Update-in-Place Analysis for True Multidimensional Arrays

STEVEN M. FITZGERALD[1] AND RODNEY R. OLDEHOEFT[2]

[1]*Department of Computer Science, California State University, Northridge, Northridge, CA 91330-8281; e-mail: sfitzger@secs.csun.edu*
[2]*Department of Computer Science, Colorado State University, Fort Collins, CO 80523; e-mail: rro@cs.colostate.edu*

ABSTRACT

Applicative languages have been proposed for defining algorithms for parallel architectures because they are implicitly parallel and lack side effects. However, straightforward implementations of applicative-language compilers may induce large amounts of copying to preserve program semantics. The unnecessary copying of data can increase both the execution time and the memory requirements of an application. To eliminate the unnecessary copying of data, the Sisal compiler uses both build-in-place and update-in-place analyses. These optimizations remove unnecessary array copy operations through compile-time analysis. Both build-in-place and update-in-place are based on hierarchical ragged arrays, i.e., the vector-of-vectors array model. Although this array model is convenient for certain applications, many optimizations are precluded, e.g., vectorization. To compensate for this deficiency, new languages, such as Sisal 2.0, have extended array models that allow for both high-level array operations to be performed and efficient implementations to be devised. In this article, we introduce a new method to perform update-in-place analysis that is applicable to arrays stored either in hierarchical or in contiguous storage. Consequently, the array model that is appropriate for an application can be selected without the loss of performance. Moreover, our analysis is more amenable for distributed memory and large software systems.

1 INTRODUCTION

Languages that follow the applicative paradigm have been proposed to define algorithms for parallel architectures [1, 2, 8]. In applicative languages, such as Sisal [3, 5], computation is carried out by the evaluation of expressions. Because the evaluation of expressions is not influenced by side effects, the order of computation is dependent only on the availability of values. As values are computed, separate copies can be provided to many independent operations that can execute simultaneously, thus exploiting parallel architectures.

An implementation that strictly adheres to the applicative model is required to copy data values when they are modified. However, the cost associated with copying large data aggregates, such as arrays, can become prohibitive, nullifying the benefits achieved through parallel execution. Optimizations are needed to remove the unnecessary copying. The Sisal 1.2 compiler, optimizing Sisal

Received April 1995
Revised June 1995

Scientific Programming, Vol. 5, pp. 147–160 (1996)
CCC 1058-9244/96/020147-14

compiler (OSC), uses build-in-place analysis [18] to preallocate memory for arrays and update-in-place analysis [4] to reduce the copying of arrays. The optimizations that are part of these analyses are applied to a graph-based intermediate form, IF2 [21, 22]. As a result of these analyses, array-intensive applications written in Sisal 1.2 execute as fast as their Fortran equivalents [4, 5, 9].

The design of Sisal 1.2 arrays is based on the vector-of-vectors array model. Under this model, multidimensional arrays are built hierarchically from one-dimensional arrays, i.e., from vectors [12], and are accessed through dope-vectors. Although hierarchical arrays are convenient for many applications, it is expensive to manipulate array values represented in this model [9]. Because the additional expense is unnecessary for applications that do not utilize the advantages of the vector-of-vectors model, other languages use the flat array model. Under this model, multidimensional arrays are built by the catenation of the subarrays of the innermost dimension to form a single one-dimensional array [12]. The array's uniform structure allows many more optimizations to be performed, such as vectorization.

The design of Sisal 2.0 arrays includes an array model we call dimensional [10], while retaining the ability to express arrays as vector-of-vectors. Fully contiguous multidimensional arrays are added to the language for two reasons. First, many high-level array operations can be both expressed succinctly and implemented efficiently [17]. These operations include subarray definitions, such as "tiling," and array comprehension. Other benefits from this array design include a constant stride in each dimension for vector processing, potentially better cache performance via loop optimizations, and faster subscripting.

Second, more efficient storage management is possible. The dynamic allocation and deallocation of multidimensional arrays are single operations instead of traversals. Substantial fractions of execution time in some Sisal 1.2 benchmarks were devoted to array creation and deletion. A new optimization was developed to alleviate the cost associated with array creation and deletion [6]. With fully contiguous arrays, the hierarchical storage of pointers associated with vector arrays is also eliminated.

We wish to extend and to enhance the performance of Sisal 1.2 to version 2.0, so current optimizations based on the vector-of-vectors model must be generalized for dimensional arrays. To that end, we explain in this article a new algorithm for update-in-place analysis that can be applied to arrays stored either as vector-of-vectors or in a contiguous, multidimensional space [11].

2 BACKGROUND: IFx

IF2 [22] is a graph-based language designed as an intermediate form for applicative languages. The language is based on and is a superset of IF1 [21]. Both languages follow the dataflow model. In this model, operations, which are represented by nodes, execute when all of their inputs (represented by edges) are available. However, IF2 does not strictly adhere to the applicative model, since a set of primitive nodes that allocate and manipulate memory are part of the language definition. Artificial dependency edges (ADEs), which are used to delay the execution of a node until some other node executes, are also included in the language.

Within IF1 and IF2, there are no explicit control nodes or control lines. Control-flow constructs, such as conditionals and iterators, are represented by predefined compound nodes. These compound nodes consist of subgraphs that define the individual functionality of a control-flow construct. For example, the LoopB node has four subgraphs: the initialization, the test, the body, and the returns. Values are passed implicitly between the subgraphs through ports (depicted as boxes as shown in Fig. 1). Since interaction between the subgraphs is implicit, the control-flow construct can be implemented differently for different architectures.

