Program Parallelization using Synchronized Pipelining

Leonardo Scandolo[1], César Kunz[1], and Manuel Hermenegildo[1,2]

[1] IMDEA Software
[2] Technical U. of Madrid, Spain
Firstname.Lastname@imdea.org

Abstract. While there are well-understood methods for detecting loops whose iterations are independent and parallelizing them, there are comparatively fewer proposals that support parallel execution of a sequence of loops or nested loops in the case where such loops have dependencies among them. This paper introduces a refined notion of independence, called *eventual independence*, that in its simplest form considers two loops, say loop_1 and loop_2, and captures the idea that for every i there exists k such that the $i+1$-th iteration of loop_2 is independent from the j-th iteration of loop_1, for all $j \geq k$. Eventual independence provides the foundation of a semantics-preserving program transformation, called *synchronized pipelining*, that makes execution of consecutive or nested loops parallel, relying on a minimal number of synchronization events to ensure semantics preservation. The practical benefits of synchronized pipelining are demonstrated through experimental results on common algorithms such as sorting and Fourier transforms.

1 Introduction

Multi-core processors are becoming ubiquitous: most laptops currently on the market contain at least two execution units, whereas servers commonly use eight or more cores. Since the number of on-chip cores is expected to double with each processor generation, there is a pressing challenge to develop programming methodologies which exploit the power of multi-core processors without compromising correctness and reliability. One prominent approach is to let programmers write sequential programs and to build compilers that parallelize these programs automatically.

Most parallelization techniques rely on some notion of independence, which ensures that certain fragments of the program only access distinct regions of memory, and thus execution of one such code fragment has no effect on the execution of the others. For example, code fragments written in a simple imperative language are guaranteed to be independent if their reads and writes are *disjoint*, in which case their sequential composition can be parallelized without modifying the overall semantics of the program. More refined notions of independence include the classical notions of absence of flow dependence, anti-dependence, or output dependence [7].

Well-understood methods exist for detecting loops whose iterations are independent (i.e., they do not contain *loop-carried dependencies*) and parallelizing them. These techniques have been used to achieve automated/correct parallelization of a number of algorithms for scientific computing such as, e.g., image processing, data mining, DNA analysis, or cosmological simulation. However, these parallelization methods do not provide significant speedups for other algorithms which contain sequences or nesting of loops whose iterations are partially dependent and/or irregular. Examples of such loops appear, for example, in sorting algorithms or Fourier transformations. On the other hand, such algorithms can be parallelized efficiently by the technique that we propose, *synchronized pipelining*, which allows loops with dependencies to be executed in parallel by making sparse use of synchronization events to ensure that the ahead-of-time execution of loop iterations does not alter the original semantics.

Our proposal is illustrated in Section 2 with a mergesort algorithm. As a warm-up to Section 2, let us first consider synchronized pipelining in its simplest form, when it deals with two consecutive loops manipulating an array structure:

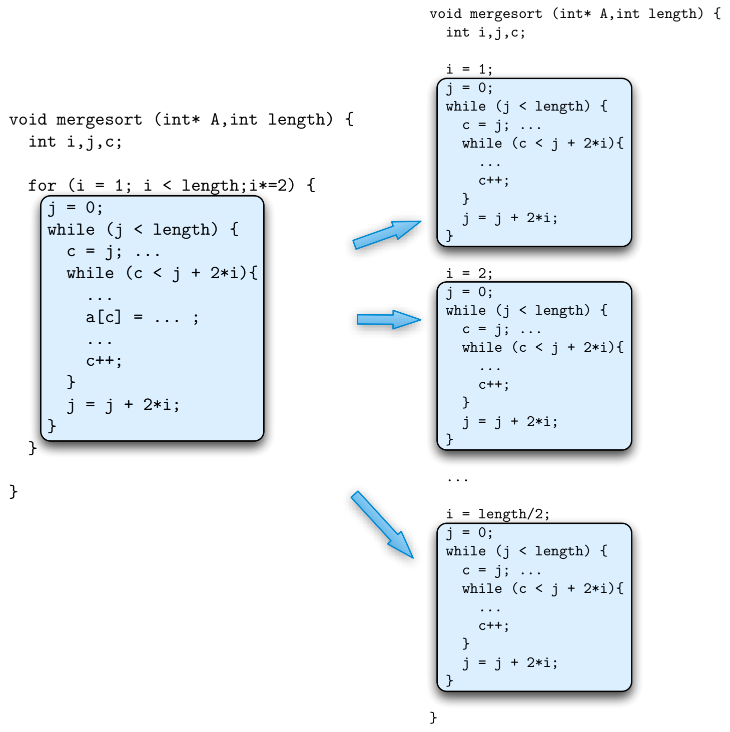
$$\texttt{while } b_1 \texttt{ do } c_1; \texttt{while } b_2 \texttt{ do } c_2$$

For simplicity, assume that the data dependence between c_1 and c_2 is restricted to the contents of the array structure. The aim is to return code that may start the execution of some iterations of c_2 before completion of the loop with body c_1. To justify such a transformation, we rely on *eventual independence*, a generalization of independence which accounts for the possibility of executing the $m + 1$-th iteration of a loop ahead of time. Informally, c_2 is eventually independent from c_1 iff for every n_2, there exists n_1 such that after n_1 iterations of c_1 and n_2 iterations of c_2, c_1 and c_2 are independent. Once eventual independence between the two loops is established, it is possible to define a semantics-preserving transformation that outputs a program:

$$\texttt{while } b_1 \texttt{ do } c'_1 \parallel \texttt{while } b_2 \texttt{ do } c'_2$$

where c'_1 is obtained from c_1 by adding event announcements to indicate that part of the computation of c_2 can be performed, and c'_2 is obtained from c_2 by inserting blocking statements that control the gradual and early computation of c_2; in both cases, the transformation of c_i into c'_i is guided by the eventual independence relation.

