Parallelism Extraction in Acyclic Code

João Paulo Luís
jpsl@inesc.pt

Carlos Gonçalves Carvalho
cgc@inesc.pt

José Carlos Delgado
jcd@inesc.pt

Parallel Architectures Group
INESC
Lisboa, Portugal

Abstract

In this article we describe a framework for parallelism extraction and partitioning in acyclic code regions. This framework is an extension of Milind Girkar's work on functional parallelism, using a Petri net model to represent parallel code, and applying modified optimization techniques to minimize the overheads of explicit synchronization. The modifications introduced are directed towards the generation of efficient multi-threaded code. We also describe a simple partitioning technique that can be used to artificially increase the granularity of the extracted parallelism up to a desired level.

1 Introduction

Most research in parallelizing compilers is directed towards loop parallelisation ([2]), however, non-loop parallelism (sometimes called *functional parallelism*) can be worthy of exploration in some applications ([8]). As this kind of parallelism is usually fine-grained, most work done in this area considers low level mechanisms for efficient scheduling ([12]).

As more operating systems start to support multi-threaded programming models at the application level promising to take advantage of parallel hardware when available (usually when hardware supports symmetric multiprocessing), we have tried to develop a set of algorithms that would allow parallelism extracted from acyclic code regions to be exploited on these systems with adequate granularity.

The framework described here is essentially based on Girkar's work on functional parallelism ([6],[7]). We use a Petri net model to represent parallel code with different optimization techniques to minimize the use of explicit synchronization mechanisms directed towards producing efficient multi-threaded code. The techniques described here can be viewed as separate stages of a larger algorithm for parallelism extraction and partitioning in acyclic code regions.

We introduce the basics of our framework in sections 2 through 4. Section 5 introduces a "code sequencing" optimization technique which is performed before the application of Girkar's algorithm for determining essential data dependences, as described in section 6. Section 7 describes the mechanism used to implement explicit synchronization, and finally, in section 8 we introduce a partitioning algorithm that takes advantage of this framework.

An actual implementation of this algorithm is being used on a paralleliser (see [10] and [3]) for the ANDF (Architecture Neutral Distribution Format) program distribution format (see [11]).

2 Input Data

We extract implicit parallelism from an acyclic code region characterized by a *Control Flow Graph* (CFG) and a *Data Dependence Graph* (DDG). We shall explain some details of the algorithm through the use of an example. Figure 1(a) and 1(b) show the CFG and DDG used in this example.

2.1 Control Flow Graph

The CFG of an acyclic code region is an acyclic directed graph defined by a set of vertices V_{CFG} and a set of directed edges E_{CFG}. There are two unique vertices $Entry, eXit \in V_{CFG}$ such that all vertices are *reachable* from $Entry$ and $eXit$ is reachable from all vertices. We say that a vertex j is *reachable* from a vertex i if there is a non-null path from i to j or if $i = j$.

$$CFG = \langle V_{CFG}, E_{CFG} \rangle$$

$$V_{CFG} = \{Entry, 1, 2, 3, ..., n, eXit\}$$

$$E_{CFG} = \{\langle i, j \rangle \mid \text{there is an edge from } i \text{ to} j\}$$

We identify each vertex by an integer number. The numbering order is calculated such that for any pair of vertices $\langle i, j \rangle$, if $i < j$ (where "$<$" is the usual total ordering *less-than* relation) then the execution of i always precedes the execution of j for all possible sequential executions of the CFG where both i and j are executed. Note that there may be more then one possible numbering order that will satisfy this condition.

2.2 Data Dependence Graph

The DDG is also an acyclic directed graph defined as follows:

$$DDG = \langle V_{DDG}, E_{DDG} \rangle$$

$$V_{DDG} = V_{CFG}$$

$$E_{DDG} = \{\langle i, j \rangle \mid i \, \delta_d \, j\}$$

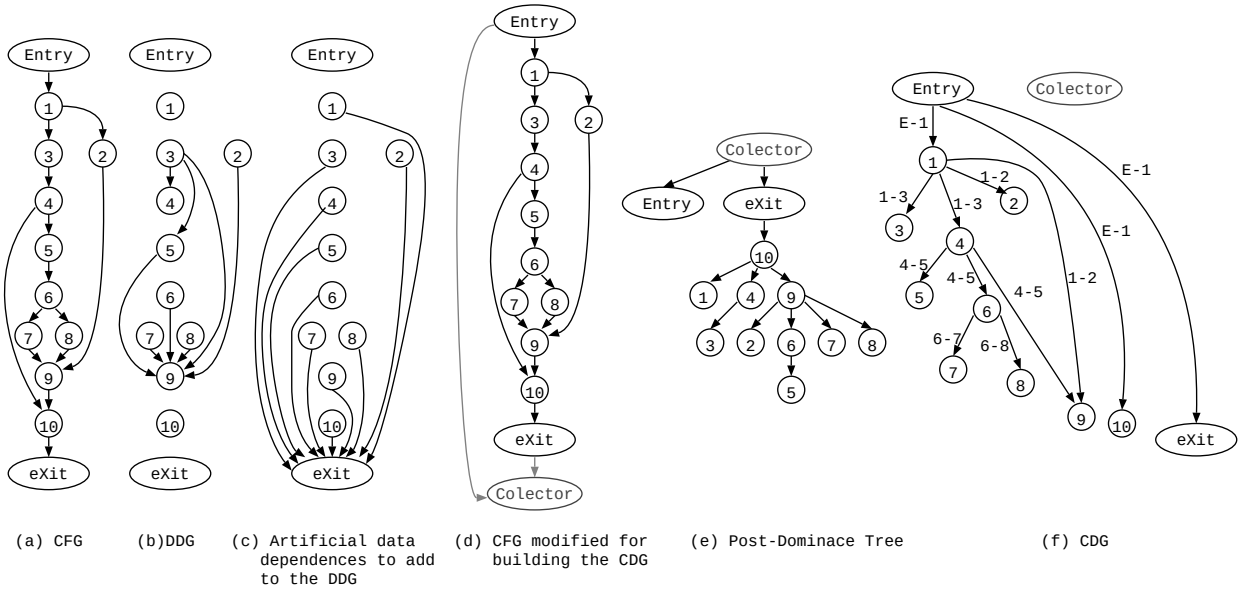


Figure 1: Graphs used by the algorithm

We write $i\,\delta_d\,j$ to say that vertex j is data dependent on vertex i. In this framework, $i\,\delta_d\,j$ only exists if $i \neq j$, j is reachable from i, and the code represented by these vertices has a memory conflict. A memory conflict occurs when two blocks of code access a common memory location with at least one access modifying it.[1] Note that if j is not reachable from i (on the CFG) then there **must not** be any edge from i to j on the DDG. Any memory conflicts will not matter as i and j will never be executed together.

