Systematic Intermediate Sequence Removal for Reduced Memory Accesses

Christophe Poucet[1], Stylianos Mamagkakis[1], David Atienza[2], Francky Catthoor[1]
[1]DDT/IMEC, Leuven, Belgium. {poucetc, mamagka, catthoor}@imec.be [*]
[2]DACYA/UCM, Madrid, Spain. datienza@dacya.ucm.es [†]

Abstract

Modern software applications are growing in complexity and demand very intensive use of data. Therefore, a wide variety of data structures are utilized to facilitate the storage and access to these vast amounts of computed information. Additionally, the need for reliable software design and the development of large applications following the object-oriented paradigm increase the amount of dynamic buffers and redundant accesses to the data stored in these buffers. In this paper, we propose a systematic, design optimization methodology to remove these intermediate dynamic buffers, thereby reducing the memory accesses of the targeted applications without altering the input-output behaviour of the algorithms. The reduction is focused on sequences and is especially relevant for embedded systems, which have limited on-chip communication bandwidth and the energy consumption of the memory subsystem is high, due to the energy consumption associated with each memory access. The effectiveness of the proposed methodology is assessed in a 3D reconstruction multimedia application and shows a significant reduction in memory accesses. In addition, the general trends for memory improvement and the scalability of our approach are supported as well by a parameterized benchmark set.

1 Introduction

Modern software applications for embedded systems have massive data storage and transfer needs. This is particularly true for applications that need to deliver a rich multimedia experience to its final user. This situation creates a bottleneck for the memory subsystem in an embedded system, which has severe memory footprint limitations due to size factors and cost factors. Also, the bandwidth of the interconnect is limited and it can become overloaded by the memory accesses. Most importantly, the accesses to the memory affect the battery life of the device through the energy consumption of the physical memory (i.e. physical memories consume a certain amount of energy per access). Therefore, the memory accesses used by the data of the software application needs to be minimized to meet the specification constraints of designers of embedded systems.

The aforementioned trend is contradicting with the trend for increased reliable design of software applications. Designers devote significant effort to introduce buffers in the source code and encapsulate data in order to make sure that the software is written in a consistent and error-free way. While this approach is valid and recommended for writing the source code of the applications in a reusable way, it introduces unnecessary intermediate variables and memory access redundancy (i.e., writes to and reads from the buffer), which has to be removed after the source code for the whole application has been developed and tested.

Additionally, more buffers are unnecessarily introduced among modules, when code development is split among many designers (for big development projects) and each designer is focusing on a single module. This situation becomes even worse in modern design environments, where tight time-to-market deadlines are the norm in industry. Therefore, software applications typically process data in different stages to allow for the aforementioned modular design. This situation results in the use of temporary dynamic buffers in between as well as within the internal functions of a complete software module.

As a result, in order to design optimized system implementations for consumer embedded devices, solving these intermediate variable and memory access problems introduced by dynamic buffers becomes a major problem because efficient software development starts dominating the overall design and system integration effort. In fact, this is especially true for dynamic buffers used in the object-oriented design paradigm [2] for modular source code development. In pure functional languages like Haskell, these problems are solved by means of *deforestation* [27, 9]. Thus, software designers are not bothered with the implementation of the buffer removal optimizations. However, the situation is different in imperative programming languages, where the designer faces the problem of manually

[*]Francky Catthoor is also Professor at ESAT/K.U.Leuven

[†]This work is partially supported by the Spanish Government Research Grant TIN 2005-5619.

removing the intermediate buffers mainly relying on his programming experience. Therefore, a similar automatic removal approach to functional languages needs to be developed for object-oriented languages like C++, since they are being gradually introduced and massively used in embedded software design.

In this paper, we propose to remove all these intermediate buffers in a systematic, consistent and stepwise way in order to reduce the memory accesses of the data storage, without altering its algorithmic I/O functionality of the C++ embedded software. The biggest challenge, which this paper addresses for the first time, is the systematic removal of data dependent, non-manifest, sequences of data elements and as a result the read/write memory accesses to them (e.g. single linked lists or vectors of the Standard Template Library - STL[23]). Such sequences are characterized by access-methods to get an iterator pointing at the elements, as well as to discern the value of this iterator, move this iterator or add values at the location pointed to by the iterator. A typical example, in pseudocode, can be seen in Code 1.

```
for (...)
  if (condition(data))
    container.append(data);

for (iterator in container)
  consume(*iterator);
```

Code 1: Pseudocode demonstrating typical STL operations

These sequences are taking a bigger part of the memory footprint of recent dynamic applications, for example, 60% of the memory footprint utilized and 13% of the overall memory accesses in a 3D reconstruction application [24]). Although our proposed methodology is currently focused at these instantiations of sequences, it is fundamentally not constrained on its scope and can also be used for other implementations or other types of access patterns. We also introduce a number of tools that help the embedded system designer in the most time consuming tasks, namely, they help to identify these intermediate variables and to calculate the possible trade-offs originated from their removal.

