Memory Access Analysis and Optimization for Efficient Streaming Software Synthesis

Mohammad H. Foroozannejad, Mohammad Motamedi, Soheil Ghiasi, University of California, Davis

An important subset of streaming applications can be synthesized from statically-analyzable synchronous dataflow (SDF) models. The software synthesis process typically implements inter-actor communication in form of buffer arrays that are written to/read from by producers/consumers. Due to practical considerations (e.g., actor IPs) and the nature of SDF models, the generated code could contain a number of redundant buffer array accesses that become evident only in the synthesized software. We identify the optimization opportunity and develop an algorithm called RACE, which optimizes the synthesized code via elimination of such redundant memory operations. Experimental results show up to 80% improvement in the runtime of the synthesized streaming software.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors,Compilers

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Streaming applications, software synthesis, synchronous data flow, code generation, memory access, optimization

ACM Reference Format:
Mohammad H. Foroozannejad, Mohammad Motamedi, Soheil Ghiasi, 2015. Memory Access Analysis and Optimization for Efficient Streaming Software Synthesis. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Synchronous Data Flow (SDF) is an effective model of computation for specification of streaming applications whose tasks exhibit constant data production and consumption rates [Lee and Messerschmitt 1987; Geilen and Basten 2004]. Such applications are commonplace in embedded systems, and form the core of many signal processing, communications, security and encryption functionalities. Constant data rates allow static analysis and optimization of SDF models and associated implementations. As a result, there exists a number of application modeling and synthesis environments that generate software implementations of SDF models [Gordon et al. 2002; Simulink 2015; Eker et al. 2003; NI 2015].

The software synthesis process involves a number of algorithmic steps, such as task scheduling and buffer allocation, and culminates in code generation. The generated code can be passed to a standard compiler to generate executable binaries. The synthesized software tends to follow specific styling conventions. For example, inter-task

Authors addresses: M. Foroozannejad (corresponding author), M. Motamedi, and S. Ghiasi, Department of Electrical and Computer Engineering, University of California, Davis, CA 95616; email: mhforoozan@ucdavis.edu.

DOI: http://dx.doi.org/10.1145/0000000.0000000

data communication is typically implemented via buffer arrays that are written to (read from) by the data producer (consumer).

In principle, specifying the application as a set of tasks and inter-task communication channels promises portability in regard to the final hardware platform. However, research shows that synthesizing software from a large SDF graph for a platform with small number of processors will result in performance loss, compared with synthesizing a smaller graph of the same application when the same platform is targeted [Hashemi et al. 2012]. It is partly because the overhead of coordinating among different tasks is justified only if sufficient amount of parallelism exists in the platform.

When the number of processing units (cores) are considerably less than the number of tasks in the graph, a large number of tasks are forced to be executed on a single core. On the other hand, the nature of dataflow streaming applications involves repeated passing of data from one task to another. This tends to result in redundant array accesses in the synthesized code if some tasks merely reorder, duplicate or drop (e.g., down sampling) data tokens[1], and pass them downstream to other tasks.

Since the behavior of SDF graph is statically analyzable, one can trace array memory accesses to characterize redundant array accesses to a large extent. We demonstrate the issue through examples and experiments, and present an algorithm called RACE (Redundant Array Copy Elimination), which exploits the statically analyzable nature of SDF execution to identify and optimize such cases. RACE is a source-to-source optimization algorithm which transforms the given generated C code into an optimized version in the same native programming language (C code). The optimization will redeem some of the performance loss caused by synthesizing software for a platform with a few number of processors from a large data flow graph.

2. BACKGROUND AND MOTIVATION

Data Flow graphs are commonly used to model streaming applications. One of the popular variations of these graphs is Synchronous Data Flow (SDF) graphs [Lee and Messerschmitt 1987]. The distinctive property of SDF graphs is that the production and consumption rates of actors are predefined [Geilen and Basten 2004]. This distinction allows software synthesis tools to statically analyze the behavior of the applications modeled by SDFs, and predict the behavior in respect to different constraints and assumptions about the system.

Software synthesis tools [Gordon et al. 2002; Simulink 2015] exploit this property to generate executable codes from the SDF modeled applications. The synthesis process involves several algorithmic steps, such as task assignment [Hashemi and Ghiasi 2010; Szymanek and Krzysztof 2003a], task scheduling [Bhattacharyya et al. 1996; Murthy et al. 1997], buffer allocation [Murthy and Bhattacharyya 2004; Foroozannejad et al. 2012], processor mapping [Foroozannejad et al. 2014; Tosun 2011], and finally code generation [Halbwachs et al. 1991]. The combined effect of these algorithmic steps undertakes to produce high quality software from the SDF modeled application for the target hardware platform. Figure 1 demonstrates these steps in the order that they are performed in the synthesis process.

In the task assignment step, every task is assigned to a processor for execution. The typical objective of task assignment is to have a balanced workload among all processors and reduce inter processor communication at the same time. At this point, the processing units are considered to be virtual processors since the physical location of the processors on the chip is ignored [Szymanek and Krzysztof 2003b].

[1] as opposed to change the data values via arithmetic and logic operations

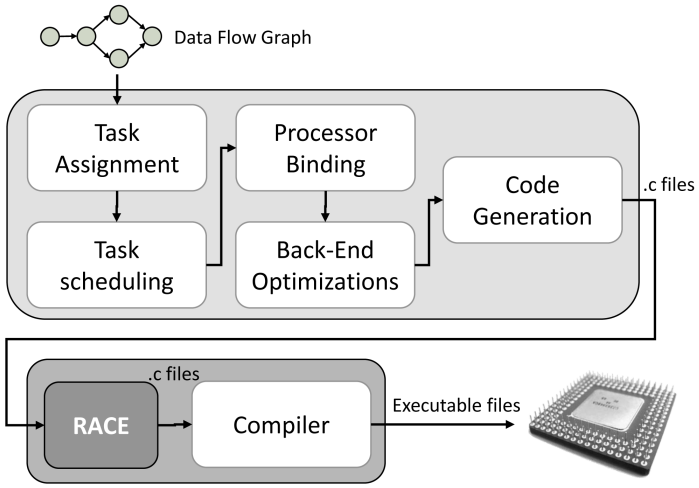


Fig. 1. The flow of automatic software synthesis for SDF modeled streaming applications based on given application graph and target manycore model. The block RACE is the contribution of this paper to this flow.

After the task assignment step, each group of tasks that are assigned to a virtual processor will be executed on a uniprocessor [2]. The order in which the tasks are executed sequentially on each core is calculated in the task scheduling step. Latency, code size , and data memory footprint are common objectives in this step. To this end, different types of algorithms have been introduced each of which has a different effect in terms of the given quality measures. Single Appearance (SA) scheduling is one of the scheduling methods which is commonly used in embedded platforms. In SA scheduling, each task appears only once in the sequential code, and hence yields the minimum code size.

Processor binding (processor mapping) allocates each virtual processor to a physical core on the chip. In this step, maximum and total communication distances between connecting cores are considered in addition to other network criteria such as dead-lock and congestion.

In the backend optimization step, post-scheduling and in-core resource allocation optimizations (e.g. buffer allocation) are performed [Foroozannejad et al. 2012].

Finally, in the code generation step of the synthesis process, the sequential code for each core is generated (usually in a high level programming language) [Halbwachs et al. 1991]. At this point, the modeled application has been completely converted to the sequential codes assigned to each core processor of the final hardware platform. The generated code can be passed to a standard compiler to generate executable binaries.

Any discussion in the literature on SDF modeled applications is aimed at the graph abstraction layer, and ends with code generation. Dealing with the produced code is considered a topic for conventional standard compilers which is a well researched area. However, we show that this auto generated code inherits many characteristics of the graph level SDF model. Therefore, it is tightly structured in accordance with the abstraction layer directives. This structure may appear rather inefficient in the final sequential form of the code.

[2]the many-core platform discussed in this paper consists of a network of on chip uniprocessors (cores)

2.1. Motivating Example

Figure 2 demonstrates a toy application modeled by a SDF graph. In the remainder of this section, we show the types of optimizations discussed in this paper using this example. In figure 2.a, edges of the graph represent inter-task (actor) data dependency. They are annotated with the names and production/consumption rates that implement inter-task dependency in the synthesized software. The name of each task is also provided in a black label adjacent to the task.

Figure 2.b represents a single appearance schedule of the tasks in the graph. The number next to each task in the schedule represents the repetition factor of the task. This repetition factor ensures that the required input data tokens to the connected tasks are produced, and the data tokens ready on the input buffers of the running task are consumed before moving on to executing the next task in the schedule.

Figure 2.c demonstrates the C code generated by the software synthesis process. The parts in gray are generated directly from the code inside each task, and the for-loops wrapped around them provide the required repetition by the given schedule. Note that tasks B and C facilitate parallel processes if multiple processors are available. In this example, they are both run sequentially on a single core processor along with other tasks.

Figure 2.d shows the paths on which two data elements of the source array travel to arrive at their final destinations (before their value changes). Lastly, some of the final destinations of the source data elements are shown in Figure 2.e. As it can be seen in figure 2, if arrays P and Q are reconstructed directly from M, then array N and all of its associated assignments can be removed from the code to optimize its performance.

3. RELATED WORK

Synthesis of streaming software from SDF modeled applications targeting single, homogeneous or heterogeneous multi-processors has been a subject of active research [Thies et al. 2003; Gordon et al. 2006; Stuijk et al. 2007; Hashemi and Ghiasi 2010; Hashemi et al. 2012; Hashemi et al. 2013]. The required memory for storage of application data and instructions is especially important for embedded platforms, where the underlying hardware is often resource-constrained.

