order to get information about the effect of interleaving among statements in different threads, we need to get the partial ordering of the statements.

Callahan and Subhlok[4] used data flow analysis to get an event ordering for explicitly parallel programs without loops. We use a data flow analysis framework similar to theirs. This is essentially the same as the common ancestor algorithm by Emrath, Ghosh and Padua [6, 7]. This method computes an conservative approximation to the guaranteed ordering. We do not use the more accurate but exponential exhaustive pairing algorithm[7]. We note that in general this problem is Co-NP-hard [10].

Our definition of parallel basic blocks ensures that a partial ordering between nodes in CCFG is a partial ordering between conflicting statements in those nodes. In this paper, we are only interested in the partial orderings between nodes inside `cobegin/coends`[3], i.e. the interleaving information between statements in different threads.

The partial ordering of the nodes in the loop is not determined at compile time. We give each node in the loop the same precedence as the loop header node. There is no harm in doing so, since we are only interested in the partial orderings between statements in different threads and there is no `set` or `wait` inside a loop by the language model.

Since all the loops in the CCFG are of the form of `while/endwhile` and there is no `goto` or `exit` statement in the language, all control flows, which go into a loop, are through the loop header[4] and the exit from the loop header is the only exit from the loop.

We now compute a relation *Prec(n)* on a modified graph where all nodes in a loop are merged into a representative node, as shown in Figure 7. If N' is the set of nodes in the modified graph, then:

$$Prec(n) = \{m \mid m \in N',\ m \text{ is guaranteed to precede } n \text{ in execution}\}$$

Details of computing *Prec(n)* can be found in [8].

We define a predicate $Precede?(m,n)$, used to get the partial ordering of two concurrent basic blocks m and n, as:

$$Precede?(m,n) = \begin{cases} true & \text{if } m \in Prec(n) \\ false & \text{otherwise} \end{cases}$$

By using the partial ordering between two conflicting statements, we can ignore some conflict edges when we generate π-assignments later.

[3] This is true because we assume that no loop construct encloses the `cobegin/coend`.

[4] The header dominates all nodes in the loop. To identify a loop header, we look for a back edge in the CCFG, then the node at the head of the back edge is the loop header[1]. Depth first search is used to identify the back edges.

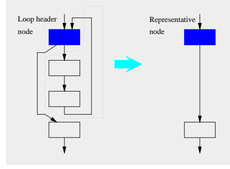

Fig. 7. To apply the data flow analysis framework, we remove the nodes and edges enclosed by the dashed line.

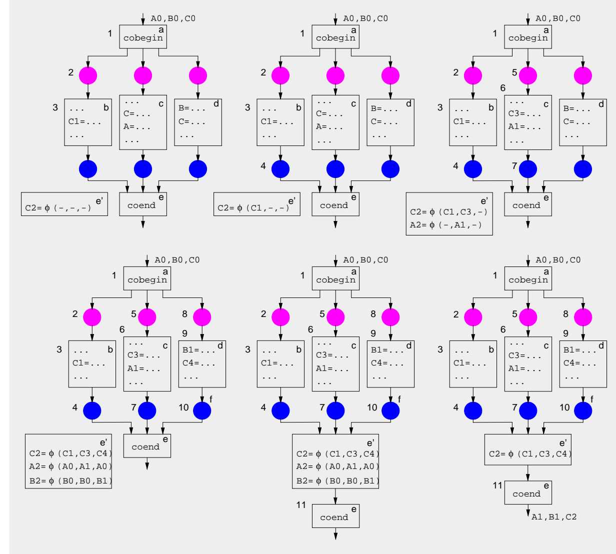

Fig. 8. Placing ϕ-assignments in the join node of *Cobegin* node.

4.2 Placing ϕ-assignments

In order to get the CSSA form, we transform the CCFG into the SSA form after synchronization analysis is done on the CCFG. We give the single assignment property to explicitly parallel programs by placing ϕ-assignments. To place ϕ-functions in a given CCFG, we consider only control flow edges and ignore conflict edges and synchronization edges in the CCFG.

There are two different classical approaches for placing ϕ-assignments [5, 2]. Because we are dealing with structured programs, our algorithm for placing ϕ-assignments can be based on that of Brandis and Mössenböck[2]. Their transformation is done at parse time on structured programs, but our algorithm is applied to a CCFG. We also extend their method to the `cobegin`/`coend` construct.