2.3 Artificial Data Dependences

The *Entry* and *eXit* vertices have the particular role of delimiting the parallel execution of this block of code. This is useful when decomposing a parallel program into hierarchical code regions, so that each code region can have a single entry and a single exit point for control flow.

This stage will introduce an artificial set of dependences to force the algorithm to produce a parallel solution where the *eXit* vertex will be the last one to be executed (after all other vertices have finished execution). If this is not needed, then this preprocessing step can be dropped.

The *eXit* vertex is artificially considered data dependent on all other vertices (with the exception of *Entry* and itself). Figure 1(c) represents these artificial dependences.

$$\forall x \in (V_{DDG} \setminus \{Entry, eXit\})\,, x\,\delta_d\,eXit$$

3 Control Dependence

We say that j *post-dominates* i if every path on the CFG from i to $eXit$ (not including i) contains j. Note that a vertex does not post-dominates itself. The *post-dominance* relation can be represented by a tree where an edge from the parent vertex j to a child vertex i indicates that j post-dominates i and there is no other vertex that post-dominates i that doesn't post-dominates j also (j is called the *immediate post-dominator* of i). Figure 1(e) shows the *post-dominance tree* for the modified CFG of figure 1(d) in this example.[2]

We say that a vertex j is control dependent on a vertex i with label $i-k$ if j does not post-dominates i and there is a non-null path from i to j such that j post-dominates all the vertices on that path excluding i and j.

We will represent the control dependence relation using a *Control Dependence Graph* (CDG). We write $j\,\delta_c\,i$ with label $i-k$ to indicate that j is control dependent on i with label $i-k$:

$$CDG = \langle V_{CDG}, E_{CDG} \rangle$$

$$V_{CDG} = V_{CFG}$$

$$E_{CDG} = \{\langle i, j \rangle \mid j\,\delta_c\,i\}$$

We use the algorithm of figure 2 to build the CDG.

Before building the post-dominance tree and the CDG, a temporary modification is made to the CFG. The *Colector* vertex and the edge $\langle Entry, Colector \rangle$ are artificially added to the CFG (as shown in figure 1(d)) so that the CDG will have all vertices (excluding *Colector*) transitively control dependent on

[1] Other algorithms (such as memory renaming) can be applied to minimize memory conflicts before extracting parallelism.

[2] An efficient algorithm for building a *dominance tree* from a CFG is presented in [9]. The post-dominance tree can be obtained using the same algorithm by reversing the edges of the CFG and exchanging the roles of the $Entry$ and $eXit$ vertices.

1 **FORALL** edges in the CFG, $\langle i,j \rangle \in E_{CFG}$ **DO:**
2 **IF** j is an ancestor of i in the post-dominance tree **THEN**
3 this edge of the CFG does not introduce any control dependences.
4 **ELSE**
3 all vertices in the only possible path on the post-dominance tree from the parent of i to j are control dependent on i (including j and excluding i itself) with label $i-j$.

Figure 2: Algorithm used to build the labelled CDG

Entry. Assuring that all vertices are transitively control dependent on the *Entry* vertex will assure that the execution of *Entry* will precede the execution of the other vertices. This artificial vertex and edge **will be ignored** for all purposes other than the building of the CDG.

Figure 1(d), 1(e) and 1(f) show the modified CFG, the post-dominance tree, and the CDG for this example.

3.1 Upper Limit for Parallelism

The CDG is used as the basis for parallelism extraction. In the absence of data dependences, the information contained in the CDG can be used for scheduling the code in a parallel execution using the scheduling algorithm described in figure 3.

1 Every vertex that is not control dependent on another vertex begins execution.
2 **DO FOREVER:**
3 When a vertex finishes execution, all vertices control dependent on that vertex with the label corresponding to the branch that would have been taken in the CFG begin execution.

Figure 3: Scheduling code for parallel execution based on the CDG

In this example, step 1 of this scheduling algorithm will start the execution only of the *Entry* vertex (due to the use of the artificial edge $\langle Entry, Colector \rangle$ when building the CDG). When *Entry* finishes executing, vertex 1 and 10 start executing. When vertex 1 finishes execution, if the branch 1→2 is taken, then the vertices 2 and 9 begin executing in parallel, otherwise if the branch 1→3 is taken, then vertices 3 and 4 begin executing in parallel and so on...

The algorithm described in this section executes the same vertices that would have been executed on an equivalent sequential execution of the CFG. This is demonstrated in [6].

4 Petri Net Representation

Formally, the Petri net (which we call R) is composed of a set of places P, a set of transitions T, an input function I and an output function O:

$$R = \langle P, T, I, O \rangle$$

The set of places P is in direct correspondence with the set of vertices and the set of transitions T is in direct correspondence with the set of edges in the CFG. For this example we have:[3]:

$$\begin{aligned} P &= \{p_E, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_X\} \\ T &= \{t_{E-1}, t_{1-2}, t_{1-3}, t_{2-9}, t_{3-4}, t_{4-5}, t_{4-10}, \\ &\quad t_{5-6}, t_{6-7}, t_{6-8}, t_{7-9}, t_{8-9}, t_{9-10}, t_{10-X}\} \end{aligned}$$

The topology of the net is defined by the input and output functions (I and O respectively).

4.1 Initial Net Topology

Initially the Petri net topology is built to model the parallel execution of the code as described previously in section 3.1.

The function O maps every place p_i on to the set of its output transitions, $O : P \rightarrow T$. Its initial mapping is obtained from set of transitions corresponding to the outgoing edges of vertex i in the CFG.

$$O(p_i) = \{t_{i-j} \mid \langle i,j \rangle \in E_{GFC}\}$$

The function O also maps every transition t_{i-j} on to the set of its output places, $O : T \rightarrow P$. Initially, this is the set of of places that correspond to the vertices that have at least one incoming edge in the CDG labelled with the branch $i-j$.

$$O(t_{i-j}) = \{p_k \mid \langle i,k \rangle \in E_{CDG} \text{ with label } i-j\}$$

The function I maps every place on to the set of its input transitions, $I : P \rightarrow T$, and every transition on to the set of its input places, $I : T \rightarrow P$. As the function I provides redundant information, we will just define it using function O.

$$I(p_i) = \{t_{j-k} \mid p_i \in O(t_{j-k})\}$$

$$I(t_{i-j}) = \{p_k \mid t_{i-j} \in O(p_k)\}$$

For this example, the initial net topology can be defined by table 1. The net can be represented graphically as shown in figure 4.