The following Sisal "for initial" expression is directly translated into the IF1 graph depicted in Figure 1.

```
C := initial
       B := A;        % the initialization
       i := 1;
     while i <= N               % the test
     repeat
       B := old B[old i : 0]; % the body
       i := old B[old i] + 1
     returns value of B     % the returns
     end for
```

Each of the subgraphs corresponds directly to an individual part of the iterative-loop construct. The expression is computed in the following manner. First, the loop variants, B and i, are initially defined. These values are used in the other sub-

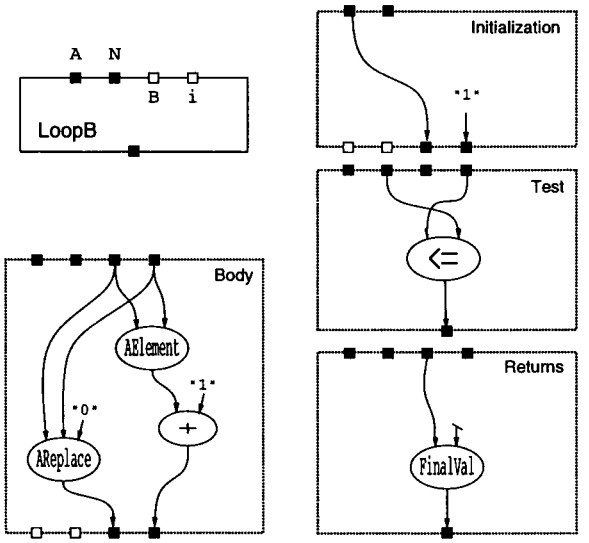


FIGURE 1 Example of a LoopB compound node and its subgraphs.

graphs. Under single-assignment semantics, only one redefinition for a loop variant is allowed per iteration. Either the previous or the current value of the loop variants can be used within the body subgraph. The previous value is accessed by prefixing the name of the variant by the keyword "old."

Second, the loop variants are supplied to the returns subgraph. The results of the LoopB node are prepared incrementally via the returns subgraph. Values flow to the returns subgraph from the initialization subgraph (corresponding to the first loop iteration) and then from each instantiation of the body subgraph.

Third, the body subgraph is executed. In our example, the `AReplace` node replaces the i^{th} element of B with the value 0, producing a new array. The next value of i is defined by the sum of one and the value of the i^{th} element of B. The `AElement` node selects the i^{th} element of the array and passes the value to the `Plus` node. Notice that the order of execution between the `AElement` node and the `AReplace` node is undefined. Consequently, a copy of array B must be created for each loop instantiation.

The second and third steps are performed iteratively, until the single value defined in the test subgraph is `False`. The result of the last instantiation of the returns subgraph, as indicated by the `Final-Val` node, is then returned as the value of the LoopB node.

3 UPDATE-IN-PLACE

Update-in-place analysis is used to eliminate unnecessary copying associated with array-update operations in applicative languages. In this section, we present an overview of the method and compare and contrast our approach with that of Cann's. We then describe, in some detail, our algorithms.

3.1 Overview

Our work is based on the methods developed by David Cann [4]. Principally, we have extended the analysis to handle arrays stored in contiguous memory. As a result, three different array models can be supported efficiently: the vector-of-vectors, the flat, and the dimensional array models.

Although the two analyses operate in different manners, both analyses determine when an array modifier, such as an `AReplace` node, is the last operation to access an array. Under this situation, a destructive update may be performed on the input array. The implicit copy operation is eliminated, thus reducing execution time. When appropriate, ADEs are introduced to ensure a valid execution order. For example, consider the graph in Figure 2.

In the graph's current form, the `AReplace` node may execute before the `AElement` node. If the `AReplace` node performs a destructive update on the input array, the result returned by the `AElement` node will be erroneous. Consequently, the `AReplace` node must always copy the input array before the update operation is performed.

To eliminate the copy operation, an ADE, which is depicted as a dashed line, must be inserted to delay the execution of the `AReplace` node, as is

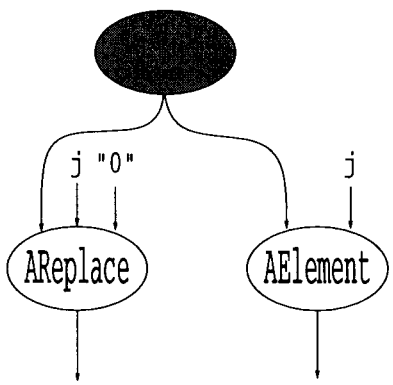


FIGURE 2 IF1 graph for the Sisal expression "A[j:0], A[j]."

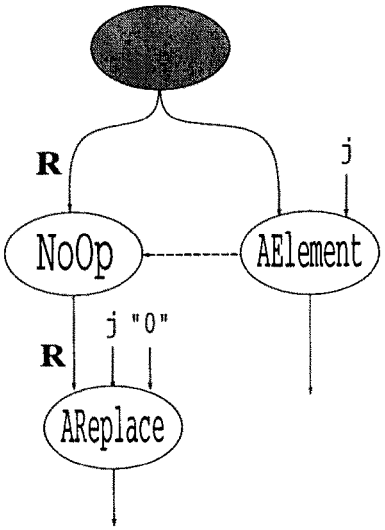


FIGURE 3 IF2 graph for the Sisal expression "A[j:0], A[j]," after update-in-place analysis has been applied.

depicted in Figure 3. Under the revised execution order, no copying is necessary. The **R** marks, which are located next to the array input edge of both the NoOP node and the `AReplace` node, indicate that the input array is not copied but is passed by reference.

The additional `NoOp` node is added to simplify analysis. `NoOp` nodes perform all necessary copying explicitly. All other array-modifier nodes now operate directly on their input array; no implicit copy operation is performed. The `AReplace` node is such an array-modifier node. Additional marks are associated with the `NoOp` node to indicate the type of copying performed. (We discuss these marks in more detail later in this article.)

Both analyses start off with the same objective. Initial assumptions, however, resulted in two different algorithms. In the original update-in-place analysis developed by Cann, the following assumptions were made.

1. The entire program is available (and necessary) for analysis.
2. Exclusive write access to an array is not presumed.
3. Arrays are always stored hierarchically.
4. The cost of copying outweighs the potential loss of parallelism.

Because the entire program was assumed to be available, an algorithm that traverses a graph in a top-down outside-in fashion was designed to determine when exclusive access to an array is guaranteed. In general, the execution path in which an array flows is examined. As the graph is traversed, a number of operations are applied. In particular, ADEs are inserted to delay the execution of array-write nodes.

Reference counts are used to maintain a dynamic count of the number of operations that access an array. Although reference counting increases execution time and produces parallel bottlenecks [7, 9], they were deemed necessary for two reasons. First, it was presumed that exclusive access to arrays could not, in general, be determined. Second, under a hierarchical array representation, subarrays may be shared. Subarray sharing creates aliases that can be hard to detect. A major component of the analysis is used to determine when reference counting operations could be eliminated.

Based on the revised execution order imposed by the ADEs, reference-count operations are eliminated. Consequently, the inserted ADEs are beneficial because they help to reduce copying and to reduce the overhead associated with reference counting. The potential loss of parallelism imposed by the inserted ADEs was assumed to be insignificant.