Memory optimization techniques developed for SDF models can be divided into scheduling-oriented and allocation-based techniques. Scheduling-oriented techniques consider the impact of task scheduling on instruction memory [Bhattacharyya et al. 1996], total buffer size [Murthy et al. 1997; Oh et al. 2005], and overall code size [Karczmarek et al. 2003]; while allocation-based schemes are typically applied after scheduling [Murthy and Bhattacharyya 2001; 2004; Foroozannejad et al. 2012]. The aforementioned memory optimization techniques only aim at minimizing the memory footprint of the system and do not address the problem on hand which is reducing the number of memory accesses in the final executable codes.

In fact, Prior work from embedded systems community deals with analysis at the level of SDF graphs, and stops after software implementation is synthesized. Optimization of the produced code is conventionally considered to be within the purview of standard compilers, and out of the scope of SDF synthesis. However, the subject problem of this paper and the proposed solution are specific to software that is synthesized from SDFs, as the ability to statically analyze the behavior is central to both detection and optimization of redundant array accesses. Since SDF models have not been of particular interest in conventional programming languages and compilers communinities, the problem has not received any attention from compiler researchers. As we will demonstrate in Section 8, state of the art compilers (e.g., gcc) are unable to handle array optimization of the type developed in this paper.

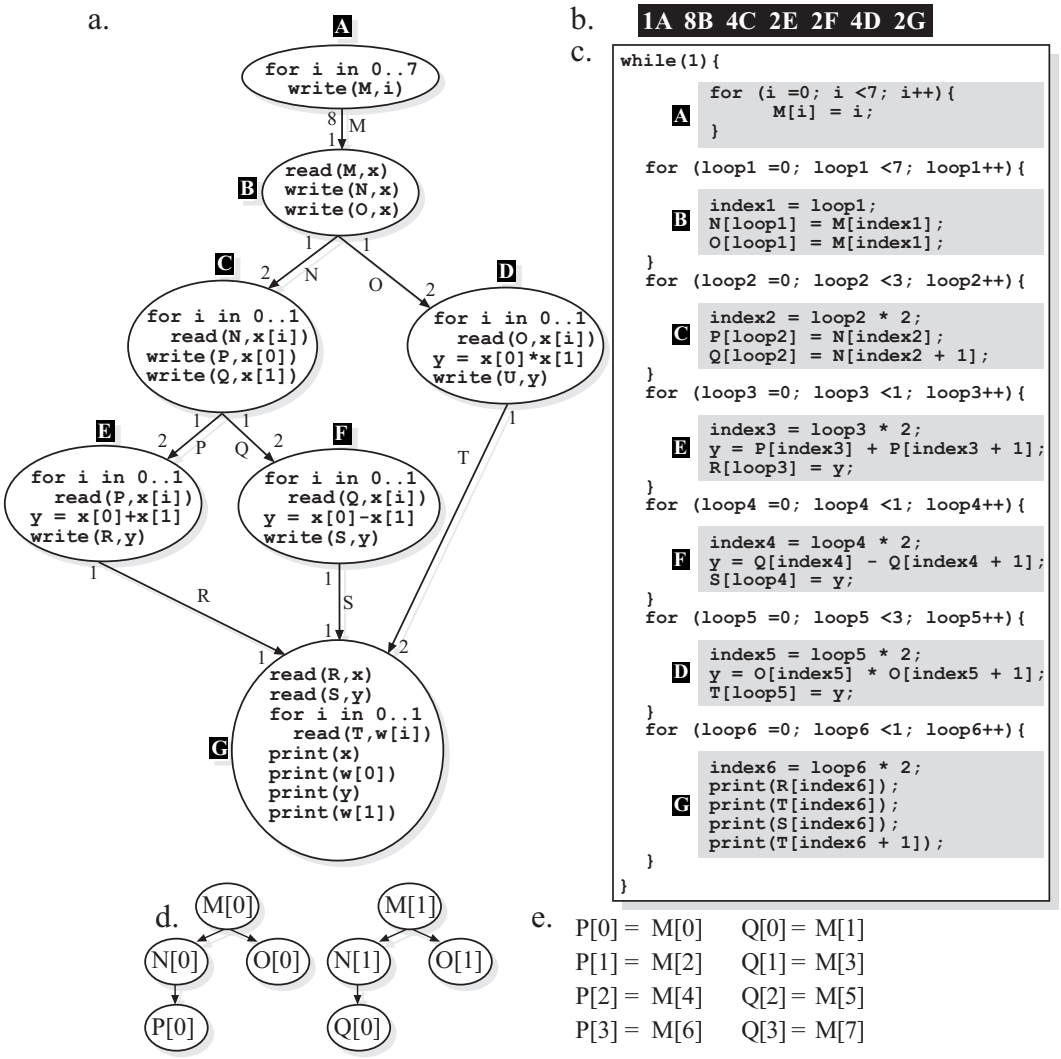


Fig. 2. a. A toy example SDF graph. The edges of the graph are annotated with the names and production/consumption rates that implement inter-task dependency in the synthesized software. The name of each task is also provided in a black label adjacent to the task. b. A single appearance schedule of the tasks in the graph. c. The synthesized code generated from the given SDF graph under the given SA schedule targeting a uniprocessor platform. The snippet codes in gray are directly reflected from the tasks of the SDF graph, and the loops implement the required repetition factor for each task. d. Two examples paths that data elements take to arrive at their final destinations. d. The mapping between the source array and two destination arrays.

4. PROBLEM STATEMENT

The problem we are tackling in this paper is to reduce the amount of memory access caused by intensive data transfer, reorder, duplicate, drop (ex. down sampling) which are commonly transpired in signal processing and streaming applications. The problem in its general form is to difficult to address. However, as it was discussed in section 2, SDF modeled applications possess characteristics that can be used to solve the problem. To this end, we assume the following criteria in the target applications:

1- The location of the data tokens within the communication buffers is not coupled with the values of these tokens, i.e. the indices of the array variables are statically resolved in the code.

2- Each iteration of the application is representative of other iterations in terms of the location of the data tokens within the communication buffers. This rule ensures that analyzing an application for only one iteration guarantees the solidity of the optimization in other iterations.

In section 8 we illustrate the practicality of these assumptions in the real-life streaming applications.

5. MEMORY ACCESS MODELING

We first explain the problem in the scalar form to introduce some of the concepts and the terminology used in the remainder of this paper. Please note that this problem in its scalar form may appear to be unnecessary since the conventional compilers are equipped with better ways to solve it, either through register allocation or other known techniques. However, the introduction of the scalar from is just a way to ease the readers to follow the main concepts on the more complex form of the problem introduced in the second half of this section where the array form is discussed. The real life applications and the solution introduced in this paper are within the array form.

5.1. Scalar Variables

We first define Run Time Dependency Graph (*RTDG*). In addition to data dependency information between different variables in the program the RTDG also contains run time information. This information is gathered by following the control flow from a high-level execution of the program under the assumption that the control flow of the code, e.g. loop variables or the condition in conditional instructions, does not depend on any of the input variables. The majority of SDF modeled applications follow this assumption since this programming model is most effective when the control of the program only relies on the availability of the input data and not the conditional structure of the code.

Definition 5.1. The Run Time Dependency Graph (RTDG) is a graph with a set of vertices and two sets of edges and is defined as follows:

$$
\begin{aligned}
G = \ & < V, E, M > \\
V = \ & \{v : a^{\alpha} \mid a \in \ \mathbf{VS},\ \alpha \in \mathbb{Z}\} \\
E = \ & \{e : a^{\alpha} \rightarrow b^{\beta} \mid a, b, ... \in \mathbf{VS},\ b := f(a, ...)\} \\
M = \ & \{m : a^{\alpha} \rightarrow a^{\alpha+1} \mid a \in \mathbf{VS}\}
\end{aligned}
$$

VS is the set of all variables in the code and α is the α^{th} time the variable a has changed value during execution of the program and is called *change count* of variable a. We use the notation ":=" to refer to the assignment operation in high-level languages, thus "$b := f(a, ...)$" means the value of $f(a, ...)$ which is a mathematical expression that relies on variable a and possibly other variables, is assigned to variable b. E is the set of dependency edges between variables, and M is the set of alteration edges. Any time that an assignment instruction is reached, a new instance of the variable accepting the new value is created as a node in the RTDG. The change count of the new instance is incremented by 1 from the last instance of the same variable or assigned zero if this is first time the assignment is reached. The new generated node is also connected to the last instance with an alteration edge. Figure 3 shows the RTDG for the code shown on the right hand side of the picture. The code is assumed to have before and after pieces.

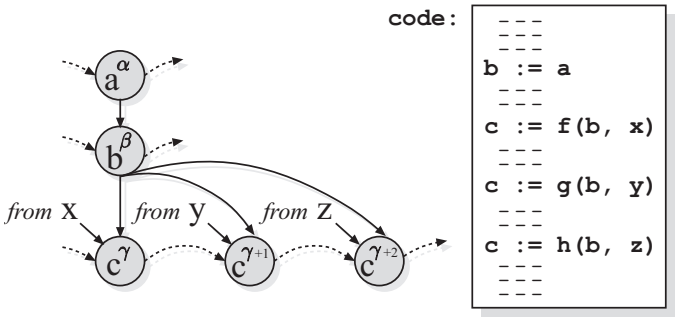


Fig. 3. Example of the Run Time Dependency Graph (RTDG) created from the given code. Solid arrows represent the dependency edges and dashed arrows demonstrate the alteration edges. Since the code is assumed to have before and after pieces, some of the arrows do not show start or stop points.

One assignment in the code can create a series of nodes in the RTDG if the assignment exists in a loop and is visited multiple times during the execution of the program.

