V. Kůrková et al. (Eds.): ITAT 2014 with selected papers from Znalosti 2014, CEUR Workshop Proceedings Vol. 1214, pp. 13–20
http://ceur-ws.org/Vol-1214, Series ISSN 1613-0073, © 2014 M. Brabec, D. Bednárek, P. Malý

Transformation of Pipeline Stage Algorithms to Event-Driven Code

Michal Brabec, David Bednárek, Petr Malý

Department of Software Engineering
Faculty of Mathematics and Physics, Charles University Prague
{brabec,bednarek, maly}@ksi.mff.cuni.cz

Abstract: Many software systems publish their interface using event-driven programming, where the application programmers create routines, called as responses to the events. One of such systems is the Bobox parallel framework, where the elements of a parallel pipeline react to events signalizing the arrival of input data. However, many algorithms are more efficiently described using classical serial approach, where the application code calls system routines to wait for required events. In this paper, we present a method of code transformation producing event-driven code from serial code and we concentrate mostly on its first part, responsible for the management of variables. Besides a friendlier programming environment, the transformation often improves the structure of the code with respect to compiler optimizations.

1 Introduction

Many frameworks and libraries present their interface as event-driven programming [1], where the user code reacts to a set of events produced by the framework at runtime.

Implementing an application in this fashion is not as difficult in areas like user interface, where most complex operations are performed by an application kernel and the interface just presents the results.

This approach is not very practical for complex algorithms, where the event-driven design can significantly increase complexity [2], making the implementation difficult to maintain and test. The difference in complexity is illustrated by a simple merge algorithm, the basic implementation is in Listing 1 and the event-driven version is in Listing 2.

The merge algorithm takes two input streams and merges them into a single output stream. The input streams provide the data one at a time, the method get() provides current value and the method next() obtains next value. The results are passed on to the output stream using the put() method.

1.1 Event-Driven Implementation

The merge algorithm, presented in Listing 1 merges two ordered sets and then it passes the results to the output stream. The implementation is simple to understand and test, but there is a hidden event management in the code. The method next() retrieves new data, and, if there is nothing in the input buffer, it waits. Here the application is waiting for an event (the arrival of input data).

```
void merge_box()
{
  input_stream   left;
  input_stream   right;
  output_stream  output;

  while (left.has_data() &&
         right.has_data())
  {
    int L = left.get();
    int R = right.get();

    if (L < R)
    { // possible waiting for data
      left.next();
      output.put(L)
    }
    else
    { // possible waiting for data
      right.next();
      output.put(R);
    }
  }
}
```

Listing 1: Basic join implementation for Bobox

An event-driven implementation is usually less comprehensive and it can be difficult to maintain or verify [2], at least in the case of complex algorithms. The Listing 2 shows an event-driven implementation of merge algorithm. It is more complex than the simple version presented in Listing 1 and it contains significant code duplication. The implementation is further complicated by the fact that we must check if the current event is the event we were waiting for.

The event driven implementation uses the method notify_next() instead of next(). This method only requests the framework to call the task again, once new data is available. The task does not wait for the data.

1.2 Method Inversion

We propose a special technique called method inversion that removes waiting for events. Instead the code reacts to an event and it ends when the event is processed. The

```
void merge_box(event_t event)
{
  // choose proper reaction based
  // on the event - available data
  switch(event)
  {
  case DATA_LEFT:
    if(right.has_data())
    {
      int L = left.get();
      int R = right.get();
      if (L < R)
      { // no waiting for events
        left.notify_next();
        output.put(L);
      }
      else
      { // no waiting for events
        right.notify_next();
        output.put(R);
      }
    }
    return;
  case DATA_RIGHT:
    if(left.has_data())
    {
      int L = left.get();
      int R = right.get();
      if (L < R)
      { // no waiting for events
        left.notify_next();
        output.put(L);
      }
      else
      { // no waiting for events
        right.notify_next();
        output.put(R);
      }
    }
    return;
  }
}
```

Listing 2: Event-driven version of merge

transformation produces an algorithm with a behavior similar to the code in Listing 2. In this paper we concentrate mostly on the first part of the optimization responsible for data management, also called movement of variables. In the first part, we must identify currently used variables, preserve their value and later restore them, so the computation can be safely interrupted and resumed.

The rest of the paper is organized as follows: Section 2 describes our motivations for this work along with the main problems we try to solve, we also present an overview of our development environment here. In section 3 we put method inversion in the context of other works related to Bobox and we discuss similar works concentrating on parallel pipelines. Section 4 contains the algorithm that implements method inversion. Section 5 concentrates on the effects the transformation has on the entire pipeline and it concludes the results of our research; we also discuss possible ways to improve presented optimizations and similar future work.