In the course of the paper, we develop the notions of eventual independence and synchronized pipelining, starting from the simple case discussed above and then dealing with sequences of loops and nested loops. In addition, we illustrate the benefits of our approach, drawing experimental results from common cases such as the above mentioned sorting algorithms and Fourier transforms. We also outline the necessary procedures and tools to automatically generate this transformation for the case in which we deal with simple data structures (arrays), and outline future lines of research to extend this approach to more general problems. In summary, the main contributions of this paper are the formal definition of eventual independence (Section 4), eventual independence



```
void mergesort (int* A,int length) {
  int i,j,c;

  for (i = 1; i < length;i*=2) {
   j = 0;
   while (j < length) {
     c = j; ...
     while (c < j + 2*i){
       ...
       a[c] = ... ;
       ...
       c++;
     }
     j = j + 2*i;
   }
  }

}
```

```
void mergesort (int* A,int length) {
  int i,j,c;

  i = 1;
  j = 0;
  while (j < length) {
    c = j; ...
    while (c < j + 2*i){
      ...
      c++;
    }
    j = j + 2*i;
  }
  i = 2;
  j = 0;
  while (j < length) {
    c = j; ...
    while (c < j + 2*i){
      ...
      c++;
    }
    j = j + 2*i;
  }

  ...

  i = length/2;
  j = 0;
  while (j < length) {
    c = j; ...
    while (c < j + 2*i){
      ...
      c++;
    }
    j = j + 2*i;
  }
}
```

Fig. 1: Iterative mergesort algorithm

criteria for the particular case of array manipulating loops, and an experimental evaluation of the benefits of synchronized pipelining (Section 6). Although many of the concepts and results of the paper only make minimal assumptions on the programming language, we carry our development in the setting of a parallel imperative language with events, introduced in Section 3.

2 Motivating Example: mergesort

Figure 1 presents the structure of an iterative mergesort algorithm. After unrolling some of the `for` loop iterations from the fragment shown on the right of the figure, we have a sequence of iterations of the inner loop `while`(`j` $<$ `length`)$\{\ldots\}$ accessing and modifying the array intervals $[0, 1]$, $[2, 3]$, $\ldots$, $[$`length`$-1,$ `length`$]$ in the first iteration, the intervals $[0, 3]$, $[4, 7]$, $\ldots$, $[$`length`$-3,$ `length`$]$ in the second iteration, and so on until the last iteration in which the intervals $[0,$ `length`$/2]$ and $[$`length`$/2 + 1,$ `length`$]$ are accessed.

One can clearly see that the first and second unrolled iteration cannot be executed in parallel (without changes) since they read and/or modify overlapping regions of the array. However, after partial completion of the first iteration, the

```
while (j < length) {
  c = j;
  ...
  while (c < j + 2*i){
    ...
    c++;
  }
  j = j + 2*i;
}
```

(a) Original

```
while (j < length) {
  c = j;
  ...
  while (c < j + 2*i){
    τ(i − 1, c) →
    {
      ...
      c++;
    }
  }
  j = j + 2*i;
  τ(i, j)!;
}
```

(b) Pipelined

Fig. 2: Illustrative example of pipelined code

second iteration can advance without waiting for the first iteration to finish. For instance, the second iteration can safely start processing the array interval $[0, 3]$, right after the first iteration has finished processing the array intervals $[0, 1]$ and $[2, 3]$. The parallelization technique we propose allows the second loop iteration to gradually progress in parallel with the first one (and successive ones), introducing synchronization primitives in order to preserve the original semantics. To this end, we rely on a heuristic oracle Ω, defined in terms of the number of steps already executed by the first and second loop, that determines at which point of the first loop it is safe to enable a partial execution of the second one.

Figure 2 gives a brief but illustrative scheme of how the code in Figure 1 is to be annotated with parallelization primitives. In this case we use τ to denote such device, using a question mark to signify a wait event on a certain subscript (or set of subscripts) that the current loop is waiting to use, and an exclamation mark to denote a signaling event which allows other threads to continue execution.

3 Setting

The target language for synchronized pipelining is a simple imperative language with arrays, extended with parallel composition and synchronization primitives.

The extension includes an empty statement nil, a standard parallel composition $\|$, and event-based synchronization primitives. We assume given a set of events $\mathcal{S}$ used for synchronization. Let $\tau \in \mathcal{S}$ and $S \subseteq \mathcal{S}$ represent a synchronization event and a synchronization event set, respectively. The statement $S!$ is a non-blocking announcement of the events in S, whereas the statement $\tau \rightarrow c$ waits for the event τ to be announced before proceeding with the execution of c.

Let *Stmt* be the set of program statements, Σ be the set of mappings from program variables to integer values, and $\mathcal{S}^\star$ be the powerset of $\mathcal{S}$. The program semantics is given by a transition relation between configurations, where a configuration is either an exceptional configuration abort, resulting, e.g., from and array-out-of-bound access, or a normal configuration, i.e., an element

$$\frac{}{\langle S!, \sigma, \epsilon\rangle \rightsquigarrow \langle \mathsf{nil}, \sigma, \epsilon \cup S\rangle} \qquad \frac{\tau \in \epsilon}{\langle \tau \to c, \sigma, \epsilon\rangle \rightsquigarrow \langle c, \sigma, \epsilon\rangle}$$

$$\frac{c \equiv d \qquad \langle d, \sigma, \epsilon\rangle \rightsquigarrow \langle d', \sigma', \epsilon'\rangle \qquad d' \equiv c'}{\langle c, \sigma, \epsilon\rangle \rightsquigarrow \langle c', \sigma', \epsilon'\rangle} \qquad \frac{c \equiv d \qquad \langle d, \sigma, \epsilon\rangle \rightsquigarrow \mathsf{abort}}{\langle c, \sigma, \epsilon\rangle \rightsquigarrow \mathsf{abort}}$$

$$\frac{\langle c, \sigma, \epsilon\rangle \rightsquigarrow \langle c', \sigma', \epsilon'\rangle}{\langle c \parallel d, \sigma, \epsilon\rangle \rightsquigarrow \langle c' \parallel d, \sigma', \epsilon'\rangle} \qquad \frac{\langle c, \sigma, \epsilon\rangle \rightsquigarrow \mathsf{abort}}{\langle c \parallel d, \sigma, \epsilon\rangle \rightsquigarrow \mathsf{abort}}$$

$$i \parallel \mathsf{nil} \equiv i \qquad i \parallel j \equiv j \parallel i \qquad i \parallel (j \parallel k) \equiv (i \parallel j) \parallel k$$

Fig. 3: Operational semantics (excerpts)

of $Stmt \times \Sigma \times \mathcal{S}^\star$. Formally, the semantics is given by a small-step relation: $\rightsquigarrow \subseteq (Stmt \times \Sigma \times \mathcal{S}^\star) \times ((Stmt \times \Sigma \times \mathcal{S}^\star) + \{\mathsf{abort}\})$.

The transition rules for synchronization and parallel execution are given in Figure 3, together with the definition of the congruence relation $\equiv \subseteq Stmt \times Stmt$; all other rules are standard. Note that event announcement is asynchronous and that event identifiers are never removed from ϵ. Thus, once an event has been announced, and until the end of the program execution, every process waiting for that event is ready to proceed.

Example 1. Consider for example the statement $(\texttt{x := 5}; \tau!) \parallel \tau \to \texttt{x := 1}$. Starting from a state where τ has not been announced, the execution terminates with the variable x holding the value 1, since `x := 1` cannot proceed before the event τ has been announced.