Each node $a^\alpha \in V$ of the graph G also stores the O_{a^α} and R_{a^α} information as described below:

Definition 5.2. R_{a^α}: The assignment instruction in the code that changes a for the α^{th} time. Note that multiple values of α might have the same R_{a^α} if the assignment instruction is iteratively executed in a loop.

Definition 5.3. O_{a^α}: The Instruction Order which is a number defining a global order on the instructions. We use the notion of instruction order as the n^{th} instruction in the representative execution on an abstract single issue architecture. Any time that a changes, the associated instruction order of the corresponding instruction at that instance is kept in O_{a^α}.

In the above definition of the RTDG, each node in graph G represents a memory transaction not differentiating between registers and main memory. The objective of the problem is to minimize $|V|$ by eliminating some of the assignment instructions from the code without changing its functionality.

Definition 5.4. Parents and Children sets of a node in the graph G: The set of nodes that directly contribute to the new value of the variable instance a^α in an assignment instruction is called the "*Parent Set*" of a^α ($PS(a^\alpha)$). The "*Children Set*" of a^α ($CS(a^\alpha)$) is the set of nodes whose values are directly influenced by the node a^α in assignment instructions.

In figure 3, $PS(b^\beta) = \{a^\alpha\}$ and $CS(b^\beta) = \{c^\gamma, c^{\gamma+1}, c^{\gamma+2}\}$.

Definition 5.5. We define *simple* and *complex* assignments as follows:
An assignment instruction with only one variable on the right hand side is referred to as a simple assignment. The corresponding execution of a simple assignment to a^α in RTDG will have $|PS(a^\alpha)| = 1$. The assignment is complex otherwise.

In figure 3 "$b := a$" is a simple assignment and "$c := f(b, x)$" is a complex assignment.

In general, the term "copy" is only used when simple assignments are in discussion as one variable is copied over another. Complex assignments suggest a form of computation in the code. Therefore we only target simple assignments for the redundant memory access elimination process in this paper. In figure 3, assume that the simple assignment "$b := a$" is the target instruction for elimination. The intuition is that we could use a in any instruction that is using the content of variable b from this point

on in the code until b receives a new value. In this example, b can be replaced by a in "$c := f(b, x)$" instruction.

LEMMA 5.6. *If the instruction R_2 $(c := f(b, x))$ is executed after the instruction R_1 $(b := a)$ in the code, the variable b can be replaced by the variable a in R_2 if the following conditions are met. α, β, and γ are assumed to be known change counts for each variable.*

$$\forall \gamma \leq \gamma_{max}: \quad (\gamma_{max} \text{ represents the last time } c \text{ changes in instruction } R_2)$$

I.	$PS(b^\beta) = \{a^\alpha\}$	R_1 *is a simple assignment*
II.	$O_{a^\alpha} < O_{c^\gamma}$	R_1 *is executed before* R_2
III.	$O_{a^{\alpha+1}} > O_{c^\gamma}$	a *doesn't change before* R_2 *is executed*

In the example presented in figure 3, b^β is the only child of a^α, thus, the assignment is simple (condition I in lemma 5.6). The instruction order of a^α is also less than that in c^γ, $c^{\gamma+1}$, and $c^{\gamma+2}$ where c is being assigned new values by f, g, and h functions respectively (condition II in lemma 5.6). Lastly, under the assumption that a is not changed during the time that variable b is being used, we can also state that $O_{a^{\alpha+1}}$ which represents the next time a is changed (not shown in the code) is bigger than O_{c^γ}, $O_{c^{\gamma+1}}$, and $O_{c^{\gamma+2}}$ (condition III in lemma 5.6). Therefore, in this example b can be replaced by a and the target assignment "$b := a$" can be eliminated.

The change count values α, β, and γ are used in this example as an indication that we do not have any prior knowledge about previous assignments, in which, the values of the variables a, b, and c might have changed.

Definition 5.7. Chain Assignment: When a series of nodes in the RTDG are connected from one to another by only one dependency edge on each connection (simple assignments), they are collectively called a *chain assignment*. The first node and the last node of a chain assignment are denoted the *source* and *destination* of the chain, respectively.

Chain assignments represent a series of redundant operations that can be optimized. We can eliminate the intermediate assignments in a chain if we could verify that the three conditions in lemma 5.6 are applicable between the source and the destination of the chain. The rules I and II in lemma 5.6 are embedded in the given definition of a chain assignment, thus, rule III in lemma 5.6 is the deciding condition.

LEMMA 5.8. *If $a^\alpha \rightarrow b^\beta \rightarrow ... \rightarrow p^\rho$ is a chain assignment, then the variable p can be replaced by the variable a in the next instruction(s) consuming p if the following condition is met (ρ_{max} represents the last time p changes in the corresponding instruction in the chain):*

$$\forall \rho \leq \rho_{max}: \quad O_{a^{\alpha+1}} > O_{p^\rho}$$

Figure 4 depicts an example of a chain assignment in the RTDG. For practical reasons, we only use the term *chain assignment* for the sequences that comply with the lemma 5.8.

Definition 5.9. Landing instruction: The instruction in which the destination variable of a chain assignment is consumed is called the *landing instruction* of the chain.

In the chain assignment shown in figure 4, we can replace d with a in the landing instruction "$e := f(d, x)$" in the code which consequently creates the opportunity for the other intermediate assignments to become candidates for elimination if their contribu-

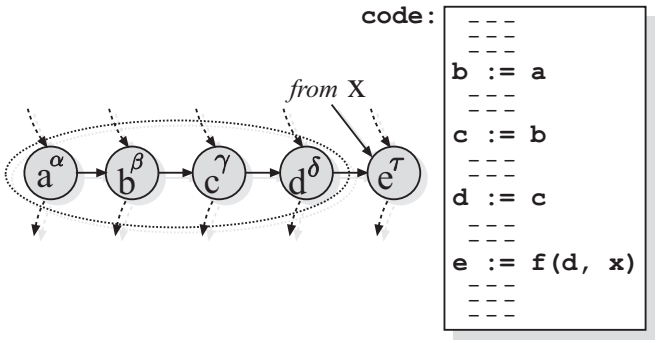


Fig. 4. Example of a Chain Assignment created form the given code. The part within the enclosed dashed area represent a chain in accordance to lemma 5.8, and the node e represents the Landing Instruction of the chain.

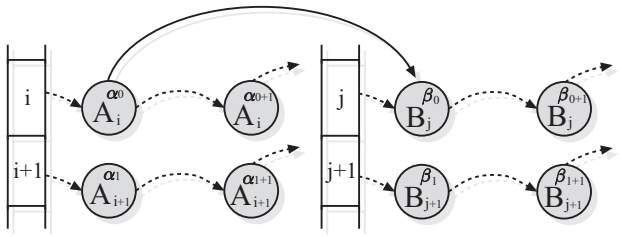


Fig. 5. Example of the RTDG in the array form.

tion to the code ends here. Note that replacing the variables in the landing instruction doesn't necessarily permit us to remove the intermediate assignments. In fact, we need to make sure that every instruction influenced by an intermediate assignment is being considered, and only if the intermediate assignment becomes completely redundant then it can be removed from the code. In section 6 we discuss how to determine if an assignment is no longer relevant in the code as part of the RACE algorithm.

Definition 5.10. Depth: The number of intermediate nodes in between the source and the destination of a chain is called *depth* of the chain.

The depth of a chain is an upper bound on the number of assignments that can be removed from the code for this chain. In figure 4, the depth of the given chain is four.

5.2. Array Variables

In the array form of the problem, each element of an array is represented in the RTDG as if it is a scalar variable. However, another dimension is added to the graph to maintain the stretch of the array along its index dimensions. The discussion in this paper focuses only on one dimensional arrays. However, arrays with more dimensions can be easily handled by either transforming the multi dimensional array into a one dimensional array or by adding more dimensions to the RTDG. Figure 5 illustrates the index dimension in the graph.

The chain assignments are recognized on an element by element basis. In other words, each element of an array is followed individually throughout the code regardless of its association with other elements of the array. Each chain assignment connects the source of the chain to the destination. The collection of all chain assignments starting with one array with all its different elements (indexes) reveal the mapping between the elements of the source array and the destination array. Figure 2.e shows an example of

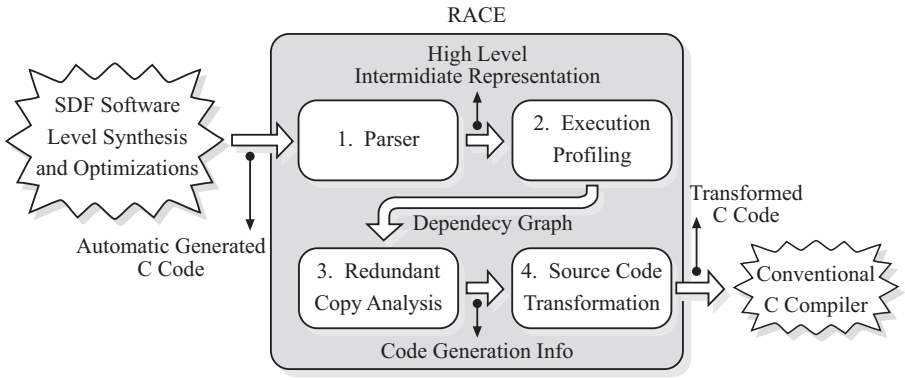


Fig. 6. RACE block diagram.

such mappings. This mapping helps us to replace the destination array with the source array in the landing instruction after the intermediate assignments are eliminated.

More discussion on finding the mappings and replacing the final arrays in the landing assignments is given in the next section (RACE Algorithm) where we propose our solution to this problem.

6. RACE ALGORITHM

RACE is a source-to-source optimization algorithm which means the input to the algorithm is a high level source code and the output is the optimized version of the same code. In this section every time we mention code or program the source code to the RACE algorithm is meant, and the word algorithm is reserved for the RACE algorithm.