2 Motivation

2.1 Parallel Pipelines

Parallel pipelines represent a great way to utilize the full capacity of contemporary computers [3], because they are highly scalable and they offer a suitable framework for many applications, including database operations [4].

A parallel pipeline is a set of independent tasks connected by data streams, where the tasks can communicate only via passing messages and data through the streams. The streams pass the output of one task to the next task which uses it as input. Tasks can be run in parallel, with each task processing different data [5]

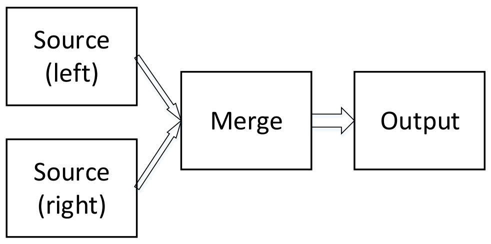

Figure 1: Simple pipeline that merges two ordered sets

Figure 1 shows a simple pipeline for merging data. Source tasks produce input, the Merge task merges its two input streams and the Output task stores the results. The arrows represent data streams that transport data between tasks.

Parallel pipelines can be treated as event-driven systems, where the main event is the arrival of input data. Each task processes input data and then waits for new data, basically reacting to new data as an event.

Calling a method that waits for an event (new data) is referred to as a blocking call in this paper, because the task is blocked until there is new data available.

One of the most important problems of pipelines is the fact that different tasks may process their data at a different rate, which may lead to a situation where parts of the pipeline are engaged in complex computations while others are waiting for input [6]. It is necessary that the waiting tasks stop their computations and release any system resources that can be used for other tasks, but doing that by hand can be difficult (Listing 2).

We propose a way to automatically produce event-driven code from serial code without any additional information, like code annotations, attributes or special comments. This allows users to develop simple tasks that can be easily maintained. The compiler removes blocking calls and transforms the code for the target environment. We designed this transformation for the Bobox framework, but it can be used for other environments.

2.2 Bobox

In this paper we focus on parallel pipelines developed for the Bobox framework. The Bobox framework provides a runtime environment used to execute a general (even non-linear) pipeline in parallel [7]. The actual execution of the pipeline is managed by the framework, which executes tasks (called boxes) in parallel according to the pipeline structure defined by the users.

There are two ways to use Bobox framework, users can either define the pipeline by hand or they can use an interface that produces the pipeline from another code, like a SPARQL query [4].

We developed the method inversion to expose all the blocking calls (waiting for data) so they can be removed. This way we can optimize separate tasks and improve the efficiency of the entire pipeline.

2.3 Development Environment

We designed a special development environment to apply presented optimizations automatically. The environment should simplify the implementation of the Bobox boxes and we intend to allow programmers to design entire Bobox pipelines in the environment [8]. The structure of the environment is in Figure 2, where the bold components are part of the development environment.

The boxes are implemented in the C# programming language and they are compiled to a CIL assembly. The optimized CIL code is then transformed to C++ source code for Bobox framework.

2.4 Goals

We have designed the method inversion to optimize tasks in parallel pipelines and to simplify their development. This way we can improve the pipeline efficiency while making the development simpler at the same time.

Second goal is to use the method inversion as a general transformation that can simplify the development of many applications.

Last goal is to improve the structure of the Bobox tasks that might allow the compiler to apply more advanced optimizations.

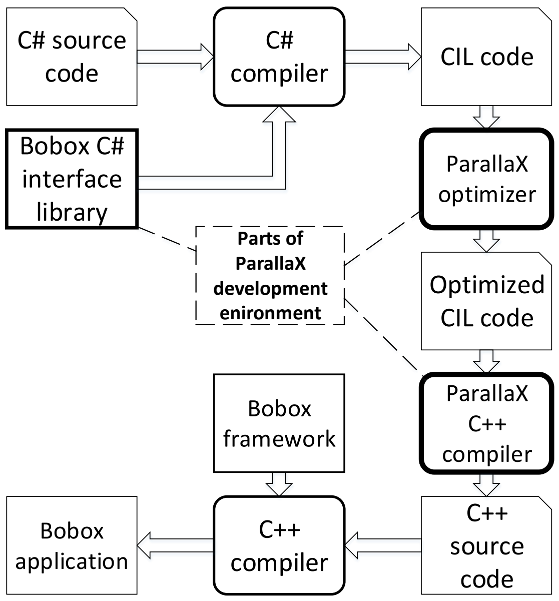

Figure 2: Development environment for Bobox framework

3 Related Work

3.1 Event-Driven Programming

There are very few works concentrating on the generation of event-driven programs from serial code. There are works that discuss event-driven programming, but they consider mostly applications already implemented as event based [9], [2]. There are few systems that try to simplify the development of event-driven programs [10], but they do not fully hide the events from users, instead they require special annotations or comments.