The algorithm does a depth first traversal on the given CCFG. Since the join nodes of the branch nodes (e.g. *Cobegin*, *Condition*, and *Header*) are known *a priori*, when a branch node is encountered we do a depth first traversal of its successor nodes to generate ϕ-functions until we encounter the corresponding join node. Since the method for *Condition* and *Header* nodes is almost the same as the method of [2], we omit the cases of *Condition* and *Header* in this paper.

We now describe the conversion of a *Cobegin* block to SSA form by means of an example. The placement of ϕ-assignments is almost the same as for the *Condition* block. However, if we treat a *Cobegin* block as a *Condition* block, many superfluous ϕ-assignments are generated. To prevent this, we check each argument in the ϕ-function after filling in all the vacant arguments in it. The

following theorem tell us when to discard superfluous ϕ-assignments.

Theorem 1 *When the join node is a Coend node, a ϕ-assignment is superfluous and can be eliminated if all of its parameters are the same, or all but one of its parameters are defined before the corresponding Cobegin node.*

Proof: See [8]. ∎

Note that if a ϕ-assignment has more than two parameters and all but one of its parameters are the same, those that are the same are all defined before the corresponding *Cobegin* node by the single assignment property. So it is superfluous.

Figure 8 shows the transformation of the *Cobegin* node. When parsing a program, the *Cobegin* node is generated by a `cobegin` statement, and its join node `coend` is known. A pointer can be created linking the *Cobegin* node to its join node, allowing the join node to be easily accessed.

1. The variable values coming into node **a** are `A0`, `B0` and `C0`. We call them the current values of `A`, `B` and `C` respectively. We do nothing at node **a**.
2. The variable values coming into node **b** are the same as those of node **a**. Since `C` is defined, a new variable `C1` is generated, replacing `C` in the left hand side of the assignment. Because a ϕ-function has not been inserted at node **e**, we create a dummy node **e**′ and place the ϕ-assignment for the variable `C` there. A new variable `C2` is generated. Initially the ϕ-function has the form ϕ`(-,-,-)` with one placeholder for each incoming control flow edges (and possible value of `C`) that reaching the join node **e**. After we make sure that the successor of node **b** is the join node, we know the value of `C` that reaches the join node **e** is `C1` and place it in the the first argument of ϕ`(-,-,-)`, producing `C2=`ϕ`(C1,-,-)`.
3. The visits to node **c** proceed similarly. After we check that the successor of **c** is the join node, we fill in the vacant argument of ϕ-functions, that corresponds to the thread in which **c** node appears. As a result, the dummy node has two incomplete ϕ-assignments, `C2=`ϕ`(C1,C3,-)`, and `A2=`ϕ`(-,A1,-)`. Similarly, after we visit node **d** , we get `C2=`ϕ`(C1,C3,C4)`, `A2=`ϕ`(-,A1,-)`, and `B2=`ϕ`(-,-,B1)`. We fill in those vacant arguments with the values of variables that go into node **a** after visiting node **f**. The final ϕ-assignments are `C2=`ϕ`(C1,C3,C4)`, `A2=`ϕ`(A0,A1,A0)`, and `B2=`ϕ`(B0,B0,B1)`.
4. After filling in all the vacant arguments in ϕ-functions we insert the dummy node **e**′ just before the *Coend* node. The predecessors of the *Coend* node become the predecessors of the dummy node. We create a new edge from the dummy node to *Coend* node.
5. Before visiting the successor of the *Coend* node, we discard superfluous ϕ-assignments from the dummy node **e**′. For example, the ϕ-assignment `A2 =` ϕ`(A0,A1,A0)` is superfluous. After updating the current value of `A` with `A1`, we discard it. Note that we do not update the current value of `A` with `A2`. The case of variable `B` is similar to `A`. This completes the visit to *Cobegin* node.