RACE consists of four main modules depicted in figure 6. The main input and output of each module are also shown in the figure. In the remainder of this section we describe each module in more detail.

6.1. Parser

The function of the parser used in RACE is to simply collect enough information from the input code for the execution profiling unit to be able to determine the index values of the arrays during the execution time, differentiate between alternate types of assignments, and finally disclose the relationship between the indexing of an array and the loop structures of the code. A high level intermediate representation of the code is generated in the parser and is sent to the execution profiling module where this information can be used to evaluate the code for further analysis.

6.2. Execution Profiling

In this module, the entire code is evaluated without having access to the input data (dynamic information). The type and number of input and output data, however, is known as they are pre-defined in SDF models and are translated into the type and size of the data arrays in the code. In each step of the execution, the high level state of the program is stored. This information consists of the current Instruction Order (analogous to program counter), the value of the index variables if they are known at the time, and also the next instruction to run.

Definition 6.1. The Known and Unknown instructions: As instruction is known if the values of the variables involved in the instruction are known and can be statically resolved at the time of execution profiling. An instruction is unknown otherwise.

The known assignments are usually associated with the index variables in which a known numeric value is assigned to a variable. The unknown assignments are generally affiliated with the input data, hence, their values are unknown to the algorithm when the code is statically analyzed. When such an assignment is encountered, a node in the RTDG is generated and connected to the proper node(s) in the graph based on the other variable(s) on the right hand side of the assignment. Please note that all index variables in the current assignment should have known values, otherwise that particular application is out of the scope of the RACE optimization.

Using the information extracted in this module, the RTDG is generated for the entire variable set of the program and sent to the next module.

Definition 6.2. Node Table: For practical reasons, we also create a set for each assignment consisting of every node of the RTDG associated with that assignment. This set is then kept in a hash table called *Node Table* using the related assignment as the key.

The Node Table is also sent to the next module along with the RTDG.

Figure 7.a demonstrates a small portion of the RTDG associated with the C code shown in figure 2.c (section 2.1). The code was generated from the toy SDF graph shown in figure 2.a under the given schedule targeting a single uniprocessor. The solid edges represent the variable dependencies and the dotted edges denote the alterations of each variable. The numbers next to each variable in each node of the RTDG represents the change count of that variable. None of the array variables change more than once in this application, however, the variable y changes frequently throughout the execution of the application, thus the alteration edges are only shown for this variable.

In each node, the instruction responsible for changing the value of the variable associated with that node is also given. as it was discussed in section 5, the nodes of the RTDG not only represent the variables but the assignments in which the values of the variables change as well. Two entries of the Node Table associated with this RTDG are also given in figure 7.d.

6.3. Redundant Memory Access Analysis

In this module, the redundant memory accesses are recognized. The first step towards this recognition is to identify the chain assignments explained in section 5. Starting from any occurrence of a variable, the dependency graph can be recursively traversed forward from parents to children to get to the last node(s) of the chain. As it is shown in figure 4, the last node in a chain is the one whose children do not represent a simple assignment. Note that since an array can be copied over multiple arrays a chain can be in the form of a tree with multiple destinations, each of which can be subject to optimization. In the RTDG given in figure 7.a, the gray area represents a chain tree with multiple branches. The node at the end of each branch has multiple dependencies, thus representing a complex assignment. Algorithm 1 shows how the RTDG is recursively traversed to extract chains that start from a given variable.

Moreover, as it was discussed in section 5 the source and the destination of the chain must satisfy the condition in lemma 5.8. The test is checking if the Instruction Order of the source is smaller than that of all destinations of the chain.

The recursive procedure allows us to start from one element of an array and discover the settling destination of the value of this element before it is engaged in any calculation in the program (consumed in the landing instruction). The number of intermediate assignments that this value goes through before getting to the destination (the depth of the chain) is, in fact, the number of redundant copies we can potentially eliminate from the code. If this procedure is run on all elements of an array, we will have the complete mapping between the source and the destination arrays.

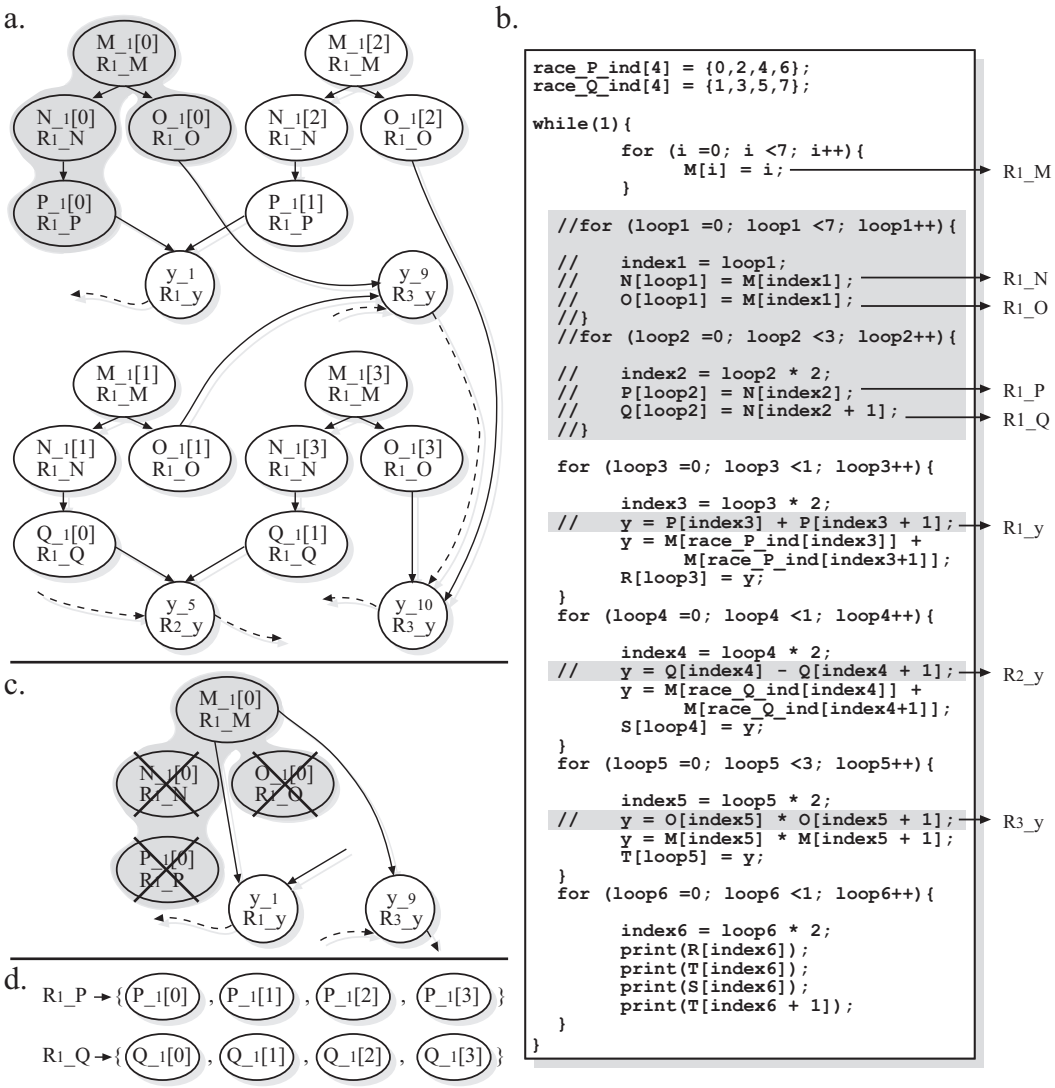


Fig. 7. a. A partial RTDG corresponding to the code illustrated in figure 2.c. The instruction responsible for changing the value of the variable associated with a node is also shown in the node. The part in gray represents an example of a chain assignment. b. The optimized version of the code shown in figure 2.c after applying the RACE optimization. Some of the instructions are named to add clarity between the RTDG and the code. The parts in gray are the eliminated assignments. c. The process of eliminating intermediate nodes from the RTDG in a chain assignment. d. Two entries of the Node Table associated with the given RTDG.

Note that different elements of an array can potentially cross different paths in the code and can be part of different chains with different depths. We define the depth of a mapping to be the maximum depth of all chains involved in the mapping. All array variables used in the code can be sorted by the depth of their mappings. Intuitively, we would want to start the redundant copy elimination process from the array with the maximum depth. It is because the chains with the depth larger than two contain chains with smaller depths depending on which node is selected as the source. Therefore,

ALGORITHM 1: chainIdentification

Input: $DG, chain, parent$

```
{/* DG : dependency graph , s: the source of the chain, chain : the data structure to keep
the results, parent : the start node for the traverse forward */}
if parent.childrenSet = φ then
   return
end if
chain.add(parent){/* the current node is added to the chain */}
for child in parent.childrenSet do
   if child.type = simple then
      chainIdentification(DG, s, chain, child)
   else
      if parent.order < s.nextAlter.order then
         parent ← end point{/* mark parent as end point */}
         {/* order is the instruction order and s.nextAlter is the next instance when the
         value of s changes following the alteration edges */}
      end if
   end if
end for
```

when the larger chain is considered first the partial chains existing within the large chain would be processed only once.

Depending on the nesting loop structure of the code, this mapping can have different patterns. In principle, there are only two major patterns that can occur between the source and the destination arrays. Figure 8 illustrates these two patterns. In figure 8.a, the source array is read and copied, element by element, onto another array with the same size (or more than one array) or multiple times onto different elements of another array(s) with a larger size.