3.2 Parallel Pipeline Optimizations

Optimizations for parallel pipelines concentrate mostly on task distribution and scheduling according to available CPUs [11] and other system resources [12], because load balancing has a major impact on the pipeline efficiency. The solution used in Bobox tasks are distributed among processors at the beginning of the pipeline execution [7] and the system migrates them very carefully to minimize communication overhead, which is especially useful in distributed environments [13].

Most parallel pipeline environments leave task optimization to the users and that is why there are not many task-specific optimizations. There are development environments that generate tasks or even pipeline structure from user code [14] and these systems can optimize the entire pipeline because they can influence its structure. Our development environment is designed for a similar purpose [7], but the current implementation concentrates on separate boxes.

3.3 Bobox

Bobox optimizations concentrate mostly on the structure and execution of the entire pipeline, leaving the task optimization for the users. Most works concentrate on task scheduling. There are general scheduling optimizations [6] as well as special strategies based on dataflow [15] [16] or input structure.

Bobox is used mostly for query processing and there are many works presenting optimizations specific for different query languages [4] [17], but there are no optimizations for the tasks produced by users.

4 Method Inversion

The inversion transformation is used to expose the parts where a pipeline task is waiting for input data to arrive. These so called blocking calls represent a problem, because it means that the task waits for data without doing anything useful thus blocking an execution unit (CPU). This is a problem, even when a task is waiting passively, because then it is necessary to swap it for another task.

The inversion requires three main steps. First, we must back up all the required data so we can safely interrupt and resume execution. Second, blocking calls must be replaced by non-blocking methods that only notify the framework without waiting. Third, it is necessary to restructure the code so it is possible to safely resume the execution.

We present the last two steps in a simplified version, but the first step is the main focus of this paper because it is necessary for any implementation of the inversion and it can be used for other optimizations.

The inversion is presented on a simple code shown in Listing 3, the code applies a function f() on all values in an input stream and it passes the results to an output stream.

```
for(;;)
{
  if (iCurrent>=iSize)
  {
    input.fetch_n(iBuf, iSize);
    iCurr = 0;
  }
  oBuf[oCurr++] = f(iBuf[iCurr++]);
  if (oCurr>=oSize)
  {
    output.put_n(oBuf,oSize);
    oCurr = 0;
  }
}
```

Listing 3: Original code before inversion

4.1 Basic and Advanced Inversion

Our ultimate goal is to restructure the code to expose all the blocking calls so we do not have to jump inside a complex control flow to resume computations. Such an algorithm would produce code that can be efficiently parallelized and further optimized.

Results of such transformation are presented in Listing 4, while the original code is in Listing 3. The switch statement manages the restoration of variables and then it resumes computation by jumping into the original code. We use goto to minimize code duplication.

```
void do_something()
{
  switch (event)
  {
  case INPUT_RECEIVED:
    ic = 0; goto loop;
  case OUTPUT_SENT:
    oc = 0; goto loop;
  }
loop:
  k = min(iSize-ic, oSize-oc);
  for (i=0;i<k;++i)
    oBuf[oc+i] = f(iBuf[ic+i]);

  if (ic+i>=iSize)
  {
    oc += k;
    return WAIT_FOR_INPUT;
  } else
  {
    ic += k;
    return WAIT_FOR_OUTPUT;
  }
}
```

Listing 4: Product of the advanced inversion

This transformation is still under development, but we implemented a simpler version. It lacks certain advantages of the advanced version, but it is easier to implement.

The simple version does not move the blocking calls outside of the loop, instead it simply jumps inside any control flow, after restoring all the necessary variables. The result of this inversion algorithm is in Listing 5, where the methods used for input and output are not blocking, they just inform the system to send or receive data when there are resources available.

4.2 Movement of Variables

The goal of our transformation is to release the thread of execution during blocking operations. Releasing the thread causes loss of the *local storage* associated with the

```
switch (state)
{
case WAIT_FOR_INPUT:
  if(event != input) return;
  iCurr = 0; iSize = size; goto iLoop;
case WAIT_FOR_OUTPUT:
  if(event != output) return;
  oCurr = 0; oSize = size; goto oLoop;
}

for(;;)
{
  if (iCurrent>=iSize)
  {
    state = WAIT_FOR_INPUT;
    input.prefetchn(iBuf, iSize);
    return;
    iLoop:
  }
  oBuf[oCurr++] = f(iBuf[iCurr++]);
  if (oCurr>=oSize)
  {
    state = WAIT_FOR_OUTPUT;
    output.sendn(oBuf,oSize);
    return;
    oLoop:
  }
}
```

Listing 5: Product of the simple inversion

thread, namely the registers and stack. Therefore, we need a *backup storage* for the local variables that are live during a blocking operation. The backup storage will usually consist of data members of a dynamically allocated class object.