6. Continue to visit the successor of the *Coend* node. The current values of variables are `A1, B1, C2`.

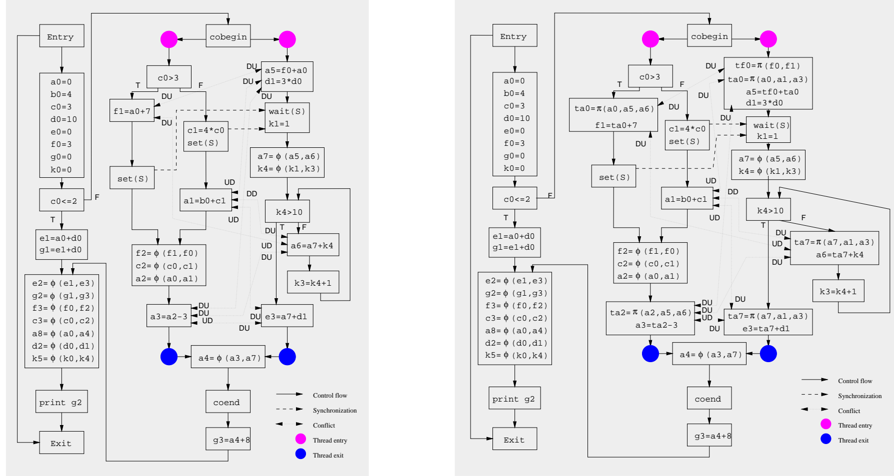

Fig. 9. ϕ-assignment placement for the code of Figure 6(A).

Fig. 10. The result CCFG of the code in Figure 6 after CSSA transformation.

The CCFG of Figure 6(B) after placing all the ϕ-assignments is given in Figure 9. The algorithm for placing ϕ-assignments is given in [8].

4.3 Placing π-assignments

In this section the placement π-assignments in a CCFG after the placement ϕ-functions is explained. Because a definition of a variable may reach a use in a different thread where no control flow join is present, there is no place to put a ϕ-function for the variable. Therefore a different notion of a join node is needed which covers the idea of conflict (*DU* edges). We begin by defining concurrent join nodes.

Definition 7 *A collection of simple paths p_i, $2 \leq i \leq n$, in a given concurrent control flow graph is said to* **converge by interleaving** *at a node y if they satisfy the following conditions.*

1. They have an end node y in common.
2. They do not have any node in common except y.
3. One and only one of these paths, say p_m, consists only of control flow edges.
4. Every path p_i, $i \neq m$, consists of only one conflict edge and the type of the edge is DU.

Definition 8 *Given a collection of paths p_i converging by interleaving at a node y, y is called the* **concurrent join node** *of the start nodes of p_i's for a variable V, if the start nodes of all p_i's contain an assignment to V, and no other node in p_i contains an assignment to V, and the collection is the largest one possible for the variable V.*

Note that by the definition of a concurrent basic block, the sequence in which a definition of a shared variable followed by the use of the shared variable never occurs in a concurrent basic block.

We locate concurrent join nodes in a given CCFG in order to place π-functions. Since a node x has an incoming *DU* edge for a variable V if and only if it is a concurrent join node of V, it is easy to locate concurrent join nodes. We just check all the *DU* edges in CCFG to locate a concurrent join node in order to place π-assignments. However we exclude the *DU* edges which violates the partial ordering among conflicting statement. To do so, we use the predicate *Precede*? defined in section 4.1 We go through an example to explain the method. Figure 11 shows the steps which is required to place π-assignments.

When we place a π-assignment for a variable in the concurrent join node of the variable, we generate a new temporary variable and assign the π-function to this variable. Then, we replace the use of the original variable with this temporary variable. Note that there is only one use of the variable in the concurrent join node by the definition of the concurrent basic block.

1. Node **b** in Figure 11(A) has three incoming *DU* edges from node **c**, **f**, and **e**. So we place a π-assignment in this node. The use of variable `A0` is replaced by the temporary `tA0`. We insert the π-assignment `tA0=`π`(-,-,-,-)` in which the number of arguments is the number of all incoming *DU* edges corresponding to the variable `A` plus 1. We fill in the first argument of the π-function with the reaching definition through the incoming control flow edge into node **a**. It is `A0`. We get `tA0=`π`(A0,-,-,-)`. We go over all incoming *DU* edges one by one. By using the predicate *Precede*? we get the partial ordering between the nodes which are the source of those *DU* edges. The Hasse diagram [5] of the partial ordering is shown in Figure 11(E). Since node **b** precedes node **e**, we do not consider the *DU* edge from node **e** for the π-function in node **b**. We remove a vacant argument in the π-function. We get `tA0=`π`(A0,-,-)`. Since there is no ordering among node **b**, **c**, and **f**, we have to consider those *DU* edges from node **c** and **f** for the π-function in node **b**. We fill in rest of the arguments with `A3, A5`. The order of argument is not important except the first one. Finally, the π-assignment become `tA0=`π`(A0,A3,A5)`.
2. When we visit node **e**, we get the ordering among the nodes in Figure 11(F). Similar to the visit on node **b**, we get the initial π-assignment `tA3=`π`(A3,-,-)`.