As usual, we can derive from the small-step semantics an evaluation semantics $\Downarrow \subseteq (Stmt \times \Sigma \times \mathcal{S}^\star) \times (\Sigma + \mathsf{abort})$, by setting:

$$\begin{array}{lcl} \langle c, \sigma, \epsilon\rangle \Downarrow \sigma' & \text{iff} & \exists \epsilon'.\ \langle c, \sigma, \epsilon\rangle \rightsquigarrow^\star \langle \mathsf{nil}, \sigma', \epsilon'\rangle \\ \langle c, \sigma, \epsilon\rangle \Downarrow \mathsf{abort} & \text{iff} & \langle c, \sigma, \epsilon\rangle \rightsquigarrow^\star \mathsf{abort} \end{array}$$

where $\rightsquigarrow^\star$ denotes the reflexive and transitive closure of $\rightsquigarrow$. In turn, the evaluation semantics can be used to define a notion of semantic equivalence.

Definition 1 (Semantic Equivalence). *Let $c_1, c_2 \in Stmt$ be two statements, $\sigma \in \Sigma$ be a state and $\epsilon \subseteq \mathcal{S}$ be a set of synchronization events. We say that c_2 simulates c_1 w.r.t. σ and ϵ, written $[\![c_1]\!] \leq_{(\sigma,\epsilon)} [\![c_2]\!]$, iff for every $\sigma' \in (\Sigma + \mathsf{abort})$, we have $\langle c_1, \sigma, \epsilon\rangle \Downarrow \sigma' \Rightarrow \langle c_2, \sigma, \epsilon\rangle \Downarrow \sigma'$. We say that c_1 and c_2 are semantically equivalent w.r.t. σ and ϵ, written $[\![c_1]\!] \equiv_{(\sigma,\epsilon)} [\![c_2]\!]$, iff $[\![c_1]\!] \leq_{(\sigma,\epsilon)} [\![c_2]\!]$ and $[\![c_2]\!] \leq_{(\sigma,\epsilon)} [\![c_1]\!]$.*

4 Eventual Independence

The purpose of this section is to introduce the notion of eventual independence, and to discuss how eventual independence relations may be inferred. For the sake of completeness, we start by recalling the semantic notion of independence between two statements.

Definition 2 (Independent Statements). *Two statements $c_1, c_2 \in Stmt$ are* independent *iff* $[\![c_1; c_2]\!] \equiv [\![c_1 \parallel c_2]\!]$.

Eventual independence aims to capture a relation between iterations of two loop bodies c_1 and c_2, and thus would be naturally formalized as a relation between natural numbers. For the clarity of the technical development, it is however preferable to view eventual independence as a relation between natural numbers and events, and assume given a function $\lambda : \mathbb{N} \to \mathcal{S}$ that assigns to each natural number m of loop_2 the event $\lambda(m)$ that will release the m-th iteration of loop_2.

Definition 3 (Eventual Independence Relation). *Statements $c_1, c_2 \in Stmt$ are eventually independent w.r.t. a relation $\Omega \subseteq \mathbb{N} \times \mathcal{S}$ iff for all $m, n \in \mathbb{N}, \epsilon \subseteq \mathcal{S}$ s.t. $(n, \lambda(m)) \in \Omega$, $\sigma \in \Sigma$ and no synchronization variables in ϵ appear in c_1 or c_2:*

$$[\![c_1^n; c_2^{m-1}; c_1^k; c_2]\!] \equiv_{(\sigma,\epsilon)} [\![c_1^n; c_2^{m-1}; (c_1^k \parallel c_2)]\!]$$

for all $k \in \mathbb{N}$. The expression c^i stands for the sequential composition of i instances of the statement c. Given Ω and $n \in \mathbb{N}$, we let $\omega(n) = \{s \mid (n, s) \in \Omega\}$.

Example 2. Consider the following program:

```
while bi do {a[i]:=a[i]+1; i := i*2}; while bj do {a[j]:=a[j]+1; j := j+1}
```

The two loop statements are not necessarily independent, but one can define an eventual independence relation over the loop bodies in order to parallelize their iterations. In this case, the loop statements are eventually independent with respect to a relation Ω, if $(n, \lambda(m)) \in \Omega$ implies $i^\star * 2^n < j^\star + m$, where $i^\star$ is the initial value of variable i and $j^\star$ is the initial value of variable j.

In practice, when considering sequential code, it is sufficient to state the semantics equivalence in terms of the event set $\epsilon = \emptyset$. From the definition of eventual independence, if $\lambda(m) = s$, then the m^{th} execution of c_2 shall wait for the event s to execute. Assuming $(n, s) \in \Omega$ then it is safe to signal the event s after executing n times the statement c_1, allowing the m^{th} execution of statement c_2 to take place. Indeed, by definition of Ω, it follows from $(n, s) \in \Omega$ that after n iterations of c_1, any and all subsequent executions of c_1 do not modify a piece of memory on which the m^{th} iteration of c_2 depends.

The main reason for defining the Ω relation is to link the iterations of the loop bodies that are safe to execute in parallel. If we take $m = 1$ in the definition, then we see that n is simply the number of iterations of c_1 that we need to execute before we can execute the first iteration of c_2 (in parallel with the remaining iterations of the first loop) without altering the semantics of the original program. Higher values of m are in relation through Ω with the values n after which it is safe to execute the m^{th} iteration of the second loop, provided that the $m - 1$ previous iterations where executed following the guidelines that Ω defines. This is the basis for the transformation we are aiming at and it is formalized in the next section.

The set $\omega(n)$, which is defined in terms of Ω, is the set of all the events that are safe to announce after n executions of the statement c_1. Since the purpose of this definition is to have a construct that will allow us to denote the set of events the first loop can safely announce after each iteration has ended, we will mainly use ω when defining our transformation.

4.1 Inferring Eventual Independence

The eventual independence relation Ω and the function λ are essential ingredients of synchronized pipelining, as they will be used to guide the insertion of synchronization statements in the original program. Therefore, it is important to be able to infer Ω and λ for a large class of code fragments. We have been able to infer this data efficiently for the algorithms under consideration, that manipulate array structures of significant size. Consider the case in which both c_1 and c_2 read and modify data from a single array `a`, iterating over the induction variables h_1 and h_2 respectively. By simple code inspection, one can easily collect the sets of syntactic expressions $\vec{e}_1$ and $\vec{e}_2$ used to read or update the array `a` inside the loop body. These array accesses are not always expressed in terms of the induction variables h_1 and h_2. However, in general, we have found that they are expressed in terms of induction variables h'_1 and h'_2 *derived* from h_1 and h_2. In those cases, induction variable analysis [4] allows one to rewrite the derived induction variables h'_1 and h'_2 in terms of the induction variables h_1 and h_2, i.e. $h'_1 = f_1(h_1)$ and $h'_2 = f_2(h_2)$ for some function expressions f_1 and f_2.