4.2 Marking and Evolution

The parallel execution in the Petri net model is initiated by depositing a mark in the p_E place. At any time, to say that a place is marked is equivalent to say that the correspondent block of code is being executed. We say for short that the place is being executed. When a place finishes execution, only one output transition of that place will fire depending on which control flow path is chosen. For example, after the execution of p_1, transition t_{1-2} fires if the branch 1→2 is taken in an equivalent sequential execution of the CFG. When a transition fires, a mark is removed from each one of its input places, and a mark is deposited in each one of its output places.

[3] We have abbreviated the *Entry* and *eXit* vertex names to E and X respectively when talking about Petri net places, so p_E is the place that corresponds to the *Entry* vertex and p_X is the place that corresponds to the *eXit* vertex.

p_i	$O(p_i)$	$I(p_i)$
p_E	$\{t_{E-1}\}$	$\{\}$
p_1	$\{t_{1-2}, t_{1-3}\}$	$\{t_{E-1}\}$
p_2	$\{t_{2-9}\}$	$\{t_{1-2}\}$
p_3	$\{t_{3-4}\}$	$\{t_{1-3}\}$
p_4	$\{t_{4-5}, t_{4-10}\}$	$\{t_{1-3}\}$
p_5	$\{t_{5-6}\}$	$\{t_{4-5}\}$
p_6	$\{t_{6-7}, t_{6-8}\}$	$\{t_{4-5}\}$
p_7	$\{t_{7-9}\}$	$\{t_{6-7}\}$
p_8	$\{t_{8-9}\}$	$\{t_{6-8}\}$
p_9	$\{t_{9-10}\}$	$\{t_{1-2}, t_{4-5}\}$
p_{10}	$\{t_{10-X}\}$	$\{t_{E-1}\}$
p_X	$\{\}$	$\{t_{E-1}\}$

(a)

t_{i-j}	$O(t_{i-j})$	$I(t_{i-j})$
t_{E-1}	$\{p_1, p_{10}, p_X\}$	$\{p_E\}$
t_{1-2}	$\{p_2, p_9\}$	$\{p_1\}$
t_{1-3}	$\{p_3, p_4\}$	$\{p_1\}$
t_{2-9}	$\{\}$	$\{p_2\}$
t_{3-4}	$\{\}$	$\{p_3\}$
t_{4-5}	$\{p_5, p_6, p_9\}$	$\{p_4\}$
t_{4-10}	$\{\}$	$\{p_4\}$
t_{5-6}	$\{\}$	$\{p_5\}$
t_{6-7}	$\{p_7\}$	$\{p_6\}$
t_{6-8}	$\{p_8\}$	$\{p_6\}$
t_{7-9}	$\{\}$	$\{p_7\}$
t_{8-9}	$\{\}$	$\{p_8\}$
t_{9-10}	$\{\}$	$\{p_9\}$
t_{10-X}	$\{\}$	$\{p_{10}\}$

(b)

Table 1: Net topology

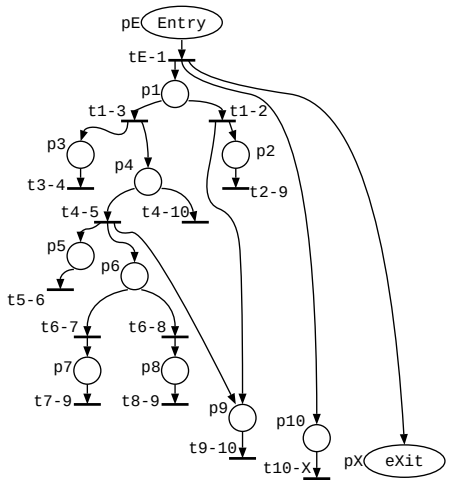


Figure 4: The Petri net R represented graphically

Given that the Petri net models the parallel execution of the acyclic code based on control dependences and due to the properties of acyclic CFGs and derived CDGs (see [4] and [6]), the Petri nets used by this algorithm present a number of properties. The net is 1-safe (ie. there will be at most one mark on any place in the net at any time). During a parallel execution every transition fires at most once, every place will not be executed more than once and the parallel execution always finishes with a single mark deposited in p_X which has no output transitions (as the sequential execution always finishes in the *eXit* vertex). The Petri net doesn't have any transitions with multiple input places (ie. it doesn't contain any `join`s, just `fork`s). There are no arcs in the net with multiplicity greater than 1 (ie. for any pair $\langle$transition, place$\rangle$ or $\langle$place, transition$\rangle$ there is at most one arc connecting them, which is enforced by the definition of the O function as a *set* and not as a *collection* as usual).

Care is taken to ensure that these properties will be preserved during the optimization stages, where the net topology will be changed.

4.3 Topological Information

The *reachability* notion that we have defined for directed graphs in section 2.1 cannot be extended in general for Petri nets, but given the specific properties of the net used by this algorithm, specifically the one that states that every transition in the net has no more than one incoming arc (only `fork`s, no `join`s), we can provide a recursive definition for a function that we call $REACH_p$ based on the net topology:

$$REACH_p : P \cup T \rightarrow 2^P$$

$$REACH_p(p_i) = \{p_i\} \cup \left(\bigcup_{t_{i-j} \in O(p_i)} REACH_p(t_{i-j})\right)$$
$$REACH_p(t_{i-j}) = \bigcup_{p_k \in O(t_{i-j})} REACH_p(p_k)$$

For the initial net topology in this example, this function would perform the mapping defined in table 2. Note that (for this initial net topology), $REACH_p(p_i)$ gives the set of places that represent the vertices on the CDG reachable from vertex i, and $REACH_p(t_{i-j})$ gives the set of places represent the vertices reachable on the CDG from the edge with label $i-j$.