[5] If R is a partial order on a set A, we construct a Hasse diagram for R by putting an edge between x and y, if $x, y \in A$ with xRy and if there is no other element $z \in A$ such that xRz and zRy. In addition, x is located above y in the diagram.

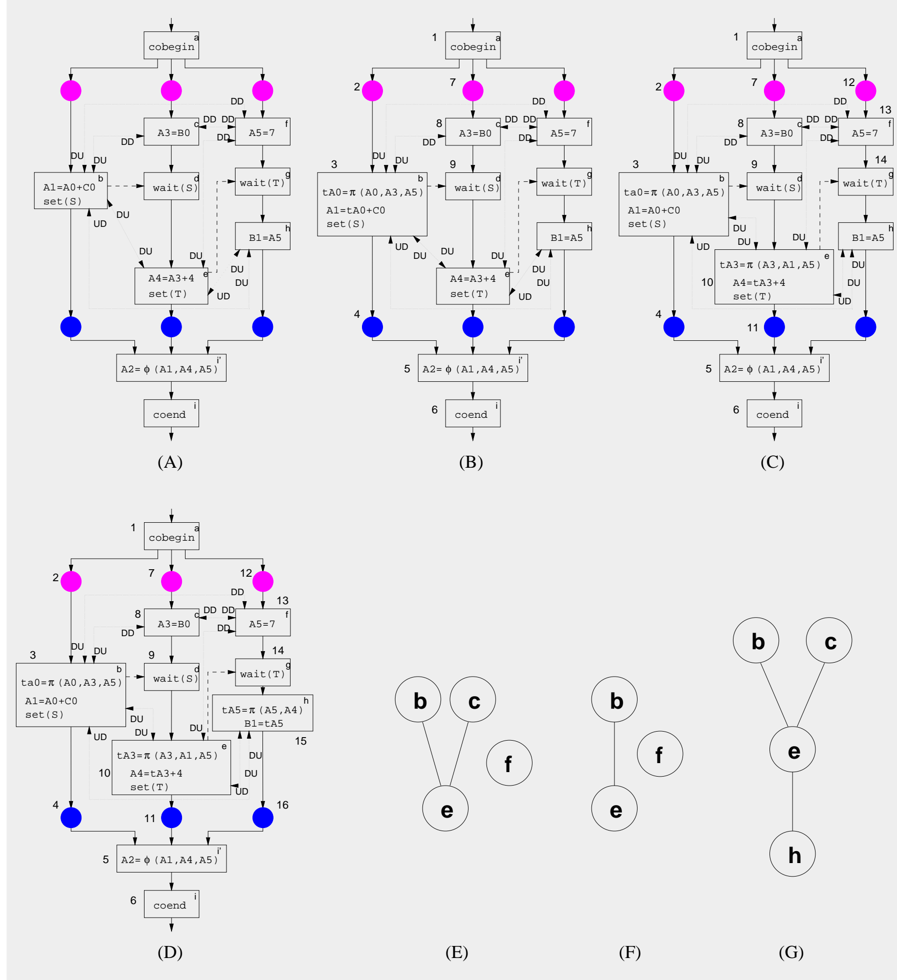

Fig. 11. Placing π-assignments in a given CCFG after placing ϕ-assignments

Since we do not have the ordering among node **e** and **f** and node **b** precedes **e**, We fill in the π-function with `A1, A5`. We get `tA3=`π`(A3,A1,A5)`.