In figure 8.b, the size of the source array is larger than the size of the destination array. Therefore, in each iteration one part of the source array is copied onto the destination array (or partially copied if the destination array has another source), and in the next iteration another part of the source is copied over the previous values of the destination. In this pattern, the mapping between the first part in the source array and the destination array is repeated in other iterations as well. The only difference is, in each iteration the indexes of the elements read from the source array is increased by the size of the destination array.

Note that in both of these mapping patterns, there is only one source array. It is not because there will never be a situation in which two arrays are copied in one destination. However, in this methodology we only consider one source at a time. When there are two or more sources for one destination, each part of this transaction is considered separately.

After uncovering the mapping of a source array and also the type of its pattern, the next step would be to start the elimination process by updating the RTDG. To this end, for every chain with the current source array, all intermediate nodes in the chain are removed from the graph, and the source and the destination become directly connected. These nodes are also removed from the Node Table described in the previous subsection. Whenever an entry (a set of nodes associated with an assignment) in this table becomes empty, the related assignment can be removed from the code. In figure 7.c, an example for the node elimination process is demonstrated. Any node removed from the RTDG will be removed from its associated entry in the Node Table as well.

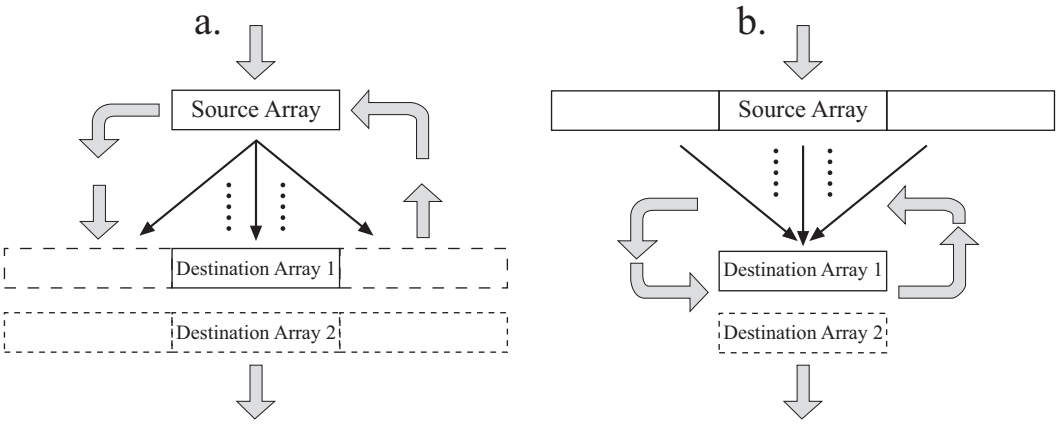


Fig. 8. Two major patterns between the source and the destination arrays.

6.4. Code Transformation

It is still necessary to insert the proper assignments connecting the source and the destination arrays to complete the code. The code transformation module is responsible for removing/generating irrelevant/required code lines. The code generation info shown in figure 6 consists of any pair of source and destination arrays that form chain assignments, the mapping between the pairs, and irrelevant code lines which needs to be removed from the code. The irrelevant code lines are the ones whose entry in the Node Table have become empty during the node elimination process meaning that the assignment will not have an observable effect on program behavior. For example in figure 7.c, the assignments $R1\_N$, $R1\_O$, $R1\_P$, AND $R1\_Q$ become irrelevant after the node elimination process.

The irrelevant code lines are simply deleted (or commented out) from the original code. When these code lines are removed, there might remain other code lines which become irrelevant as well, such as the instructions in charge of the indexing of the deleted array assignments or some of the loop structures. We leave this part to the conventional compilers (e.g. gcc) as they are equipped with algorithms for dead code elimination [Debray et al. 2000].

To generate the necessary code lines, we first discuss mappings of pattern type $a.$ in figure 8. Assume that the mapping between the source and the destination arrays is recognized as follows: $B[0] = A[i_0]$, $B[1] = A[i_1]$, ..., $B[n] = A[i_n]$ where A and B are the source and the destination arrays, respectively. In general, we can always replace the the destination array with the source array in the landing instruction as it is shown in figure 9.a.

Definition 6.3. Mapping Array: A constant array in the size of the destination array, which discloses the relationship between the source and the destination arrays. In other words, mapping array is essentially a lookup table for the indices of the two arrays.

indA in figure 9 is an example of a mapping array. The preferred location for the mapping array would be outside of the main loop of the program. The C code shown in figure 7.b is the optimized version of the the original code given in figure 2.c after performing the RACE optimization. The mapping patterns in this example are all of the type a. in figure 8. The first two lines of the code instantiate the mapping arrays

a.

```
1:    indA = {i0, i1, i2, ..., in};
      \* C[j] = f(B[k + const]); -->    the old instruction *\
2:    C[j] = f(A[indA[k + const]]);    \* the new auto generated instruction *\
```

b.

```
1:    indA = {i0, i1, i2, ..., in};
2:    jump = -offset;

...

3:    loop{
4:        jump = jump + offset;
          \* C[j] = f(B[k + const]); --> the old instruction *\
5:        C[j] = f(A[indA[k + const + jump]]); \* the new auto generated instruction *\
      }
```

Fig. 9. Examples of the optimized codes associated with the patterns a. and b. shown in figure 8, respectively.

between the array M and the arrays P and Q in the code. The landing instructions in which the arrays P and Q are used ($R1\_y$ and $R2\_y$) are changed accordingly.

In the case of mappings of pattern type $b.$ the same principle is applied. However, since the source array is larger than the destination array and periodically copied onto the destination with some offset in each iteration, the generated code needs to be adjusted as it is shown in figure 9.b.

As it was discussed in section 5, *offset* is usually equal to the size of the destination array. The first two lines of the code shown in figure 9.b should be located outside of the main loop of the program.

7. EXTENSIONS TO RACE

In this section, we explore two more optimizations which can be performed in addition to the main RACE algorithm to further improve the quality of the generated final code. These optimizations use the same information produced in the parser and execution profiling modules but apply them differently in the copy elimination and code transformation modules.

7.1. Reverse Application

Previously discussed in section 5, the source and the destination of a chain must satisfy the condition in lemma 5.8, which guarantees that the source is not changed before it is used in the landing instruction. Algorithm 1 allows us to start from one element of an array and discover the settling destination of the value of this element by traversing the RTDG in the forward direction. This direction also complies with the actual flow code. However, after applying RACE on the code in this direction, there remains a possible case in which traversing the graph in the reverse direction can uncover additional copy elimination opportunities.

Figure 10.b shows a snipped of code copied (and changed to suit as an example) from the FFT application, which is a benchmark application presented in section 8. Figure 10.d illustrates a small portion of the RTDG associated with the given code. As it is shown in the figure, the value of $results[0]$ is copied to $N[0]$ but changes before it can be used in the landing instruction. It violates the condition in lemma 5.8, therefore this copy will not form a chain in the RTDG.

However, if the RTDG is traversed in the reverse direction, and the the condition in lemma 5.8 is applied in the reverse order, more chains can be revealed in the RTDG. An example chain formed in this order is shown in figure 10.e. If the RTDG is traversed

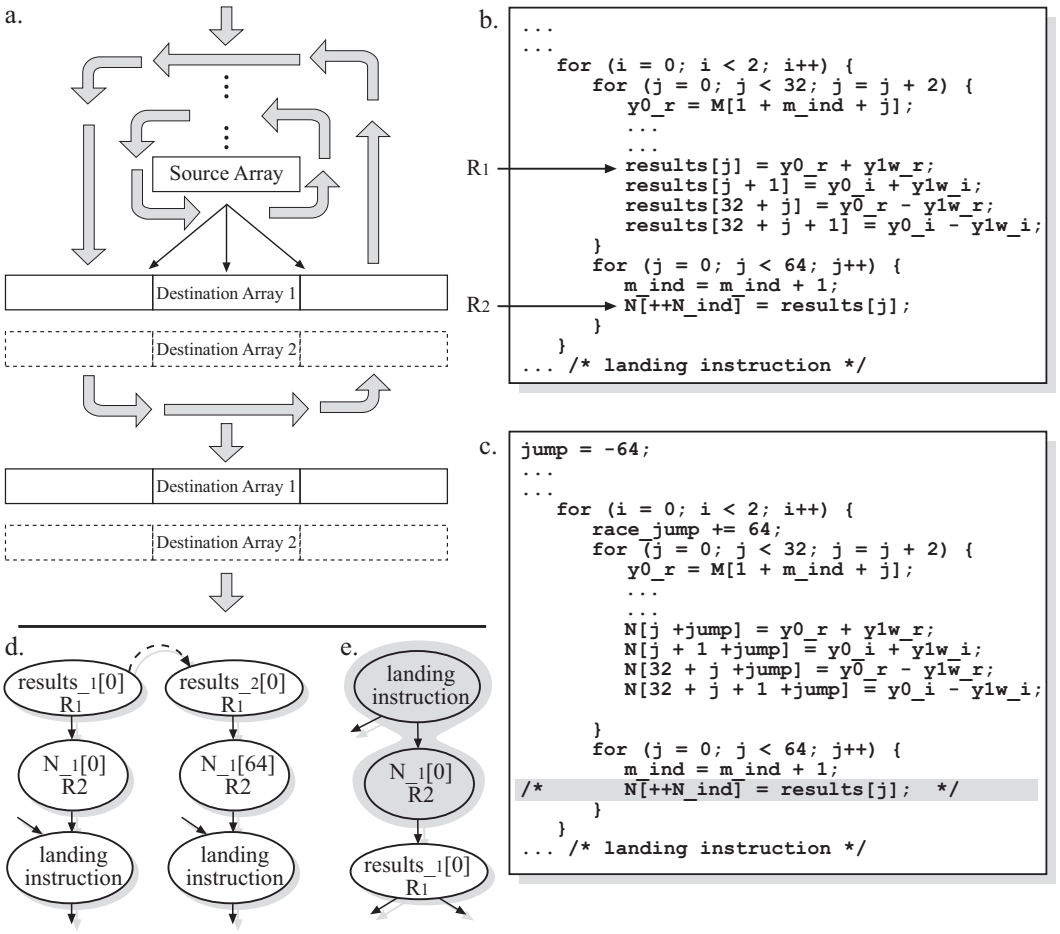


Fig. 10. a. the general case in which the reverse RACE optimization is applicable. b. a simplified sippet code from the FFT2 application, which is a benchmark application presented in section 8. The landing instruction is assumed to appear later in the code and is not shown here. c. The optimized version of the code after applying the reverse RACE optimization. d. A partial RTDG associated with the given code. e. An example chain formed in reverse order.

in reverse for all the elements of array N, the mappings between the source and the destination arrays are revealed.

