FP-MAP - AN APPROACH TO THE FUNCTIONAL PIPELINING OF EMBEDDED PROGRAMS

Ireneusz Karkowski and **Henk Corporaal**
Delft University of Technology

Technical Report No. 1-68340-44/1997/06

Abstract

Practice shows that increasing the amount of instruction level parallelism offered by an architecture (like adding instruction slots to VLIW instructions) does not necessary lead to significant performance gains. Instead, high hardware costs and inefficient use of this hardware may occur. Mapping embedded applications onto multiprocessor systems forms a very interesting extension to ILP. In this report we propose a functional pipelining approach to the mapping of embedded programs written in ANSI C onto a pipeline of application specific processors. Our novel functional pipelining algorithm has low computational complexity and was especially developed to form the parallelization engine of a (semi)-automatic system for multi-processor embedded system design. The report explains the proposed algorithm and demonstrates its applicability.

Keywords: *heterogeneous multi-processors, parallelization techniques, high performance embedded system design*

Chapter 1

Introduction

Previous research has proven the existence of the large potential of fine-grain [15] and coarse-grain [10] parallelism in sequential programs. Unfortunately this potential is not always easy to exploit. In practice, when exploiting exclusively the instruction level parallelism (ILP), only average speedups in the range of 2-5 are observed [9]. This is far less than the parallelism upper bounds predicted by trace analysis ($\approx$60, [15]). We are confronted with a fundamental question about the limitations of the ILP exploitation methods. The obstacles seem to lie both in the area of software parallelization techniques, as in the architecture of the nowadays ILP processors (i.e. superscalar, VLIW).

In the past, numerous ILP exploitation methods were proposed and used. Among them, the techniques for the ILP in loops received most attention. The software pipelining [13] is the most popular one in this class. It has been successfully applied during code generation for the VLIW processors. Unfortunately the method has its limitations. Namely, few software pipelining algorithms are able to effectively schedule loops which lie high in the call tree of the program and/or contain control constructs. This significantly limits their application scope. Handling of conditional statements received most attention. In the past, several solutions have been proposed. *Hierarchical reduction* [12] hierarchically reduces conditional statements to single nodes having resource requirements being the union of both branches and the latency of the longer branch. If a much longer branch is much less frequently executed, a very inefficient schedule results. The *All Paths Pipelining* [14] extracts different iteration paths in a loop body, applies software pipelining to each path in isolation and inserts code that is executed from one schedule to another when a subsequent iteration executes a different path. While in many cases quite successful, the algorithm exhibits for some loops a substantial code expansion. Another software pipelining algorithm, called *Enhanced Pipeline Scheduling* [6, 7], uses extremely aggressive speculative scheduling combined with register renaming to pull up as many future computations as possible, including over loop-back branch moves. Disadvantage of the approach is its inability to deal with complex resource patterns [1]. Inefficiencies in the kernels generation have been reported as well. In addition, on architectures where the FU pipelining is visible at the architectural level (as is the case with our MOVE ASIP processors [5, 9]) the implementation of the *Enhanced Pipeline Scheduling* is tedious.

The conditionals are not the only high level language constructs hampering the effective software pipelining. Loops and function calls present in the loop's body form even a greater challenge. While loop unrolling and function inlining can be used in some cases, no good universal solution exists. Application of all these methods may result in a substantial code expansion, which is usually undesired in the "embedded" context.

Another more fundamental problem arises. To further exploit the parallelism, all possible control paths through the program should be scheduled separately in parallel. Clearly, this leads to disastrous code size increase, and becomes even infeasible in the presence of the loops with compile time unpredictable loop counts. This problem is serious.

