Data-Race Detection: The Missing Piece for an End-to-End Semantic Equivalence Checker for Parallelizing Transformations of Array-Intensive Programs

Kunal Banerjee [*]
Dept of Computer Sc & Engg
IIT Kharagpur, India
kunalb@cse.iitkgp.ernet.in

Soumyadip Banerjee   Santonu Sarkar
Dept of CSIS
BITS Pilani- Goa, India
{soumyadipb, santonus}@goa.bits-pilani.ac.in

Abstract

The parallelizing transformation (hand-crafted or compiler-assisted) is error prone as it is often performed without verifying any semantic equivalence with the sequential counterpart. Even when the parallel program can be proved to be semantically equivalent with its corresponding sequential program, detecting data-race conditions in the parallel program remains a challenge. In this paper, we propose a formal verification approach that can detect data-race conditions while verifying the computational semantic equivalence of parallelizing loop transformations. We propose a coloured array data dependence graph (C-ADDG) based modeling of a program for verification of program equivalence as well as data-race condition detection across parallel loops. We have tested our tool on a set of Rodinia and PLuTo+ benchmarks and shown that our method is sound, whereby the method does not produce any false-positive program equivalence or data-race condition.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Translation validation, coloured array data dependence graph (C-ADDG), parallelizing transformation, data-race

1. Introduction

Massively parallel computing infrastructure comprising of coprocessors like GPGPU can provide a significant speedup when the processing logic of a sequential program that deals with a large amount of data (hereinafter we refer to such a program as an array-intensive program) is suitably transformed into a data-parallel counterpart that can maximally exploit the computing power of the underlying data-parallel processor.

Even though there have been a significant research effort to improve parallelizing compilers, it has still not found inroads to the experienced programmer community who often resorts to a hand-crafted transformation of a sequential program, followed by careful tuning of the transformed program for the optimal performance. In such a case, it is all the more important to verify if the parallel transformation (i) preserves the semantic equivalence of the sequential code and (ii) does not contain problems related to the parallel execution of the code, namely the data-race conditions, delay in memory access due to conflicts, cache misses and so on.

In this paper, we propose a verification approach, implemented as a prototype tool, to detect data-race conditions in a parallel program which may be introduced during the parallelizing transformation. This approach is different because our verification technique is coupled with the equivalence checking approach. Therefore, it works on the model of the program which is used to detect the computational equivalence of programs. However, typical program verification techniques rely on property verification which create a set of predicates from a parallel program and check for the satisfiability. Our approach extends the existing array data dependence graph (ADDG) model (Shashidhar 2008) of a program that captures the data transformation and the iteration domain of a loop jointly. The extended model C-ADDG brings the notion of multi-threaded execution that is necessary to detect possible data-race situations. Subsequently, we propose a verification technique to detect data-race conditions which works along with the computational equivalence checking. Finally, we propose an end-to-end verification framework for translation validation of parallelizing transformations, which involves additional steps such as, converting a program into single assignment (SA) form. Program verification being an undecidable problem, our method is sound, i.e., it can correctly report an equivalence, without any false-positive cases. However, when it reports a non-equivalence, the two programs may be equivalent; in such cases, our tool reports *possibly* faulty program paths which serve as strong hints for program correction. In summary, the contributions of this paper are: (i) a data-race detection mechanism in the context of equivalence checking, and (ii) an end-to-end verification framework for parallelizing transformations of programs.

The rest of the paper has been organized as follows. Section 2 highlights the verification challenges using a motivating example. Section 3 illustrates the methodology with this running example. We formally introduce the C-ADDG model and its computational semantics in Section 4. Identification of data-race conditions is explained in Section 5. Next we describe the overall verification approach along with its soundness and complexity in Section 6. The experimental result in Section 7 illustrates testing of the verifier on a variety of benchmark applications. Section 8 provides an overview of existing work, and Section 9 concludes the paper with the limitations of the current approach and a future direction.

2. Motivating Example

To explain our approach, let us consider the example given in Figure 1; Figure 1(a) shows the original sequential program while Figure 1(b) shows its parallelized counterpart using OpenMP constructs. Let us observe the changes in the transformed version other than the obvious parallelization.

[*] K. Banerjee is presently a Research Scientist at Intel Parallel Computing Lab, Bangalore, India.

ARRAY'16, June 14, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4384-8/16/06...$15.00
http://dx.doi.org/10.1145/2935323.2935324

```
void function1(
 int A[], int Z[]) {
  int i, B1[1000], B2[1000];
  S0: Z[0] = A[0];
  for (int i=1; i<N; i++) {
    S1: B1[i] = A[i];
    S2: B2[i] = A[N-i-1];
    S3: Z[i] = Z[i-1] +
          B1[i]*B2[i] + B1[i];
  }
}
          (a)
```

```
void function2(int A[], int Z[]){
  int i, t, B[1000];
  T0: Z[0] = A[0];
  #pragma omp parallel {
    #pragma omp for nowait
    for (i=0; i<N; i++) {
      T1: B[i] = A[i];
    }
    #pragma omp for
    for (i=0; i<N; i++)
    {
      T2: t = N-i-1;
      T3: Z[i] = B[t]+1;
      T4: Z[i] = Z[i]*B[i]+Z[i-1];
} } }
              (b)
```

Figure 1. (a) Original sequential program. (b) Transformed parallel program.

Loop fission: The `for` loop in the original program has been split into two.
Elimination of temporary array variable: Instead of two temporary array variables `B1` and `B2` in the original program, the transformed program has only one namely, `B`.
Introduction of temporary scalar variable: The variable `t` appears in the transformed program.
Non-compliance to SA form: In the original program, each variable (array element) is assigned a value only once and hence it is in SA form whereas, in the transformed program, the array element `Z[i]` is defined twice in statements `T3` and `T4` in each iteration of the second `for` loop in Figure 1(b); also the scalar variable `t` is redefined across all iterations of the second `for` loop.
Application of arithmetic transformations: The operation in statement `S3` in the original program has been captured in the statements `T3` and `T4` in the transformed program by using the following properties of arithmetic expressions: (a) commutativity of the addition operation, (b) distributive property of multiplication over addition.

It is important to note that if we ignore the parallelizing constructs in the transformed program (i.e., treat Figure 1(b) as a sequential code), then the programs are semantically equivalent; however, parallelism introduces data-race in the transformed program which renders the programs in Figure 1 non-equivalent – this clearly underlines the significance for data-race detection.