Moving aggregate variables, i.e. arrays and structures, is expensive or even impossible (due to aliasing); in most cases, the best strategy is storing aggregate variables in the backup storage for their complete lifetime.

Unaliased scalar variables (including dissociated structures) may be moved easily and their movement between the backup storage and the local storage offers the advantage of faster access to the local storage. The faster access to the local storage is primarily given by the ability of compilers to allocate registers for important local-storage objects (i.e. local variables); furthermore, scalar local-storage variables allow the compilers to apply many advanced optimization steps while similar optimization for backup storage members would require difficult inter-procedural analysis of code.

The movement of unaliased scalar variables between the backup storage and the local storage must be carefully statically planned in order to achieve correctness and efficiency. For the planning of movement, we developed the *static movement planner* algorithm described in the following paragraphs. The algorithm inserts the necessary movement operations at appropriate places of the code; in addition, it may minimize the number of movements at the cost of duplication of code.

Let $G_{\texttt{CFG}} = (V_{\texttt{CFG}}, E_{\texttt{CFG}})$ be the control-flow graph of a procedure, where $V_{\texttt{CFG}}$ are inidividual statements and $E_{\texttt{CFG}}$ the transitions between them. Let $I \in V_{\texttt{CFG}}$ denotes the *initial* node (entry point) of the control-flow graph and $F \subseteq V_{\texttt{CFG}}$ denotes all the *final* nodes (return statements). Each statement is either *non-blocking* or *blocking*.

We will assume in the following paragraphs that there is only one type of blocking operation and that there are no parameters (nor return values) to this operation. The set of blocking statements is denoted $B \subseteq V_{\texttt{CFG}}$.

The non-blocking statements include access to variables and memory, calculations, and calls to non-blocking procedures. For simplicity of the description, we will assume that all write operations in a statement are ordered after all read operations in the same statement. The relations $R, W \subseteq V_{\texttt{CFG}} \times X$ where X is the set of variables determine which statements read/write which variables.

Let $x \in X$ be a scalar local variable with no alias. With respect to x, the input code may contain only two operations: read (R_x) and write (W_x). In the output code, we keep the read/write operations working on the local storage and we add two additional operations: store (S_x) for copying the variable from the local storage to the backup storage and load (L_x) for copying in the opposite direction. The load/store operations are always inserted between the original statements, i.e., they are attached to the edges of the original control flow graph. Note that after an S_x or L_x operation, both the local storage and the backup storage contain identical copies of the variable.

In the input code, the blocking statements do not affect the local storage. In the inverted code, blocking statements are broken into a pair of a return statement and an additional entry point. The inversion causes that the local storage is destroyed by every B operation and must be substituted by the backup storage.

The purpose of the static movement planner is inserting L and S operations so that the semantics of the code is not affected by the destruction of the local storage by B operations. The simplest solution would be inserting a S-L pair around every B operation; however, it is not an optimal solution with respect to the number of load/store operations required.

Essentially, the variable x is used to transport a value from a W operation to an R operation, across a path in the control-flow graph which contains no other W operations.

If such a path contains a B operation, the transported value must be saved by inserting S somewhere between the W and the first B, and loaded back to local storage using an L operation inserted after any B followed by R.

4.3 Static Movement Planner

The first phases of the algorithm determine the following set of relations between the control flow graph nodes and the variables:

$$N_{\mathrm{B}}, N_{\mathrm{L}}, \overline{N}_{\mathrm{B}}, \overline{N}_{\mathrm{L}}, P_{\mathrm{B}}, P_{\mathrm{L}}, \overline{P}_{\mathrm{B}}, \overline{P}_{\mathrm{L}} \subseteq V_{\mathtt{CFG}} \times X$$

For a statement $s \in V_{\mathtt{CFG}}$ and a variable $x \in X$, $\langle s,x \rangle \in N_{\mathrm{B}}$ indicates that there is a control-flow path starting at s which *needs* x located in the *backup* storage. Similarly, $\langle s,x \rangle \in N_{\mathrm{L}}$ indicates that there is a path where x is *needed* in the *local* storage.