Also on the hardware side several problems tempering effective exploitation of the ILP exist. The most important one is that of the complexity of the internal interconnection network of the known ILP processors. It increases rapidly with the number of available FUs. For example on VLIW processors this network complexity is quadratic in the number of FUs ([5], chapter 4). The run-time instruction scheduling hardware of the superscalar processors reaches complexities difficult to handle even faster. This results in large area con-

sumption and puts limitation on the achievable clock cycles. The MOVE (Transport Triggered Architecture) processors, remove most of the limitations of the traditional ILP processors. Still however, by increasing number of available FUs, we also obviously increase the load on the buses and therefore the transport delays. This load can be reduced, but exclusively by some very sophisticated architectural measures (for example internal clustering of FUs, [4], section 7.4.2).

Clearly, at certain moment the limit on the ILP exploitation is reached. Further progress is only possible by a combined fine- and coarse-grain parallelism exploitation. The advent of sub-micron processing, allowing integration of several millions of transistors on a single carrier, has brought new opportunities in the embedded system design. The embedded multi-processors become nowadays a feasible and very attractive option. Especially if the system consists of heterogeneous processors, each having the functionality optimized for the part of the computation that it has to perform. This way both fine- and coarse-grain parallelism can be efficiently exploited.

What we need is a set of algorithms which can be used for the coarse-grain partitioning of the embedded application onto different multi-processors architectures. In the past, several parallelizing compilers for symmetric shared memory multi-processors were implemented [2]. They partitioned the iteration space of the loops in a sequential program among available processors. Code transformations were usually necessary to ensure correct parallel execution of those partitions. Unfortunately, in the context of the embedded systems, often the symmetric shared memory architecture is not cost-efficient, because it results in large hardware duplication and requires a complex memory system. Also exploiting only data parallelism in loops may not be sufficient to justify the use of the multi-processor framework. Some other, coarse-grain parallelism exploitation methods, oriented towards simpler multi-processor architectures, are necessary.

This report is devoted to one such parallelization technique - the so called "functional pipelining". The technique could be shortly defined as a mapping of a selected loop nest in a sequential program onto a pipeline of processors, in such way that a pipelined style of execution is obtained (analogous to the way automobile assembly line works). We propose an efficient algorithm for automatic "functional pipelining" of a selected loop nest in a program written in ANSI C onto a pipeline of the application specific processors. While applicable to any loop nest, the technique was especially designed to map loop nests laying very high in the call hierarchy of a program and containing all kinds of control statements. We plan to use it as a component in our multi-processor embedded system design framework.

The remainder of the report is organized as follows. In chapter 2 we provide necessary definitions and formally state the problem. Chapter 3 is devoted to the presentation of our mapping algorithm called **FP-Map**. Its performance is demonstrated on an example C code in chapter 4. In chapter 5 we discuss how to extend the scope of the application of the algorithm. Chapter 6 concludes the report.

Chapter 2

Preliminaries

Our problem could be formally stated as that of finding a mapping Γ of a program graph $G_P(V, E)$ onto an architecture template graph $G_A(P, C)$, such that the resulting hardware-software solution satisfies the price-performance specification for the design. In the graphs, V is the set of program statements, E the set of data and control dependencies between the statements, P the set of processing elements and C the interconnection network between them. The problem of finding a good mapping for arbitrary G_P and G_A is clearly too complex. The problem which we address in this report is simpler, however, still far from being trivial. We start namely with a subgraph of G_P containing a set of vertices from one of the loop nests in the program. Also an architecture template graph is given. Our goal is to find a mapping, which optimizes the performance of the system.

Let us start with few definitions. Usually there exists a number of data dependencies between statements in a loop body. The calculation of these dependencies in embedded programs written in ANSI C has been discussed in [11]. The data dependencies can be divided into two categories:

Definition 1 *The* intra iteration *data dependency is a dependency between the executions of two statements belonging to the same loop iteration. Conversely, the* inter iteration *dependency exists between statements executed in different iterations. A* backward inter iteration *dependency is an* inter iteration *dependency going from a statement b to a, with the property that there exists a path $a \rightarrow b$ composed of the* intra iteration *data dependencies.*