To verify the semantic equivalence of the programs in Figure 1, the verification methodology must be able to address: (i) loop transformations, (ii) arithmetic transformations, and (iii) synchronization transformations.

Furthermore, (iv) the sequential and parallel versions involve recurrences (`S3` and `T4`), whereby an element of an array gets defined in a statement S inside a loop in terms of some other element(s) of the same array which have been previously defined through the same statement S, and (v) the program Figure 1(b) is not in SA form (note that exact data flow analyses can be performed for programs in SA form).

The computational equivalence checking approach namely, (Shashidhar 2008) can handle loop transformations only (i.e., (i) but not (ii)–(v)). Recent work by (Verdoolaege et al. 2012) can only deal with (i), (iv) and (v). The work by (Schordan et al. 2014) can handle (i) and (ii) to some extent for constant loop bounds. Another work (Karfa et al. 2011, 2013) can handle (i) and (ii) to a larger extent while allowing parametric loop bounds (such as, `N` in Figure 1). The method reported in (Banerjee 2016) extends the work of (Karfa et al. 2013) to address (iv). The approach proposed here further extends the capability of (Banerjee 2016) to handle synchronization transformations. Additionally, conversion of a program from non-SA form to SA form is carried out using the LLVM compiler (Lattner and Adve 2004). Thus, our work for the first time, to the best of our knowledge, addresses the full spectrum

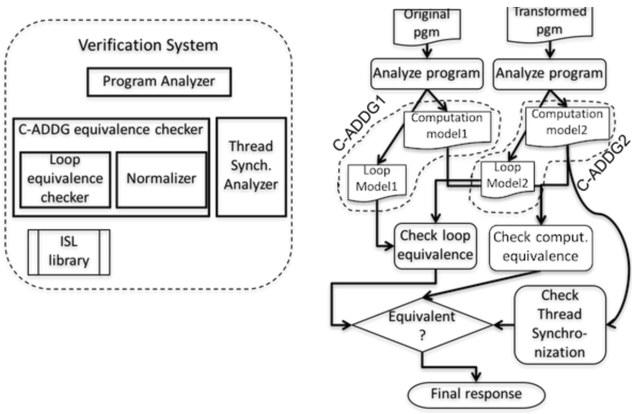

Figure 2. Tool Overview and Verification Process

of semantic equivalence checking between a sequential program and its parallelized counterpart.

3. Methodology

A schematic diagram of the verification tool is shown in Figure 2(a). The tool comprises of three main modules namely the *program analyzer*, the *C-ADDG based equivalence checker* and the *Thread Synchronizer Analyzer*. The program analyzer module processes a `C` source code to generate its corresponding C-ADDG model. A detailed description of the model, the equivalence checking calculus and the equivalence checker are described in the subsequent sections. The C-ADDG equivalence checker has two important submodules namely, the loop equivalence checker and the normalizer. The loop equivalence checker uses an external ISL library (Verdoolaege 2010) to verify the equivalence of loop expressions. The normalizer module is used to evaluate integer expressions. While there are several techniques to represent a Boolean expression in a canonical form, no canonical form exists for integer expressions. For this purpose, we have adopted the normalization technique from our previous work (Karfa et al. 2013; Banerjee et al. 2014b). The normalizer is used for comparing two arithmetic expressions for their equivalence. The thread synchronizer module analyzes array variables that are accessed concurrently by multiple threads and checks for any data inconsistency that may arise after the completion of a data parallel loop.

3.1 Verification Activities

Here the tool takes the sequential program and the transformed parallel program and generates their corresponding C-ADDG models, which is an extension of the ADDG model as explained later. The ADDG model provides a formalism that can succinctly model subscripts to array variables and loop bounds that are (piece-wise) affine expressions (Shashidhar 2008). As a result, effective equivalence checking mechanism (such as the one described in this work) can be used to prove equivalence for groups of elements at once.

The loop equivalence checker module compares equivalence of the iteration space of the source and the target program. The computation models of the source and the target program are also analyzed for their equivalence. A flowchart describing various activities of the verification tool is shown in Figure 2(b). We illustrate the overall equivalence checking technique with the help of the example shown in Figure 1.

Program Analysis This step analyzes the input programs to determine whether they can be directly fed to the C-ADDG equivalence checker or whether some preprocessing needs to be carried out to make them amenable for our checker. In particular, a preprocessing

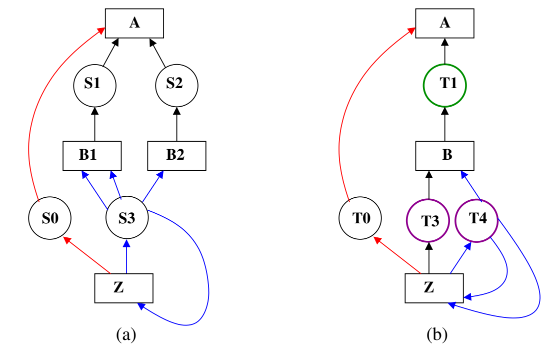

Figure 3. (a) C-ADDG of the program in Figure 1(a). (b) C-ADDG of the program in Figure 1(b).

step is applied to the programs given in Figure 1(b) to convert them into SA form. A program which is not presently in SA form can be made to be so by increasing the dimensionality of the variables of the program which are being written into (Vanbroekhoven et al. 2007). For example, let us consider the statements T3 and T4 in Figure 1(b). The SA form of these statements are:

```
    Z[i][0] = B[t]+1;
    Z[i][1] = Z[i][0]*B[i]+Z[i-1][1];
```

To maintain uniformity of array dimensionality, statement T0 is converted to:

```
    Z[0][1] = A[0];
```

The scalar variable t in Figure 1(b), similarly, can be converted into a single dimensional array to conform to SA form. Without loss of generality, henceforth we shall consider all variables (except the loop iterators) in a program as array variables; note that scalar variables are basically treated as array variables of zero dimension.

ADDG Construction The ADDG corresponding to Figure 1(a) and Figure 1(b) are given in Figure 3(a) and 3(b) respectively. A formal definition of C-ADDG will be given later in Definition 1. Informally, a node in ADDG can either represent an array variable, or an operator (or a barrier which is required for synchronization purpose, as elaborated later). For brevity, an operator node has been marked with the corresponding statement id. For statement S3, the operator node has an incoming edge (hereinafter called write-edge) from array node Z and four outgoing edges (hereinafter called read-edges) to arrays B1, B1, B2 and Z; note that there are two edges to array B1 because this array appears twice in the right hand side (rhs) expression twice. During ADDG construction, each operator node also stores information about the loop body where the corresponding statement belongs.