In the code transformation module, the same principle described in 6.4 applies, only this time the destination array replaces the source array in the initiation instruction (the instruction in which the source array is last changed in a complex assignment). Figure 10.c demonstrates the optimized code after applying the reverse RACE optimization.

Please note that if the forward RACE optimization is applied first on the code, the reverse RACE optimization will only find chains with the depth of one as the intermediate copies are captured in the forward RACE. Figure 10.a depicts the general case in which the reverse RACE optimization is applicable. In this scenario, the source array is generated within a loop, and then copied over to a larger array. This process continues until the larger array receives the generated data in the right order which will be used later in the code.

a. Main

```
...
    \* C[j] = f(B[k + const]); --> the old instruction *\
    C[j] = f(A[mapping_addr(k + const)]);\* the new auto generated instruction *\
...
```

b. Matrix Multiplier

```
1:    int mapping_addr( int in_index) {
2:        int out_index;
3:        int j;
4:        out_index = in_index & 1;
5:        for (j = 1; j < 7; j++) if (in_index & (1 << j))
              out_index = (out_index + (1 << (7 - j)));
6:        return out_index
7:    }
```

c. FFT

```
1:    int mapping_addr( int in_index) {
2:        return ((in_index % 2 == 0) ? (10*(in_index/200) + (in_index % 20)/2)
           : (100 + (in_index % 200)/20 + 10 * (in_index % 20)/2));
3:    }
```

Fig. 11. a. An example of the optimized code when a mapping function is used. b. The manually implemented mapping function for the Matrix Multiplier application. c. The manually implemented mapping function for the FFT2 application.

7.2. Mapping Patterns

Inserting the mapping array ($indA$) in the code as it is shown in the example codes in figure 9 is not always necessary. There might be some mappings with known patterns between the indices of the source and the destination arrays that can be reflected on the landing instruction without using the mapping array. Some of the simple possibilities would be if the source array is copied onto the destination array with the exact same indexing order, or if only the odd or even indices are copied. For example in figure 7.b, the arrays M and O have exactly the same indexing order, thus the reconstruction of the landing instruction ($R3\_y$) is simply done by replacing array O with array M.

The mapping arrays will also cause the overhead of increasing the code size of the application especially when the associated source and destination arrays are large. If the mapping between the source and destination arrays can be formulated in a function, we could use this function to regenerate the proper indices required to reconstruct the landing instruction. Assume $mapping\_addr\ (int\ \ addr)$ is a mapping function which accepts an index of the destination array B and calculates its mapping to the source array A. In this case, the landing instruction $f(B[k + const])$ can be reconstructed as it is shown in the main code given in figure 11.a.

In general, finding a pattern between two sets of numbers that are regenerated inside one or a series of nested for-loops is a complex problem and is out of the scope of this paper. However, to show the level of difficulty arising from this problem when dealing with real life applications, we manually created the mapping functions for two of the benchmark applications used in our evaluation in section 8. The mapping functions for “Matrix Multiplier” and “FFT” applications are shown in figures 11.b and 11.c.

In section 8 we further discuss the potential performance gain/loss of applying this approach using the manually calculated functions given in figures 11 regardless of the origin of these functions (manually or automatically obtained). This discussion

Table I. **The specifications of the Nios II processors**

	name	branch prediction	HW mult/dev	instruction cache	data cache
Proc. I	Nios II/e	-	-	-	-
Proc. II	Nios II/f	X	X	16KB	-
Proc. III	Nios II/f	X	X	16KB	16KB

will determine if future work on the automatic realization of the mapping functions is justified.

8. EXPERIMENTAL EVALUATION

In this section, we present our experimental results to demonstrate the effectiveness of RACE in further optimizing the auto-generated executable codes outputted from MIT StreamIt compiler [Gordon et al. 2002]. Although we use StreamIt as an example of existing academic software synthesis tools for evaluation purposes, our proposed technique is not conceptually specific to StreamIt implementation.

8.1. Setup

The benchmark applications presented in this section are implemented in StreamIt language [Gordon et al. 2002] which conforms to the SDF semantics, by modeling an application as a graph of interconnected but independent "filters" with statically-defined input and output rates. The StreamIt compiler translates stream programs to C, which can be passed to any standard C compiler to generate executable binaries. In this paper, before the generated C codes are sent to a standard compiler, they are processed once again in the RACE algorithm for further optimizations.

RACE is a source-to-source optimization algorithm which transforms the given generated C code into the optimized version in the same native programming language (C code).This version of RACE only supports the basic syntax of the C programming language such as basic data types, arrays, loops, variable assignments, prints, etc. This decision was made primarily because the more complex syntax of the language do not appear in automatically generated codes, such as in the StreamIt benchmark applications we use for evaluation.

We report the results from Nios II processor running the original auto-generated benchmark codes and the ones after RACE optimization. Nios II is a 32-bit RISC architecture designed specifically for the Altera family of FPGAs [Altera 2015].

The original and the optimized codes were also compiled and executed on a Unix machine to ensure that functional correctness is preserved after our transformation.

We create six different hardware settings by generating three Nios II processors shown in table I, each of which is accompanied by a separate Floating Point Unit (FPU) hardware that is enabled in only half of the settings. These settings are selected to comply with commonly used general purpose embedded processors.

Figure 12 depicts the flow of execution and the tool chain used in our experiments on the benchmark applications from programming in StreamIt language to running the final executable binary codes on the PPGA platform. RACE is our only contribution to this flow.

8.1.1. Benchmark Applications. To evaluate the proposed technique we selected four different streaming kernels as our benchmarks. They include matrix multiplication, the fast Fourier transform (FFT), time delay estimation (TDE), and a ten-stage lattice filter. These kernels frequently appear in many higher-level applications that are used in portable and handheld embedded systems. Table II shows the benchmark applications along with some of the specifications of the original C code and executable binaries outputted form StreamIt compiler.

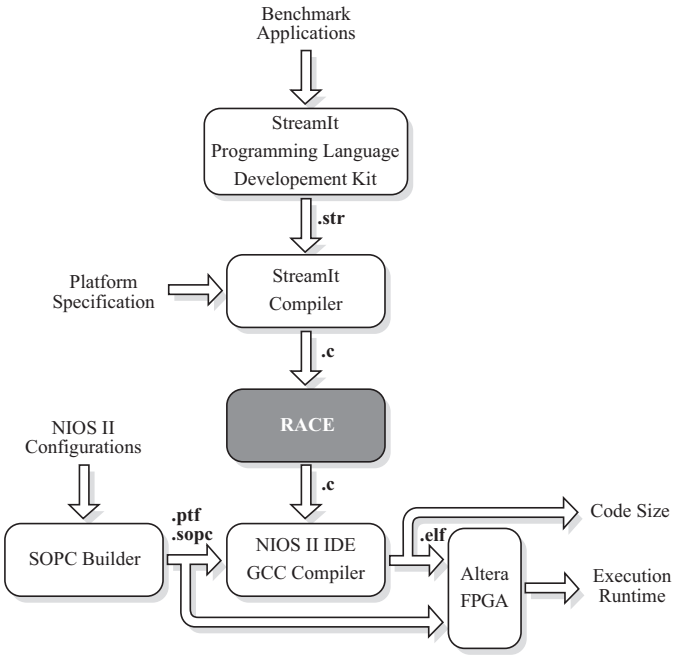


Fig. 12. Experimental setup and tool flow.

8.2. Results

Table II also reports the results on the benchmark applications after the RACE optimization. The runtime numbers given in this table are reported from running the original codes and the codes after the RACE optimization is applied on the given hardware settings. The original and after RACE codes are compiled using GCC version 4.5.3 included in the Altera Quarts 14.0 tool set. The reported runtime numbers are for one iteration of the applications averaged from 10 iteration runs.

Three of the applications in the benchmark set show substantial improvements after RACE optimization is applied. The last application (Lattice) does not provide us with any significant improvement. In fact, it is worth mentioning that horizontally paralleled applications in which the data is copied and spread among multiple actors are better suited for RACE optimization in contrast to vertically paralleled applications where a series of actors work in form of a pipeline.

In the bottom part of the table II we also report some of the parameters from the RACE algorithm in order to compare different factors which contribute to the effectiveness of RACE optimization. For example, although in Matrix Multiplier there is only one instance of a source-destination pair with the maximum chain depth of three, a large number of chain assignments are involved in this instance. It is because the source array is copied multiple times into the destination array using the intermediate assignments. Therefore, a large number of assignments are eliminated in the elimination process. Figure 13 depicts a sample chain for one of the elements in the source array for Matrix Multiplier application along with a sample chain for FFT2 application.

Figure 14.a demonstrates the speed-up made by RACE on the runtime of running the application on the given hardware settings over the original benchmark codes for the benchmark applications. As shown in this figure, Matrix Multiplier achieves maximum of 80% improvement on runtime after applying RACE optimization.