Examining the CFG we find that (for all possible sequential executions) when some branches are taken, there are some vertices that definitely won't be executed. For example, when branch 1→2 is taken we know that vertex 3 won't get to be executed. Adapting this concept to the Petri net (for all possible parallel executions), we define the NEG_p function to map every place p_i in the net to the set of transitions that when fire avoid the execution of p_i definitely.

$$NEG_p : P \rightarrow 2^T$$

$$NEG_p(p_i) = \{t_{j-k} \mid \begin{array}{l} p_i \neq p_j \wedge \\ p_i \in REACH_p(p_j) \wedge \\ p_i \notin REACH_p(t_{j-k})\} \end{array}$$

(Note that t_{j-k} is an output transition from p_j). Table 3 shows the NEG_p function for this example.

p_i	$REACH_p(p_i)$
p_E	$\{p_E, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_X\}$
p_1	$\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9\}$
p_2	$\{p_2\}$
p_3	$\{p_3\}$
p_4	$\{p_4, p_5, p_6, p_7, p_8, p_9\}$
p_5	$\{p_5\}$
p_6	$\{p_6, p_7, p_8\}$
p_7	$\{p_7\}$
p_8	$\{p_8\}$
p_9	$\{p_9\}$
p_{10}	$\{p_{10}\}$
p_X	$\{p_X\}$

(a)

t_{i-j}	$REACH_p(t_{i-j})$
t_{E-1}	$\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_X\}$
t_{1-2}	$\{p_2, p_9\}$
t_{1-3}	$\{p_3, p_4, p_5, p_6, p_7, p_8, p_9\}$
t_{2-9}	$\{\}$
t_{3-4}	$\{\}$
t_{4-5}	$\{p_5, p_6, p_7, p_8, p_9\}$
t_{4-10}	$\{\}$
t_{5-6}	$\{\}$
t_{6-7}	$\{p_7\}$
t_{6-8}	$\{p_8\}$
t_{7-9}	$\{\}$
t_{8-9}	$\{\}$
t_{9-10}	$\{\}$
t_{10-X}	$\{\}$

(b)

Table 2: $REACH_p$ for the initial net topology

p_i	$NEG_p(p_i)$
p_E	$\{\}$
p_1	$\{\}$
p_2	$\{t_{1-3}\}$
p_3	$\{t_{1-2}\}$
p_4	$\{t_{1-2}\}$
p_5	$\{t_{1-2}, t_{4-10}\}$
p_6	$\{t_{1-2}, t_{4-10}\}$
p_7	$\{t_{1-2}, t_{4-10}, t_{6-8}\}$
p_8	$\{t_{1-2}, t_{4-10}, t_{6-7}\}$
p_9	$\{t_{4-10}\}$
p_{10}	$\{\}$
p_X	$\{\}$

Table 3: NEG_p

This means that during any parallel execution if p_i is not to be executed, there will be one (and just one) transition in $NEG_p(p_i)$ that fires. Otherwise p_i is executed and no transition in $NEG(p_i)$ fires:[4]

$$\forall p_i \in P,$$
$$\left(\exists^1 t_{j-k} \in NEG_p(p_i) \mid t_{j-k} \text{ fires}\right) \dot{\vee} \left(p_i \text{ is executed}\right)$$

This property is demonstrated in [6] for Girkar's framework based on the CFG and CDG properties for acyclic code. This demonstration is valid in this framework for the Petri net built from the CDG as described in section 4.1. As the topology of the net will be changed during the optimization stage, care has been taken in order to preserve this property.

4.4 Execution Precedence

Let i and j be any two vertices in the CFG, which are in direct correspondence with places p_i and p_j on the net. We use $j \prec i$ to denote that p_j finishes execution before p_i begins, in all possible parallel executions in which both places p_j and p_i are executed. The function $PREC$ will be used to map a place p_i onto the set of places that are *known* to precede the execution p_i:

$$PREC : P \rightarrow 2^P$$

$$p_j \in PREC(p_i) \Rightarrow \text{we know that } j \prec i$$

Given the algorithm for parallel execution of the CDG as modeled by the Petri net, it can be demonstrated that[5]:

$$p_i \in REACH_p(p_j) \Rightarrow j \preceq i$$

This equivalence can be used to build the $PREC$ table as:

$$PREC(p_i) = \{p_j \mid p_i \in REACH_p(p_j) \wedge p_i \neq p_j\}$$

Table 4 shows the $PREC$ function resulting from the net in figure 4. This precedence relationship satisfies control dependence constraints only.

p_i	$PREC(p_i)$
p_E	$\{\}$
p_1	$\{p_E\}$
p_2	$\{p_E, p_1\}$
p_3	$\{p_E, p_1\}$
p_4	$\{p_E, p_1\}$
p_5	$\{p_E, p_1, p_4\}$
p_6	$\{p_E, p_1, p_4\}$
p_7	$\{p_E, p_1, p_4, p_6\}$
p_8	$\{p_E, p_1, p_4, p_6\}$
p_9	$\{p_E, p_1, p_4\}$
p_{10}	$\{p_E\}$
p_X	$\{p_E\}$

Table 4: Initial precedence "$\prec$" relationship

[4] Note that the "$\dot{\vee}$" denotes the logical *exclusive-or* operation, and "$\exists^1$" means "exists one and just one".

[5] $i \preceq j$ if $i \prec j \vee i = j$

5 Sequencing Code

The existing net executes code satisfying control dependence constraints (ie, only the places that would be executed in an equivalent sequential execution of the CFG are executed). Additionally, every data dependence $y\,\delta_d\,x$ constraints the execution of p_x to start only after the execution of p_y (if p_y is executed).

In this stage, an optimization algorithm will attempt to sequence code without loss of implicit parallelism to satisfy data dependences (and thus avoiding the use of explicit synchronization mechanisms whenever possible). The precedence relationship table will be used to check if a given data dependence is already enforced by the net topology or if it needs to be enforced in some other way.

The optimization algorithm is shown in figure 5. Figure 6 illustrates the incremental changes that the algorithm performs on the net as it handles data dependences on DDG. (Dashed arcs represent annotated data dependences that might have to be enforced by explicit synchronization.) Table 5 shows the precedence relationship that results from this optimization.

p_i	$PREC(p_i)$
p_E	$\{\}$
p_1	$\{p_E\}$
p_2	$\{p_E, p_1\}$
p_3	$\{p_E, p_1\}$
p_4	$\{p_E, p_1, p_3\}$
p_5	$\{p_E, p_1, p_3, p_4\}$
p_6	$\{p_E, p_1, p_3, p_4\}$
p_7	$\{p_E, p_1, p_3, p_4, p_6\}$
p_8	$\{p_E, p_1, p_3, p_4, p_6\}$
p_9	$\{p_E, p_1, p_2, p_3, p_4, p_6, p_7, p_8\}$
p_{10}	$\{p_E\}$
p_X	$\{p_E, p_{10}\}$

Table 5: $PREC$ table after code sequencing

6 Determining Essential Dependences

After sequencing code, unsatisfied data dependences will have to be enforced by explicit synchronization, but by doing so for some data dependences, other data dependences may be indirectly satisfied (thus avoiding the need for more explicit synchronization mechanisms).