It may be important to note that whenever an array variable is assigned with a constant value (not shown in this example), we assume that those values are supplied by some fictitious array having dimension equal to that of the array variable being defined and each member of this fictitious array has value equal to that constant.

Recurrence Identification Recurrences lead to cycles in an ADDG. Since the same array may be recursively defined through multiple statements, instead of simple cycles, strongly connected components (SCCs) are identified using the mechanism proposed by Tarjan (Tarjan 1972). However, cycles (SCCs) in an ADDG do not always mean recurrences; for example, the cycle involving Z[i] on the rhs of the statement T4 in Figure 3(b) does not represent a recurrence because the variable Z[i] defined in statement T4 does not depend upon an earlier definition of Z[i] through the same statement, it simply depends on the definition of Z[i] at statement T3. The cycle involving Z[i-1] (in statements S3 and T4), on the other hand, does represent a recurrence because it has a loop-carried dependency on the same statement. We refer to such a vertex (Z) as a recurrence array vertex (Banerjee 2016).

Recurrence subgraph construction Once an SCC is found to lead to a recurrence, we need to construct a subgraph from the ADDG which represents the recurrence. For this, we first identify a basis subgraph. To illustrate the process, let us consider Figure 3(a). Here the SCC comprises only two nodes: Z and S3 with Z being the recurrence array vertex. The basis subgraph consists of the nodes Z, S0 and A along with the connecting edges shown in red colour. Next, we create an induction subgraph which consists of the nodes Z, S3, B1, B2 along with the connecting edges shown in blue colour. Finally, the recurrence subgraph in Figure 3(a) comprises the nodes Z, S0, A, S3, B1, B2 and the connecting edges shown in red and blue colours. Similarly, the recurrence subgraph in Figure 3(b) is also constructed whose connecting edges have also been shown in matching red and blue colours. The details about the recurrence subgraph construction can be found in (Banerjee 2016).

Extension of the recurrence subgraph It is to be noted that till now all the steps were carried out in isolation for each ADDG, that is, we did not try to compare the original and the transformed ADDGs prior to this step. However, now we need to compare the recurrence subgraphs from the two ADDGs. In order to do so, the elements of the recurrence array vertex must be defined in terms of the elements of some *common arrays*, i.e., arrays which appear in both the original and the transformed programs. In Figure 3(a), the recurrence array vertex Z is defined in terms of A, B1 and B2 whereas, in Figure 3(b), the recurrence array vertex Z is defined in terms of A and B. Thus, there is a difference due to the temporary arrays B1, B2 and B; also, there is a mismatch in the definition of Z in the two recurrence subgraphs (vide statements S3 and T4 in Figure 1). So, the recurrence subgraphs are extended until Z can be defined in terms of some common arrays only. As it happens in case of Figure 3, extension results in the recurrence subgraphs covering the entire original and the transformed C-ADDGs, i.e., Z gets defined in terms of A.

In case there are multiple recurrence subgraphs in each C-ADDG then we pair up individual extended recurrence subgraphs (one from each C-ADDG) based on the common arrays appearing in a recurrence subgraph. Failure to pair up some recurrence subgraph results in immediate termination of the equivalence checker reporting possible non-equivalence and that recurrence subgraph as the possible source of mismatch between the two C-ADDGs (programs).

Equivalence of recurrence subgraphs To establish the equivalence of two recurrence subgraphs, one has to first establish equivalence of their basis subgraphs followed by establishing equivalence of their induction subgraphs. Note that after converting to SA form, we are now trying to establish equivalence of $Z_1[i] = Z_2[i][1]$, $\forall i$, $0 \leq i < N$, where Z_1 and Z_2 represent the values assumed by the array Z in the source program and the transformed program, respectively. Obviously, the data computations and the array index spaces for the basis subgraphs (statements S0 and T0) are equivalent, i.e., $Z_1[0] = Z_2[0][1]$. Next, we concentrate on the induction subgraphs and compute their data transformations by employing the normalizing technique described in (Karfa et al. 2013; Banerjee et al. 2014b). It can be deduced using backward substitution and applying the distributive and the commutative properties of arithmetic expressions mentioned in Section 2 that $Z[i] = A[i] + A[i] * A[N - i - 1] + Z[i - 1]$ in the original program whereas, for the transformed program, the data transformation is the same with $Z[i]$ and $Z[i - 1]$ replaced by $Z[i][1]$ and $Z[i - 1][1]$, respectively. The index spaces of the two induction subgraphs can be verified to be equivalent using a standard equiva-

lence checker for polyhedral models (Verdoolaege 2010). This step establishes $Z_1[i] = Z_2[i][1],\ \forall i,\ 1 \leq i < N$, and thus concludes the proof of equivalence of the recurrence subgraphs.
A pertinent point to note is that in a normalized expression all the variables are ordered using a consistent ordering of the variable names in the program; when the same array variable with different index expressions occur in the same normalized expression (such as, `Z` in statement `T4`), they are ordered according to the variables appearing in their index expressions (Karfa et al. 2013); this is in sharp contrast with the method of (Verdoolaege et al. 2012) which tries out all different permutations for such arrays which occur multiple times.

Equivalence of recurrence-free subgraphs Had there been recurrence-free subgraphs in the ADDGs of Figure 3, then those subgraphs could have been proved to be equivalent by the conventional ADDG based equivalence checking method of (Karfa et al. 2013). Basically, for establishing equivalence of recurrence-free subgraphs of ADDGs, one does not have to incur the overhead of identifying basis and induction subgraphs and establish their equivalence in isolation.

Synchronization Checking Next, the thread synchronization checker module (*checkSynch()*) takes the C-ADDG model corresponding to the transformed program and determines whether the transformed program has barrier synchronization constructs to avoid data-races. In the current implementation the tool is capable of understanding a subset of the OpenMP constructs. Note that if we ignore the parallel constructs (`#pragma`'s) in the program shown in Figure 1(b), i.e., we consider it as a sequential program, then the two programs in Figure 1 are indeed equivalent; however, presence of multiple threads without proper synchronization actually renders them (potentially) non-equivalent. We shall deliberate on the synchronization issue and data-race detection in Section 5. Once the source and the transformed programs are found to be equivalent with respect to the array index spaces and data computations and the transformed program is found to be free from data-races, our tool declares that the source and the transformed programs are equivalent. The algorithm is described in Algorithm 1.