The next two relations are complementary: $\langle s,x \rangle \in \overline{N}_{\mathrm{B}}$ means that there is a path starting at s where x is *not needed* in the *backup* storage, similarly for $\overline{N}_{\mathrm{L}}$ and the *local* storage.

Furthermore, $\langle s,x \rangle \in P_{\mathrm{B}}$ indicates that there is a control-flow path ending at s which leaves x *present* in the *backup* storage; similarly for $\overline{P}_{\mathrm{L}}$ and the *local* storage. Finally, $\overline{P}_{\mathrm{B}}$ and $\overline{P}_{\mathrm{L}}$ indicate the ends of paths where x is *not present* in the *backup* or *local* storage, respectively.

The sets N_{B}, N_{L}, $\overline{N}_{\mathrm{B}}$, and $\overline{N}_{\mathrm{L}}$ are computed by backward propagation Algorithm 1.

Algorithm 1 Backward propagation algorithm

```
Require: ($V_{\mathtt{CFG}}, E_{\mathtt{CFG}}$) control flow graph; $F, B \subseteq V_{\mathtt{CFG}}$; $X$
    set of variables; $R, W \subseteq V_{\mathtt{CFG}} \times X$
Ensure: $N_{\mathrm{B}}, N_{\mathrm{L}}, \overline{N}_{\mathrm{B}}, \overline{N}_{\mathrm{L}} \subseteq V_{\mathtt{CFG}} \times X$
 1: $N_{\mathrm{B}}, N_{\mathrm{L}}, \overline{N}_{\mathrm{B}}, \overline{N}_{\mathrm{L}} := \emptyset$
 2: $N'_{\mathrm{B}} := \emptyset$
 3: $N'_{\mathrm{L}} := R$
 4: $\overline{N}'_{\mathrm{B}} := W \cup (F \times X)$
 5: $\overline{N}'_{\mathrm{L}} := W \cup ((B \cup F) \times X)$
 6: while $N'_{\mathrm{B}} \cup N'_{\mathrm{L}} \cup \overline{N}'_{\mathrm{B}} \cup \overline{N}'_{\mathrm{L}} \neq \emptyset$ do
 7:     $N_{\mathrm{B}} := N_{\mathrm{B}} \cup N'_{\mathrm{B}}$ ; $N_{\mathrm{L}} := N_{\mathrm{L}} \cup N'_{\mathrm{L}}$
 8:     $\overline{N}_{\mathrm{B}} := \overline{N}_{\mathrm{B}} \cup \overline{N}'_{\mathrm{B}}$ ; $\overline{N}_{\mathrm{L}} := \overline{N}_{\mathrm{L}} \cup \overline{N}'_{\mathrm{L}}$
 9:     $N'_{\mathrm{B}}, N'_{\mathrm{L}}, \overline{N}'_{\mathrm{B}}, \overline{N}'_{\mathrm{L}} := \emptyset$
10:     for $\langle a,b \rangle \in E_{\mathtt{CFG}}$ do
11:         $N'_{\mathrm{B}}[a] := N'_{\mathrm{B}}[a] \cup (N_{\mathrm{B}}[b] \setminus W[a] \setminus N_{\mathrm{B}}[a])$
12:         $\overline{N}'_{\mathrm{L}}[a] := \overline{N}'_{\mathrm{L}}[a] \cup (\overline{N}_{\mathrm{L}}[b] \setminus R[a] \setminus \overline{N}_{\mathrm{L}}[a])$
13:         if $a \in B$ then
14:             $N'_{\mathrm{B}}[a] := N'_{\mathrm{B}}[a] \cup (N_{\mathrm{L}}[b] \setminus W[a] \setminus N_{\mathrm{B}}[a])$
15:         else
16:             $N'_{\mathrm{L}}[a] := N'_{\mathrm{L}}[a] \cup (N_{\mathrm{L}}[b] \setminus W[a] \setminus N_{\mathrm{L}}[a])$
17:             $\overline{N}'_{\mathrm{B}}[a] := \overline{N}'_{\mathrm{B}}[a] \cup (\overline{N}_{\mathrm{B}}[b] \setminus \overline{N}_{\mathrm{B}}[a])$
18:         end if
19:     end for
20: end while
```

The algorithm essentially enlarges the four relations from the initial state until stable. The primed versions of the relations represent the increments added to the relations at lines 7 and 8. The initial state, formed at the lines 2 to 5, reflects the fact that a read operation induces immediate need for local storage while a write operation or a return statement causes that the previous value is no longer needed. In addition, any blocking operation aborts any need for local storage.