Definition 2 *Let Λ be a computer system consisting of N_{proc} processing elements (or shortly processors), connected with a number of unidirectional and asynchronous communication links.* Functional pipelining *is a mapping that maps a graph G_L containing vertices from the body of a loop nest in a program (a subgraph of G_P), onto Λ in such a way that the resulting architecture is processing several iterations of the loop in a pipelined fashion.*

Consider the sequential program in figure 2.1. This loop can be mapped onto a 3-processor computer system,

```
forall (samples) {
  a=do_fft(sample);
  b=freq_mult(a,c[sample_id-2]);
  c[sample_id]=bandpass(b);
  }
```

Figure 2.1: Example loop nest

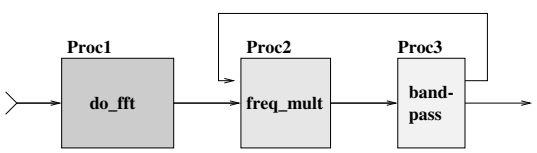

Figure 2.2: The 3 processor architecture implementing the code from figure 2.1

in which the *do_fft* function is mapped onto the first processor in the chain, the *freq_mult* onto the second one, and *bandpass* operation onto the third one (figure 2.2). Clearly when we partition data dependent statements in G_L between two processors in Λ, a communication link has to be created. In our case the interconnection network will consist of three unidirectional links, $proc_1 \rightarrow proc_2$, $proc_2 \rightarrow proc_3$ and $proc_3 \rightarrow proc_2$. The system can start working on the next iteration before the previous one(s) are finished. At any time (excluding

the startup) the computer system will be processing 3 different samples in a pipelined fashion. The existing *backward inter iteration* dependency going from the third to the second statement requires some attention. Thanks to the fact that the iteration distance of this dependence is equal 2, no pipeline stalls (assuming a balanced processor load) are necessary - when the second processor starts processing iteration i, the third processor will be already done with iteration $i-2$ and starting iteration $i-1$. As the result the value of $c[sample_id-2]$ can be retrieved via the backward communication link between both processors. Note that this would not be possible without a stall if the iteration distance of this dependency was 1.

Clearly, by successfully applying *functional pipelining* to parts of an embedded program substantial speedup can be obtained. In the next chapter we will show how this mapping can be found, and how to estimate the speedup.

Chapter 3

The methodology

As stated in definition 2 our goal is to map a body of a loop nest onto a computer system Λ consisting of N_{proc} processing elements. In this chapter we will show how this can be obtained by first making a graph model of the loop and mapping this graph onto Λ. The process consists of the following steps:

1. Build a dependency graph $G_{LR}(V, E)$.
2. Perform the mapping.
3. Estimate the speedup.

Figure 3.1 illustrates the mapping process graphically. In the following sections the algorithm will be discussed in detail.