The remainder of this paper is structured in the following way. In Section 2 we present the related work. Then, in Section 3 we illustrate with a motivational example the sort of transformations we are targeting. The methodology is then further exposed in Section 4 and the use of our tools is described in this section as well. Then, our systematic approach is applied to a real multimedia application and to a number of additional set of representative code samples of embedded benchmarks, and the results achieved are shown in Section 5. Finally, in Section 6 we draw our conclusions.

2 Related Work

The current state of the art indicates that a large body of research to find solutions for scalar intermediate variable removal exists [15]. Additionally, De Greef [7] looks at ways to reduce indexed arrays accessed within manifest loop nests to their minimum necessary size. In [20] a similar model, known as the *Polyhedral model*, is used. Both approaches depend on compile-time analysable for-loops with (piece-wise) linear dependencies on the for-loop iterators in the addressing of arrays. More recently, work has been performed on inter-array analysis [26], which focuses on copy-propagation and employs the Polyhedral model. This framework employs sets of equations with linear affine indices and then utilizes matrix manipulations to enable transformations. All the numbers used in the expressions must be fully-manifest, and the methodology can not cope with extended data dependent behaviour. In addition, this work does not look at removing copies that are not pure copies of larger arrays. Finally, work was performed on stream like applications [1], however this work also requires that the number of elements being produced in each phase is linearly related to the number of elements consumed. Therefore, although the work also streams data consumption and production, it can only perform this optimization for linear state space systems, and not for non-manifest runtime dependent data consumption and production.

Within the literature of functional programming languages have been studied under the term *deforestation*[27, 9]. This approach is specific to pure functional programming languages such as Haskell[17]. Their compiler tries to remove intermediate data that is produced between two different functions to lower the data access overhead between functions that are chained together. An advanced form of deforestation is known as *fusion*[25], which targets also higher-order functions. Both deforestation as well as fusion are specifically targeted towards pure functional languages where the basic data-type is the list and side-effects are not present. However, this can not be directly reused within the context of object-oriented software applications due to the fact that the way data types are accessed there is much more complex than the way lists are accessible in functional languages and because *equational reasoning* [22] is not possible for object-oriented languages.

Data management and data optimizations at design time for embedded applications have been extensively studied in the related literature [11]. [18, 3] are good overviews about the vast range of proposed techniques to improve memory footprint and decrease energy consumption in statically allocated data. Finally, from the methodology viewpoint, several approaches have been proposed to tackle this issue at the different levels of abstraction (e.g., memory hierarchies), such as the Data Transfer and Storage Exploration

(DTSE) methodology [6]. However, all these approaches focus only at optimizations of global and stack data, which is allocated at compile-time. In this paper, we propose optimizations of dynamic, heap data, which is allocated at run-time. Obviously, the aforementioned approaches are 100% compatible with our approach and can complement ours in order to optimize data of embedded applications, which is allocated both at compile-time and at run-time.

In summary, so far no research focuses on developing a systematic approach to remove *dynamic* data types from software applications with *non-manifest* and *data-dependent* behaviour. Moreover, research has to be done on systematically eliminating variables that are not pure copies of global or larger dynamic variables, but also injective control-flow or processing relationships. Related, manual experiments were performed on such applications [13] focusing on reducing memory footprints, but no attempt was made yet to formalize the process in a systematic methodology, which also targets memory accesses. The main limitation in the previous methodologies with respect to dynamic data-types is that the link between array-position and iterator is no longer manifest nor piece-wise affine. Even though the actual *iterator* used is linearly related to the for-loop instance in which it executes, due to the semantics of sequences, the link between this iterator and the actual storage inside the sequence is non-manifest and in most cases data-dependent. This is due to the fact that any time an element is *added* or *removed* to a sequence (two operations that an array does not support) the index-to-value mapping of all the other subsequent elements are changed.

3 A Motivational Example

To get a better understanding, we present a motivational example demonstrating a common use of STL in C++ source code. This section first demonstrates a typical use of an intermediate sequence to highlight where the memory accesses are being consumed as well as some features of the problem that our methodology tries to tackle. Then, the proposed solution is demonstrated, and it is shown how the memory access overhead is removed.

3.1 Problem snapshot

If we take the source code in Figure 1, we can see the different features of code that usually makes intermediate sequence removal difficult. The first thing to note about the example is that the intermediate sequence inter is not a pure copy of the input. Additionally, the data contained within it is data dependent, as the condition test depends on data already within the container as well as data to be inserted. Therefore, the mapping of positions of the data elements in inter to their originals in input is not known as the addition happens in a data-dependent way. Notice that this example is a simple example to demonstrate the problem, and due to its simplicity the mapping of the position of elements in inter to the elements that generated them in input is still piecewise linear incremental, which need not be true in the general case. Due to the fact that the mapping from input element position to output element position is not affine, previously mentioned techniques can not be employed. As such, only a runtime technique that takes into account the data-dependent control-flow can be used to remove the intermediate sequence.

```
vector<int> produce(vector<int>& input) {
  vector<int> inter;
  for (vector<int>::iterator i = input.begin();
       i != input.end(); ++i) {
    if (test(*i)) {
      inter.push_back(f(*i));
    }
  }
  return inter;
}
int main () {
  vector<int> inter = produce(input);
  consume(inter); // Pseudocode
}
```