3. When we visit node **h**, we get the partial ordering between node **b**, **c**, **e**, and **h** in Figure 11(G). Similar to the visit on node **b**, we get the initial π-assignment `tA5=`π`(A5,-,-,-)`. Since both node **b** and **c** precede node **e** and node **e** precedes node **h**, we do not count the definition from node **b** and **c**. This is because the definitions of the variable `A` from node **b** and **c** are killed by the definition in node **e**. Thus, we remove two vacant arguments in `tA5=`π`(A5,-,-,-)` and get `tA5=`π`(A5,-)`. We fill in the vacant argument with the value `A4` from node **e** and get `tA5=`π`(A5,A4)`.

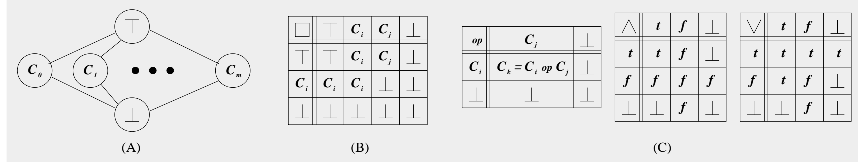

Fig. 12. (A) The lattice for constant propagation. (B) The meet operation $\sqcap$ for the lattice.($i \neq j$) (C) The expression evaluation rules for arithmetic operations **op** such as `+`, `-`, `*`, `/` and logical operations such as $\wedge$, $\vee$.

The final CSSA form of the code in Figure 6 is given in Figure 10. The algorithm for placing π-assignments is given in [8].

5 Concurrent Sparse Conditional Constant Propagation

In this section, we describe a constant propagation algorithm for explicitly parallel programs based on CSSA representation which is an extension of the Sparse Conditional Constant Propagation (SCC) (Wegman and Zadeck[15]) to explicitly parallel programs. We call this extended algorithm the Concurrent Sparse Conditional Constant (CSCC) propagation algorithm.

5.1 The lattice for constant propagation

The lattice used in constant propagation is shown in Figure 12(A). During constant propagation, each variable used or defined in the program is mapped to an element of the lattice. There are three types of lattice elements. $\top$ is the highest element of the lattice and means that nothing may yet be asserted about the variable in question. $\bot$ is the lowest element of the lattice and means that the variable in question has been determined not to have a constant value. C_i is less than $\top$ and greater than $\bot$ in the lattice and means that the variable has the constant value C_i. There are infinite number of C_i's in the lattice. The meet operation $\sqcap$ of the lattice is defined in Figure 12(B). In addition, we need rules for evaluating expressions. These rules are summarized in Figure 12(C). The case where an operand in an expression has the value $\top$ never occurs in CSCC algorithm if all the variables are initialized in the original program.

5.2 The concurrent sparse conditional constant propagation algorithm

After we transform an explicitly parallel program into its CSSA form, we add def-use edges between statements. We call them CSSA edges (analogous to SSA edges in [15]).

Definition 9 *A* **CSSA edge** *goes from the statement where a variable is defined to a use of the variable. The edges may go between statements in different threads.*

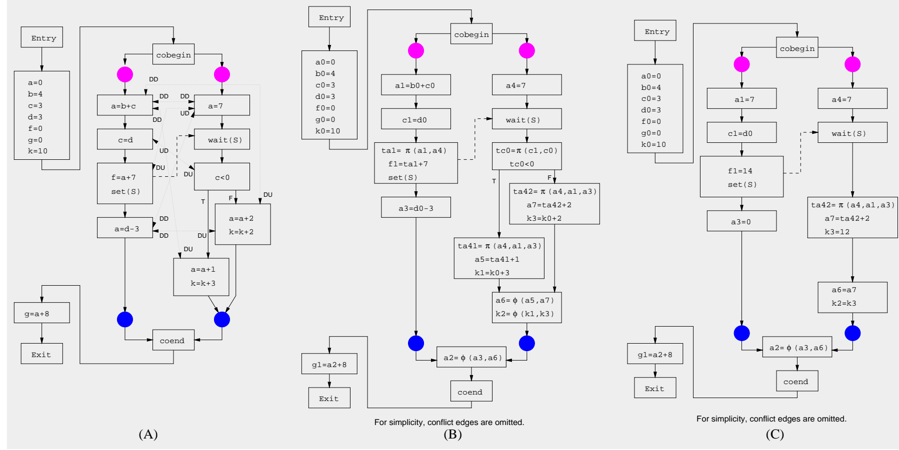

Fig. 13. (A) An example CCFG for constant propagation. (B) CSSA form. (C) After constant propagation is done.