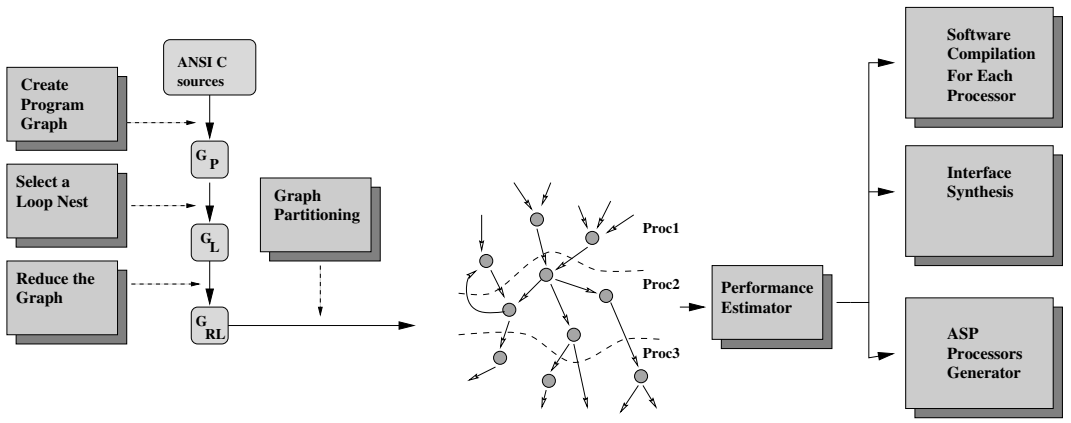

Figure 3.1: The *functional pipeling* methodology

3.1 Build a dependency graph

The reduced dependency graph $G_{LR}(V, E)$ is a directed graph, where V is the set of vertices and E the set of edges. The name "reduced" comes from the fact that it models groups of operations from the G_L by single vertices, and that it excludes some "relatively unimportant" operations (as will be described in section 3.2). With every vertex and edge a number of coefficients is associated. In case of vertices we have a 5-tuple (l, ρ, c, g, b) and in case of edges a single value $\vec{p}$. Their meaning is explained in the table 3.1. Vertices are created according to the following *vertex creation rules*:

Symbol	Description
$l(v)$	time necessary to perform the operation(s) modeled by the vertex (in clock cycles)
$\rho(v)$	probability that a given operation(s) will be executed during a single iteration
$c(v)$	id of the conditional statement to which the vertex belongs or $\perp$ if not conditional
$g(v)$	value of the conditional statement test expression, which selects the branch where v belongs
$b(v)$	equal $\top$ if the vertex lies on the longest branch of the conditional, $\perp$ otherwise
$\vec{p}(e)$	the iteration distance vector

Table 3.1: The vertex and edge coefficients in $G_{LR}(V, E)$

- All loop and call statements (including all statements within it) are mapped into single vertices. The latency l of these vertices is set to the maximum time that an execution of this loop or call takes.
- For every conditional statement we create a *test* vertex with latency l being the time necessary to calculate the test expression. Then we include all vertices on the branches (recursively applying the *vertex creation rules*). Also the values of c, g, b are appropriately set for all the vertices.
- All other groups of statements located between the above mentioned constructs are mapped into single vertices with the latency calculated as the latency of the most critical path in the instruction level data dependency graph.

To each node the execution probability $\rho(v)$ is attached as measured during profiling. For edges we only have a single value $\vec{p}$ which is the dependence distance. An edge between vertex u and v denotes that the operation corresponding to v in iteration $\vec{i}$ should be executed at least $l(u)$ clock cycles after the operation corresponding to u in iteration $\vec{i} - \vec{p}$. There is also an edge between two vertices if there exists a control dependency between those vertices. We assume that every vertex $v \in V$ represents an atomic computational entity to be allocated on a processor, and that the instructions inside it will be scheduled separately. Therefore by the longest branch we mean the branch B with maximal sum of vertex latencies $\sum_{v \in B} l(v)$.

3.2 Additional graph reduction

As we wrote in the previous section the G_{LR} is a "reduced" graph in the sense that multiple instructions from the G_L are mapped onto its vertices. Additional reduction of the graph can be obtained by skipping vertices which are "relatively unimportant" - have low execution probability $\rho(v)$ and/or small latency $l(v)$. Considering these vertices would lead to situations where unnecessarily long schedules are generated in cases when we have a much longer branch which is almost never executed. We approach this problem by optimizing the "most common case". In its "stable" state the pipeline will only execute the "most common case" statements. When one of the other branches is triggered, the whole pipeline will possibly have to be stalled. To obtain such an effect, only the vertices with the execution probability $\rho(v)$ higher than a user given cut value P_α will be considered. Further reduction can be obtained by in addition merging some "relatively short" vertices with their predecessors. This leads to the following definition of the *the MCC set* (most common case set):

$$V_{MCC} \equiv \{v \in V \mid \rho(v) \geq P_\alpha \wedge \frac{|V| \cdot \rho(v) \cdot l(v)}{\sum_{u \in V} \rho(u) \cdot l(u)} \geq \beta\} \tag{3.1}$$

where β is a user given cut value. In practice the value of P_α should be low enough to avoid excluding any essential computation. Also the value of β should be carefully selected to avoid precision loss. When creating the $G_{LR}(V, E)$ we should consider only *the MCC set* vertices. The skipped vertices form the subsets V_α, V_β. The reduced graph serves as the input to the mapping step. Note that this graph may still contain complete conditionals.