Figure 1: Motivational example showing the dynamic creation of an intermediate variable

Depending on how often the condition test is true, the sequence inter may be as large as input. Assuming that the intermediate buffer is then only used once, which is true for multimedia applications working in phases (as observed in [12]), all the writes to inter as well as the subsequent reads from inter will be completely removed. In this simple motivational example, removing inter should therefore lead to a reduction in memory accesses of up to 66%. Instead of the writes to inter followed by the reads inter, reads to input will happen in the original application nonetheless to build the inter sequence. Additionally, in more complex examples of real software applications, when data sets are being filtered or processed, they are typically copied from one sequence into the next thereby creating these sorts of consumptions. This means that data will be processed in different stages leading to further (possibly transformed) copies of the primary copies as the data is transformed. This eventually leads to a large memory access overhead that our systematic methodology aims to remove. Manually for real applications, it would indeed be a large effort to rewrite all the source code and debug it.

3.2 Proposed optimized solution

We propose to completely remove the production logic inside produce as well as the sequence inter and instead replace it by a custom implementation of a vector that contains the logic to ensure that the correct elements are returned. Doing this will completely reclaim the storage that is currently used by the intermediate vector `inter` and remove the accesses to this vector.

In Figure 2 our final, optimized implementation is shown. As can be seen in the code, the logic that dictates how an iterator should arrive at a new element is delayed to the time when that element is actually needed. The special logic generated by `append_sequence` then computes at runtime where the next element should come from. Because multimedia applications process their data in phases, we can introduce this new data-type which does not need to be modified further.

This transformation changes the code in such a way that data is only lazily evaluated when it is actually being demanded, or read. It is a non-trivial transformation and depends on how the sequence was originally constructed. Thus, we create an *on-demand sequence* (which is similar to the concept of lazy streams[8]) that produces data on demand and this data matches the data that would originally have taken place in memory, thereby consuming only the memory accesses to first produce and then consume.

This example illustrates the optimization of only one loop. The `delayed_vector` takes as one parameter the underlying sequence, which in this case is empty. In the same way that the production loop for the intermediate sequence was empty. In general, however, it is possible to stack these different transformations on top of each other, thereby completely removing the storage of a sequence. By removing the storage of this sequence, which due to their typical size happens to expensive off-chip memory[14], one write to and one read from off-chip memory is avoided for each element in the sequence.

In Figure 3 the gains are demonstrated. In the first example, before the optimizations, for each element in the input, there was one read and then one possible write to the intermediate sequence `inter`. When the data is then consumed from `inter`, then another read is performed for each element. In the transformed code, on the other hand, we create an object that saves the state of the loop where we are at. As such, every time we read from the delayed sequence, we will only cause one read from the original input.

4 Methodology

In this section we explain our methodology to derive the aforementioned transformations. Additionally, the implementation characteristics of the methodology using operators similar to STL are explained in more detail. It consists of four distinctive steps. They are detailed below along with the terminology used in the remainder of this section. The methodology has actually been implemented for STL sequences. However, our methodology is generally applicable

```
APPEND_THUNK(merged, int)
  append_thunk_merged(const vector<int> & input)
   : input_(input) { initialize(); }
  const vector<int> & input_;
  vector<int>::const_iterator i;
  APPEND_BODY(
    for (i = input_.begin();
         i != input_.end(); ++i){
      if (test(*i)) {
        cr_return(1, f(*i));
    }})};

typedef append_sequence<
  empty_sequence<int>,
  append_thunk_merged
> delayed_vector;

delayed_vector produce(vector<int>& input) {
  delayed_vector inter(
      empty_sequence<int>,
      append_thunk_merged(input));
  return inter;
}

int main () {
  delayed_vector inter = produce(input);
  consume(inter); // Pseudocode
}
```

Figure 2: Motivational example after intermediate variable removal

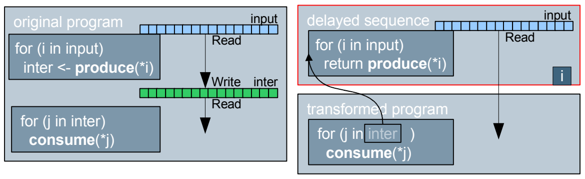

Figure 3: Memory accesses before and after our transformation

to any implementation of sequences (for instance [10]) that works on the basis of *iterators*.