4. The C-ADDG Model and its Computational Semantics

Let us consider the following generalized loop structure for the model.

```
for($i_1 = L_1; i_1 \leq H_1; i_1 += r_1$)
  for($i_2 = L_2; i_2 \leq H_2; i_2 += r_2$)
    .....
    for($i_x = L_x; i_x \leq H_x;\ i_x += r_x$)
      if ($C$)
        S : $Z[e_1] \ldots [e_k] =$
            $f(Y_1[e'_{1,1}] \ldots [e'_{1,l_1}], \ldots, Y_m[e'_{m,1}] \ldots [e'_{m,l_m}]);$
```

A program in *single assignment form with a static control flow, affine indices and bounds, and a valid schedule* can be represented as an ADDG (Shashidhar 2008). An in-depth analysis of the ADDG model and its associated methodology can be found in (Shashidhar 2008; Karfa et al. 2013); here we present only the essential concepts required to comprehend the extended model.

DEFINITION 1 (Coloured-ADDG (C-ADDG)). *The C-ADDG is a directed graph $G = (V, E, \mathcal{H}, \mathcal{L})$, where the vertex set $V = \mathcal{A} \cup \mathcal{F}$ and the edge set $E = \{\langle A, f\rangle \mid A \in \mathcal{A}, f \in \mathcal{F}\} \bigcup \{\langle f, A\rangle \mid f \in \mathcal{F}, A \in \mathcal{A}\}$. The set $\mathcal{A}$ is a set of array nodes and $\mathcal{F}$ is a set of operator and barrier nodes denoted by f and b respectively. Edges of the form $\langle A, f\rangle$ or $\langle A, b\rangle$ are called write edges; they capture the dependence of the left hand side (lhs) array node on the operator corresponding to the right hand side (rhs) expression. Edges of the form $\langle f, A\rangle$ or $\langle b, A\rangle$ are called read edges; they capture the dependence of the rhs operator on the (rhs) operand arrays.*

The loop labeling function $\mathcal{L} : \mathcal{F} \to \mathbb{Z}$ associates an integer number to each operator node.

The function $\mathcal{H} : \mathcal{F} \to \mathscr{C}$ marks each operator node in the C-ADDG a colour from the set $\mathscr{C}$ of infinite colours.

While the above definition captures the structure of the graph, the association of program constructs with various C-ADDG nodes is defined as follows:

DEFINITION 2 (C-ADDG Program Model). *An assignment statement S of the form $Z[\overline{e_z}] = f(Y_1[\overline{e_1}], \ldots, Y_k[\overline{e_k}])$, where $\overline{e_1}, \ldots, \overline{e_k}$ and $\overline{e_z}$ are respectively the vectors of index expressions of the arrays $Y_1, \ldots, Y_k$ and Z, appears as a subgraph G_S of G, where $G_S = \langle V_S,\ E_S\rangle$, $V_S = \mathcal{A}_S \bigcup \mathcal{F}_S$, and $\mathcal{A}_S = \{Z, Y_1, \ldots, Y_k\} \subseteq \mathcal{A}$, $\mathcal{F}_S = \{f\} \subseteq \mathcal{F}$ and $E_S = \{\langle Z, f\rangle\} \bigcup \{\langle f, Y_i\rangle,\ 1 \leq i \leq k\rangle\} \subseteq E$. The write edge $\langle Z, f\rangle$ is associated with the statement name S. If the operator associated with an operator node f has an arity k, then there will be k read edges $\langle f, Y_1\rangle, \ldots, \langle f, Y_k\rangle$. The operator f applies over k arguments which are elements of the arrays $Y_1, \ldots, Y_k$, not necessarily all distinct.*

The loop labeling function $\mathcal{L}(f)$ denotes the innermost loop body in which the statement corresponding to f belongs. $\mathcal{L}(f) = 0$ indicates that f does not belong to any loop.

A barrier statement is associated with a barrier node $b \in \mathcal{F}$. For a barrier node b, its data transformation is considered to be identity. At a barrier, for each array node corresponding to an array variable defined within that parallel construct, a duplicate array node (with the same array name) is created with an intermediary barrier operator node b.

After defining the program model, we now define the colour mapping.

DEFINITION 3 (Colour mapping). *The colour set $\mathscr{C}$ of infinite colours has two special colours – black and blue – that imposes the following constraints on the mapping function $\mathcal{H}$:*
(i) the operator nodes (f) corresponding to statements which are executed by a single thread are marked with black colour,
(ii) operator nodes corresponding to all statements in a parallel for construct are marked with a distinct colour,
(iii) a barrier node (b) is marked with blue colour.

We now formalize the notion of the iteration domain.

DEFINITION 4 (Iteration domain of statement S (I_S)). *For each statement S within a generalized loop structure with nesting depth x, the iteration domain I_S of the statement is a subset of $\mathbb{Z}^x$ defined as*

$$I_S = \{[i_1, i_2, \ldots, i_x] \mid C \wedge \bigwedge_{k=1}^{x} (L_k \leq i_k \leq H_k \wedge \exists \alpha_k \in \mathbb{Z}(i_k = \alpha_k r_k + L_k))\},$$ *where, for $1 \leq k \leq x$, the loop iterators i_k are integers, L_k, H_k are affine expressions over the loop iterators and some integer variables, and r_k are integer constants.*

Using the iteration domain, we define the notion of usage and definition of array variables.

DEFINITION 5 (Definition mapping (${}_SM_Z^{(d)}$)). *The definition mapping of a statement S describes the association between the elements of the iteration domain of the statement S and the elements of its lhs array Z (depicted as a suffix). ${}_SM_Z^{(d)} = I_S \to \mathbb{Z}^k$ s.t. $\forall \overline{v} \in I_S,\ \overline{v} \mapsto [e_{Z,1}(\overline{v}), \ldots, e_{Z,k}(\overline{v})] \in \mathbb{Z}^k$.*

The range of the function ${}_SM_Z^{(d)}$ is called the *definition domain* of the lhs array Z, defined as ${}_SD_Z$. So, ${}_SD_Z = {}_SM_Z^{(d)}(I_S)$. Due

```
#pragma omp parallel
{
 #pragma omp for nowait
 for (i = 1; i < 100; i++) {
  T1: B[i] = A[i] * A[i];
  T2: D[i] = g(B[i], B[i-1]);
 }
 #pragma omp for
 for (i = 1; i < 50; i++)
  T3: C[i] = B[i]*2;
}//barrier b
#pragma omp parallel for
for (i = 50; i < 100; i++)
 T4: C[i] = B[i]*2;
        (a)
```

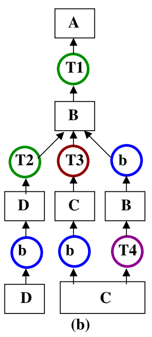

Figure 4. (a) An OpenMP program. (b) Its corresponding C-ADDG.

to single assignment form, each element of the iteration domain of a statement defines exactly one element of the lhs array of the statement. Therefore, the mapping between the iteration domain of the statement and range of ${}_SM_Z^{(d)}$ is injective (one-one). Hence, $({}_SM_Z^{(d)})^{-1}$ exists.