Figure 7 presents Girkar's algorithm (modified to suit our framework) for determining which dependences are to be satisfied by explicit synchronization (called "essential dependences"). We annotate the *essential dependences* found by this algorithm in the $SYNCH$ set:

$$SYNCH = \{\langle p_j, p_i\rangle \mid j\,\delta_d\,i \text{ is essential}\}$$

Girkar's algorithm steps through all pairs of places in the net that can be both executed in a parallel execution, determining the actual enforced precedence relationships, and adding all unsatisfied data dependences to the $SYNCH$ set.

Figure 8(a) shows the final annotated net, while table 6 shows the final precedence relationship.

p_i	$PREC(p_i)$
p_E	$\{\}$
p_1	$\{p_E\}$
p_2	$\{p_E, p_1\}$
p_3	$\{p_E, p_1\}$
p_4	$\{p_E, p_1, p_3\}$
p_5	$\{p_E, p_1, p_3, p_4\}$
p_6	$\{p_E, p_1, p_3, p_4\}$
p_7	$\{p_E, p_1, p_3, p_4, p_6\}$
p_8	$\{p_E, p_1, p_3, p_4, p_6\}$
p_9	$\{p_E, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$
p_{10}	$\{p_E\}$
p_X	$\{p_E, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\}$

Table 6: Final $PREC$ table

7 Explicit Synchronization Mechanism

Considering any data dependence $i\,\delta_d\,j$ (j is data dependent on i) the execution of p_i must finish before p_j begins to execute in order to satisfy the dependence. If only data dependences were to be considered and if it was assumed that both p_i and p_j would always be both executed, it would suffice to have p_j to wait for p_i to finish executing. As control dependences are also considered, every time that p_j is going to be executed this wait must terminate if it is known that p_i will not be executed.

When any of the output transitions of p_i fires, we know that p_i has finished executing. When any of the transitions of $NEG_p(p_i)$ fires, we know that p_i will not be executed. Given this, we can define a function DDC that maps every place p_i on to a set of transitions that, when any one of them fires it is known that p_i has finished execution or that p_i will not be executed:

$$DDC : P \rightarrow 2^T$$

$$DDC(p_i) = O(p_i) \bigcup NEG_p(p_i)$$

Table 7 defines DDC for the current example.

p_i	$DDC_p(p_i)$
p_E	$\{t_{E-1}\}$
p_1	$\{t_{1-3}, t_{1-2}\}$
p_2	$\{t_{2-9}, t_{1-3}\}$
p_3	$\{t_{3-4}, t_{1-2}\}$
p_4	$\{t_{4-5}, t_{4-10}, t_{1-2}\}$
p_5	$\{t_{5-6}, t_{1-2}, t_{4-10}\}$
p_6	$\{t_{6-7}, t_{6-8}, t_{1-2}, t_{4-10}\}$
p_7	$\{t_{7-9}, t_{1-2}, t_{4-10}, t_{6-8}\}$
p_8	$\{t_{8-9}, t_{1-2}, t_{4-10}, t_{6-7}\}$
p_9	$\{t_{9-10}, t_{4-10}\}$
p_{10}	$\{\}$
p_X	$\{\}$

Table 7: DDC function tabled for the current example

For every place p_i such that there is at least one annotation $\langle p_j, p_i\rangle \in SYNCH$ we use Girkar's

```
1 FOR x = Entry, 1, 2, ..., eXit DO:
2   FOR y = x − 1, x − 2, ..., Entry DO:
3     IF y δ_d x THEN
        /* this dependence will fall within one of the following three possibilities: */
4       IF p_x ∈ REACH_p(p_y) THEN
          /* This data dependence is already satisfied due to control dependences or to previous code sequencing. Do nothing. */
5       ELSE IF ∃t_{i−j} | t_{i−j} ∈ I(p_x) ∧ p_y ∈ REACH_p(t_{i−j}) THEN
          /* it is possible to solve this dependence by sequencing code. */
6         FORALL t_{i−j} | t_{i−j} ∈ I(p_x) ∧ p_y ∈ REACH_p(t_{i−j}) DO:
7           CALL Delay(t_{i−j})
8       ELSE
          /* This data dependence might have to be satisfied by explicit synchronization. Leave it for the next stage. */
```

```
  PROCEDURE Delay(t_{i−j})
1   IF t_{i−j} ∉ NEG_p(p_y) THEN
2     Remove the arc connecting t_{i−j} to p_x.
      /* There exists one (and just one) exit place from t_{i−j} (let it be called p_z) from where p_y is reachable.
      Find that unique p_z. */
3     ∃¹p_z | p_z ∈ O(t_{i−j}) ∧ p_y ∈ REACH_p(p_z)
4     FORALL t_{z−k} | t_{z−k} ∈ O(p_z) DO:
5       IF t_{z−k} is not yet connected to p_x THEN
6         Connect t_{z−k} to p_x.
7     IF p_z ≠ p_y THEN
8       FORALL t_{z−k} | t_{z−k} ∈ O(p_z) DO:
9         CALL Delay(t_{z−k})
```

Figure 5: Algorithm for sequencing code

method to determine a boolean expression that evaluates to *true* when all conditions are met for p_i to start execution:[6]

$$\left(\bigvee_{t_{k-l} \in I(p_i)} t_{k-l}\right) \wedge \left[\bigwedge_{\langle p_j, p_i \rangle \in SYNCH} \left(\bigvee_{t_{m-n} \in (DDC(p_j) \setminus NEG_p(p_i))} t_{m-n}\right)\right]$$

For this example, the $SYNCH$ set is:

$$SYNCH = \{\langle p_5, p_9 \rangle, \langle p_9, p_X \rangle\}$$

The resulting expressions for p_9 and p_X are shown in table 8.

p_i	Execution conditions for p_i
p_9	$(t_{2-9} \vee t_{7-9} \vee t_{8-9}) \wedge [(t_{5-6} \vee t_{1-2})]$
p_X	$(t_{10-X}) \wedge [(t_{9-10} \vee t_{4-10})]$

Table 8: Execution conditions for places dependent on explicit synchronization

An explicit synchronization mechanism like the one illustrated in figure 9 can be used to implement the necessary synchronization for dependent places. The resulting net (from replacing the input arcs of the dependent places by the explicit synchronization mechanism) is shown in figure 8(b). We designate the explicit synchronization mechanisms by *synchronization variables*, and we refer to them as s_9 and s_X in this example.