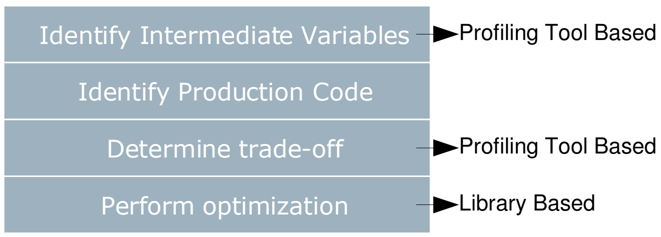

Figure 4: Overview of the methodology for Intermediate Sequence Removal

An overview of the methodology can be seen in Figure 4. In a first step, the intermediate sequences are identified through the use of our profiling tool which is tuned towards sequences. The sequences should be identified as first being created and then being read. This step is further explained in Section 4.1. In a second step, the production sites of the sequences are identified. Here it is important to determine how the sequences are being created, and if this creation is intermingled with any other code that causes observable output behaviour (see Section 4.2). In a third step, the tradeoff between removing a sequence and not removing it is evaluated, as explained in Section 4.3. The easiest way to determine whether an optimization is fruitful is through profiling the application prior and after the transformation with our profiling tool. Finally, the optimization is performed through a transformation of the source code, as detailed in Section 4.4.

4.1 Intermediate Sequence Identification

In the first step of our methodology, it is important to actually identify which sequences are intermediate sequences. Because we rely on the existence of STL-like containers, we have duplicated this interface in our existing profiling tool to obtain profiling not at the memory access level but at the semantic operation level (append, remove, ...). The profiling sequence mirrors a `vector` implementation of STL.

One key characteristic of intermediate sequences is that they are typically first produced and then consumed. As such, first there will be a set of operations that mutate the sequence and at some point after the completion of these operations there will be a set of operations that access the sequence through iterators. It is important that these two phases are distinct and do not overlap, otherwise the transformation might not work. Currently, this transformation step of the methodology supports only sequences that are used through incremental consumption of data. Typically, this will be most if not all the intermediate sequences due to standard practices regarding STL usage[16].

A specific pattern that was observed in the application used in Section 5 was that some intermediate sequences are defined at a top scope and then created and consumed in each loop-iteration, before being cleared at the end of the loop instance. This pattern can easily be rewritten to a form that can be transformed by moving the definition of the intermediate sequence to the inside of the loop and removing the `clear` call. This could even be automated by performing a lifetime analysis and then through pattern matching and rewrite rules. The reason this is not done a priori is that, from the point of view of STL, creating and destroying a sequence is not as efficient as the allocated space can be reused for the next loop-iteration, and as such this pattern can be expected in source codes that need to generate and then consume intermediate data of the same type several times in a loop. Other transformations that reduce the lifetime of intermediate sequences to their absolute minimum are beneficial and complementary to this methodology.

4.2 Production Code Analysis

In the second step of the analysis, we use the previously obtained profiling information to perform analysis and pruning of the code. Specifically, the production sites of the intermediate sequences need to be identified and then pruned. In this step of our methodology it is important to discern what other variables the intermediate sequences depend on. An overview of this part of the methodology is given in Algorithm 1. The details of the individual steps are further explained in the following paragraphs.

Algorithm 1 Production Code Analysis

```
1:  Profile the identified sequences $S_{sequences}$
2:  **for all** identified sequences $seq_n \in S_{sequences}$ **do**
3:    Identify production sites $Psite_n$ for sequence $seq_n$
4:    **for all** code blocks $c_n^i \in Psite_n$ **do**
5:      **if** code block $c_n^i$ has side-effects **then**
6:        Try to disentangle side-effect code
7:        Otherwise, continue with next identified sequence $seq_{n+1}$
8:      **end if**
9:      Discern dependency variables $Var_i^n$ used in $c_n^i$
10:     **for all** dependency variables $var \in Var_i^n$ **do**
11:       **if** dependency variable $var$ is non-scalar **then**
12:         Ensure lifetime of $var$ is extended to consumption site of $seq_n$
13:       **end if**
14:     **end for**
15:     Introduce sub-sequence $sub_n^i$ for $seq_n$ built from code block $c_n^i$ and the previous sub-sequence $sub_n^{i-1}$ or the empty sub-sequence.
16:   **end for**
17: **end for**
```

Once the intermediate sequences have been identified, the profiling information concerning their production will indicate where they are produced. Here it is important to determine whether the code of data production has any observable side-effects besides the creation of the intermediate variables. If this is so, then either this code will have to be disentangled from the production logic, or it will not be possible to apply the transformation of this specific variable. This is due to the fact that the optimization transformation reorganizes the global ordering of loops. In multimedia applications side-effects do not tend to occur in the code that is responsible for the computation and as such this tends to not be a limitation. In the case that non-scalar variables are used for the production of the intermediate variable, and these variables no longer exist at the consumption site of the intermediate variable, then either their lifetime will have to be extended to that point or they will have to be copied. If their lifetime already extends until the consumption site of the intermediate sequence, then either a pointer or reference to those variables can be stored in the delayed sequence construct, unless it is a scalar, at which point a simple copy can be made. It is assumed that the input data that is used to generate the intermediate sequence does not change between the production and consumption site of the intermediate sequence. Through the analysis of the code producing the data, and by looking at the profiling information, this can be validated. The profiling information is only used to drive the analysis of *where* the code should be transformed not *how*, thereby maintaining correctness of the transformation.