Table II. **The specifications of the benchmark applications before and after RACE optimization**

			Matrix Multiplier		FFT2		TDE_PP		Lattice	
			original	after RACE	original	after RACE	original	after RACE	original	after RACE
total inst. order			17063	8863	10924	9644	1033216	892096	6418	6414
total num. of array elements			2700	200	3828	1788	85112	61848	2044	2042
code size (kB)	without FPU	proc. I	679	689	712	709	833	829	686	686
		proc. II	676	686	709	706	830	826	682	682
		proc. III	676	686	710	707	830	826	683	683
	with FPU	proc. I	678	688	708	705	827	823	683	683
		proc. II	675	685	705	702	824	820	680	680
		proc. III	676	685	706	703	825	821	681	681
run time cycles (k)	without FPU	proc. I	22451	16667	14423	13531	3411060	3232712	8760	8757
		proc. II	3143	1438	2448	2198	298999	263913	1286	1286
		proc. III	878	512	689	642	122049	107569	377	371
	with FPU	proc. I	7449	1662	3540	2692	643871	463251	1681	1681
		proc. II	2158	434	1065	812	117786	85341	504	504
		proc. III	503	132	254	207	44697	33814	123	123
maximum chain depth			3		5		9		1	
num. of src.-dst. pairs			1		2		4		1	
largest src. array			200		256		6480		2	
largest dst. array			2000		128		256		2	
RACE runtime (sec)			6		4		193		2	

The effectiveness of RACE is expressed differently in different hardware settings. Figure 14.a shows a major increase in the speed-up made by RACE between the processors when the FPU accelerator is enabled as opposed to when it is disabled. It is because in the absence of the FPU accelerator, the floating point computations are done in software, hence the ratio of memory operations and non-memory operations is changed. Since the RACE algorithm decreases the number of memory operations by eliminating redundant memory accesses, less improvement is observed on the systems without a FPU.

The same effect is noted when cache memory is introduced to the system. The difference between processor II and processor III is that the former only has instruction cache and the latter has both instruction and data caches. In the presence of data cache memory operations impose less delay, and consequently RACE becomes less effective compared to the system without data cache.

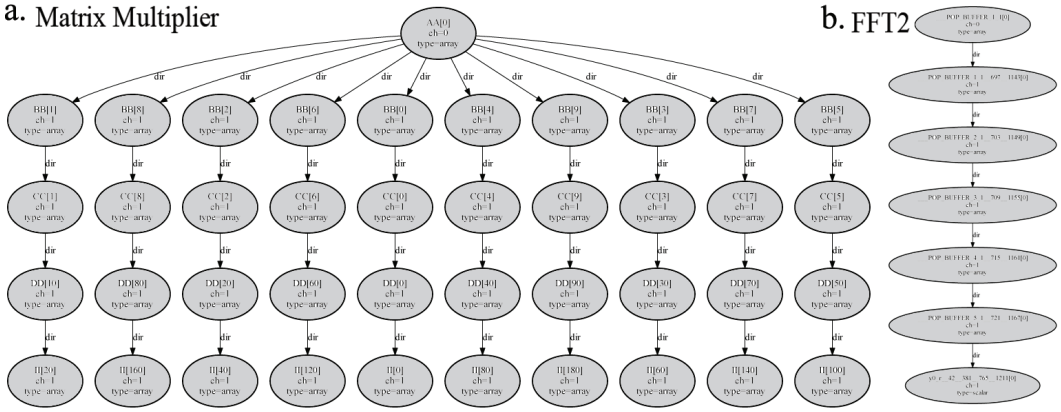


Fig. 13. Sample chains for two of the applications.

Figure 14.b demonstrates the effect of the RACE optimization on the size of the final compiled code. Introducing mapping arrays to the code (discussed in section 6.4) increases the code size. On the other hand, eliminating the irrelevant array assignments results in code size reduction. Therefore, the combination of these additions and eliminations determines the final code size of the application. For the Matrix Multiplier application, the RACE optimization results in an increase in the size of the code memory. The RACE optimization has a negligible impact on the code size of the other two applications.

8.3. Results on the Extensions

In Section 7 tow more optimizations in addition to the main RACE algorithm are explored. In this part of this section we show the impact of these additional optimizations on the same benchmark applications.

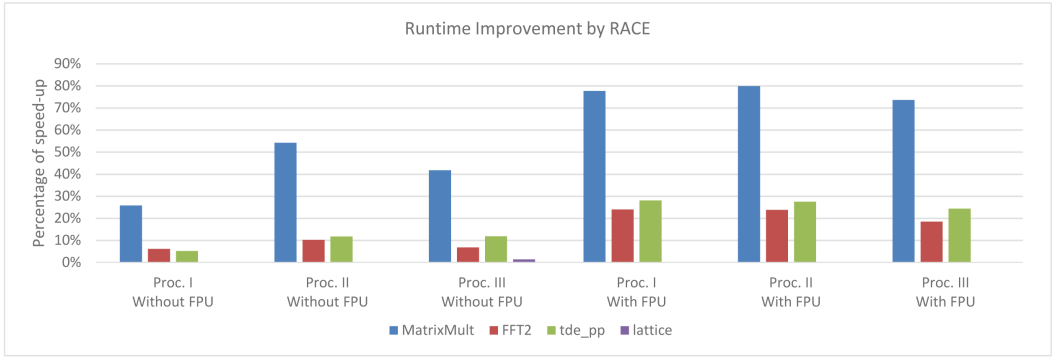
8.3.1. Results on Reverse Application. The opportunity to apply the reverse application optimization (described in section 7.1) appears only in FFT2 and TDE_PP applications. Table III shows the results of applying this optimization in addition to the RACE main approach (forward) for the two applications on the given hardware settings. The results for the RACE main approach without the additional optimization are also given for comparison.

Figure 15 demonstrates the improvements made by the reverse application optimization compared to the original codes. For comparison, the improvements made by the RACE main approach are also depicted in the figure. The impact of reverse application on the code size is negligible, and thus, not shown in the results.

8.3.2. Results on Mapping Patterns. As discussed in section 7.2, it is possible to formulate the mapping between the source and destination arrays in a function. Although automatic realization of such functions is a hard problem on its own, we show the effect of this formulation on the runtime and the code size of the applications by manually creating the mapping functions for two of the benchmark applications used in this section. The mapping functions for the Matrix Multiplier and FFT2 applications are shown in figures 11.b and 11.c, respectively. In order to reduce the overhead of function calls in the code, we also implement the mapping functions as inline macros.

Table IV shows the results of implementing the mapping formulations in the form of both functions and macros for the Matrix Multiplier and FFT2 applications on the given hardware settings.

a.



b.

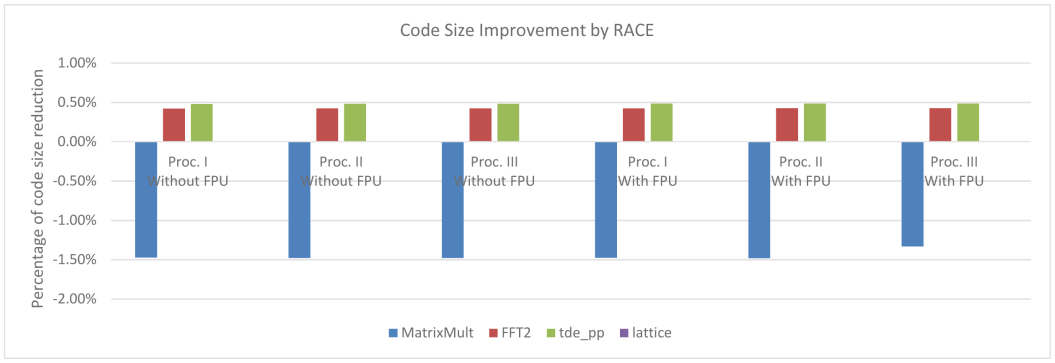


Fig. 14. The percentage of improvements made by RACE over the original codes (before RACE) in both runtime and code size reduction.

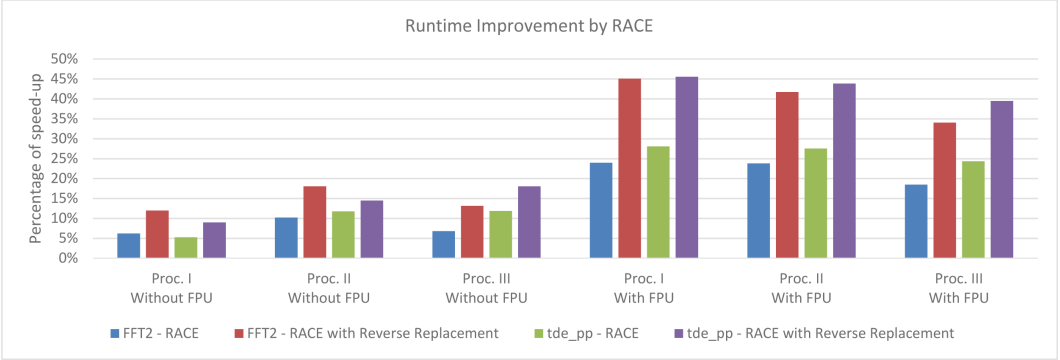


Fig. 15. The percentage of improvements made by reverse replacement optimization over the original codes (before RACE) in runtime reduction. The same results on the RACE main approach is also given for comparison.

Table III. **Reverse Application**: The results of adapting Reverse Application (RA) in RACE for the FFT2 and TDE_PP applications. The results for the RACE main approach are also given for comparison.