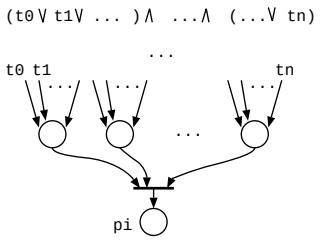


Figure 9: An explicit synchronization mechanism with its associated boolean expression.

Note that after placing the synchronization variables on the net, the conditions for the definition of the function $REACH_p$ (and any other function defined in terms of $REACH_p$) will **no longer be valid**. For this reason, its placement on the net is delayed until the optimizations based on such functions are finished.

In the code generated, we have implemented a synchronization variable as a counter that is initialized

[6] In this boolean expression, net transitions are treated as boolean variables that become *true* after firing.

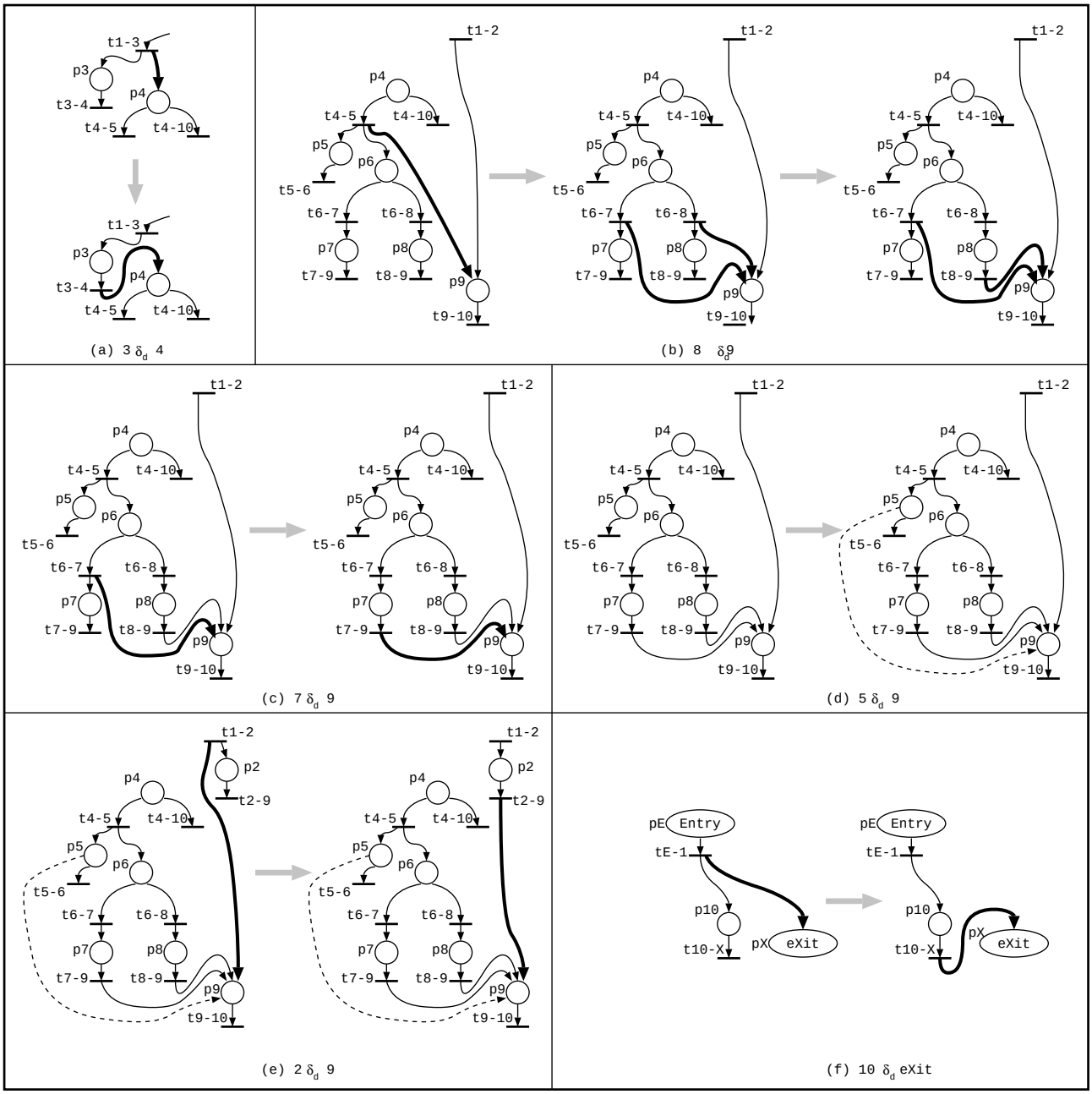
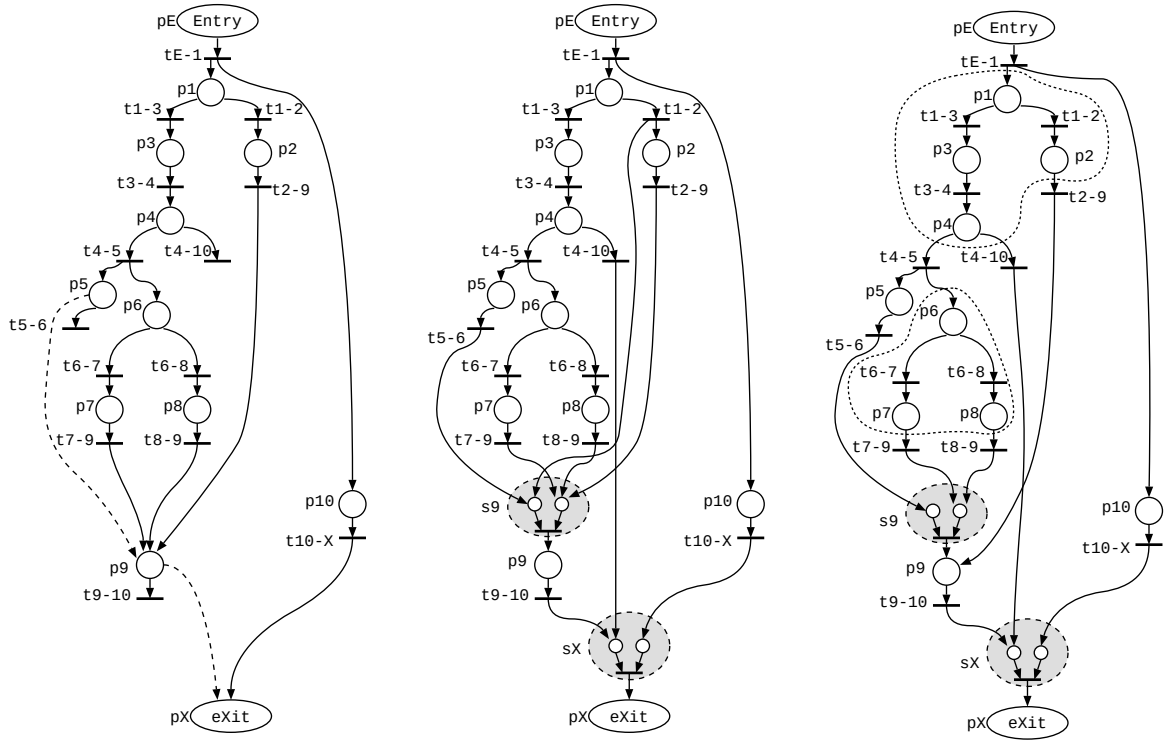