It is important to identify the different loops that create the intermediate sequences and to prune the code to create intermediate sub-sequences. For each region of code that modifies the sequence, a new sub-sequence copy is created. This is done by creating a new sub-sequence that is a copy of the previous sub-sequence and then modify it with the new loop. These copies will not lead to an overhead in memory accesses as they will all be removed in the optimization step of the methodology. Through this transformation, this comes close to the *static-single-assignment* form [5] of the source code but at the sub-sequence-level. Abstractly we can say that each intermediate copy is then formed by the fusion of two streams, the data originally in the sequence and the data being added by the code-structure surrounding the modification statement. Although the easiest approach is to have one sub-sequence per loop, if two adjacent loops both use the append operation (push_back in STL) then they can be treated as one block, thereby reducing the number of lazy combinators that need to be used.

The specific operations that can appear in one code block generating one sub-sequence from the previous subsequence are listed below. Initially, the presented method does not support mixing different operations (for instance both insertion and appending) within one loop. Although this may seem like a strong limitation, in typical applications these operations are not usually intermingled at such a fine level of granularity. It is, however, possible, to have one loop with one type of operations following a loop with another type of operations, which is the idea behind the different sub-sequences.

- either any number of `push_back` (append) operation as demonstrated in Code 2,
- or any number of `insert` operations using one specific iterator, with any number of ++ operations on that iterator and starting exactly at where the iterator is set to the beginning of the previous sub-sequence as demonstrated in Code 3,
- or any number of `remove` operations using one specific iterator, with any number of ++ operations on that iterator and starting exactly at where the iterator is set to the beginning of the previous sub-sequence as demonstrated in Code 4

```
for (...)
  // Any set of nested loops...
  if (...)
    // ... or conditions
    // Any number of appends
    inter.push_back(...);
// No other operations accessing inter
```

Code 2: Code block for appends

```
// Start of insert block introducing
// a new sub-sequence
iterator o = inter.begin();
for (...)
  // Any set of nested loops...
  if (...)
    // ... or conditions
    // Any number of insertions as long as
    // they are based on the same iterator o
    o = inter.insert(o, ...);
    // Any number of increments of the same iterator
    o++;
// No other operations besides insert
// based on the iterator o accessing inter
```

Code 3: Code block for insertions

```
// Start of remove block introducing
// a new sub-sequence
iterator o = inter.begin();
for (...)
  // Any set of nested loops...
  if (...)
    // ... or conditions
    // Any number of removals as long as
    // they are based on the same iterator o
    inter.remove(o);
    // Any number of increments of the same iterator
    o++;
// No other operations besides remove
// based on the iterator o accessing inter
```

Code 4: Code block for removals

For `insert` operations, the return value of the operation needs to be stored back into the iterator being used to insert values into the sub-sequence. This is a requirement by STL as otherwise the iterator could possibly become invalid if the insertion causes a reallocation. Note that it is not necessary to literally introduce the sub-sequence copies, merely to annotate where their production begin and ends and on which prior sub-sequence they depend.

4.3 Cost Tradeoff

To be able to determine whether the removal of an intermediate sequence will positively influence the memory accesses, an analysis of the cost tradeoff needs to be performed. This tradeoff is non-trivial as the removal of an intermediate sequence does not just lower the memory accesses. As noted in the previous section, it is possible that certain other variables need to be extended in lifetime to enable the removal of one intermediate variable. Additionally, if the intermediate sequence is consumed more than once, then the gain in memory accesses is also less clear.

In general, however, a few heuristics can help to determine whether a variable should be removed or not. If an intermediate sequence of N elements is consumed only once, which tends to be the case for phased multimedia kernels, then there will be a gain in memory accesses of N reads and N writes. Nonetheless, that specific variable was removed as well as it depended on an intermediate sequence which depended on other variables that lived longer than that specific variable. Although the delayed computation introduces a few scalar variables to enable the pausing of the production control-flow, these are negligible as they can be placed into registers which have a much lower memory access cost than off-chip SDRAM where the intermediate sequences and input data reside. In the case that each element in the intermediate is dependent on only one element in the input data, then the sequence can be consumed multiple times and the transformation will still result in a reduction in memory accessed.

Although execution time is not studied in this paper, the overhead of pausing the computation is negligible as modern day VLIWs can execute simple conditional code without big penalties [21]. Therefore, in the most typical case when the intermediate sequence is only consumed once, there will be no loss in performance and a gain in performance is even envisioned as the data no longer needs to be stored and retrieved. The reason there is no loss in performance is that the computation of the sequence is simply being delayed until when the consumption of this sequence requires it; consuming it once therefore does not lead to recomputation.