In a similar way, the usage mapping for each operand array variable of a statement can be defined as follows.

DEFINITION 6 (Usage mapping (${}_SM_{Y_n}^{(u)}$)). *The n^{th} operand mapping of a statement S describes the association between the elements of the iteration domain of the statement S and the elements of its rhs array Y_n; specifically, ${}_SM_{Y_n}^{(u)} = I_S \rightarrow \mathbb{Z}^{l_n}$ s.t. $\forall \overline{v} \in I_S$, $\overline{v} \mapsto [e_{n,1}(\overline{v}), \ldots, e_{n,l_n}(\overline{v})] \in \mathbb{Z}^{l_n}$.*

The range ${}_SM_{Y_n}^{(u)}(I_S)$ is the *operand domain* of the rhs array Y_n in the statement S, denoted as ${}_SU_{Y_n}$. One element of the operand array Y_n may be used to define more than one element of the array Z. It means that more than one element of the iteration domain may be mapped to one element of the operand domain. Hence, ${}_SM_{Y_n}^{(u)}$ may not be injective.

5. Data-race Detection

A program is said to satisfy *synchronization constraint* if *all* its executions are free from data-races namely, read after write (RAW), write after read (WAR) and write after write (WAW) (Hennessy and Patterson 2012). It is worth noting that for a program in SA form, WAR and WAW cannot occur. So, in effect, it is sufficient to show the absence of RAW race condition in the programs which we consider. Barrier is a well-known synchronization construct, that forces all the running threads to join. If l is a barrier, it ensures any write operation defined in statement S prior to l to finish memory updation before executing statement T posterior to l in the control flow of a program.

The *checkSynch()* module checks the existence of a RAW race condition on C-ADDG as follows:
Let there be operator nodes corresponding to two statements S and T which are in the same path but $\nexists b$ s.t. $\mathcal{H}(b)$ =blue (b is the barrier node) in between S and T. The module reports a potential RAW race condition if any of the two rules is satisfied:

R1: $\mathcal{L}(S) = \mathcal{L}(T)$ and $\mathcal{H}(S), \mathcal{H}(T) \neq$black and $\exists \overline{i_1}, \overline{i_2} \in I_f$ (where f represents the for loop within which S and T appear) s.t. ${}_SM_A^{(d)}(\overline{i_1}) = {}_TM_A^{(u)}(\overline{i_2}) \wedge \overline{i_1} \neq \overline{i_2}$.

R2: $\mathcal{H}(S) \neq \mathcal{H}(T)$.

It may be noted that $\mathcal{L}(S) = \mathcal{L}(T)$ implies $\mathcal{H}(S) = \mathcal{H}(T)$ although the converse is not true (because there can be multiple for loops which are executed by a single thread).

We explain these rules using the example shown in Figure 4(a) and 4(b). The first rule *R1* considers a potential RAW race condition when the parallel for-loop comprising of statements T1 and T2 in Figure 4(a) is executed by multiple threads (in a GPU for instance). Here, B[i-1] may not have been written into by statement T1 of some other thread by the time it is read in statement T2 in the current thread. The *R1*-rule states that there are two distinct iteration vectors $\overline{i_1}$ and $\overline{i_2}$ (with a possibility of them being executed by different threads) such that some element of A defined in S during iteration $\overline{i_1}$ is used in T during iteration $\overline{i_2}$. The *checkSynch()* does not report a RAW race condition for the first operand B[i] in statement $T2$ of Figure 4(a) because no such distinct $\overline{i_1}$ and $\overline{i_2}$, as explained above, can be found but it does so for the second operand B[i-1]. Programmers typically place a barrier in between T1 and T2 to avoid this case.

As for the rule *R2*, consider two statements T1 (green coloured) and T3 (red coloured) that are in different loop bodies as shown in Figure 4. The threads computing the values of array B and D will not wait to join after T1 and T2, but will rather proceed to compute the values of array C in statement T3; this may result in reading some element B[i] in T3 before it has been written into in T1. Once again, a blue barrier node is required in between T1 and T3 in Figure 4(b) to avoid a potential data-race hazard.

Now, let us revisit the program given in Figure 1(b) whose corresponding C-ADDG is given in Figure 3(b); for this example, potential data-race hazards will be detected for the statement pairs ⟨T1, T3⟩ and ⟨T1, T4⟩, and hence the two programs in Figure 1 will be declared as possibly non-equivalent.

Algorithm 1 equivalenceChecker(C-ADDG G_1, C-ADDG G_2)

Inputs: Two C-ADDGs G_1 and G_2.
Output: Boolean value *true* if G_1 and G_2 are equivalent and G_2 adheres synchronization constraints, *false* otherwise; in case of failure, it reports possible source of non-equivalence or synchronization violation.

```
1: if $G_1 \equiv G_2$, i.e., the output arrays have identical data computations
   and mappings with respect to the input arrays, by the method of (Banerjee 2016) then
2:     if checkSynch($G_2$) reports NO RAW-race then
3:         return true.
4:     else
5:         return false and report the path in $G_2$ which violates synchronization constraint.
6:     end if
7: else
8:     return false and report the subgraph in a C-ADDG which has no
       equivalent subgraph in the other C-ADDG.
9: end if
```

6. The Overall Equivalence Checking Method

The overall equivalence checking method is shown in Algorithm 1. The soundness and the complexity of this algorithm is provided next.

6.1 Soundness

Program verification being an undecidable problem, completeness of a verification procedure cannot be guaranteed; hence, due to page limitation, here we provide only a sketch of the proof of soundness of our method. The following lemmas are in order before the final proof is presented.