Figure 6: Minimizing the need for explicit synchronization by sequencing code

```
1  SYNCH = {}
2  FOR x = Entry, 1, 2, ..., eXit DO:
3    FOR y = x − 1, x − 2, ..., Entry DO:
4      IF x is reachable from y in the CFG THEN
5        IF NOT cont_path(p_y, p_y, p_x) THEN
           /* All paths from y to x on the CFG have at least one vertex z such that p_y ∈ PREC(p_z) ∧ p_z ∈
           PREC(p_x). This assures us that p_y precedes p_x. */
6          FORALL p_t | p_x ∈ REACH_p(p_t) DO:
7            Add p_t to PREC(p_x)
8        IF y δ_d x ∧ p_y ∉ PREC(p_x) THEN
           /* y δ_d x is an essential dependence that must be enforced by the use of explicit synchronization */
9          Add ⟨p_y, p_x⟩ to SYNCH
10         FORALL p_t | p_x ∈ REACH_p(p_t) DO:
11           Add p_t to PREC(p_x)
```

```
   FUNCTION cont_path (z)
     /* There is a path from y to z on the CFG. Returns True if there is at least one path from z to x on the
     CFG such that there is no vertex s on that path for which p_y ∈ PREC(p_s) ∧ p_s ∈ PREC(p_x) */
1    IF p_z has been visited before for the current ⟨y, x⟩ pair THEN RETURN False
2    IF p_y ∈ PREC(p_z) ∧ p_z ∈ PRECENDENTS(p_x) THEN RETURN False
3    IF p_z = p_x THEN RETURN True
4    FORALL p_t | ⟨z, t⟩ ∈ E_CFG DO:
5      IF cont_path(p_y, p_t, p_x) THEN RETURN True
6    RETURN False
```

Figure 7: Girkar's Algorithm for detecting essential dependences



(a) Final annotated net

(b) Explicit snychronization variables added to the net

(c) Optimized for multi-threaded code generation

Figure 8: Output net

to zero and is incremented by one every time a mark is deposited in any place inside the variable. When the counter reaches the number of places inside that variable, the place dependent on it begins execution. This approach is also described in [5].[7] The necessary code to atomically increment and test, and start the execution of the dependent place if necessary, is generated for every transition in the net that is linked to a synchronization variable.

To minimize updates to synchronization variables we define the IMP_t function for the annotated net (as in figure 8(a)) recursively as follows:

$$IMP_t : P \cup T \rightarrow 2^T$$

$$IMP_t(p_i) = \bigcap_{t_{i-j} \in O(p_i)} IMP_t(t_{i-j})$$

$$IMP_t(t_{i-j}) = \{t_{i-j}\} \cup \left(\bigcup_{p_k \in O(t_{i-j})} IMP_t(p_k) \right)$$

$IMP_p(p_i)$ gives the set of transitions that will surely fire after p_i is executed. $IMP_p(t_{i-j})$ gives the set of transitions that will surely fire after t_{i-j} fires (including t_{i-j} itself). Table 9 shows the IMP_t function for this example.

p_i	$IMP_t(p_i)$
p_E	$\{t_{E-1}, t_{10-X}\}$
p_1	$\{\}$
p_2	$\{t_{2-9}, t_{9-10}\}$
p_3	$\{t_{3-4}\}$
p_4	$\{\}$
p_5	$\{t_{5-6}\}$
p_6	$\{t_{9-10}\}$
p_7	$\{t_{7-9}, t_{9-10}\}$
p_8	$\{t_{8-9}, t_{9-10}, \}$
p_9	$\{t_{9-10}\}$
p_{10}	$\{t_{10-X}\}$
p_X	$\{\}$

t_{i-j}	$IMP_t(t_{i-j})$
t_{E-1}	$\{t_{E-1}, t_{10-X}\}$
t_{1-2}	$\{t_{1-2}, t_{2-9}, t_{9-10}\}$
t_{1-3}	$\{t_{1-3}, t_{3-4}\}$
t_{2-9}	$\{t_{2-9}, t_{9-10}\}$
t_{3-4}	$\{t_{3-4}\}$
t_{4-5}	$\{t_{4-5}, t_{5-6}, t_{9-10}\}$
t_{4-10}	$\{t_{4-10}\}$
t_{5-6}	$\{t_{5-6}\}$
t_{6-7}	$\{t_{6-7}, t_{7-9}, t_{9-10}\}$
t_{6-8}	$\{t_{6-8}, t_{8-9}, t_{9-10}\}$
t_{7-9}	$\{t_{7-9}, t_{9-10}\}$
t_{8-9}	$\{t_{8-9}, t_{9-10}\}$
t_{9-10}	$\{t_{9-10}\}$
t_{10-X}	$\{t_{10-X}\}$

Table 9: IMP_t function tabled for the current example

With this information, when calculating the expression of explicit synchronization for a place p_i from the set of its annotated data dependences we can substitute any transition t_{m-n} from $(DDC(p_j) \setminus NEG_p(p_i))$ by a transition from $IMP_t(t_{m-n})$ whenever this allows any simplification.