4.4 Code Transformation and Optimization

The final step is to optimize the source code by transforming it and removing the code of the production of the intermediate sub-sequences as well as the intermediate subsequences themselves that were selected in Section 4.3.

After the tradeoff has been performed, the next step is to apply source-code transformation to remove the code producing the different sub-sequences and move this to a delayed sequence. Beginning with the first sub-sequence of the first sequence in the data-dependency list, the code of the data production is hoisted out of its context and placed into a thunk as demonstrated in Figure 2. The `APPEND_THUNK` macro expands to a partial class definition that can then be extended. For each local variable or parameter that is used in the production loop, an appropriate constructor parameter and instance variable is made, using references or pointers for those variables that are not scalar.

When the code is hoisted out, instead a specific sequence type should be created that constructs on top of the previous sub-sequence. The specific transformation is dependent on what type code-block is being hoisted out. In the case of an append-block, such as in Code 2, all the `push_back` operations should be replaced by the macro call `cr_return` operation with two parameters, first a unique number and secondly the value that was going to be stored in the sequence. This call will save the current location in the production loop and return the value whenever the thunk is called. Through the use of a typedef, the delayed sub sequence type is defined, with as parameter the type of the previous sub sequence of the intermediate sequence as well as the thunk that has the delayed code producing the data. Finally, through a proper constructor call, such a delayed sub sequence can easily be constructed. Then, a delayed sub-sequence is built as shown in Code 5.

```
APPEND_THUNK(FOO, type_of_element)
  append_thunk_FOO(parameters) {...}
  local variables...;
  APPEND_BODY(
    // code producing the data with
    // 'append' replaced by
    cr_return(UNIQUE NUMBER, value);
  )
};
typedef append_sequence<
  previous_delayed_sub_sequence_type,
  append_thunk_FOO
> delayed_sub_sequence_type;
...
// Where the code producing the data was:
delayed_sequence_type current_sub_sequence(
  previous_sub_sequence,
  append_thunk_FOO(parameters));
```

Code 5: Code transformation for append blocks

On the other hand, if the code producing the data is an insert-block, such as in Code 3, then all the insert calls are replaced by a call to `push` with the value that would have been inserted. On the other hand, in every location where the insertion iterator was being incremented, a call needs to be made to `cr_void` with a unique number as parameter. The prior will save the value in a local stack to be returned as soon as the original ++ operator is encountered. The latter will then return a value, such that the latest inserted value is available to be used. Similarly as with the append delayed sub-sequence, a typedef is used to actually define the sub-sequence in terms of the previous one and the delayed code block, and a simple constructor call with the parameters that were used in the code producing the data creates such a sub-sequence. One extra detail is that the original call get the beginning iterator for insertion is no longer kept, it is simply used to identify where the begin of the insert-block should be to make this transformation correct. The code for a delayed insert sub-sequence is shown in Code 6.

```
INSERT_THUNK(FOO, type_of_element)
  insert_thunk_FOO(parameters) {...}
  local variables...;
  INSERT_BODY(
    // code producing the data with
    // 'insert' replaced by
    push(value);
    // and the ++ calls replaced by
    cr_void(UNIQUE NUMBER)
  )
};
typedef insert_sequence<
  previous_delayed_sub_sequence_type,
  insert_thunk_FOO
> delayed_sub_sequence_type;
...
// Where the code producing the data was:
delayed_sequence_type current_sub_sequence(
  previous_sub_sequence,
  append_thunk_FOO(parameters));
```

Code 6: Code transformation for insert blocks

Finally, if the code producing the data was actually a remove-block, such as in Code 4, the transformation is nearly identical to that of an insert-block. However, in this case, instead of calling push with the value to be inserted, each time a remove statement is found, a the function `drop` should be called. The code for a delayed remove sub-sequence is shown in Code 7.

As noted, each sub sequence is built on top of a previous sub sequence, therefore it is important to have an original sub sequence which represents an empty sequence. For this, the special template class `empty_sequence<type_of_element>` is used which always returns an iterator that is equal to its end iterator. A standard empty `vector` of the STL library would also suffice, except that this would create a larger overhead in both memory accesses as well as memory footprint, the `empty_sequence` specifically designed to be a read-only empty sequence.

It is important to ensure that *all* code blocks producing the data that relate to an original intermediate sequence can be removed, meaning all the sub-sequence copies that were introduced to remove any modification statements. This is necessary as otherwise removing only some of the copies

```
REMOVE_THUNK(FOO, type_of_element)
  remove_thunk_FOO(parameters) {...}
  local variables...;
  REMOVE_BODY(
    // code producing the data with
    // 'remove' replaced by
    drop();
    // and the ++ calls replaced by
    cr_void(UNIQUE NUMBER)
  )
};
typedef remove_sequence<
  previous_delayed_sub_sequence_type,
  remove_thunk_FOO
> delayed_sub_sequence_type;
...
// Where the code producing the data was:
delayed_sequence_type current_sub_sequence(
  previous_sub_sequence,
  append_thunk_FOO(parameters));
```

Code 7: Code transformation for remove blocks

will lead to code where only some parts of the intermediate sequence are not stored, and as such the gains in the memory accesses would be less.