LEMMA 1. *If two ADDGs are declared equivalent by the equivalence checker of (Banerjee 2016), then they are indeed equivalent.*

Proof: The detailed proof of this lemma can be found in (Banerjee 2016). ■

LEMMA 2. *If two statements in a parallel for loop are free from* R1 *RAW race condition, then there cannot be any (other) RAW race condition between these two statements.*

Proof: We prove this lemma by the method of contradiction. So, let there be an array D which is defined in statement P and used in statement Q within the same for loop f' for which there does exist a RAW race condition.

First possibility is there may be a barrier b between the statements P and Q due to which *R1* RAW race free condition was reported. Obviously, due to b, the elements of D written into in statement P across all iterations of f' must have been updated before statement Q is executed in any iteration of f'. (contradiction)

We consider the next possibility. Substituting A by D, S by P, T by Q, and f by f' in the condition *R1* we get, $\exists \overline{i_1}, \overline{i_2} \in I_{f'}(= I_P = I_Q)$ s.t. ${}_PM_D^{(d)}(\overline{i_1}) = {}_QM_D^{(u)}(\overline{i_2}) \wedge \overline{i_1} \neq \overline{i_2}$. which must have been found to be *false* by *checkSynch()*. Therefore, either ${}_PM_D^{(d)}(\overline{i_1}) = {}_QM_D^{(u)}(\overline{i_2})$ or $\overline{i_1} \neq \overline{i_2}$ is evaluated as *false*. Now, if the first conjunct is considered to be *false* then that implies ${}_PM_D^{(d)}(\overline{i_1}) \neq {}_QM_D^{(u)}(\overline{i_2})$ which is a scenario where there is no define-use relationship and therefore no possibility of a RAW race condition; otherwise, if we consider the second conjunct to be *false* then that implies $\overline{i_1} = \overline{i_2}$ which must be executed by a single thread and hence again there is no possibility of RAW race condition. (contradiction) $\blacksquare$.

LEMMA 3. *If two statements in two different parallel for loops are free from* R2 *RAW race condition, then there cannot be any (other) RAW race condition between these two statements.*

Proof: We again prove this lemma by method of contradiction. So, let there be a RAW race condition for an array, A say, which is defined in statement P in some parallel for loop and latter used in statement Q in a different parallel for loop, i.e., ${}_PD_A \bigcap {}_QU_A \neq \emptyset$. Since *checkSynch()* found R2 to be *false*, and since the operator nodes corresponding to P and Q are differently coloured, *checkSynch()* must have found a blue barrier node b between the operator nodes for P and Q. The presence of b will enforce a barrier and hence there cannot be a RAW race condition between statements P and Q. (contradiction) $\blacksquare$

THEOREM 1. *If Algorithm 1 terminates in step 3 then the two C-ADDGs are indeed equivalent and the transformed ADDG is free from synchronization constraint violation.*

Proof: It is to be noted that Algorithm 1 can erroneously output *true* only in one of the following cases: (i) the checker erroneously declares two *actually* non-equivalent C-ADDGs as equivalent in step 1 – this is impossible by Lemma 1; (ii) the checker erroneously declares the C-ADDG corresponding to a parallel program to satisfy synchronization constraint in step 2 – this is impossible by Lemma 2 and Lemma 3. $\blacksquare$

6.2 Complexity Analysis

In this subsection, we concisely analyze the worst case time complexity of Algorithm 1. The time complexity for finding data-race condition is $O(V + E) = O((a + s) + (a + s \times t))$, where the number of arrays in each C-ADDG is a, the number of statements (write edges) is s, and the maximum arity of a function is t. The costliest step in the method of (Banerjee 2016) is finding the transitive dependence (among multiple statements) and the union of the mappings for which the ISL tool (Verdoolaege 2010) has been used, whose worst case time complexity is the same as the deterministic upper bound on the time required to verify Presburger formulae, i.e., $O(2^{2^{2^n}})$, where $n = O(t \times a)$ is the length of a Presburger formula. Thus, the worst case time complexity of Algorithm 1 is

Table 1. Transformations applied

Benchmark	Transformations
DCT (discrete cosine transform)	Loop tiling, speculative code motion
LU (matrix LU decomposition)	Loop tiling, loop fission
FDTF-2D (finite difference time domain)	Loop tiling, boosting down
FLOYD (shortest path calculation)	Loop tiling
NW (Needleman Wunsch algorithm)	Loop fission, constant folding
K-MEANS (clustering algorithm)	Loop reversal, partial evaluation
MATMUL (matrix multiplication)	Loop reordering, loop tiling
CFD (computational fluid dynamics)	Loop tiling, loop fission

Table 2. Results for sequential to parallel benchmarks

Benchmark	Original			Trans				C-ADDG (sec)		DFG EC	Our EC
	#Ar	#Op	Dep	#Ar	#Op	Dep	#Par	Orig	Tran	Time (sec)	Time (sec)
DCT	5	5	3	5	5	6	14	0.53	0.54	$\times$	1.43
LU	1	2	3	3	8	5	10	0.37	0.40	1.73	1.21
FDTD-2D	3	4	3	5	16	5	16	0.44	0.46	$\times$	1.32
FLOYD	1	1	3	3	3	5	5	0.34	0.32	0.71	0.95
NW	6	8	2	8	20	4	4	0.97	1.30	$\times$	4.22
K-MEANS	4	10	3	5	12	3	4	1.02	1.43	$\times$	3.78
MATMUL	3	2	3	3	2	4	3	0.34	0.38	0.40	0.96
CFD	6	12	2	6	16	4	8	1.12	1.33	2.16	4.56

$O(2^{2^{2^n}})$; the worst case behaviour, however, is never exhibited in our experiments.

7. Experimental Results

The equivalence checking procedure has been implemented in `C` on a 3.0-GHz Intel$^{®}$ Core$^{\text{TM}}$2 duo CPU machine with 2-GB RAM and satisfactorily tested on several parallel benchmarks taken from PLuTo (Acharya and Bondhugula 2015) and Rodinia (Che et al. 2009) benchmark repositories. Table 1 depicts the benchmarks (first four are from PLuTo and the next four are from Rodinia) and corresponding transformations applied on those benchmarks. It is to be noted that other than the loop and arithmetic transformations (as mentioned in Table 1), loop parallelization was also carried out for each of our benchmarks.

For the first four benchmarks, the translations are carried out by PLuTo (Acharya and Bondhugula 2015) compiler where the original (sequential) program is fed to the PLuTo compiler to obtain the transformed behaviour, followed by application of some human-guided transformations. For the remaining benchmarks, only human-guided transformations are applied. The original and the transformed programs are then fed into our equivalence checker.