The original update-in-place analysis was implemented and is part of the current Sisal compiler, OSC. The analysis has shown to be effective in eliminating unnecessary copy operations. In most cases, no array copying and no reference counting are performed. We were able to use this experience to take a more aggressive approach. At the same time, we were able to relax some of the restrictions placed on the original analysis.

Our analysis was designed based on the following assumptions.

1. The entire program is not available for analysis.
2. Exclusive write access to an array is presumed.
3. Arrays may be stored hierarchically or contiguously.
4. A performance tradeoff exists between retaining parallelism and eliminating copying.

In many cases, an entire program is not available for analysis. Moreover, the time required to analyze a large application in its entirety can be prohibitive. This assumption has led to the design of an inside-out, bottom-up algorithm. Our ap-

proach allows subprograms to be analyzed separately, stored for later use, and incorporated into an application.

We assume that each subprogram has exclusive write access to its input arrays. This assumption simplifies analysis within the subprogram. Moreover, this assumption reduces the need for reference counting. The assumption that arrays might be stored in contiguous memory further decreases the need for reference counting. Since individual subarrays may not be shared indirectly, aliases can be more easily detected. Consequently, our analysis does not rely on reference-counting operations to determine when copying is necessary.

To ensure program semantics, we must guarantee that each function has exclusive write access to its input arrays. A NoOp node is inserted into the caller's graph to perform any copy operation that is necessary before the function is called. The application of our analysis to that graph will determine if the copy operation is necessary. These extra NoOp nodes allow the tradeoff between the loss of parallelism and the cost of copying to be examined more fully.

Consider the graph depicted in Figure 4. Two functions are given the array **A** as input. The first input edge to the `Call` node specifies the function applied. The function "alpha" performs a destructive update on the array, and the function "beta" uses the array as read-only data. A **W** mark indicates that the array is updated within the function. This information was determined when these functions were analyzed. The other marks are explained in a subsection of 3, "Examination Phase."

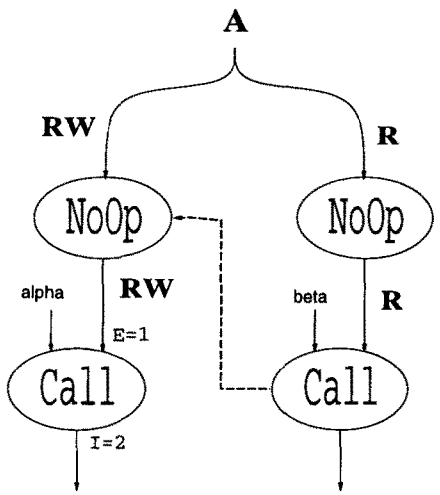


FIGURE 4 IFx graph with two function applications.

Copying is only necessary if the function alpha is not guaranteed to execute last. The ADE from the rightmost `Call` node to the leftmost NoOp node ensures that the correct execution order is maintained. However, the ADE can adversely affect the execution time of the program because parallelism is lost. If the time required to copy the array is less than the time to execute the function "beta," it is better to copy the array. The array can be copied by either the leftmost or the rightmost NoOp node.

3.2 Our Algorithm

Our analysis is performed in two phases: a preparation phase and an examination phase. During the preparation phase, the graph undergoes several changes, which include the addition of marks that are annotated on edges, the insertionof NoOp nodes, and the reconstruction of subgraphs that perform multidimensional-array updates. These operations are applied as the graph is traversed in reverse dataflow order.

During the examination phase, each compound node is reexamined. Seven interrelated operations are applied. Five of these operations are applied during a bottom-up traversal of each subgraph, and the remaining two operations are applied after each subgraph has been fully analyzed. Together, these operations annotate input edges of NoOp nodes, insert ADEs, insert grounding edges (a mechanism to deallocate arrays), identify aliases, and propagate marks to the outside of the compound node.

Each function within an IFx graph is optimized independently. Once optimized, the set of marks assigned to the graph's input and output edges are retained. These marks are then used to annotate the edges connected to each `Call` node that references the function. The marks provide additional information needed to perform update-in-place analysis in the graph that contains the `Call` node. Under this scheme, a set of fully optimized library functions may be created. When one of these functions is referenced, the marks associated with the function are used to determine if an input array must be copied prior to function invocation.

To ensure that each function-definition graph is optimized before it is referenced, all function graphs are optimized in reverse topological order as defined by a standard call graph. Since recursive functions introduce cycles into the call graph, these functions cannot be analyzed before they are referenced. Consequently, these functions require special handling. We have taken a conservative ap-

proach to preserve program semantics. However, this approach prevents the identification and elimination of some unnecessary copy operations. To fully optimize recursive functions, a more aggressive approach is required.

Preparation Phase

Three main operations are performed during the preparation phase: edge classification, NoOp insertion, and subgraph inheritance. The first two operations are similar to the operations performed under Cann's analysis with only slight modifications. The third operation, subgraph inheritance, is a reformation of reference inheritance, which was developed by David Cann [4]. We have also added a second component, known as MSD graph detection, to subgraph inheritance to help identify redundant copy operations. Although MSD graph detection is part of the examination phase, we describe subgraph inheritance and MSD graph detection together to simplify the discussion.

Edge Classification. Input edges are classified to determine how arrays are manipulated within functions and compound nodes. The classification of input edges is based on both the type of node being considered and the classification of the node's output edge. Due to the order of application, marks are propagated in reverse dataflow order through the graph. Edges may be classified as *dope-vector* write, *array-data* write, *aggregate* read, or *subarray* move. Based on the classification, edges are annotated with one or more marks. Table 1 indicates the marks inserted during the edge classification procedure. Aggregate-read edges are not annotated.

In our analysis, we have added two classifications: the dope-vector write and the subarray-move classification. Some operations modify only an array's descriptor, i.e., the array's dope-vector. In the second phase of update-in-place analysis, ADEs are inserted to delay write operations to prevent unnecessary copying. In some cases it is advantageous to copy a dope-vector rather than to restrict parallelism. The additional classification helps to differentiate the type of array modification that occurs within functions and compound nodes.

The subarray-move classification indicates that the elements of a subarray must be copied into another array. In general, this copy operation is necessary to ensure that the final array is stored in contiguous memory. Under certain circumstances this copy operation is unnecessary and is removed during the second phase of the analysis, via MSD graph detection (see footnote ‡ on page 153).