			FFT2		TDE_PP	
			RACE/main	RACE/RA	RACE/main	RACE/RA
run time cycles (k)	without FPU	proc. I	13531	12694	3232712	3105552
		proc. II	2198	2006	263913	255781
		proc. III	642	598	107569	100047
	with FPU	proc. I	2692	1945	463251	350584
		proc. II	812	621	85341	66151
		proc. III	207	167	33814	27062

Table IV. **Mapping Formulations**: The results of adapting mapping formulations in RACE in both function and macro implementations for the Matrix Multiplier and FFT2 applications. The mapping formulations are manually generated and are shown in figure 11.

			Matrix Multiplier		FFT2	
			RACE/function	RACE/macro	RACE/function	RACE/macro
code size (kB)	without FPU	proc. I	689	692	708	709
		proc. II	686	689	705	706
		proc. III	686	689	706	706
	with FPU	proc. I	688	691	704	705
		proc. II	685	688	701	702
		proc. III	685	688	702	702
run time cycles (k)	without FPU	proc. I	24315	23794	13452	13524
		proc. II	2321	2187	2216	2211
		proc. III	1072	1032	689	644
	with FPU	proc. I	9751	9177	2700	2639
		proc. II	1314	1190	831	821
		proc. III	691	648	212	210

Because the mapping between the source and the destination arrays is usually the result of complex reordering, duplicating, or dropping the input data in streaming applications, mapping formulations tend to be complex as well. Consequently, employing mapping formulations predominantly increases the amount of computations in the code. On the other hand, removing the mapping arrays from the code can possibly reduce the size of the final compiled code if the inserted functions are smaller than the required mapping arrays in code size.

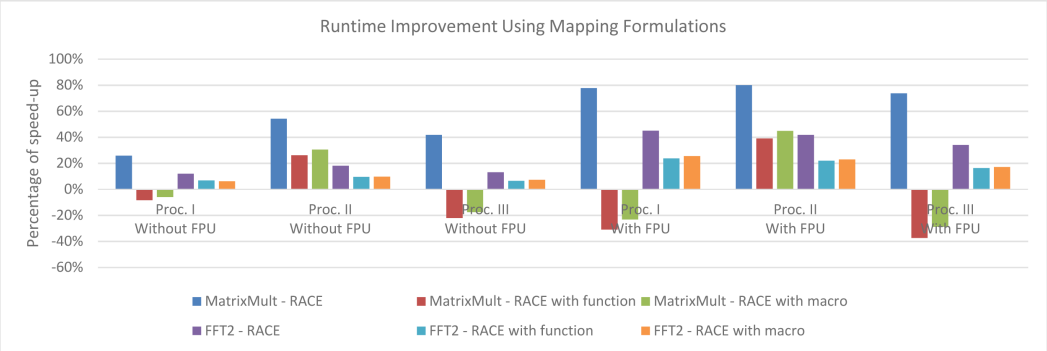
Figure 16.a illustrates the effect of employing mapping formulations instead of mapping arrays on the runtime of the applications in both function and macro implementations. Figure 16.b demonstrate the same effect on the code size of the applications.

As it is depicted in figures 16.a and 16.b, employing mapping formulations reduces or in some cases reverses the effectiveness of the RACE optimization with almost no improvement gained on the size of the final compiled code.

9. CONCLUSIONS

In this paper, we study the automatic generated codes for streaming applications outputted from software synthesis tools. We show that the inherited properties in the code

a.



b.

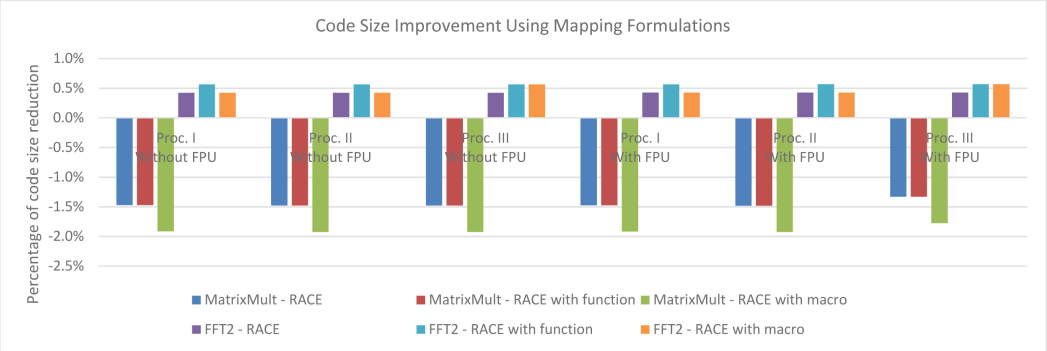


Fig. 16. The effect of employing mapping formulations instead of mapping arrays on the runtime and code size of the applications in both function and macro implementations. The original RACE implementation (using mapping arrays) is also given for comparison.

from the abstraction layer allow us to statically analyze the code for further optimizations. We introduce a technique called Redundant Array Copy Elimination (RACE) that potentially minimizes the number of memory instructions in the final code. Experimental results show up to 80% improvement in the runtime of the optimized code in our benchmark application set.

REFERENCES

Altera. 2015. Nios II Processor: The World's Most Versatile Embedded Processor. (2015). available online at https://www.altera.com/products/processors/overview.tablet.html.

Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer.

Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler Techniques for Code Compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 378–415. DOI:http://dx.doi.org/10.1145/349214.349233

J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. 2003. Taming Heterogeneity—the Ptolemy Approach. *Proc. IEEE* 91, 2 (January 2003).

Mohammad H Foroozannejad, Matin Hashemi, Alireza Mahini, Bevan M Baas, and Soheil Ghiasi. 2014. Time-Scalable Mapping for Circuit-Switched GALS Chip Multiprocessor Platforms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 33, 5 (2014), 752–762.

Mohammad H. Foroozannejad, Trevor Hodges, Matin Hashemi, and Soheil Ghiasi. 2012. Postscheduling buffer management trade-offs in streaming software synthesis. *ACM Trans. Des. Autom. Electron. Syst.* 17, 3 (July 2012), 1 – 31.

Marc Geilen and Twan Basten. 2004. Reactive process networks. In *International Conference on Embedded Software*. 137–146.

M. I. Gordon, W. Thies, and S. Amarasinghe. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 151–162.

Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman P. Amarasinghe. 2002. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 291–303.

Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. 1991. Generating efficient code from data-flow programs. In *Programming Language Implementation and Logic Programming*, Jan Maluszyski and Martin Wirsing (Eds.). Lecture Notes in Computer Science, Vol. 528. Springer Berlin Heidelberg, 207–218. DOI:http://dx.doi.org/10.1007/3-540-54444-5_100

Matin Hashemi, Mohammad H. Foroozannejad, and Soheil Ghiasi. 2013. Throughput-memory Footprint Trade-off in Synthesis of Streaming Software on Embedded Multiprocessors. *ACM Trans. Embed. Comput. Syst.* 13, 3, Article 46 (Dec. 2013), 26 pages. DOI:http://dx.doi.org/10.1145/2539036.2539042

Matin Hashemi, Mohammad H. Foroozannejad, Soheil Ghiasi, and Christoph Etzel. 2012. FORMLESS: Scalable Utilization of Embedded Manycores in Streaming Applications. *SIGPLAN Not.* 47, 5 (June 2012), 71–78. DOI:http://dx.doi.org/10.1145/2345141.2248429

Matin Hashemi and Soheil Ghiasi. 2010. Versatile Task Assignment for Heterogeneous Soft Dual-Processor Platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2010).

M. Karczmarek, W. Thies, and S. Amarasinghe. 2003. Phased scheduling of stream programs. In *Conference on Languages, Compilers and Tools for Embedded Systems*. 103–112.

Edward A. Lee and David G. Messerschmitt. 1987. Synchronous Data Flow. *Proc. IEEE* 75, 9 (September 1987), 1235–1245.

Praveen K. Murthy and Shuvra S. Bhattacharyya. 2001. Shared Buffer Implementations of Signal Processing Systems Using Lifetime Analysis Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 2 (2001).

Praveen K. Murthy and Shuvra S. Bhattacharyya. 2004. Buffer merging__ powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Trans. Des. Autom. Electron. Syst.* 9 (April 2004), 212–237. Issue 2.

Praveen K. Murthy, Shuvra S. Bhattacharyya, and Edward A. Lee. 1997. Joint Minimization of Code and Data for Synchronous Dataflow Programs. *Formal Methods in System Design* 11 (1997), 41–70. Issue 1.

NI. 2015. National Instruments' LabVIEW. (2015). available online at http://www.ni.com/labview/.

Hyunok Oh, Nikil D. Dutt, and Soonhoi Ha. 2005. Single appearance schedule with dynamic loop count for minimum data buffer from synchronous dataflow graphs. In *CASES*. 157–165.

Mathworks Simulink. 2015. Simulation and Model-Based Design. (2015). available online at http://www.mathworks.com/products/simulink/.

S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. 2007. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Design Automation Conference*. 777–782.

Radoslaw Szymanek and Krzysztof Krzysztof. 2003a. Partial Task Assignment of Task Graphs Under Heterogeneous Resource Constraints. In *Proceedings of the 40th Annual Design Automation Conference (DAC '03)*. ACM, New York, NY, USA, 244–249. DOI:http://dx.doi.org/10.1145/775832.775895

Radoslaw Szymanek and Krzysztof Krzysztof. 2003b. Partial Task Assignment of Task Graphs Under Heterogeneous Resource Constraints. In *Proceedings of the 40th Annual Design Automation Conference (DAC '03)*. ACM, New York, NY, USA, 244–249. DOI:http://dx.doi.org/10.1145/775832.775895

W. Thies, J. Lin, and S. Amarasinghe. 2003. Partitioning a structured stream graph using dynamic programming. In *5th Workshop Media Streaming Processors*.

Suleyman Tosun. 2011. New heuristic algorithms for energy aware application mapping and routing on mesh-based NoCs. *Journal of Systems Architecture* 57, 1 (2011), 69 – 78.