Lines 11 to 18 propagate the relations backward across a control-flow edge $\langle a,b \rangle$. Line 11 states that a need for backup propagates backward unless a write is encountered. Line 12 propagates the absence of requirement for local storage unless a read is encountered.

Line 14 corresponds to blocking operations – if a variable is needed in the local storage after a blocking statement, it is needed before the blocking statement in the backup storage, unless the blocking statement writes the variable. Lines 16 and 17 propagate through non-blocking statements – the need for local storage propagates unless a write is encountered, the absence for backup storage propagates unconditionally.

The sets P_{B}, P_{L}, $\overline{P}_{\mathrm{B}}$, and $\overline{P}_{\mathrm{L}}$ are computed by forward propagation Algorithm 2.

Algorithm 2 Forward propagation algorithm

```
Require: ($V_{\mathtt{CFG}}, E_{\mathtt{CFG}}$) control flow graph; $F, B \subseteq V_{\mathtt{CFG}}$; $X$
    set of variables; $R, W \subseteq V_{\mathtt{CFG}} \times X$
Ensure: $P_{\mathrm{B}}, P_{\mathrm{L}}, \overline{P}_{\mathrm{B}}, \overline{P}_{\mathrm{L}} \subseteq V_{\mathtt{CFG}} \times X$
 1: $P_{\mathrm{B}}, P_{\mathrm{L}}, \overline{P}_{\mathrm{B}}, \overline{P}_{\mathrm{L}} := \emptyset$
 2: $P'_{\mathrm{B}} := \emptyset$
 3: $P'_{\mathrm{L}} := W \cup R$
 4: $\overline{P}'_{\mathrm{B}} := W \cup (I \times X)$
 5: $\overline{P}'_{\mathrm{L}} := (B \cup I) \times X$
 6: while $P'_{\mathrm{B}} \cup P'_{\mathrm{L}} \cup \overline{P}'_{\mathrm{B}} \cup \overline{P}'_{\mathrm{L}} \neq \emptyset$ do
 7:     $P_{\mathrm{B}} := P_{\mathrm{B}} \cup P'_{\mathrm{B}}$ ; $P_{\mathrm{L}} := P_{\mathrm{L}} \cup P'_{\mathrm{L}}$
 8:     $\overline{P}_{\mathrm{B}} := \overline{P}_{\mathrm{B}} \cup \overline{P}'_{\mathrm{B}}$ ; $\overline{P}_{\mathrm{L}} := \overline{P}_{\mathrm{L}} \cup \overline{P}'_{\mathrm{L}}$
 9:     $P'_{\mathrm{B}}, P'_{\mathrm{L}}, \overline{P}'_{\mathrm{B}}, \overline{P}'_{\mathrm{L}} := \emptyset$
10:     for $\langle a,b \rangle \in E_{\mathtt{CFG}}$ do
11:         $P'_{\mathrm{B}}[b] := P'_{\mathrm{B}}[b] \cup (P_{\mathrm{B}}[a] \setminus W[b] \setminus P_{\mathrm{B}}[b])$
12:         $\overline{P}'_{\mathrm{L}}[b] := \overline{P}'_{\mathrm{L}}[b] \cup (\overline{P}_{\mathrm{L}}[a] \setminus R[b] \setminus W[b] \setminus \overline{P}_{\mathrm{L}}[b])$
13:         if $b \in B$ then
14:             $P'_{\mathrm{B}}[b] := P'_{\mathrm{B}}[b] \cup (P_{\mathrm{L}}[a] \setminus W[b] \setminus P_{\mathrm{B}}[b])$
15:         else
16:             $P'_{\mathrm{L}}[b] := P'_{\mathrm{L}}[b] \cup (P_{\mathrm{L}}[a] \setminus P_{\mathrm{L}}[b])$
17:             $\overline{P}'_{\mathrm{B}}[b] := \overline{P}'_{\mathrm{B}}[b] \cup (\overline{P}_{\mathrm{B}}[a] \setminus \overline{P}_{\mathrm{B}}[b])$
18:         end if
19:     end for
20: end while
```

The algorithm is analogous to the backward propagation with different propagation rules reflecting the following observations: After any write or read operation, the value is present in the local storage (line 3, 12); after any write, the value of backup storage is obsolete (line 4, 11). The backup storage is initialized by moving from local storage before a blocking statement (line 14, 17); any blocking statement invalidates the local storage (line 5, 16).

The final stage of the static movement planner is the determination of control-flow edges where the variables shall be loaded or stored, i.e. the determination of the relations $L, S \subseteq E_{\mathtt{CFG}} \times X$.