Table 1. Annotations Applied to Edges During Edge Classification

Classification	Annotation
Dope-vector write	**w**
Array-data write	**W**
Subarray move	**m**

NoOp *Node Insertion.* Many IFx nodes induce copying, some copy dope-vectors, and some copy arrays. A NoOp node is inserted for every array-modifier node. Each array-modifier node is also transformed into its AT-node equivalents, e.g., an AReplace node is transformed into an AReplaceAT node. By definition these nodes allow in-place operations to occur.*

Once NoOp nodes are inserted, all copying is isolated to a single-node type. Marks are used to indicate the type of copying performed by the NoOp node. The NoOp node's input edge is annotated with a **P** mark based on the classification of the node's output edge, i.e., the input edge of the array-modifier node. A **P** mark indicates that the array's dope-vector should be copied as opposed to the array's data.

In our analysis, we insert NoOp nodes more liberally. Two additional classes of NoOp nodes are inserted: *external* and *final* NoOp nodes. These NoOp nodes are not necessary to preserve program semantics, but their presence simplifies analysis. An external NoOp is inserted for each array-input edge connected to either a Call or a compound node. Their placement allows the necessary internal-copy operations to be expressed within the external graph environment, where the tradeoff between the cost of copying and the loss of parallelism is more appropriately evaluated.

NoOp nodes are then inserted for each array-output edge connected to a graph boundary. These nodes, which are known as final NoOps, are used to copy arrays before they are exported from a graph. These copy operations may be necessary to prevent unwanted aliases from being created and to prevent side effects to loop variants associated with either a LoopA or a LoopB node.† A final

* Under Cann's analysis, the AReplace node is not transformed into a AReplaceAT node.

† A LoopA node corresponds to the repeat-until loop.

NoOp is also associated with the third input edge of some AReplace nodes.

Subgraph Inheritance Transformation and MSD Graph Detection. To update a single value in an N-dimensional array, a series of write operations is required, one for each dimension. Since each of these write operations accesses the same array, up to $N-1$ subarrays may be copied. To help identify the extra copy operations, each subgraph that performs a multidimensional array-update operation is transformed. The transformation is applied for each AReplaceAT and AElement node pair in which an output edge of the AElement node is annotated with an **m** mark. This mark indicates that the subarray extracted by the AElement node will be moved.

For example, consider the graph depicted in Figure 5. This graph illustrates the effect of applying subgraph inheritance to a typical multidimensional array-update operation. The subarray-update operation is forced to execute after the NoOp node and before the AReplaceAT node associated with the update operation for the outermost dimension. As part of the transformation, a NoOp node (identified as the minor NoOp) and an ADE are inserted into the graph. Together, they ensure that the selected subarray is copied to a new location before it is overwritten via the AReplaceAT node. In most cases, the copy operation is not necessary and is eliminated in the second phase.

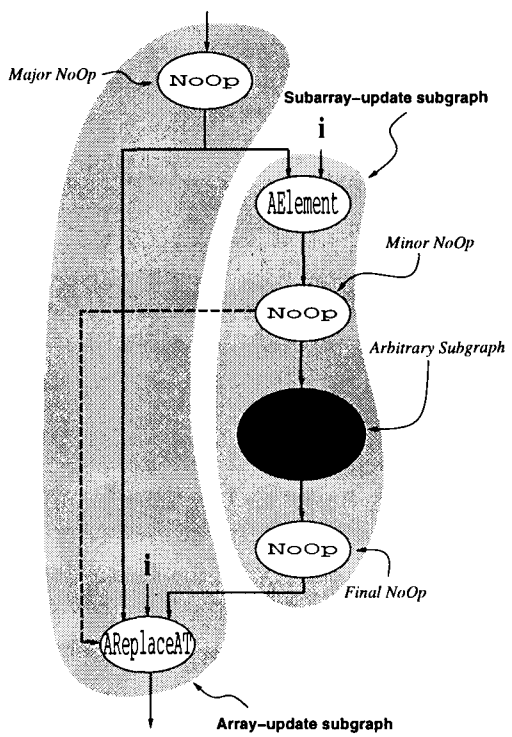


FIGURE 5 Typical graph template for a multidimensional array-update operation after subgraph inheritance.

Under the contiguous array model, the AReplaceAT must copy the subarray into the correct position within the destination array, i.e., the subarray must be moved to its final location. (Under hierarchical storage, only a single pointer has to be updated.) If the subarray is updated in place and the result of the operation is passed to the AReplaceAT node of the array-update operation, the subarray-move operation is redundant. Under this situation, the multidimensional array-update operation has the form depicted in Figure 5. We refer to such a graph as mutually strong-dependent (MSD).‡

MSD graph detection is used in the second phase to determine when the copy operation performed by the minor NoOp and the move operation performed by the AReplaceAT node are unnecessary. The determination is easily made by examining the minor NoOp node's global-leaf set. (This set is discussed in a subsection of Section 3, "Examination Phase.") If the graph is of the correct form and the copy operations are not necessary, the array-input edge of both NoOp nodes is annotated with an **R** mark and the third input of the AReplaceAT node is annotated with a **P** mark. The **P** mark indicates that the subarray has been built in place [18].

MSD graph detection is also beneficial for arrays stored hierarchically. Each AReplaceAT node that is annotated with a **P** mark allows a pointer-update operation to be eliminated. The corresponding performance gain can be substantial if the operation is nested within an iterative construct. Moreover, on a distributed memory architecture, the pointer-update operation can greatly degrade performance [13]. Consider the case where each subarray resides on a different processing unit. By eliminating the operation, we also eliminate the corresponding communication between the processing units.

Examination Phase

During the second phase of the algorithm, an IFx function graph is analyzed to remove all unneces-

‡ We have borrowed the term mutually strong-dependent from Kim [14]. We use the term in a similar sense.

```
Procedure Graph_Examination(graph G)
  For each array-generation node N of G
                in reverse dataflow order Do
    S_N := Local-usage_Set_Construction(N);
    Control, Copy, NoCopy
          := Copy_Identification(S_N);
    If N is a major-NoOp node Then
      Copy := MSD_Graph_Detection(S_N, Copy);
    End If
    Graph_Reordering(Control, Copy, NoCopy);
    S_N.global_leaves :=
      Global-leaf_Set_Construction(Control, Copy,
                                   NoCopy, S_N.F);
  End For
  Information_Propagation();
  Array_Deallocation();
End Procedure
```

FIGURE 6 Pseudocode for the graph-examination phase of update-in-place.

sary copy operations. These copy operations are associated with the inserted NoOp nodes. Via the analysis, each NoOp node is potentially annotated with an **R** mark to indicate that no copying is performed. Each function graph is analyzed starting at the innermost compound nodes. This ordering ensures that any necessary copy operations are always performed at the outermost level possible.