5 Assessment of the proposed approach

In this section we demonstrate the gains of the previously exposed methodology when applied to a real 3D reconstruction application [19] and a number of different, smaller STL benchmarks, derived from real multimedia applications.

The first case study is a 3D reconstruction algorithm that resembles 3D perception of humans, where the relative displacement between two 2D projections (i.e., one for each eye) is used to reconstruct the 3rd dimension. The experimental results are taken from the source code that is one of the basic building blocks in many current 3D vision algorithms. More specifically, the source code under study has been extracted from the original code of the 3D image reconstruction system (see [24] for the full code of the global 3D algorithm, which contains 1.75 million lines of high level C++) and creates the mathematical abstraction from the images or related frames that is used in subsequent phases of the global algorithm. This implementation matches corners [19] detected in 2 subsequent frames and was chosen due to its memory usage intensive nature.

In the core of the application, point-matches are selected by comparing neighbourhoods of the two frames. These points are put into a local sequence named `ImageCandidates`. For each feature point of the first image, this process is repeated and the points are selected from this sequence to be put into a more global sequence `CandidateMatches`. Once all the candidate matches have been found, a second phase of the application tries to find the best candidates. It utilizes the fact that the points in `CandidateMatches` are sorted by the point in the first image. The best candidates are filtered out and stored into another sequence `BestMatches`. After the algorithm finishes, the best matches are copied into the final output sequence named `NewMatches`. Finally, the matches in `NewMatches` is copied into the structure `OldMatches`.

The results can be seen in Table 1. The accesses to the input variables are not counted, because the accesses to this variables do not increase after the different transformations in this specific algorithm. It should be noted that these are not the real memory accesses, but only the sequence accesses (the accesses to the sequence interface, where adding an element is counted as a write and getting an element from an iterator is considered a read). If real accesses were measured, then the removed overhead of each sequence would be even bigger as the internal management of the vectors would introduce a lot of spurious accesses that are not present after our transformation. We have decided not to count them in our metric, although this presupposes an optimized data type definition, something which is nontrivial. Therefore, in general, our transformation should yield even higher gains in memory accesses.

We have explored the removal of each of the sequences that exist in the application, to see what the effect is of removing just a subset of the sequences. Through this exploration we were able to validate that the number of memory accesses to the input in this application did not change due to our transformation. The results can be seen in Table 1. To keep the data presentable, the different sequences have been given mnemonics defined here: `A` for `ImageCandidates`; `B` for `CandidateMatches`; `C` for `BestMatches`; and `D` for `NewMatches`.

To study the effect of how this methodology will scale, we have introduced several different benchmarks that illustrate a behaviour similar to that found in real-life applications. They are derived from the 3D reconstruction application as well as from the STL benchmarks found in [4]. In these benchmarks, we have started from code samples found in previous sources and extended them along two different axes to test scalability. In theses test benches we have also explored the global effect when the number of accesses to the input change due to the optimization. As such, the accesses to the input are also counted. Specifically, we were interested in seeing the effect of dynamism as well as the effect of re-consuming an intermediate sequence type. Specifically we model a set of consecutive loops where data is read from an input, conditionally placed into the first intermediate sequence, transformed several times from one sequence to the next, and finally consumed several times from the last sequence. The number of iterations (kept at 1000 for legibility) in the first loop are not varied in the set of benchmarks as the results have been found to scale linearly with it. The different parameters that have been explored can be seen below, and the results can be found in Table 2. As can be observed, in one few case, the transformation can sometimes lead to an increase in memory accesses. Therefore it is important to use the cost analysis to determine when performing the optimization is beneficial.

Optimized Sequences	sequence reads	sequence writes	relative sequence accesses	sequence access reduction
None	85828	63950	100.00%	0.00%
[A]	54375	32497	58.00%	42.00%
[B]	32149	32497	43.16%	56.84%
[A,B]	696	1044	1.16%	98.84%
[C]	85480	63602	99.54%	0.46%
[A,C]	54027	32149	57.54%	42.46%
[B,C]	31801	32149	42.70%	57.30%
[A,B,C]	348	696	0.70%	99.30%
[D]	85480	63602	99.54%	0.46%
[A,D]	54027	32149	57.54%	42.46%
[B,D]	31801	32149	42.70%	57.30%
[A,B,D]	348	696	0.70%	99.30%
[C,D]	85132	63254	99.07%	0.93%
[A,C,D]	53679	31801	57.07%	42.93%
[B,C,D]	31453	31801	42.23%	57.77%
[A,B,C,D]	0	348	0.23%	99.77%

Table 1: Relative sequence accesses of the 3D Reconstruction application depending on which sequences are removed