The relations are calculated locally from the relations created in the previous phases using the following rules:

$$\langle\langle a,b\rangle,x\rangle \in S \Leftrightarrow \langle a,x\rangle \in \overline{P}_{\mathrm{B}} \wedge \langle b,x\rangle \in N_{\mathrm{B}} \setminus (\overline{N}_{\mathrm{B}} \setminus P_{\mathrm{B}})$$

$$\langle\langle a,b\rangle,x\rangle \in L \Leftrightarrow \langle a,x\rangle \in \overline{P}_{\mathrm{L}} \wedge \langle b,x\rangle \in N_{\mathrm{L}} \setminus (\overline{N}_{\mathrm{L}} \setminus P_{\mathrm{L}})$$

The two rules are based on the same observation: A copy towards a storage may be placed at an $\langle a,b\rangle$ edge if the value was not present ($\overline{P}$) in the storage after the statement a and it may be required (N) by a path starting at b. Nevertheless, if the value is not needed ($\overline{N}$) at some of the other paths starting at b, copying would be premature and shall be postponed until the point where all paths need the variable. However, if the value may already be present (P) at b (due to a different path ending there), postponing is not possible because the copy operation will disturb the existing value.

The intersection $Z_{\mathrm{B}} = N_{\mathrm{B}} \cap \overline{N}_{\mathrm{B}} \cap P_{\mathrm{B}} \cap \overline{P}_{\mathrm{B}}$ denotes the places where the rules create suboptimal code: In such nodes, there exist outgoing paths that need the validity of the storage as well as outgoing paths where the storage is not needed; at the same time, incoming paths that ensure the presence as well as incoming paths without the presence exist. Consequently, placing a store operation here is both necessary and premature. Similar sub-optimality exist at the intersection $Z_{\mathrm{L}} = N_{\mathrm{L}} \cap \overline{N}_{\mathrm{L}} \cap P_{\mathrm{L}} \cap \overline{P}_{\mathrm{L}}$ for the load operation.

These sub-optimal placements may be avoided by code duplication: For each node-variable pair $\langle b,x\rangle$ in the intersection Z_{B}, the node b shall be split into b_1 and b_2 and the incoming edges arranged so that $\langle b_1,x\rangle \notin P_{\mathrm{B}}$ and $\langle b_2,x\rangle \notin \overline{P}_{\mathrm{B}}$. The definition of P_{B} and $\overline{P}_{\mathrm{B}}$ ensures that this process is plausible because the source nodes for the incoming edges will be split, too. Similar splitting process shall be applied for Z_{L}. When repeated for all variables, the Z_{B} and Z_{L} relations become empty and, consequently, the insertion of the S and L operations becomes optimal. However, it comes at the cost of code expansion, in the worst case exponential with respect to the number of variables.

4.4 Simple Method Inversion

We can perform the actual inversion, once all the necessary variables are identified and saved. The simple inversion algorithm is shown in Listing 6. The steps are explained in following text. Listing 5 shows an example of a code produced by this algorithm.

The blocking methods are internal to the Bobox interface library (provided as part of the development environment), so they are called only inside the library methods and we expose them by integrating (inlining) all the library methods into the inverted method.

Locate blocking calls This step is very simple, because we have a complete list of all potentially blocking methods and this means that we simple check the code and record all the places where the methods are called.

```
void Invert()
{
  // locate all the blocking methods
  LocateBlockingCalls();
  // save a state for each call
  StoreStates();
  // use non-blocking methods instead
  ReplaceBlockingCalls();
  // insert return and label
  InsertReturnResume();
  // state machine for event dispatch
  CreateStateMachine();
  // save and restorelocal variables
  PatchLocalData();
}
```

Listing 6: Original code befor inversion

Store states We must create a separate state for each located blocking call, not just each method, but for each actual call instruction in the code. This way we can resume the computation at the exact location where we left off. We simply create State1 - StateN, where N is the number of located blocking calls. We store the state into a member variable of the box.

Replace blocking calls In this step we must replace the blocking methods for methods that do not wait for data. These methods just notify the system to call the task, once an event occurs. There is such a method for each blocking method and we can assure this, because all the methods are hidden in the Bobox interface library that is part of the development environment. So we simply replace the called blocking method for its safe version.

Insert return and resume Now we must insert constructs that allow us to interrupt the computation and later resume it at the same place. To do this we must place a return statement after the former blocking call and immediately after the return we place a label so we can jump back via a goto statement.

Create state machine We must create switch statement to complete the the state machine mechanics, so we can jump to a correct place, based on the actual state. We create a switch that takes the actual state as a parameter and it jumps on the appropriate label placed in the code in the previous step. There is one thing we must do before the jump, we must check if the received event is the event we are waiting for. If the received event is not the correct one, we simply jump out. We can ignore events, since they inform us about new data in an input buffer and we will process it later.