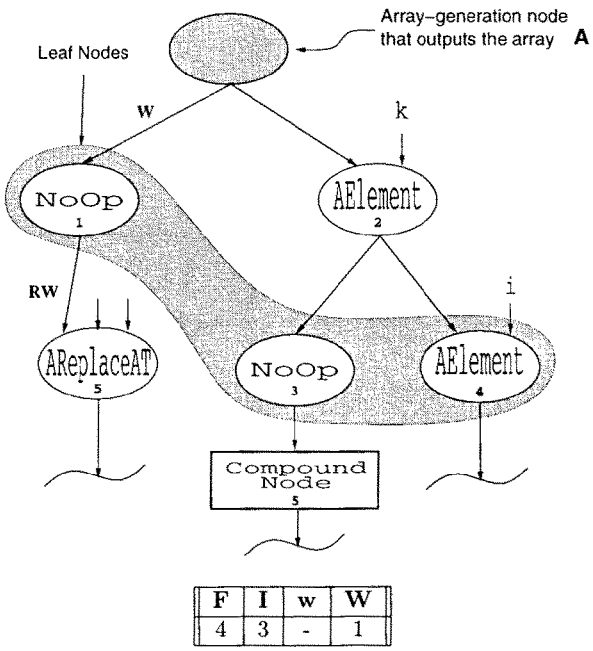
The examination phase contains seven interrelated operations. Three of the operations are used to prepare sets that summarize the usage of nodes within a graph. Each array-generation node is associated with two sets: a local-usage set and a global-leaf set. Together, these sets are used to construct three more sets, which are used to identify which NoOp nodes must copy their input arrays. These sets are constructed via the copy identification procedure.

These sets are constructed incrementally through a bottom-up traversal of the graph. As the sets are constructed, two optimizations are performed: MSD-graph detection and graph reordering. Graph reordering is used to insert ADEs into the graph to delay write operations. Once the entire compound node has been analyzed, two final operations are performed: information propagation and array deallocation. We present the pseudocode for the examination phase in Figure 6.

Local and Global Usage-Set Construction. Each node that conceptually creates a new array is associated with a local-usage set. This set records the nodes in which the generated array is an input and is not an output, i.e., the array is consumed or copied. These nodes are referred to as leaf nodes. For example, an `ASize` node, which returns the size of an array, is a leaf node because it has an array as an input but only a scalar as an output. The NoOp nodes are also considered leaf nodes because they may copy an array to produce a new array.

To build a local-usage set, each subgraph that is dominated by an array-generation node is traversed in depth-first-search order. The search is bounded by the array-generation node and leaf nodes. When a leaf node is encountered, its node number is recorded in one of four groups: final read (**F**), intermediate read (**I**), dope-vector write (**w**), or data write (**W**). NoOp nodes are recorded in a group based on the edge classification of their array-input edge. All non-NoOp nodes are recorded in the **F** group.

Consider the graph fragment in Figure 7. The local-usage set for the topmost array-generation node contains two NoOp nodes and one `AElement` node. One NoOp node (no. 1) is recorded in the **W** group, and the other NoOp node (no. 3) is recorded in the **I** group. Only one of the `AElement` nodes is recorded into the local-usage set. The topmost `AElement` node (no. 2) is not recorded because



F	I	w	W
4	3	-	1

FIGURE 7 Partial IF*x* graph illustrating leaf nodes. The local-usage sets for the array **A** are also shown.

the final result produced by this node is a subarray contained within the original array **A**.

The local-usage set is used to identify copy operations within a localized region. In general, each potential copy operation, represented by a NoOp node, is recorded in one of three sets: *Control*, *Copy*, or *NoCopy*. Based on the set in which a NoOp is recorded, ADEs are inserted and edges are annotated. (We defer the discussion of this process until later.) A global-usage set is then created based, in part, on these sets.

A global-usage set records the final nodes that access an array. This information is used to identify copy operations within a larger region of the graph. Each NoOp node recorded in the local-usage set also has a global-leaf set. These sets are pieced together to form the global-leaf set for the current array-generation node. In general a global-leaf set is formed by the concatenation of

1. The nodes listed in the **F** group of the current local-usage set
2. The nodes listed in the global-leaf set of each NoOp node that does not perform any copy operation
3. The nodes listed in the global-leaf set of each NoOp node that performs a dope-vector copy operation
4. The NoOp nodes that perform an array-copy operation

In effect, a single global-leaf set is constructed incrementally for each array imported into the graph.

Copy Identification and Graph Reordering. Once the local-usage sets are built, they are used to identify which of the NoOp nodes must perform a copy operation. If only a single NoOp node is recorded in the local-usage set, the corresponding copy operation can always be eliminated. If, however, multiple NoOp nodes are recorded in the local-usage set, copying is necessary. In either case, ADEs are inserted to delay the execution of a single NoOp node if the cost of copying is greater than the loss of parallelism.

To determine if the insertion of an ADE results in a performance gain, estimated execution costs can be associated with each node in the graph. During local-usage set construction, these estimated costs are combined to determine the expected execution cost of a subgraph. These costs are then used to estimate both the expected copy cost and the expected delay cost of NoOp nodes. The delay cost of a NoOp node is defined to be the cumulative execution time of the subgraph bounded by the NoOp node and the nodes recorded in its global-leaf set.

Many different methods can be used to determine the expected time to execute each type of node: simple or compound. In particular, the current Sisal compiler, OSC, uses a simplified version of the techniques developed by Vivek Sarkar [19] to assign an execution cost to each node. Although these techniques were developed for partitioning and scheduling parallel programs, they are sufficient to calculate the cost of copying and the loss of parallelism.

To identify which of the NoOp nodes in the local-usage set should perform a copy operation, three steps are taken. As part of these steps, each NoOp node is recorded into one of three sets: *Control*, *Copy*, or *NoCopy*. The steps are:

1. A single-write NoOp node is recorded in the *Control* set.

   If multiple-write NoOp exists, the selection of a NoOp node has an effect on the total amount of copying that can be eliminated. In general, the NoOp node that performs the largest amount of copying should be selected. However, the amount of copying is partially determined by run-time information making it difficult to predetermine. In many cases, there is only one such NoOp node from which to choose.