Each node n in the given CCFG has a function $n.LatticeValue$. For each variable v which appears in node n, $n.LatticeValue$ maps v to its lattice value in node n.

The algorithm proceeds by modifying the mapping $n.LatticeValue$. The value corresponding to each variable is lowered as more information for the variable is discovered. It continues until a fixed point is reached. All the variables map to $\top$ initially. However, the variable, which appears in `read` or is uninitialized in the original CCFG, initially maps to $\bot$. Without loss of generality, we assume all the variables are initialized in the original program.

The algorithm is a work list algorithm. A work list $CTFE$ contains control flow edges and the other work list $CSSAE$ contains CSSA edges. Initially, the $CTFE$ contains *Entry* node and $CSSAE$ is empty.

Each control flow edge has a flag *ExecutionFlag* for the possibility of execution of the edge. If an edge is executable, its *ExecutionFlag* is **True**, otherwise it is **False**. Each node has a flag *ExecutionFlag*. Its function is the same as the one of control flow edges.

The full Concurrent Sparse Conditional Constant Propagation algorithm is given in [8]. Figure 13(A) is the initial CCFG before CSSA transformation. Figure 13(B) is the one after CSSA transformation. The result of the constant propagation is given in Figure 13(C). In this figure, all the unreachable nodes are removed. Figure 14 shows the correct constant propagation for the example in the Figure 1.

The essential difference between CSCC and SCC[15] is the existence of π-function and the definition of CSSA edges. When we evaluate a π-function, we apply the meet operation $\sqcap$ of the lattice to all of the arguments in the π-function. For example, assume the arguments a3, a6, and a8 of a π-assignment

```
flag = 0
b = 4
cobegin
  while flag = 0 do
  endwhile
  a = b
  print a
//
  b = 3
  flag = 1
coend
```

(A)

```
flag0 = 0
b0 = 4
cobegin
  while π(flag0,flag1) do
  endwhile
  tb = π(b1,b2)
  a1 = tb
  print a1
//
  b2 = 3
  flag1 = 1
coend
b3 = φ(b1,b2)
```

(B)

```
flag0 = 0
b0 = 4
cobegin
  while π(flag0,flag1) do
  endwhile
  tb = π(b1,b2)
  a1 = tb
  print a1
//
  b2 = 3
  flag1 = 1
coend
b3 = φ(b1,b2)
```

(C)

Fig. 14. Correct constant propagation in the presence of busy-waiting by using concurrent sparse conditional constant propagation. (A) Before constant propagation. (B) CSSA form. (C) After concurrent sparse conditional constant propagation. There is no difference between (B) and (C); i.e., the constant propagation is correct in this case.

`ta3=`π`(a3,a6,a8)` have the lattice value **3**, **3**, and $\top$, respectively, after constant propagation. `a8` has the value $\top$ means that the definition node of `a8` is unexecutable. `ta3` has the constant value **3** by applying $\sqcap$ operation to `a3`, `a6`, and `a8`. Thus, this assignment is replaced by `ta3=3`. CSSA edges contains def-use edges across different threads in addition to SSA edges[15]. It is easy to find CSSA edges since all the uses of conflicting variables are the arguments of π-functions in a given CSSA form of a CCFG.

5.3 Conservativeness of CSCC

We show in this section that the CSCC algorithm is conservative in the sense that CSCC does not say that a variable is a constant if it is not. It is well known that the SCC algorithm is conservative [1]. We show next that CSCC treats π-function conservatively in the presence of inexactly introduced arguments. These inexact arguments may be introduced by the inexact synchronization analysis (the common ancestor algorithm which is used to get guaranteed ordering among statements). One of these situations is depicted in Figure 16. The argument `a2` in the π-assignment is inexactly introduced due to the lack of information about execution ordering. This comes from common ancestor algorithm at the stage of synchronization analysis. The situation can be avoided by the more exhaustive synchronization analysis as mentioned before[7].

Without loss of generality, we assume the π-function has only two arguments v_0 and v_1 and v_1 is inexactly introduced. We apply the meet operation $\sqcap$ to the arguments of the π-function for all possible cases of v_1. The result of evaluation is in Figure 15.