3.3 Perform the mapping

In the discussion below we assume that the code contains only non-nested multi-branch conditional statements (**switch** statement of the C language). This limitation has been introduced here only to increase the readability of our presentation. The **FP-Map** algorithm can be easily extended to handle nested conditionals.

Our goal is to map our loop nest onto the computer system Λ consisting of N_{proc} processors, in such a way that a *pipelined* style of execution is obtained. A good mapping should result in maximal speedup in comparison with the sequential implementation. In other words it should minimize the so called *initiation interval* of the pipeline (the time between two subsequent iterations of the loop are initiated). Obviously, the value of the *initiation interval* is equal to the time that the most loaded processor in the pipeline needs to perform its part of the computation.

Definition 3 *The* load $L(n)$ *of the processor* n *is defined by:*

$$L(n) \equiv \sum_{v \in V_n} (c(v) = \perp \ ? \ l(v) : (b(v) = \top \ ? \ l(v) : 0)) \tag{3.2}$$

where V_n *is the subset of vertices which are mapped onto processor* n*. The expression* $(expr \ ? \ value_1 : value_2)$ *should be understood as follows: if* $expr$ *is* **true** *then the value of the whole expression is* $value_1$*, otherwise* $value_2$

Our mapping problem is equivalent to the following partitioning problem:

Definition 4 The partitioning*: Let* $G_{LR}(V, E)$ *be the dependency graph as defined in section 3.1 and* N_{proc} *the number of processors. Partition the vertices* $v \in V$ *between the processors in such way that the load on the most heavily loaded processor is minimized. In other words find the optimal mapping* $(P : v \rightarrow \{1 \ldots N_{proc}\})$*. In addition obey the following rules:*

1. *All data dependency edges with exception of the backward edges go from processor* i *to* j*, where* $j > i$*.*
2. *All the vertices laying on a cycle* ϕ *of* G_{LR}*, are distributed among at most* $\sum_{e \in \phi} p(e)$ *processors.*
3. *If a conditional statement is partitioned among two processors* $i, j, \ j > i$ *then the value of the "test" expression should be forwarded to the processor* j*, and a cost of an extra branch statement added there.*

The rule 1 guarantees that all the operations are executed in the proper order (as defined by the data dependencies). The second one ensures that no pipeline stalls caused by the inter-iteration dependencies occur. The last rule takes care of the proper conditional statements partitioning between processors - clearly a continuing processor should in case of a partitioned conditional execute an extra branch instruction. This may introduce an extra cycle or two to the load on that processor.

Consider a *decision* version of the optimization problem from definition 4. Given a value of the load L on the most heavily loaded processor, the question is: *"Can a partitioning be found such that the load on each processor is not higher than* L*?"*. The problem is NP-complete. This can be shown by *polynomial-time reduction* from another NP-complete problem - the *bin packing problem* ([8], p.226). Therefore we propose a heuristic, which is a derivative of both the FFD (first fit decreasing) algorithm used for the *bin packing problem* and the *modulo-scheduling* algorithm used for *software pipelining*.

Consider the *bin packing problem*. There are N bins, each with capacity L (height) and a number of blocks v_i with length and width equal to that of the bins. The height of every block is $l(v_i), \forall_{v_i} l(v_i) \leq L$ and $\sum_{v_i} l(v_i) \leq N * L$. The *decision problem* which has to be solved is *"Can the blocks be placed in the bins?"*. The *FFD* algorithm places the largest blocks first in a bin until no blocks can be found which fit there. Then the procedure goes to the next bin. If after processing all bins there are no blocks left, the answer is "Yes", otherwise the answer is "No".