2. The tradeoff between copying and the loss of parallelism is evaluated for the other NoOp nodes.

   Because only one write NoOp is selected in Step 1, all other write NoOps must perform a copy operation. However, for each read NoOp, a copy operation may or may not be beneficial to overall program performance. To determine if a read NoOp should perform a copy operation, the delay and the copy cost of the NoOp node are compared. If the copy cost is greater than the delay cost, the NoOp node is recorded in the *NoCopy* set. Otherwise, it is recorded in the *Copy* set.

3. The control NoOp is identified as either a copy or a noncopy operation.

   If the control NoOp does not perform a copy operation, it will be delayed. The delay time

can be estimated by the cumulative execution times of the nodes that must execute after the current node and before the control NoOp. The difference between the copy cost and the delay time can be used to make the appropriate identification. If the control NoOp is identified as a copy operation, it is recorded in the *Copy* set and erased from the *Control* set.

Once all copy operations are identified, ADEs are inserted to delay the execution of the control NoOp. The insertion of ADEs, in effect, reorders the graph. In general, an ADE is inserted for each node recorded in the following sets: the *Copy* set, the global-leaf set of the nodes contained in the *NoCopy* set, and the **F** group of the local-usage set. Additionally, the input edge of each NoOp node that does not perform a copy is annotated with an **R** mark. These nodes are recorded in the *NoCopy* set.

In some situations, only a dope-vector must be copied. If the input edge of a NoOp node is annotated with a **w** mark, the edge is annotated with a **P** mark, indicating that only the dope-vector is copied. Since the dope-vector copy operation will create an alias, the position of the ADEs must be adjusted to preserve program semantics. An ADE is inserted for each node recorded in the global-leaf set of a NoOp node that performs a dope-vector copy operation.

Reconsider the graph fragment in Figure 7. Because a single-write NoOp node is listed in the local-usage set, this node (no. 1) is recorded in the *Control* set. The other NoOp node (no. 3) is recorded in either the *Copy* or the *NoCopy* set. To prevent the copy operation associated with the selected NoOp, two ADEs must be inserted. The first ADE is connected to the `AElement` node because it is recorded in the **F** group of the local-usage set. The other ADE is connected to either the rightmost NoOp node (no. 3) or the compound node (no. 5).

The position of the second ADE is based on the set in which the rightmost NoOp node is recorded. If this node performs a copy operation, i.e., it is recorded in the *Copy* set, the ADE extends from this NoOp node. If, however, no copying is performed, the selected NoOp must be delayed until after all nodes contained in the rightmost NoOp's global-leaf set have executed. In this graph we assume that node 5 is the only node recorded in that set.

Ideally, a graph partitioning and scheduling scheme, e.g., [14, 19, 23, 24], should be used in conjunction with our analysis. Under this scheme, the current subgraph would first be partitioned. Our analysis would then be performed. The appropriate comparison function to apply during our analysis would be based on the partition obtained. The resulting subgraph would then be scheduled to a processor. On a distributed memory processor, if a NoOp node that performs a copy operation dominates a partition, the copy operation could be used as a data-transfer mechanism.

Information Propagation and Array Deallocation. After an entire graph is analyzed, the global-leaf sets associated with the import ports of a graph are used to classify the usage of all input arrays. Together with the global-leaf sets associated with array-generation nodes, we can identify both the node that creates an array and the final set of nodes that access the array. With respect to the current graph being analyzed, each array is classified into one of four classes:

1. An array is neither created nor *totally* consumed within the graph, i.e., it passes through the graph.
2. An array is created within the graph.
3. An array is imported into the graph and is (partially) consumed within the graph.
4. An array is both created and totally consumed within the graph.

If an array is not created within the graph and is exported from the graph (i.e., the array is a member of the first class), this information must be propagated to the external graph environment. If, however, the array is both created and totally consumed within the graph, the array can be deallocated.

The information-propagation procedure examines each global-leaf set associated with a graph-import port. If a global-leaf set contains a final NoOp node, the imported array passes through the graph. Recall that a final NoOp is connected to the graph boundary. Both the array's export-port number, say P, and the array's import-port number, say Q, are used to indicate which of the graph's ports are annotated. The P^{th} import port is annotated with an **E**=P mark, and the Q^{th} export port is annotated with an **I**=Q mark. These marks are then propagated to the external graph environment.

The array-deallocation procedure examines each global-leaf set that is not associated with an

array import port. These are the arrays that are created within the graph. If any of these arrays are not exported from the graph, the corresponding array may be deallocated. The procedure inserts a *grounding* edge from the array-generation node to port zero of the graph node.

Port zero semantics are nonstandard with respect to both the syntax and the semantics of IFx [4]. Syntactically, this port allows multiple input edges, i.e., fan-in. Fan-in occurs when more than one array is deallocated within the graph. Semantically, the associated array is not available as an output value. Instead, the allocated memory for an array is deallocated at run-time.

4 PRELIMINARY INDICATIONS

To date, a full implementation of our algorithms has not been developed. To determine the effectiveness of our analysis, we have manually applied our algorithms to several Sisal 1.2 applications. In this article, we present execution times that result from our analysis for two programs: the Hohn-Aufenkamp state-removal algorithm [20] and a matrix inversion algorithm. To create the executables, we first manually generated IFx graphs for the programs. The Sisal compiler, OSC, was then used to complete the compilation process. To simplify implementation, multidimensional arrays were represented under the vector-of-vectors array model, but under the assumption that the arrays were stored in contiguous memory.

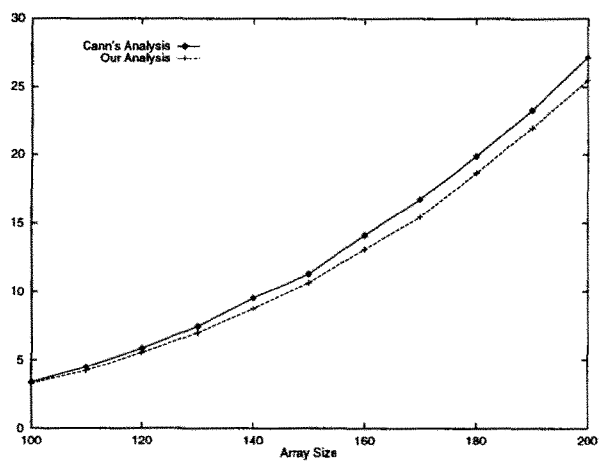


FIGURE 8 Execution times for the expresison "state-removal(T), T[4]" when optimized both by Cann's analysis and by our analysis. The ordinate indicates the execution time in seconds.