Most frequently, when h'_i is an induction variable derived from h_i, then f_i is a linear function on h_i. More complex cases may arise, for instance when f_i is defined as a polynomial or geometric function on h_i. In those cases, the expressions $\vec{e}_1(h'_1)$ and $\vec{e}_2(h'_1)$ are easily rewritten in terms of the inductive variables, i.e., as $\vec{e}_1(f_1(h_1))$ and $\vec{e}_2(f_2(h_2))$. By static interval analysis, we can approximate the regions of data that are read and modified by c_1 and c_2, in terms of the induction variables h_1 and h_2, and the expressions $\vec{e}_1(f_1(h_1))$ and $\vec{e}_2(f_2(h_2))$.

Assume $[l_1^{rw}, u_1^{rw}]$ represents the interval of the array `a` that is written or read by c_1, where l_1^{rw}, u_1^{rw} are integer expressions that depend on h_1 (and similarly with c_2). Since $\vec{e}(f_1(h_1))$ and $\vec{e}(f_2(h_2))$ are linear (or polynomial) functions on h_1 and h_2, one can determine whether they are monotonic (or determine the points from which they are monotonic). If the l and u expressions are increasing as the h variables grow (the decreasing case is symmetrical) one can propose an eventual independence relation Ω. For instance when l_1^{rw} and u_2^{rw} are increasing functions, we determine the pairs (a,b) of values for h_1 and h_2 such that $u_2^{rw} < l_1^{rw}$, and then, since the b^{th} iteration of c_2 is independent of the a^{th} iteration of c_1, we can have $(a, \lambda(b)) \in \Omega$.

Example 3. We show in this paragraph how to determine an eventual independence relation for this simple pair of loop statements

$$\texttt{while } b_1 \texttt{ do } c_1; \texttt{while } b_2 \texttt{ do } c_2$$

where c_1 and c_2 are defined as

$$\begin{array}{lcl} c_1 & \doteq & \mathsf{a[x] := 1; x := x + 1} \\ c_2 & \doteq & \mathsf{y := y + a[z]; z := z + 1} \end{array}$$

First of all, notice that statements c_1 and c_2 access the array a, so they are not independent. By examining statements c_1 and c_2, it is immediate that the indexes of the array accesses are monotonically increasing and the relation between the initial values of program variables (denoted $x^\star$ for a variable x) define the eventual independence relation. In this case, a simple induction variable analysis will define $\vec{e}_1$ and $\vec{e}_2$, and thus l_1^{rw},l_2^{rw}, u_1^{rw} and u_2^{rw}, as a linear function of the induction variables: $l_1^{rw}(h_1) = u_1^{rw}(h_1) = h_1 + x^\star$ and $l_2^{rw}(h_2) = u_2^{rw}(h_2) = h_2 + z^\star$. Thus, the procedure's requirements translate into: $h_2 + z^\star < h_1 + x^\star$. The argument above allows us to propose an eventual independence relation Ω.

$$\begin{array}{l} (z^\star - x^\star + 1, \lambda(1)) \in \Omega_{c_1,c_2} \\ \forall x.\ x \leq z^\star - x^\star + 1 \Rightarrow (x, \lambda(1)) \notin \Omega_{c_1,c_2} \end{array}$$

This Ω relation formalizes the intuition that c_1 and c_2 can be executed in parallel as long as every iteration k of c_2 executes after the iteration number $z^\star - x^\star + k$ of c_1. Furthermore, since the size of the array a ($|\mathsf{a}|$) is bounded, if c_1 is executed more than $|\mathsf{a}| - x^\star$ times, we end up at an exceptional state abort, in which case any execution of c_2 is independent. In conclusion, the following relation Ω determines the eventual independence between c_1 and c_2:

$$\begin{array}{l} x + x^\star \leq |\mathsf{a}| \wedge y \leq x + z^\star - x^\star - 1 \Rightarrow (x, \lambda(y)) \in \Omega_{c_1,c_2} \\ x + x^\star > |\mathsf{a}| \Rightarrow (x, \lambda(y)) \in \Omega_{c_1,c_2} \end{array}$$

5 Synchronized Pipelining

We now define synchronized pipelining, starting from two consecutive loops, and then extending the transformation to sequences of loops and nested loops.

Consider a program c of the form `while` b_1 `do` c_1; `while` b_2 `do` c_2, where c_1 and c_2 are compound statements that access an array. We assume that the boolean conditions b_1 and b_2 are not affected by the execution of c_2 and c_1, respectively. Further, we let h_1 and h_2 be program counters that determine the number of iterations already performed for the first and second loop respectively. Our aim is to transform the program so that it executes both loops in parallel. To preserve the program semantics, the transformation must insert code that ensures a correct synchronization between the two loops, so the resulting program will be of the form `while` b_1 `do` $c_1' \parallel$ `while` b_2 `do` c_2', where c_1' is derived from c_1 by adding event announcements and c_2' is derived from c_2 by adding synchronization guards. Both transformations are guided by a relation Ω of eventual independence and by a function λ that are given as input to the transformation.

Definition 4. *The* synchronized pipelining *of c is statement $\bar{\bar{c}}$ defined as:*

$$\bar{\bar{c}} = (\texttt{while } b_1 \texttt{ do } c_1'); S! \parallel \texttt{while } b_2 \texttt{ do } c_2'$$

where $c'_1 = c_1; \omega(h_1)!$, $c'_2 = \lambda(h_2) \rightarrow c_2$, and S is the set of all events on which statement c'_2 can wait.

Statement $S!$ is introduced after the execution of c'_1 to ensure that all events are indeed announced, and thus the progress of the original program is preserved. In order to accomplish that, statement $S!$ simply announces all events, in any order. Since all events in which statement c'_2 is waiting are eventually announced by $S!$, statement c'_2 cannot block indefinitely. For the same reason, $c \leq \bar{\bar{c}}$. Notice that the set of events announced by c'_1 and $S!$ may be redundant. In practice, one can reduce program size and synchronization overhead by statically removing duplicated events. Similarly, c_2 may be simplified by removing synchronization primitives that wait on the same event. We assume, however, the definition given above for notational simplicity.

The eventual independence condition determined by Ω is enough to show that the semantics is preserved. That is, every execution state reached by the final program is also reachable by the original one.

Proposition 1 (Semantics Preservation). *For every initial state $\sigma \in \Sigma$ and every event set ϵ disjoint from the fresh synchronization variables introduced by the transformation, we have that $[\![c]\!] \equiv_{(\sigma,\epsilon)} [\![\bar{\bar{c}}]\!]$.*

5.1 Extensions

We first analyze the case of a sequence of loops. Then, we explain how we proceed in the presence of nested loops.