We notice several similarities between the *bin packing problem* and the partitioning problem defined in definition 4. The bins are the processors, the blocks are vertices in the G_{LR}, the heights their latencies. The

bin capacity is equivalent to the processor capacity which we define here to be the maximum *load* (def. 3) that a processor can accept. If we knew that the *initiation interval* of our pipeline was L, our partitioning problem would also reduce to a *decision problem*: *"Can the vertices of the G_{LR} be partitioned among the processors if the maximal capacity of each processor is L?"*. This observation forms the motivation for the following strategy:

- Determine the search range for the values of the bin capacities.
- Perform a search over this range. In every step try to "pack" the vertices in the bins of capacity L.
- Use the "packing" with minimum value of L as the solution to the partitioning problem.

Let us concentrate now on the "packing". The *FFD* strategy seems to be very attractive - vertices with longest delay should be placed first in the bins. To satisfy data dependencies, which exist among the vertices in the G_{LR}, several extensions are necessary however:

- The vertices should be placed in a bin only if the vertices on which they are data dependent have already been placed in the current or one of the previous bins.
- If a candidate vertex belongs to a cycle ϕ of G_{LR} with $pc \equiv \sum_{e \in \phi} p(e) = 1$ then all vertices $v \in \phi$ should be placed in this bin. If $pc > 1$ then the vertices $v \in \phi$ can be distributed among at most pc bins.
- If a candidate vertex belongs to a conditional statement, its effective load (or height in the bin's terminology) may be smaller than its real latency. Consider a situation when a number of vertices $v \in V_x$ with total load X from one, and a number of vertices $v \in V_y$ with total load Y from another branch have already been placed in a bin. The effective load of a candidate vertex v_z belonging to the second branch will be:
$$l_{eff}(v_z) = \min\{l(v_z), \max\{Y + l(v_z) - X, 0\}\} \tag{3.3}$$
The above formula can be generalized for the multi-way conditional statement.
- When starting a new bin, we have to check for unfinished (non-completely placed in the previous bins) conditional statements. We have to add a constant number of cycles (for the extra branch) to the first vertex of the conditional, which is placed in next the bin.

All the goals can be achieved by an extended version of the *list scheduling* algorithm, wrapped inside a search over values of the bin capacity L. Figures 3.2-3.5 contain its pseudo-code. The table 3.2 summarizes all used symbols (except of those introduced already in section 3.1, table 3.1). The top level procedure **FP-Map** first finds a set of *strongly connected components* (SCC) in the graph. Note that some SCCs may consist of single vertices and that $\bigcup_{s \in S} = V$. After calculating some temporary coefficients it calculates the lower and upper bound values of our search range (L_{min}, L_{max}). Then the search for L over this range is performed. Note that it is not possible to use the binary search for this purpose, because the "packing feasibility" is not a monotonous function of L. Instead we use a linear search starting at L_{min} and with step of ϵ. At every step the **Packing** procedure is used to attempt the "packing". Because of the upper bound tightening done at every step (as explained further), only a small number of steps is expected.

The values of (L_{min}, L_{max}) are calculated as follows. The lower bound on the value of the *initiation interval* L_{min} (minimum *initiation interval*) can be calculated relatively easy using following three observations:

- The value of the L_{min} will certainly not be lower then the maximum latency of any operation/vertex:
$$L_{min,Op} = \max_{v \in V} l(v).$$
- Assuming the absence of conditional statements, in best case all the computation will be partitioned among the processors in such a way that the load on every processor will be equal. Therefore:
$$L_{min,Res} = \left\lceil \frac{\sum_{v \in V} l(v)}{N_{proc}} \right\rceil.$$