The 'exact result' in the above table means the result of evaluating the π-function without inexactly introduced arguments. In this case, it is the same as the value of v_0. Since the value of π-function is not higher than 'exact result' in the lattice, the inexactly introduced argument v_1 does not do any harm in

value of v_0	value of v_1	value of $\pi(v_0, v_1)$	exact result
v_0	$\top$	v_0	v_0
C	C	C	C
C	K	$\perp$	C
v_0	$\perp$	$\perp$	v_0

Fig. 15. The result of evaluation of the π-function with two arguments v_0 and v_1. $(C \neq K)$

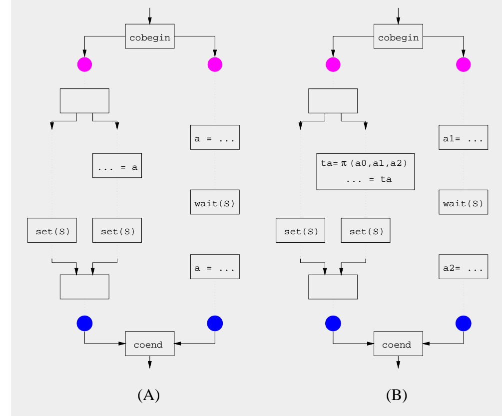

Fig. 16. (A) Before CSSA transformation. (B) After CSSA transformation.

constant propagation. However, the number of constants we will have found is less than the number of constants without inexactly introduced arguments.

6 Conclusions and Future work

We introduced concurrent control flow graphs (CCFG) which contains information about conflicting statements in explicitly parallel programs. The CCFG is used as an intermediate representation for explicitly parallel programs in order to convert them into concurrent static single assignment form. The concurrent static single assignment (CSSA) form proposed in this paper is for explicitly parallel programs with interleaving semantics and post-wait synchronization. It uses both ϕ- and π-assignments. The π-function is a new concept and it summarizes the interleaving of statements at the point where it is placed. Using this concurrent static single assignment form as an intermediate representation, we extended the classical sparse conditional constant propagation algorithm[15] to one for explicitly parallel programs. We call it the concurrent sparse conditional constant propagation algorithm.

An extended version of this paper[8] describes the application of the CSSA data structure to the common subexpression elimination problem.

An extension of the work in this paper will be CSSA form for the `parallel do` constructs with post-wait synchronization. Our goal is to develop conservative and efficient scalar optimization techniques (e.g. common subexpression elimination, partial redundancy elimination, code motion, register allocation) that avoid the above mentioned constraints for explicitly parallel programs. These scalar optimization techniques also could be applied in instruction level to explicitly parallel programs for code scheduling and register allocation.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.
2. Mark M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single assignment form for structured languages. *ACM Transactions on Programming Language and Systems*, 16(6):1684–1698, 1994.
3. Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 159–170, June 1994.
4. David Callahan and Jaspal Subhlok. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, May 1988.
5. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, and Mark N. Wegman. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
6. Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Proceedings of Supercomputing*, pages 580–588, 1989.
7. Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Detecting nondeterminacy in parallel programs. *IEEE Software*, pages 69–77, January 1992.
8. Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and concurrent sparse conditional constant propagation for explicitly parallel programs. Technical Report TR#1525, CSRD, University of Illinois at Urbana-Champaign, July 1997.
9. S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing, Vol. II Software*, pages 105–113, August 1990.
10. Robert H. B. Netzer and Barton P. Miller. On the complexity of event ordering for shared memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing, Vol. II Software*, pages 93–104, August 1990.
11. Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Gloabal value numbers and redundant computations. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Laguages*, pages 12–27, January 1988.
12. Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In *The Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, August 1993.
13. Harini Srinivasan and Dirk Grunwald. An efficient construction of parallel static single assignment form for structured parallel programs. Technical Report CU-CS-564-91, University of Colorado at Boulder, December 1991.
14. Harini Srinivasan, James Hook, and Michael Wolfe. Static single assignment for explicitly parallel programs. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 260–272, January 1993.
15. Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

16. Michael Wolfe and Harini Srinivasan. Data structures for optimizing programs with explicit parallelism. In *Proceeedings of the First International Conference of the Austrian Center for Parallel Computation*, September 1991.

This article was processed using the LaTeX macro package with LLNCS style