Patch local data In the last step we use the results of the Variable movement algorithm (Section 4.2) to save and restore necessary variables.

5 Conclusion and Future Work

We have successfully implemented a first version of method inversion for CIL intermediated code. We are able to eliminate blocking calls and we can produce an event-driven code from sequential implementation.

The actual implementation produces a modified CIL code that can be used in .NET implement of Bobox, but the code cannot be used in the main C++ implementation, because the compiler from CIL to C++ is not yet ready. We are not yet able to measure the effect of the method inversion on the efficiency of C++ Bobox pipeline, on the other hand it produces working boxes that we tested in the .NET implementation of Bobox.

We are not yet able to present reliable experimental results, because we can use the optimization only in the .NET implementation of Bobox which is designed mostly for testing and prototype evaluation. The .NET version is not optimized for speed and the method inversion has only a minimal effect on its efficiency.

In the future we will be mostly concentrating on the development of the advanced method inversion that would produce a code with better structure and behavior.

Acknowledgment

This work was supported in part by the grant P103/13/08195S of the Grant Agency of the Czech Republic and the grant 122214 of the Charles University Grant Agency.

References

[1] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter, "Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks," in *Chilean Society of Computer Science, 2007. SCCC'07. XXVI International Conference of the*. IEEE, 2007, pp. 3–12.

[2] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 2006, pp. 29–42.

[3] A. Navarro, R. Asenjo, S. Tabik, and C. Caşcaval, "Load balancing using work-stealing for pipeline parallelism in emerging applications," in *Proceedings of the 23rd international conference on Supercomputing*. ACM, 2009, pp. 517–518.

[4] Z. Falt, D. Bednarek, M. Cermak, and F. Zavoral, "On parallel evaluation of SPARQL queries," in *DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications*. IARIA, 2012, pp. 97–102.

[5] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism," in *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, 2009, pp. 281–290.

[6] Z. Falt and J. Yaghob, "Task scheduling in data stream processing." in *DATESO*, 2011, pp. 85–96.

[7] D. Bednárek, J. Dokulil, J. Yaghob, and F. Zavoral, "The Bobox project parallelization framework and server for data processing," *Charles University in Prague, Technical Report*, vol. 1, p. 2011, 2011.

[8] M. Brabec and D. Bednárek, "Programming parallel pipelines using non-parallel C# code," *CEUR Workshop Proceedings*, vol. 1003, pp. 82–87, 2013. [Online]. Available: http://ceur-ws.org/Vol-1003

[9] A. Ghosal, T. A. Henzinger, C. M. Kirsch, and M. A. Sanvido, "Event-driven programming with logical execution times," in *Hybrid Systems: Computation and Control*. Springer, 2004, pp. 357–371.

[10] J. Fischer, R. Majumdar, and T. Millstein, "Tasks: language support for event-driven programming," in *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, 2007, pp. 134–143.

[11] M. N. Garofalakis and Y. E. Ioannidis, "Parallel query scheduling and optimization with time-and space-shared resources," *SORT*, vol. 1, no. T2, p. T3, 1997.

[12] J. Subhlok and G. Vondran, "Optimal latency-throughput tradeoffs for data parallel pipelines," in *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1996, pp. 62–71.

[13] M. Cermak and F. Zavoral, "Achieving high availability in D-Bobox," in *DBKDA 2014, The Sixth International Conference on Advances in Databases, Knowledge, and Data Applications*. IARIA, 2014, pp. 92–97.

[14] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "Flumejava: easy, efficient data-parallel pipelines," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 363–375.

[15] D. Bednárek, J. Dokulil, J. Yaghob, and F. Zavoral, "Data-flow awareness in parallel data processing," in *Intelligent Distributed Computing VI*, ser. Studies in Computational Intelligence, G. Fortino, C. Badica, M. Malgeri, and R. Unland, Eds. Springer Berlin Heidelberg, 2013, vol. 446, pp. 149–154. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32524-3_19

[16] ——, "Data-flow awareness in parallel data processing," in *Intelligent Distributed Computing VI*, ser. Studies in Computational Intelligence, G. Fortino, C. Badica, M. Malgeri, and R. Unland, Eds. Springer Berlin Heidelberg, 2013, vol. 446, pp. 149–154. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32524-3_19

[17] D. Bednárek, "Output-driven xquery evaluation," in *Intelligent Distributed Computing, Systems and Applications*. Springer, 2008, pp. 55–64.