Loop Sequences. Now suppose the original program is of the form:

$$\texttt{while } b_1 \texttt{ do } c_1; \ldots; \texttt{while } b_n \texttt{ do } c_n$$

The idea is to parallelize the whole program by progressively applying the basic transformation to each pair of interfering loops. Therefore, we must provide for all i, j such that $i < j$ an eventual independence relation $\Omega_{i,j}$ and a function $\lambda_{i,j} : \mathbb{N} \rightarrow \mathcal{S}$. By definition of eventual independence, we must have for every $(n, \lambda_{i,j}(m)) \in \Omega_{i,j}$ and for all state σ and event set ϵ:

$$[\![c_i^n; c_j^{m-1}; c_i^k; c_j]\!] \equiv_{(\sigma,\epsilon)} [\![c_i^n; c_j^{m-1}; (c_i^k \parallel c_j)]\!]$$

Since the parallel execution of the i^{th} loop may interfere not only with its immediately preceding loop, but with every preceding one, we synchronize each pair of non-independent loops. Thus, the i^{th} loop of the final program becomes:

$$\texttt{while } b_i \texttt{ do } \bigcup_{1 \leq j < i} \lambda_{i,j}(h) \rightarrow \left(c_i; \bigcup_{i < j \leq n} \omega_{i,j}(h)! \right); \forall_{i<j \leq n} S_{i,j}!$$

where $S_{i,j}$ stands for all the synchronization events used to synchronize execution between $\texttt{while } b_i \texttt{ do } c_i$ and $\texttt{while } b_j \texttt{ do } c_j$, for every $i < j$. From the expression above, it may seem that excessive synchronization overhead is introduced. However, the actual number of synchronization primitives depends on the definition of λ and ω, and on the removal of duplicated synchronization events.

Nested Loops. We now turn our attention to a different but more common program structure: nested loops. Consider the following program as the target of the parallelization: `while` a `do` $(c_1;$ `while` b `do` $c; c_2)$. In order to be able to apply our transformation we take the following assumptions:

1. We assume that the number of iterations of the outer loop (or an overapproximation) can be computed at runtime. In the rest of this section we let β stand for the number of iterations that may be computed at runtime and, for simplicity, we assume that the boolean condition a is of the form $l \leq \beta$, where l is the induction variable of the outer loop, incremented with step 1 from the initial value 1. In practice, the exact form of a may differ from this assumption, but we assume that it is possible to evaluate the number of iterations at runtime based on the current memory state. Intuitively, if we can determine the exact number of iterations of the outer loop, we can unroll it and parallelize the resulting program by applying the transformation on sequences of loops as explained above. However, assuming that we can statically determine the exact number of iterations is an unnecessary and too strong assumption.
2. We assume also that there is no interference between the scalar variables read and modified in c_1 and c. We can reduce the interference between loop iterations by vectorizing each scalar variable v into an array $\hat{v}$, with the cost of extra memory usage. For every statement c and boolean condition b, we denote $\hat{c}[l]$ and $\hat{b}[l]$ the result of vectorizing scalar variables in c and b, respectively. The value of the variable l determines which position of the vectorized variables is in use. At the end of the transformed program, a `sync` operation takes each vectorized variable $\hat{v}$, and transforms it back into the original scalar variable v, i.e., executes $v = \hat{v}[\beta]$. The reason for this vectorization is to avoid clashes between the values that are accessed by the fragments `while` $\hat{b}[i]$ `do` $\hat{c}[i]$, for different values of i.
3. The last hypothesis we make is that the scalar variables initialized by the statement c_1 are not modified by c or c_2 after vectorization. This is a reasonable assumption to make, since data structure accesses are in most cases confined to the inner loop. This allows us to ignore dependencies between these instructions and the rest of the loop.

As before, for every $i, j \in \mathbb{N}$ s.t. $i < j \leq \beta$ we need a function $\lambda_{i,j} : \mathbb{N} \to \mathcal{S}$ mapping iterations to synchronization events. In this case, the parametric relation $\Omega_{i,j}$ takes into account the last instructions of the outer loop. We require, if $(n, \lambda_{i,j}(m)) \in \Omega_{i,j}$ and for every $\epsilon \subseteq \mathcal{S}$ and $\sigma \in \Sigma$, that:

$$[\![\hat{c}[i]^n; \hat{c}[j]^{m-1}; \hat{c}[i]^k; \hat{c_2}[i]; \hat{c}[j]]\!] \equiv_{(\sigma,\epsilon)} [\![\hat{c}[i]^n; \hat{c}[j]^{m-1}; (\hat{c}[i]^k; \hat{c_2}[i] \parallel \hat{c}[j])]\!]$$

The transformation is similar to the one performed for sequences of loops. Since inner loops are syntactically equal, the value of induction variable l corresponding to the outer loop is used to distinguish between different iterations.

The transformation follows, thus, the scheme:

$$\begin{array}{l}\texttt{while}\ a\ \texttt{do}\quad \tau_{c,l-1} \to (\hat{c_1}[l]; \tau_{c_1,l}!)\,; \\ \quad\texttt{while}\ \hat{b}[l]\ \texttt{do}\ \Big(\bigcup_{1\leq j<l} \lambda_{l,j}(h) \to \Big(\hat{c}[l]; \bigcup_{l<j\leq\beta} \omega_{l,j}(h)!; \hat{c_2}[h']\Big)\Big)\,; \\ \texttt{sync}\end{array}$$

Notice that the order in which the instances of $\hat{c}[l]$ are executed is preserved.

5.2 Motivating Example Revisited

Our motivating example, `mergesort`, was annotated with synchronization statements that follow the guidelines described in our transformation. If we take two consecutive iterations of the main loop of the program, we can sketch the constructs we have presented in our theoretical model.

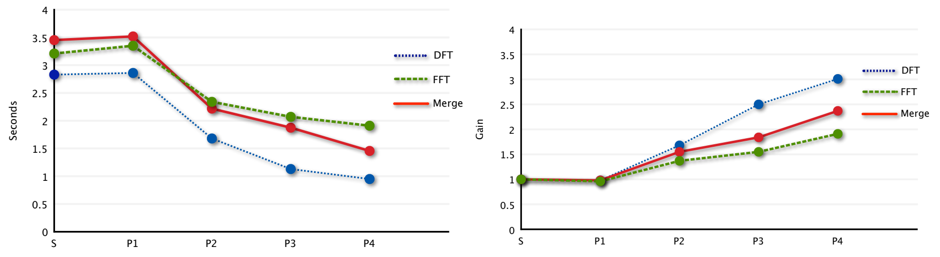
Starting from the original code, we need first to vectorize the variables that parameterize our inner loop. In our example this is variable `i`. Since we need to spawn a new procedure in order to launch (possibly) a new thread, we encapsulate the inner loop in a function call, which receives `i` as a parameter. Then, the stack allocation scheme automatically vectorizes variable `i` for us, since now each iteration will possess its own copy of `i`, independent from the others, and initialized to the value which each iteration would see in a sequential execution. The only problem here consists in working with a language which allows function calls to be made to run in parallel. Later we will explain how we deal with this issue in practice.