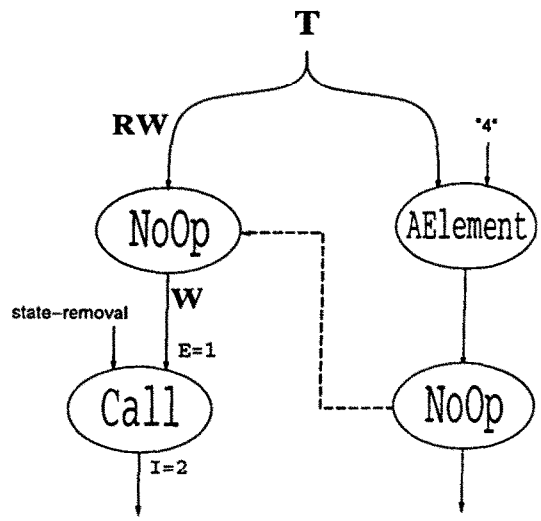


FIGURE 9 IFx graph for the Sisal expression "state-removal(T), T[4]."

The IF2-to-C code translation, if2gen, was modified to insert code to simulate the copy operations required to preserve the contiguity requirement. The inserted code performed full multidimensional-array copy operations and subarray move operations whenever necessary. These copy operations are associated with `NoOp` and `AReplaceAT` nodes, respectively.

For comparison purposes, we also applied the analysis developed by Cann [4] to our test programs. Recall that the OSC compiler represents arrays under the vector-of-vectors array model. Although we are comparing execution times for two different array models, the comparison provides a reasonable measure of the effectiveness of our analysis. All test programs were executed on a uniprocessor, a DECstation 5000. Fifty separate runs were performed for each test case. Execution times that varied more than 5% from the average were eliminated to minimize the effect of system sharing. We report the average execution times of the remaining times.

4.1 Hohn-Aufenkamp State-Removal Algorithm

The state-removal algorithm of Aufenkamp and Hohn [20] is a method of finding all the possible paths of length less than n between two nodes of a net, where n is the number of nodes. In general, the algorithm reduces an $n \times n$ matrix, say T, to a 1×1 matrix, incrementally, by the following equation.

$$T'_{i,j} = \frac{T_{i,j} + T_{i,n} \times T_{n,j}}{1 - T_{n,n}}$$

$$where \quad 1 \le i,j \le n-1$$

In the Sisal 1.2 implementation, three iterative loops were used. Although a more natural implementation using parallel loops is possible, update-in-place analysis is not, in general, applicable under such an implementation.

Under both analyses, all copying was eliminated and execution times were similar. Different times, however, were obtained when we assumed that the algorithm was analyzed in isolation. Recall that under Cann's method the entire program must be available for analysis. In Figure 8, we present the execution times for expression "state-removal(T), T[4]."

In this expression, the fourth row of T is shared. Consequently, under Cann's analysis, reference counting is used to determine when the row is copied. Under our analysis, the row is copied prior to the call of the state-removal function. The graph for the expression is depicted in Figure 9. Under Cann's analysis, the increased execution times are due to the reference-counting operations.

4.2 Matrix Inversion

We have chosen a particular algorithm for the matrix inverse function that exhibits better performance when the matrix is stored as a vector-of-vectors. Within the algorithm, rows of the matrix

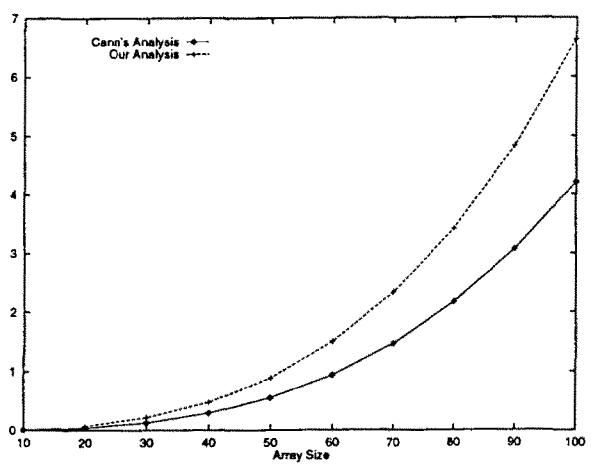


FIGURE 10 Execution times for the matrix inversion function when optimized both by Cann's analysis and by our analysis. The ordinate indicates the execution time in seconds.

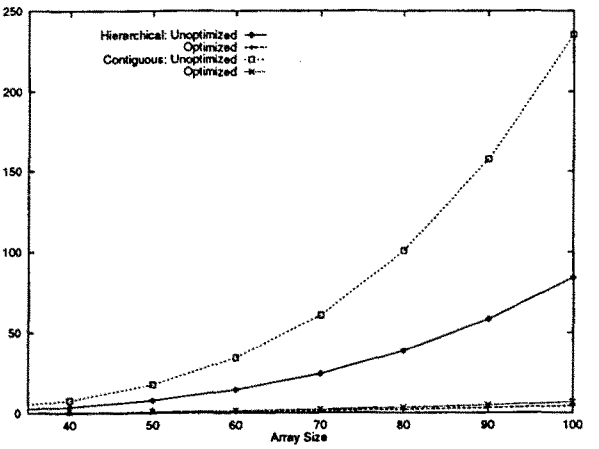


FIGURE 11 Execution times for the matrix inversion function using both hierarchical storage and mimicked contiguous storage for multidimensional arrays. The ordinate indicates the execution time in seconds.

are swapped. The swap operation can be performed quickly under the vector model—via a pointer copy operation. Under contiguous storage, the individual elements of the swapped rows must be copied.

A random matrix generator was used to create the input, a square matrix with a single non-zero entry in each row and in each column. In Figure 10 the execution times for both versions of the program are shown. Not surprisingly, the vector-of-vectors version runs faster because pointer swapping is used instead of row swapping, which is required under the dimensional version. To put this difference into perspective, Figure 11 depicts the execution times of unoptimized versions of both programs, along with the data repeated from Figure 10. We see that the copying penalty in the dimensional model is small compared to the penalty of not applying the optimizations at all.