Symbol	Description
s, S	a SCC and a set of all SCCs
ϕ, Φ	a cycle in the graph, set of all cycles
c, C	a conditional, set of all conditionals in the program
$lp(v)$	longest path from vertex v to one of the sink vertices (excluding backward edges)
L, L_{min}, L_{max}	current bin capacity, the lower and upper bounds of the search region
L_t	required bin capacity as set by **Packing**
P	a partitioning, mapping $(P : v \to \{1 \dots N_{proc}\})$
ϵ	precision of the linear search
$Ready$	priority list keeping vertices which are "ready" to be placed in the bin
$part(s), pc(s)$	maximum number of bins in which the SCC s has to fit
$L_{scc}(s)$	total size of not yet placed vertices of s
n	current bin (processor) number
L_{bin}	total size of vertices already in the current bin
$sFlag(s)$	temporary, takes value of $\top$ if one of vertices $v \in s$ has been placed in the current bin
l_{eff}	effective size of a candidate vertex
L_{left}	space left in the current bin
G_c	a set of branches of a conditional statement c
$L_{bin}(c, g)$	a temporary variable, which keeps for each conditional c and its branch g the total size of vertices from this branch which have been allocated in the current bin
T	set of test vertices of partially allocated conditionals
η	number of cycles necessary to perform a branch

Table 3.2: Summary of symbols used in the *pseudo-code* of the **FP-Map** algorithm

In case that the G_{LR} contains conditional statements, the calculation of the total load becomes slightly more complicated, because we have to take only the latency of vertices laying on the longest branch of every conditional:

$$L_{min,Res} = \left\lceil \frac{\sum_{v \in V} (c(v) = \bot \ ? \ l(v) : (b(v) = \top \ ? \ l(v) : 0))}{N_{proc}} \right\rceil .$$

- The G_{LR} may contain cycles caused by backward inter-iteration dependencies. A cycle ϕ has to be partitioned among at most $\sum_{e \in \phi} p(e)$ processors, to avoid processor stalls. All cycles with $\sum_{e \in \phi} p(e) > 0$ define the third lower bound on the value of L_{min}:

$$L_{min,Rec} = \left\lceil \max_{\phi \in \Phi} \frac{\sum_{v \in \phi} l(v)}{\sum_{e \in \phi} p(e)} \right\rceil .$$

These three values together define the lower bound of our search interval:

$$L_{min} = \max\{L_{min,Op}, L_{min,Res}, L_{min,Rec}\}. \tag{3.4}$$

In worst case all the vertices will remain on a single processor. This may happen for example if there is a backward inter-iteration dependence with distance $p = 1$ going from the last loop body statement to the first one. Using this observation:

$$L_{max} = \sum_{v \in V} (c(v) = \bot \ ? \ l(v) : (b(v) = \top \ ? \ l(v) : 0)). \tag{3.5}$$

```
FP-Map(G_LR)
{
    Find all  strongly connected components s ∈ S in G_LR;
    Calculate the values of L_min, L_max;
    Remove backward edges in G_LR;
    Calculate the longest path  length lp(v) from every vertex v in G_LR
       to one of the sink vertices in the graph;
    L =  L_min;                                  // linear search over values of L
    do {
         P =  Packing(L, &L_max);                // attempt packing
         L =  L + ε;                             // go one step up
    } while (L < L_max);
    return P;
}
```

Figure 3.2: The mapping algorithm - top level view

The **Packing** procedure attempts to pack all vertices of the G_{LR} inside N_{proc} bins, each with size L. If successful it returns a partitioning P and sets value $L_t = L$. Otherwise, the obtained partitioning (with remaining nodes allocated in the last bin) and $L_t > L$ is returned. This way the upper bound value L_{max} is automatically tightened, speeding up the search process. In every step the candidate vertex is selected from a priority list of "ready" vertices by the **SelectVertex** procedure. The priority is calculated using the **Priority** function (figure 3.4). The update operation of the $Ready$ list in the **Packing** routine requires some explanation. We have to check if for the SCC s $part(s) == 1$ (recall that in this case the whole SCC s has to be placed in the bin), if yes check incoming dependency edges for all the vertices $z \in s$.