In the original program, the variable `c` is the expression used for writing in the array, and furthermore it is the lowest variable which is read or written in the array. On the other side, the variable `r` is the highest variable which is read, this is a consequence of the initial state of the inner loop and is preserved in the loop body. We can analyze the loop and determine that `c` is monotonically increasing. It follows that if we have two consecutive iterations,i and $i+1$, of the loop, the latter cannot proceed unless it can assure that the value of[3] $\texttt{c}^{\texttt{i}}$ is bigger than that of $\texttt{r}^{\texttt{i+1}}$.

Thus, the following piece of code is added to the original code:

```
...
while (j < length){
      while (c-j<2*i){
        event_wait(r);
        fromQueue = last(Q);
        if (l-j > i){
            ...
            A[c] = dequeue(Q);
        }
        event_announce(c);
        c++;
      }
...
```

[3] We use superscripts to denote which loop variables belong to and subscripts to refer to the value of the variable at a given iteration of its loop.



(a) Computation Time (b) Speedup

Fig. 4: Experimental Results

Our function λ essentially maps $m \to \mathtt{r}_{\mathtt{m}}$. It becomes apparent now that our Ω relation must relate every tuple $(n,\lambda(m))$ where $\mathtt{c}_{\mathtt{n}}^{\mathtt{i+1}}$ is larger than $\mathtt{r}_{\mathtt{m}}^{\mathtt{i}}$.

We now need to determine λ and Ω for every other possible combination of iterations. But since the same loop is repeated, with the same properties, we require the same condition to advance, namely $\mathtt{r}_{\mathtt{m}}^{\mathtt{i}} < \mathtt{c}_{\mathtt{n}}^{\mathtt{j}}$, and thus $\Omega_{i,j}$ again contain pairs $(n,\lambda_{i,j}(m))$ which meet that condition.

6 Experimental Results

We have experimented with the parallelizing transformation taking as input a program written in a subset of *C* and returning a *Cilk* [5] program. *Cilk* is an extension of *C* for multithreaded parallel programming, that provides a light-weight thread model based on job stealing.

We proceed by annotating the source program with *Cilk* statements for thread creation and synchronization, using *Cilk locks* and *spawn* procedures to implement event signaling and efficient variable synchronization. We encapsulate inner loops in *spawned* procedures, and use the *C* stack allocation scheme to efficiently allocate memory for vectorization.

The proposed transformation has been applied to well-known algorithms that traverse arrays to obtain information as to the applicability and the efficiency of our approach. In all cases, the transformation yields good results unless the input size is tiny enough to make the synchronization overhead relatively significant.

For our tests we have used a 64bit Intel(R) Core(TM)2 Quad CPU at 2.4 GHz clock speed, 1GB of DIMM 800 MHz memory, running GNU/Linux.

In all cases we have labeled the graphics with `S` for the sequential (unmodified) algorithm, running on a single processor, and we have labeled `Pn` for our modified, pipelined algorithm with `n` processors.

Figure 4 shows the computing time and the relative performance gain of the DFT[4], FFT[5], and MergeSort algorithms run under the different conditions we

[4] Discrete Fourier Transform
[5] Fast Fourier Transform

have explained. The pipelined version of our DFT program is slightly slower while running with only one processor, due to the overhead of synchronization variable allocation and signaling. Once we augment the number of available processors the amount of time spent computing starts to decrease as the several runs on the array on which we are working start to (safely) overlap. The efficiency gain is almost linear, but of course the overhead of signaling and also the thread creation and manipulation overhead add some extra work to the computation. The algorithm used is well suited for our transformation since it copies the input array and then modifies one element at a time incrementally, allowing several elements to be modified at the same time without interference.

Our experiments with an FFT algorithm also yield good results, though not as good as with the DFT algorithms. The reason for this is that unlike DFT, FFT traverses the input array heavily and performs the computation in-place, so it slowly gives up resources and thus the overlapping of different traversals is smaller. Nevertheless, some performance gain is indeed achieved in our pipelined version of the algorithm, roughly a 50% gain with 4 processors. The pipelined version is still outperformed by the sequential one in the case we have a single processor available, again due to synchronization overheads.

The last benchmark we present is that of our motivating example, namely mergesort. This algorithm also traverses an array several times incrementally, which allows us to obtain greater benefits from our transformation. The benchmarks were made sorting an array of one million elements. The results show that our transformation yields a 240% efficiency increase by overlapping the merging steps that are otherwise run sequentially, for a 4 processor machine.

7 Related Work

Ottoni et al. [13] proposed a technique called Decoupled Software Pipelining (DSWP) to extract the fine-grained parallelism hidden in most applications. The process is automatic, and general, since it considers non-scientific applications in which the loop iterations have heavy data dependencies. It provides a transformation that is slightly different to typical loop parallelization, in which each iteration is assigned alternately to each core, with an appropriate synchronization to prevent data races. As a result, no complete iteration is executed simultaneously with another one, since every iteration has a data dependence with every other one. Instead of alternating each complete loop iteration on each core, DSWP splits each loop body before distributing them among the available cores. This technique improves the locality of reference of standard parallelization techniques, and thus reduces the communication latency. It is effective in a more general set of loop bodies, but it does not take advantage of the eventual data independence hidden in scientific algorithms.

A recent experimental study [10] analyzes particular cases in which standard automatic parallelization fails to introduce significant improvements. This is the case of applications that manipulate complex and mutable data structures, such as Delauney mesh refinement and agglomerative clustering. The authors propose a practical framework, the *Galois* system, that relies on syntactic constructs to enable programmers to hint to the compiler on parallelization opportunities and an optimistic parallelization run-time to exploit them. Due to the unpredictability of irregular operations on mutable and complex data structures, the *Galois* framework is mostly based on runtime decisions and backtracking, and does not exploit statically inferred data dependence.

Data Parallel Haskell [14] (DPH) provides nested data parallelism to the existing functional language compiler GHC. Flat parallelism is restricted to the concurrent execution of sequential operations. Nested parallelism generalizes flat parallelism by considering the concurrent execution of functions that may be executed in parallel, and thus provides a more general and flexible approach, suitable for irregular problems. DPH extends Haskell with parallel primitives, such as *parallel arrays* and a set of *parallel operations* on arrays. The compiler compiles these parallel constructions by desugaring them into the GHC Core language, followed by a sequence of Core-to-Core transformations. DPH is a notable framework for the specification of concurrent programs, but the compiler is not intended to automatically discover parallel evaluations.