In this example, when calculating the synchronization expression for p_9 we know that t_{1-2} from $(DDC(p_5) \setminus NEG_p(p_9))$ can be substituted by t_{2-9}. This allows the application of usual algebraic simplification rules to simplify the expression from:

$$(t_{2-9} \vee t_{7-9} \vee t_{8-9}) \wedge [(t_{5-6} \vee t_{1-2})]$$

to:

$$t_{2-9} \vee [(t_{7-9} \vee t_{8-9}) \wedge [(t_{5-6})]]$$

[7] Other approaches are also described in [12], but are, in our opinion, less efficient for implementation on conventional multi-threaded systems.

Figure 8(c) illustrates the net with synchronization variables placed after this optimization.

8 Partitioning

The first step in partitioning consists in identifying sequential code in the net. This can be done by a trivial algorithm (such as one described in [10]) that gathers places into sequential regions. The net in figure 8(c) identifies sequential regions composed of more than one place by delimiting them with a dashed line. These regions can be replaced on the net by a single equivalent place.

Typically, as more data dependences are present in the input data, larger regions of sequential code are produced. By observing this property the following empiric partitioning algorithm was designed for use when the execution time of a sequential region (including the execution time of its output transitions) can be estimated relatively to the cost of thread management operations: The partitioning algorithm tries to find sequential regions that have an estimated execution time too small to justify its parallel execution. If such regions are found, artificial data dependences are introduced between those regions and other regions that are started in parallel with them. The parallelism extraction and optimization stages are then re-executed to produce a new solution with less parallel code.

In this example, if we consider that the sequential region $\{p_5\}$ has an estimated execution time that is too small to justify the cost of having t_{4-5} creating a new thread for executing $\{p_5\}$ or $\{p_6, p_7, p_8\}$, then the partitioning algorithm introduces artificial data dependences between $\{p_5\}$ and other regions potentially started by t_{4-5} ($\{p_6, p_7, p_8\}$ in this example, and always respecting that a data dependence $i\,\delta_d\,j$ is only meaningful if j is reachable from i on the CFG) and re-executes the parallelism extraction stages. Figure 10(a) illustrates the artificially introduced data dependences while figure 10(b) shows the resulting net.

9 Conclusion

In this paper we presented a framework for determining useful parallelism in acyclic code regions. We showed, through the use of an example, the application of techniques for parallelism extraction with optimizations directed towards multi-threaded code generation.

By attempting to sequence code to satisfy data dependences while preserving program correctness based on information extracted from the net topology, combined with Girkar's algorithm for determining essential data dependences, we are able to eliminate the need for use of explicit synchronization to satisfy most data dependences, achieving better results.

The behavior of this optimization technique allows it to be used in a simple partitioning scheme to artificially increase the granularity of the parallelism extracted up to a desired level, which is essential when using multi-threaded systems with high overheads in thread management operations.

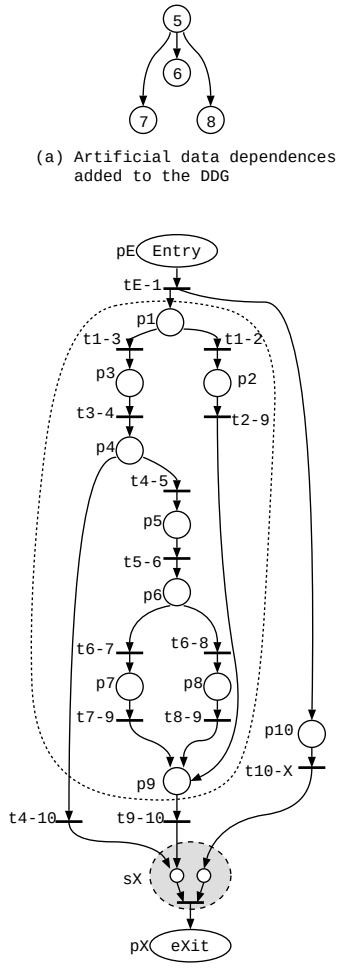


Figure 10: Partitioning (artificially sequencing $\{p_5\}$)

10 Acknowledgment

The authors would like to thanks Paulo Reis for his useful suggestions in improving the readability of this paper.

This work was partially supported by ESPRIT Project OMI/GLUE 6062.

References

[1] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh and V. Sarkar, "A Framework for Determining Useful Parallelism", *International Conference on Supercomputing*, St. Male, France, July 1988.

[2] U. Barnjee, R. Eigenmann, A. Nicolau, "Automatic Program Parallelization", *Proceedings of the IEEE*, 81(2), February 1993.

[3] C. Carvalho, J. Luís, J. Delgado, "Design of the Paralleliser for a Shared Memory Target", INESC, December 1993.

[4] R. Cytron, J. Ferrante, V. Sarkar, "Experiences using control dependence in PTRAN", *Languages and Compilers for Parallel Computing*, D.Gelertner, A. Nicolau, and D.A. Padua, Eds. Cambridge, MA: MIT Press, 1990.

[5] M. Furnari, "Managing Parallelism in Lisp: Problems and Perspectives", *Parallel Computing From Theory to Sound Practice*, IOS Press, 1992.

[6] M. Girkar, "Functional Parallelism, Theoretical Foundations and Implementations", CSRD Report 1182, Ph.D thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, December 1991.

[7] M. Girkar and C.D. Polycronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs", *IEEE Transactions on Parallel and Distributed Systems*, vol.3, no. 2, March 1992

[8] M. S. Lam, Robert P. Wilson, "Limits of Control Flow on Parallelism", *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Cost, Australia, May 1992.

[9] T. Lengauer, Robert Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph", *ACM Transactions on Programming Languages*, vol. 1, no. 1, July 1979.

[10] J. Luís, "Paralelização Automática de ANDF/TDF" ("Automatic parallelisation of ANDF/TDF" in Portuguese), M.S. thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, October 1994.

[11] S. Macrakis, "Virtual Binary: An Enabling Technology for Software and Hardware Innovation", *ANDF Technology Collected Papers*, vol. 4, OSF Research Institute, December 1993.

[12] J.E. Moreira, C.D. Polychronopoulos, "On the Implementation and Effectiveness of Autoscheduling", CSRD Report No. 1372, Technical Report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, May 1994.