Table 2 characterizes the original and the transformed C-ADDG models in terms of their array nodes, operator nodes, maximum depth of loop nestings and the number of parallelized *for* loops in the transformed program. The next two columns respectively record the C-ADDG construction time for the original and the transformed programs. The last two columns compare the end-to-end equivalence checking (EC) time taken by the data-flow graph (DFG) equivalence checking tool reported in (Verdoolaege et al. 2012) and our equivalence checking tool (including the C-ADDG construction time). Note that the tool of (Verdoolaege et al. 2012) reports non-equivalence of programs (indicated by $\times$) DCT, FDTD-2D, NW and K-MEANS, whereas our tool is able to establish the equivalence. The reason for this is that the tool (Verdoolaege et al. 2012) cannot handle arithmetic transformations and reports non-equivalence for cases where such transformations have been applied. Furthermore, we observed that the tool of (Verdoolaege et al. 2012) is able to establish equivalence in lesser time because: (a) our tool invokes ISL (Verdoolaege 2010) as a separate process whereas, ISL comes as an integrated package within (Verdoolaege et al. 2012) itself, (b) the tool of (Verdoolaege et al. 2012) ignores the parallel constructs since it does not perform any check for synchronization constraint.

Table 3. Equivalence checking of faulty translations

Error	Benchmark	#Op chngd	C-ADDG covered
Type1	LU, DCT	4, 2	50%, 33%
Type2	NW	2	40%
Type3	K-MEANS, CFD	2, 3	20%, 33%

Additionally, we take the original behaviours and manually inject the following three types of errors in the behaviours in order to check the efficacy of the equivalence checker in detecting incorrect parallelizing code transformations.

Type1 Computation error: In LU and DCT, we introduced a *false* data dependency from one branch of an if-then-else block to the block preceding it. This is a typical case of non-uniform boosting up code motion.

Type2 Index space error: In NW we have wrongly written the loop iterators to enforce erroneous loop boundary calculation.

Type3 Synchronization error: We have injected a synchronization constraint violation by placing dubious `nowait` signals in K-MEANS and CFD benchmarks. Table 3 depicts the type of the error along with the benchmarks in which it has been introduced, the number of operations changed in the transformed program and the percentage of the transformed C-ADDG covered before the error is revealed by our equivalence checker.

Furthermore, some hand-fabricated code snippets were tested upon for *R1* and *R2* RAW race conditions; these test cases were constructed from the examples provided in (Xu 2015; Breshears 2009). Our tool terminated reporting possible non-equivalence in all these cases; the details of these experiments is omitted for page limitation.

8. Related Work

Transformation of programs for parallel execution has gained renewed focus in recent times. Recent body of literature reports various techniques such as graph partitioning (Andión et al. 2013), polyhedral loop transformation (Jimborean et al. 2014), affine loop transformation in PLuTo (Acharya and Bondhugula 2015). These attempts have mostly focused on improving the performance of the transformed program but have not analyzed if the transformation preserves the semantics.

There are two well known verification techniques in general. First, the *property verification* technique typically focuses on certain aspects of the system, such as potential race conditions, safety property, liveness etc. Second, the *behavioural verification* on the contrary, checks the computational semantics of the system formally. Unlike property verification, which is performed through model checking or theorem proving, behavioural verification is carried out by equivalence checking. As our main target is to validate semantics preserving transformations, our verification method falls under equivalence checking category, which was pioneered by Pnueli (Pnueli et al. 1998), followed by (Necula 2000), and (Kundu et al. 2008). This body of research primarily focused towards structure preserving transformation of scalar programs. The body of literature such as (Chang et al. 2005a; Necula and Gulwani 2005; Rosu 2015) reports several validation techniques using sequential models of computation for scalar programs using bisimulation based equivalence; therefore they cannot validate programs with different control structure. Subsequently, the work by (Banerjee et al. 2014a) proposes a path based validation method which can validate programs with modified control structures. However, these methods can handle only scalar programs; thus making them unsuitable for array-intensive programs and several loop parallelizing transformations.

A bisimulation method for concurrent program verification was first reported by Milner et al. (Leifer and Milner 2000), which cannot handle parallel code motion transformations beyond the basic block boundaries. The work by (Raudvere et al. 2008) performs a piece-wise verification of every non-semantic preserving transformation and then performs a global analysis of the entire program.

There have been several attempts of property verification of a parallel program. In (Chen et al. 2003), verification is done by translating the program into PROMELA description which SPIN model checker (Holzmann 1997) can analyze. A recent work by (Leung et al. 2012) verifies the control-flow of a GPU kernel considering all possible interleaving statically, without considering the data path. The work by (Li and Gopalakrishnan 2010) reports a symbolic model checking technique for a real life GPU kernel to verify data-race conditions, incorrect synchronization barriers and possible bank conflicts. Parametrized symbolic equivalence checking between two CUDA programs is reported in (Li et al. 2012).

Mateev et al. (Menon et al. 2003) proposed a fractal symbolic analysis (which was used to verify the `C11` compiler (Vafeiadis et al. 2015)) for verification of loop transformations. However, this technique cannot handle data parallel loop transformations for both scalar and array handling programs. A path based validation method for validating data parallel loop transformations is reported in (Bandyopadhyay et al. 2015) which also cannot handle data parallel loop transformations for array handling programs. The work by (Chang et al. 2005b) can validate loop transformation using symbolic approach. However, this method cannot handle recurrence as well as several arithmetic transformations. Array data dependence graph, first proposed by Sashidhar et al. (Shashidhar et al. 2005), is one of the popular models for array-intensive programs. Our approach is based on this representation. Several validation techniques for loop transformations are reported using ADDG model in (Shashidhar et al. 2005; Karfa et al. 2013). Similar to (Chang et al. 2005b), these attempts also cannot validate programs with recurrences. Verdoolaege et al. (Verdoolaege et al. 2012) proposes a DFG based bisimulation method where validation of loop transformation can handle recurrences. However, this work cannot handle arithmetic transformations.

For parallel programs, thread synchronization is an important aspect for correctness. An early work (Flanagan et al. 2002) proposes a new programming model and a static checker to analyze various thread synchronization issues. A similar work by (Deng et al. 2002) also proposes an approach to synthesize synchronization implementation from a high level specification. Recently, the work (Sharma et al. 2015) proposes a property verification technique for the correctness of warp-specialized kernels in terms of deadlock-freedom, barrier recycling and shared memory data-race freedom.