In a different line of work, the Manticore project is developing a parallel programming language for heterogeneous multi-core processor systems [3]. A main feature of the language is the support for both implicit and explicit threading. Nevertheless, as a design choice, it avoids implicit parallelism (i.e., it requires the programmer to hint parallelism by providing annotations) since they claim implicit parallelism to be only effective for dense regular parallel computations.

The goal of the Paraglide project at IBM is to assist the construction of highly-concurrent algorithms. The Paraglider tool [17] is a linearization-based framework to systematically construct complex algorithms manipulating concurrent data structures, from a sequential implementation. This approach combines manual guidance with automatic assistance, focusing mainly on fine-grained synchronization.

8 Conclusion

Synchronized pipelining is a parallelization technique that relies on eventual independence, a new refinement of the established notion of independence, to successfully transform programs with nested loops. This paper has set the theoretical foundations of the transformation, and showed its practical benefits on representative examples.

Future work includes applying this transformation to general recursive procedures, which is a possibility if the program is first transformed into an iterative version of itself. This is a widely studied optimization problem [11] which can significantly improve performance. Other lines of research include applying the transformation to languages that manipulate the heap. Many concepts developed in this paper are largely independent of the underlying programming language, and the main issue is rather to find an analysis to detect independence. Recent work on the use of shape analysis and separation logic for detecting data dependence and for parallelization provide a good starting point (e.g., [15, 16, 8, 6, 12]).

References

1. V. Allan, R. Jones, R. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.
2. T. Fahringer and B. Scholz. A unified symbolic evaluation framework for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, PDS-11(11):1105–1125, November 2000.
3. M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in manticore. In J. Hook and P. Thiemann, editors, *ICFP*, pages 119–130. ACM, 2008.
4. M. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, 1995.
5. Supercomputing Technologies Group. Cilk 5.4.6 reference manual, 1998.
6. S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.
7. J. Hennessy and D. Patterson. *Computer Architecture: a quantitative approach.* Morgan Kaufman, 2003.
8. J. Hummel, L. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *PLDI*, 1994.
9. M. Joyner, Z. Budimlic, and V. Sarkar. Optimizing array accesses in high productivity languages. In Ronald H. Perrott, Barbara M. Chapman, Jaspal Subhlok, Rodrigo Fernandes de Mello, and Laurence Tianruo Yang, editors, *High Performance Computing and Communications, HPCC 2007, Houston, USA, September 26-28, 2007, Proceedings*, volume 4782 of *LNCS*, pages 432–445. Springer, 2007.
10. M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In J. Ferrante and K. McKinley, editors, *PLDI*, pages 211–222. ACM, 2007.
11. Y. Liu and S. Stoller. From recursion to iteration: What are the optimizations? In *PEPM*, pages 73–82, 2000.
12. M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. Identification of Heap-Carried Data Dependence Via Explicit Store Heap Models. In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.
13. Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO*, pages 105–118. IEEE Computer Society, 2005.
14. Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag.
15. M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *ESOP*, 2009. To appear.
16. R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *PPOPP*, 1999.
17. M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In R. Gupta and S. Amarasinghe, editors, *PLDI*, pages 125–135. ACM, 2008.

Appendix

A Complete mergesort Algorithm

Original algorithm.

```
void mergesort (int* A,int length){
  int i,j,c,l,r,fromQueue;
  struct queue Q;

  for (i = 1; i < length;i*=2) {
   j = 0;
   while (j < length) {
     c = j;l = c; r = j + i;
     while (c < j + 2*i){
       fromQueue = last(Q);
       if (l >= j + i){
 if ((last(Q) < A[r] && last(Q) != -1) || r >= j + 2*i )
   A[c] = dequeue(&Q);
 else
   A[c] = A[r++];
       }
       else if (r >= j + 2*i)
 A[c] = dequeue(&Q);
       else{
 if (fromQueue < A[l] && fromQueue < A[r] && fromQueue != -1){
   enqueue(&Q,A[l++]);
   A[c] = dequeue(&Q);
 }
 else if (A[r] < A[l]){
   enqueue(&Q,A[l++]);
   A[c] = A[r++];
 }
 else /* (A[l] < A[r]) */
   l++;
       }
       c++;
     }
     j = j + 2*i;
   }
  }
}
```

Transformed algorithm

```
  for (_forIndex = 0; _forIndex < _A_length;_forIndex++)
    event_announce(_A_events[0][_forIndex]);


  for (_outerCount = 0; _outerCount < _outerLoopCant;_outerCount++) {
    spawn _innerLoop(A,_outerCount,i);
    i *= 2;
  }
  sync;
  return;
}

cilk void _innerLoop(int* A,int _outerCount, int i){
  int j = 0,c = j,l = j,r = j+i,fromQueue;
  int length = _A_length;
  struct queue Q;

  while (j < length) {
    c = j;l = c; r = j + i;
    while (c < j + 2*i){
      event_wait(_A_events[_outerCount][r]);
      fromQueue = last(Q);
      if (l >= j + i){
if ((last(Q) < A[r] && last(Q) != -1) || r >= j + 2*i ){
  A[c] = dequeue(&Q);
}
else{
  A[c] = A[r];r++;
}
      }
      else if (r > j + 2*i)
A[c] = dequeue(&Q);
      else{
if (fromQueue < A[l] && fromQueue < A[r] && fromQueue != -1){
  enqueue(&Q,A[l]);l++;
  A[c] = dequeue(&Q);
}
else if (A[r] < A[l]){
  enqueue(&Q,A[l]);l++;
  A[c] = A[r];r++;
}
else /* (A[l] < A[r]) */
  l++;
      }
      event_announce(_A_events[_outerCount+1][c]);
      c++;
    }
    j = j + 2*i;
  }
}
```

B Loop Sequence Transformation Example

We wish to apply our loop sequence transformation to the program in Figure 5a. From simple observation we can deduce that any starting state under which the loops will be executed will be such that variables `x`, `y` and `z` have values 1, 2 and 3 respectively. .[6] We can also ensure that in $\texttt{Loop}_1$ variables `y` and `z` will not be modified, and neither will variable `z` in $\texttt{Loop}_2$.

Another property we assume is that array `a` has only 10 elements, and since it is the only shared resource, we use one event per array element to synchronize the different loops.

Taking these properties into account it is easy to establish the Ω functions by analyzing how the variables are modified by these loops.