5 CONCLUDING REMARKS

We have presented a new approach to perform update-in-place analysis. Our methods improve upon the methods developed by Cann by allowing an array to be stored either as a vector-of-vectors or in a contiguous, multidimensional space. Additionally, our approach allows functions to be fully optimized separately, stored for later use, and incorporated into an application. This ability supports the construction of large software systems, and can also result in better performance.

In the state-removal algorithm, the ability to optimize the function independently resulted in a decrease in execution time. Under Cann's approach, it must be assumed that subarrays, e.g., rows of a matrix, are shared. Consequently, reference-counting operations, which increase execution time, must be used to determine when row copying is necessary. Under the contiguous storage of arrays, no such assumption is necessary.

In some situations, however, it is advantageous to store arrays hierarchically. The matrix-inverse program is an example of such a situation. For contiguous multidimensional arrays, subarray-copy operations are necessary, which increase the execution time (as compared to hierarchical arrays). Specific analysis could be devised to determine which array storage model should be used. Our analysis is applicable for arrays stored hierarchically, provided that row sharing is not permitted. For the matrix-inverse program, each copy operation would only copy a pointer. Under this situation, we expect similar execution times to be achieved.

As a result of our analysis, three different array models can be supported efficiently: the vector-of-vectors, the flat, and the dimensional array models. The lack of reliance on a particular array model permits other benefits to be achieved. In particular, array representation under the dimensional-array model permits better expressibility of array operations, more efficient storage management, and decreased execution time.

Much work is needed to achieve these benefits. Our immediate plan is to first implement our algorithms. We will then test and verify our implementation using a dataflow simulator, TWINE [16]. This simulator allows a user-defined package, which can gather statistical information, to monitor the execution of an IFx graph. The result of these statistics will be used to guide our specific research directions.

ACKNOWLEDGMENTS

We thank John Motil for suggesting the Hohn-Aufenkamp state-removal algorithm as a test case. We also thank the anonymous reviewers for their comments and suggestions. Their input has helped to improve the article.

REFERENCES

[1] Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Trans. Computers*, vol. 39, pp. 300–318, March 1990.

[2] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications ACM*, vol. 21, Aug. 1978.

[3] A. P. W. Böhm, R. R. Oldehoeft, D. C. Cann, and J. T. Feo, *SISAL Reference Manual Language*, Version 2.0. Colorado State University—Lawrence Livermore National Laboratory, 1992.

[4] D. C. Cann, "Compilation techniques for high performance applicative computation," PhD Thesis, Colorado State University, CSU Tech. Rep. CS-80-108, 1989.

[5] D. C. Cann, "Retire Fortran? A debate rekindled," *CACM*, vol. 35, pp. 81–89, Aug. 1992.

[6] D. C. Cann and P. Evripidou, "Advanced array optimizations for high performance functional languages," *IEEE Trans. Parallel Distrib. Systems*, vol. 6, pp. 229–239, March 1995.

[7] D. C. Cann and R. R. Oldehoeft, "Reference count and copy elimination for parallel applicative computing," Department of Computer Science, Colorado State University, Tech. Rep. CS-88-129, Nov. 1988.

[8] J. B. Dennis, "Data flow supercomputers," *IEEE Computer*, pp. 48–56, Nov. 1980.

[9] J. T. Feo, "Arrays in SISAL," Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. UCRL-JC-106081, Sept. 1990. *Published in* The First International Workshop on Arrays, Functional Languages, and Parallel Systems.

[10] S. M. Fitzgerald, "Copy elimination for true multidimensional arrays," in *The Proceedings of the Third SISAL Users and Developers Conference*, LLNL CONF-9310206, October 1993.

[11] S. M. Fitzgerald, "Array memory optimizations for high speed parallel computing," PhD Thesis, University of Massachusetts Lowell, Nov. 1994.

[12] G. R. Gao, *A Code Mapping Scheme for Dataflow Software Pipelining*. New York: Kluwer Academic Publishers, 1990.

[13] M. Haines and W. Böhm, "A virtual shared addressing system for distributed memory SISAL," in *The Proceedings of the Third SISAL Users and Developers Conference*, October 1993, pp. 151–163.

[14] S. J. Kim, "A general approach to multiprocessor scheduling," PhD Thesis, The University of Texas at Austin, Austin, Texas 78712–1188, Feb. 1988.

[15] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, C. Kirkham, B. Noyce, and R. Thomas, *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual*, Version 1.2, M-146 ed. Livermore, CA: Lawrence Livermore National Laboratory, March 1985.

[16] P. J. Miller, "TWINE: A portable, extensible

SISAL execution kernel," in *Proceedings of the Second SISAL Users' Conference* (San Diego, CA), Lawrence Livermore National Laboratory, CONF-9210270, December 1992, pp. 243–256.

[17] R. R. Oldehoeft, "Implementing arrays in SISAL 2.0," *Proceedings of the Second SISAL Users' Conference*, December 1992, pp. 209–222.

[18] J. E. Ranelletti, "Graph transformation algorithms for array memory optimization in applicative languages," PhD Thesis, University of California Davis, UCD Tech. Rep. UCRL-53832, 1987.

[19] V. Sarkar, "Partitioning and scheduling parallel programs for multiprocessors," PhD Thesis, Stanford University, 1987.

[20] S. Seshu and M. B. Reed, *Linear Graphs and Electrical Networks*. Addison-Wesley, 1961.

[21] S. Skedzielewski and J. Glauert, *IF1—An Intermediate Form for Applicative Languages*, M-170 ed. Livermore, CA: Lawrence Livermore National Laboratory, July 1985.

[22] M. Welcome, S. Skedzielewski, R. K. Yates, and J. Ranelletti, *IF2—An applicative Language Intermediate Form with Explicit Memory Management*, M-195 ed. University of California—Lawrence Livermore National Laboratory, November 1986.

[23] L. M. Wilkens, "Modeling parallel computation via the fusion of timed Petri nets with an application to the mapping problem," PhD Thesis, Department of Computer Science, University of Massachusetts at Lowell, Lowell, MA, May 1992.

[24] R. M. Wolski, "Program partitioning and scheduling for NUMA computer architectures," PhD Thesis, University of California, Davis, 1994.