- p : Probability that an element is used from the input
- n_i : Number of input reads required to make an element in the first intermediate sequence
- L : Number of subsequent loops transforming one intermediate sequence to the next
- n_o : Number of times the last intermediate sequence is consumed

6 Conclusion

In this paper we presented a methodology to remove intermediate dynamic buffers, specifically sequences. Due to the common trend towards reliable embedded software design, these sequences are increasingly wasting memory bandwidth and increase the energy consumption through the memory accesses. Our results show that we can achieve real gains in removing these sequences with a systematic, stepwise approach. The proposed optimization actually performs a source-to-source transformation which removes the intermediate data and introduces on-demand sequences,

p	n_i	L	n_o	Accesses Before	Accesses After	Relative Gains
50%	1	1	1	3000	1000	66.67%
50%	1	1	5	5000	5000	0.00%
50%	1	5	1	7000	1000	85.71%
50%	1	5	5	9000	5000	44.44%
50%	2	1	1	4000	2000	50.00%
50%	2	1	5	6000	10000	-66.67%
50%	2	5	1	8000	2000	75.00%
50%	2	5	5	10000	10000	0.00%
100%	1	1	1	5000	1000	80.00%
100%	1	1	5	9000	5000	44.44%
100%	1	5	1	13000	1000	92.31%
100%	1	5	5	17000	5000	70.59%
100%	2	1	1	6000	2000	66.67%
100%	2	1	5	10000	10000	0.00%
100%	2	5	1	14000	2000	85.71%
100%	2	5	5	18000	10000	44.44%

Table 2: Scalability of methodology in function of the number of loop-nests and the number of data elements

without altering the input-output behaviour of the application. As a by-product of removing these memory accesses, the memory footprint is also reduced as the storage for the intermediate sequences is no longer necessary. Nevertheless, the systematic approach to calculate the relevant memory footprint tradeoffs is not within the scope of this paper and will be investigated in future work. Although the transformations are systematic, they are not yet fully formalized and thus resort to a pattern-based approach. Future work will address the formalization of this transformation to extend it to handle sloppier code.

References

[1] S. Agrawal et al. Optimizing stream programs using linear state space analysis. In *CASES*, ACM, 2005.

[2] D. J. Armstrong. The quarks of object-oriented development. volume 49, USA, 2006. ACM Press.

[3] L. Benini et al. System-level power optimization: techniques and tools. *ACM TODAES*, 2000.

[4] J. Beyer. Cbench: Compiler benchmark, 2006. http://cbench.sourceforge.net.

[5] G. Bilardi et al. Algorithms for computing the static single assignment form. *Journal of the ACM*, May 2003.

[6] F. Catthoor et al. *Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Boston, USA, 1998.

[7] E. De Greef. *Storage Size Reduction for Multimedia Applications*. PhD thesis, Katholieke Universiteit Leuven, January 1998.

[8] D.Renz et al. Implementing lazy streams in c++. *Morehead Electronic Journal of Applicable Mathematics*, 4, May 2005.

[9] A. Gill et al. A short cut to deforestation. In *FPCA* , June 1993.

[10] S. M. Inc. The collections framework, 2005. http://java.sun.com/docs/books/tutorial/collections/.

[11] M. Kandemir et al. Improving cache locality by a combination of loop and data transformations. *IEEE Transactions on Computers*, February 1999.

[12] M. Leeman et al. Methodology for refinement and optimisation of dynamic memory management for embedded systems in multimedia applications. In *SIPS*, August 2003.

[13] M. Leeman et al. Intermediate variable elimination in a global context for a 3d multimedia application. In *ICME*, July 2003.

[14] Micron Technology, Inc. 128MSDRAM. *http://www.micron.com/dram*.

[15] S. Muchnick. *Advanced compiler design & implementation*. Morgan Kaufmann Publisher, San Francisco, CA, 1997.

[16] D. Musser et al. *STL Tutorial and Reference Guide*. Addison-Wesley Publishing Company, 2nd edition, 2001.

[17] Glassgow haskell compiler. http://www.haskell.org/ghc/.

[18] P. R. Panda, et al. Data and memory optimizations for embedded systems. *ACM TODAES*, April 2001.

[19] M. Pollefeys et al. Metric 3D surface reconstruction from uncalibrated image sequences. In *Lecture Notes in Computer Science*, 1998.

[20] F. Quillere et al. Optimizing memory usage in the polyhedral model. *ACM TOPLAS*, September 2000.

[21] P. Raghavan et al. Distributed loop controller architecture for multi-threading in uni-threaded VLIW processors. In *DATE*, 2006.

[22] A. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, 1995.

[23] SGI. Standard template library, 2006. http://www.sgi.com/tech/stl/.

[24] Target jr, 2002. http://www.targetjr.org.

[25] D. van Arkel et al. Fusion in practice. In *In Implementation of Functional Languages, (LNCS)*, 2002.

[26] P. Vanbroekhoven et al. Advanced copy propagation for arrays. In *LCTES*, 2003.

[27] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP*, 1988.