The result of a simple analysis on the loops taking into account the initial conditions yields:[7]

- $\Omega_{(1,2)}$ will relate values $(n,\lambda(m))$ such that $n > m + 1$.
- $\Omega_{(2,3)}$ will relate values $(n,\lambda(m))$ such that $n > 2 * m + 1$.
- $\Omega_{(1,3)}$ will relate values $(n,\lambda(m))$ such that $n > 2 * m + 3$.

This relation between loops is unsurprisingly the relation followed by the index of the accesses accesses made to array `a`, since it is the only shared resource. It can be preserved by simply requiring succeeding loops that want to access an element to wait for the preceding ones to modify it. This is accomplished by having an array element access wait on an event associated to that element, and after having done so, announce an event that allows the succeeding loops to modify or read elements it will no longer touch, in this case the lowest read or modified index of the iteration, since the indexes are increasing.

The modified program is presented in Figure 5b[8].

---

[6] Actually in more complicated programs a more advanced analysis may need to be run in order to obtain information about the starting state of the loops.

[7] The analysis is omitted on account of relevancy.

[8] The transformed program is annotated with the necessary synchronization primitives, though we have left out the announcement of the full set of events after a loop, because the construct is easy enough to analyze in order to determine that to be redundant.

```
x = 1
y = 2
z = 3
while (x < 10)
  a[x] = 3
  x = x + 1
while (y < 10)
  a[y] = a[y] + a[y-1]
  y = y + 1
while (z < 10)
  a[z] = 2 * a[z]
  z = z + 2
```

(a) Original

```
x = 1
y = 2
z = 3
( while (x < 10)
        a[x] = 3
        x = x + 1
        $\tau_{1,2,x}$ !
        $\tau_{1,3,x}$ !
||while (y < 10)
        $\tau_{1,2,y}$ →
        a[y] = a[y] + a[y-1]
        y = y + 1
        $\tau_{2,3,y-1}$ !
||while (z < 10)
        $\tau_{1,3,z}$ & $\tau_{2,3,z}$ →
        a[z] = 2 * a[z]
        z = z + 2
)
```

(b) Pipelined

Fig. 5: Loop sequence example code

C Semantics Preservation Proof *(sketch)*

The main idea is to prove semantics preservation inductively, relying on the definition of eventual independence and the restrictions it imposes on event signaling. If we execute the transformed statement, both loops of the original statement c ($\texttt{Loop}_1$ and $\texttt{Loop}_2$) will be launched in parallel, but since the second one has to wait for signaling, only $\texttt{Loop}_1$ is able to proceed initially.

We assume that iteration n of $\texttt{Loop}_1$ is the first one to announce the event the $\lambda(1)$, namely the event the first iteration of $\texttt{Loop}_2$ waits on.

Once this happens, we need to check if the definition of our dependence relation indeed preserves the semantics if the first iteration of $\texttt{Loop}_2$ is executed.

Thus, we have that

$$(n, \lambda(1)) \in \Omega_{(i,j)}$$

and thus, by the definition of our eventual independence relation we have that execution the first iteration ahead of time preserves our semantics for every $\epsilon \subset \mathcal{S}$, since the synchronization variables in our transformation are chosen to be fresh:

$$[\![c_1^n; c_1^k; c_2]\!] \equiv_{(\sigma,\epsilon)} [\![c_1^n; (c_1^k \parallel c_2)]\!] \equiv_{(\sigma,\epsilon)} [\![c_1^n; c_2; c_1^k]\!]$$

For the sake of simplicity we assume an ascending order in Ω, meaning that $(n_1, m_2), (n_2, m_2) \in \Omega \wedge m_2 > m_1 \Rightarrow n_1 \geq n_2$. This avoids having to use maximum values between values of n for the proof, but it is otherwise equivalent.

The proof for $m = 2$ will give an insight at what we are trying to prove. We have:

$$[\![c_1^k; c_2; c_2]\!] \equiv_{(\sigma,\epsilon)} [\![c_1^{n_1}; c_1^{k-n_1-n_2}; c_1^{n_2}; c_2; c_2]\!] \equiv_{(\sigma,\epsilon)} [\![c_1^{n_1}; c_2; c_1^{n_2-n_1}; (c_2 \parallel c_1^{k-n_2})]\!] \equiv_{(\sigma,\epsilon)}$$

$$\equiv_{(\sigma,\epsilon)} [\![c_1^{n_1}; (c_2 \parallel c_1^{n_2-n_1}); (c_2 \parallel c_1^{k-n_2})]\!]$$

The last expression is equivalent to the result of applying the transformation to the first expression and executing the c_2 statements as early as possible. We want to prove that we can interleave the execution of the c_2 statements guided by Ω in a safe way for an arbitrary m. We take as our inductive hypothesis:

$$[\![c_1^k; c_2^m;]\!] \equiv_{(\sigma,\epsilon)} [\![c_1^{n_1}; (c_2 \parallel c_1^{n_2-n_1}); (c_2 \parallel c_1^{k-n_2}); \ldots; (c_2 \parallel c_1^{n_m-n_{m-1}}); (c_2 \parallel c_1^{k-n_m})]\!]$$

Our inductive step is then:

$$[\![c_1^k; c_2^{m+1}]\!] \equiv_{(\sigma,\epsilon)} [\![c_1^k; c_2^m; c_2]\!] \equiv_{(\sigma,\epsilon)}$$

$$\equiv_{(\sigma,\epsilon)} [\![c_1^{n_1}; (c_2 \parallel c_1^{n_2-n_1}); (c_2 \parallel c_1^{k-n_2}); \ldots; (c_2 \parallel c_1^{n_m-n_{m-1}}); (c_2 \parallel c_1^{k-n_m}); c_2]\!] \equiv_{(\sigma,\epsilon)}$$

$$\equiv_{(\sigma,\epsilon)} [\![c_1^{n_1}; (c_2 \parallel c_1^{n_2-n_1}); (c_2 \parallel c_1^{k-n_2}); \ldots; (c_2 \parallel c_1^{n_m-n_{m-1}}); (c_2 \parallel c_1^{k-n_{m+1}-n_m}); c_1^{k-n_{m+1}}; c_2]\!] \equiv_{(\sigma,\epsilon)}$$

$$\equiv_{(\sigma,\epsilon)} [\![c_1^{n_1}; (c_2 \parallel c_1^{n_2-n_1}); (c_2 \parallel c_1^{k-n_2}); \ldots; (c_2 \parallel c_1^{n_m-n_{m-1}}); (c_2 \parallel c_1^{k-n_{m+1}-n_m}); (c_2 \parallel c_1^{k-n_{m+1}})]\!]$$

The last step is possible due to the fact that we can "push" the first m c_2 statements to the right in order to use the definition of the Ω function for the proof.