9. Conclusion and Future Directions

In this paper we proposed a novel approach to verify the equivalence of parallelizing loop transformation based on computational semantics. Our approach, based on the C-ADDG model, can evaluate arithmetic expressions of array variables, statements that are not in SA form, as well as recurrence relations. We have proposed a technique to verify thread synchronization after a data parallel loop.

We have considered benchmarks which have been transformed by PLuTo compiler, as well as a few from Rodinia benchmark suite, transformed by human experts and have successfully showed them to be equivalent by our method. In addition, deliberately injected errors such as, computation errors, loop boundary errors and synchronization errors, were also correctly identified as (possibly) faulty transformations by our tool. Program verification being an undecidable problem, while reporting possible non-equivalence, our tool also produces strong hints (in terms of possibly erroneous program snippets) which may be used for program correction by the user.

Our current approach has a few limitations. Firstly, our equivalence checking method does not presently support co-induction (mutual recurrence) and nested recurrence. While the former can be tackled by tracking SCCs with multiple recurrence array vertices, the latter can be alleviated by incorporating a recursive call in our equivalence checking algorithm such that the innermost recurrence is first checked for equivalence before moving on to outer recurrences – these schemes are yet to be implemented in our tool.

Our method cannot verify programs if the recurrence in the original program is replaced by some iterative code Without recurrence in the transformed program or vice versa since each recurrence subgraph in the original program is paired with a recurrence subgraph from the transformed program by our equivalence checker. However, it may be worth noting that we have not found any automated code optimizer to apply such transformation. Above mentioned scenarios are the possible improvements of our current work which we intend to take up in future.

Acknowledgments

The last author has been supported by the SERB grant SB/S3/EECE /0170/2014 for this work.

References

A. Acharya and U. Bondhugula. PLUTO+: Near-Complete Modeling of Affine Transformations for Parallelism and Locality. In *PPoPP*, pages 54–64, 2015.

J. M. Andión, M. Arenaz, G. Rodríguez, and J. Tourino. A novel compiler support for automatic parallelization on multicore systems. *Parallel Computing*, 39(9):442–460, 2013.

S. Bandyopadhyay, D. Sarkar, K. Banerjee, and C. A. Mandal. A path-based equivalence checking method for petri net based models of programs. In *ICSOFT-EA*, pages 319–329, 2015.

K. Banerjee. *Translation Validation of Optimizing Transformations of Programs using Equivalence Checking*. PhD thesis, IIT Kharagpur, India, 2016.

K. Banerjee, C. Karfa, D. Sarkar, and C. A. Mandal. Verification of code motion techniques using value propagation. *IEEE TCAD*, 33(8):1180–1193, 2014a.

K. Banerjee, D. Sarkar, and C. Mandal. Extending the FSMD framework for validating code motions of array-handling programs. *IEEE TCAD*, 33(12):2015–2019, 2014b.

C. P. Breshears. *The Art of Concurrency - A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly, 2009.

B. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck. Type-based verification of assembly language for compiler debugging. In *PLDI 2005*, pages 91–102, 2005a.

B. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck. The open verifier framework for foundational verifiers. In *PLDI 2005*, pages 1–12, 2005b.

S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, pages 45–54, 2009.

X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Formal verification for embedded system designs. *Design Automation for Embedded Systems*, 8:139–153, 2003.

X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based Specification, Synthesis, and Verification of Synchronization in Concurrent Programs. In *ICSE*, pages 442–452, 2002.

C. Flanagan, S. N. Freund, and S. Qadeer. Thread-Modular Verification for Shared-Memory Programs. *Programming Languages and Systems, LNCS*, 2305:262–277, 2002.

J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

G. J. Holzmann. The model checker spin. *IEEE TSE*, 23:279–295, 1997.

A. Jimborean, P. Clauss, J.-F. Dollinger, V. Loechner, and J. M. M. Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *IJPP*, 42(4):529–545, 2014.

C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Equivalence checking of array-intensive programs. In *ISVLSI*, pages 156–161, 2011.

C. Karfa, K. Banerjee, D. Sarkar, and C. A. Mandal. Verification of loop and arithmetic transformations of array-intensive behaviors. *IEEE TCAD*, 32 (11):1787–1800, 2013.

S. Kundu, S. Lerner, and R. Gupta. Validating high-level synthesis. In *CAV*, pages 459–472, 2008.

C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

J. J. Leifer and R. Milner. Deriving bisimulation congruences for reactive systems. In *CONCUR 2000*, pages 243–258, 2000.

A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner. Verifying GPU kernels by test amplification. In *PLDI*, pages 383–394, 2012.

G. Li and G. Gopalakrishnan. Scalable smt-based verification of GPU kernel functions. In *FSE*, pages 187–196, 2010.

P. Li, G. Li, and G. Gopalakrishnan. Parametric flows: automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *SC*, page 29, 2012.

V. Menon, K. Pingali, and N. Mateev. Fractal symbolic analysis. *ACM TOPLAS*, 25(6):776–813, 2003.

G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94, 2000.

G. C. Necula and S. Gulwani. Randomized algorithms for program analysis and verification. In *CAV 2005*, page 1, 2005.

A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, pages 151–166, 1998.

T. Raudvere, I. Sander, and A. Jantsch. Application and verification of local nonsemantic-preserving transformations in system design. *IEEE TCAD*, 27(6):1091 –1103, 2008.

G. Rosu. From rewriting logic, to programming language semantics, to program verification. In *Logic, Rewriting, and Concurrency*, pages 598–616, 2015.

M. Schordan, P. Lin, D. J. Quinlan, and L. Pouchet. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *ISoLA*, pages 493–508, 2014.

R. Sharma, B. M., and A. A. Verification of Producer-Consumer Synchronization in GPU Programs. In *PLDI*, pages 88–98, 2015.

K. C. Shashidhar. *Efficient Automatic Verification of Loop and Dataflow Transformations by Functional Equivalence Checking*. PhD thesis, Katholieke Universiteit Leuven, 2008.

K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *DATE 2005*, pages 1310–1315, 2005.

R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Z. Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL 2015*, pages 209–220, 2015.

P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. A practical dynamic single assignment transformation. *ACM TODAES*, 12(4), 2007.

S. Verdoolaege. ISL: An Integer Set Library for the Polyhedral Model. In *ICMS*, pages 299–302, 2010.

S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM TOPLAS*, 34(3):11, 2012.

Z. Xu. Lecture 10: Introduction to OpenMP(part 2). `http://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-11-02.pdf`, 2015. [Online accessed: 29-Mar-2016].