Procedures **InitIfBins, UpdateIfBins** and **FitInIfBins** allow calculation of the effective size of a candidate vertex belonging to a conditional statement. The **InitIfBins** procedure initializes work variables for all conditional statements in the G_{LR}. The function **FitInIfBins** calculates the effective size of a candidate vertex (recall eq. 3.3). The **UpdateIfBins** routine updates the working variables after a vertex has been placed in the bin.

It is interesting to point out the analogies between the **FP-Map** algorithm and the *modulo scheduling* algorithm used for *software pipelining*. We could call our method a "processor based" scheduling (as opposite to the "cycle" or "operation" based schedulers). We fill the processors instead of cycles. In place of the resource constraints only a single constraint on the maximum load on a processor is considered.

3.4 Computational complexity

The computational complexity of this algorithm is easily determined. The set of *strongly connected components* of G_{LR} can be calculated in $\mathcal{O}(|V| + |E|)$ time using the algorithm from [3], page 489. The longest paths from every vertex to all sinks can be calculated by the modification of the Bellman-Ford algorithm [3], page 532, which runs in $\mathcal{O}(|V||E|)$ time. The values of L_{min} can be calculated in $\mathcal{O}(|V|)$ time. The number of steps of the linear search does not directly depend on the problem size, and is expected to be low, due to the instant tightening of the upper bound value by the **Packing** procedure (via the L_t variable). The complexity of the "packing" procedure depends on the implementation of the *Ready* priority list. If the "heap" implementation is used ([3], page 151), any operation on the list will take $\mathcal{O}(\lg |V|)$ time. The complexity of the SSC and conditional code handling operations is $\mathcal{O}(1)$. This results in the total $\mathcal{O}(|V| \lg |V|)$ complexity for the **Packing** procedure. Since $\lg |V| < |E|$, the overall complexity becomes $\mathcal{O}(|V||E|)$.

```
Packing(L, L_t)
{
    forall (v ∈ V) P(v) = ⊥;                                          // unmap all nodes
    Ready = root nodes of V ;                                         // init the priority list
    forall (s ∈ S) {                                                  // init SCC temporary
        part(s) = pc(s) = max{1, min_(φ∈ s) Σ_{e∈φ} p(e)};            //   variables
        L_scc(s) = Σ_{v∈ s} ( c(v) = ⊥ ? l(v) : (b(v) = ⊤ ? l(v) : 0));
    }
    for (n = 1; n ≤ N_proc; n = n + 1) {                              // for all processors (bins)
        L_bin = 0;                                                    // initially the bin is empty
        forall (s ∈ S) sFlag(s) = ⊥;                                  // set no SCC allocated
        InitIfBins();                                                 // init cond. code handling
        while ( (v = SelectVertex(L−L_bin, &l_eff)) ≠ ⊥ ) {           // select a vertex which fits
            L_bin = L_bin + l_eff;                                    // update the bin's load
            P(v) = n;                                                 // assign the vertex
            UpdateIfBins(v);                                          // update conditionals
            if (v ∈ s | s ∈ S) sFlag = ⊤;                             // flag this SCC allocated
                                                                      // update the priority list
            Ready = Ready − v + {u ∈ V | P(u) = ⊥ ∧ ( ( u ∈ s|s ∈ S && part(s) > 1) ?
                                  (∀_{(w,u)∈ E} P(w) ≠ ⊥) : (∀_{z∈ s} ∀_{(w,z)∈ E} P(w) ≠ ⊥) ) };
        }
        forall (s ∈ S) if (sFlag(s) = ⊤) part(s) = part(s) − 1;       // update part(s) if SCC s
    }                                                                 // was allocated in this bin
    L_t = L_bin;
    forall (v ∈ V|P(v) == ⊥) {
        P(v) = n;                                                     // add remaining nodes to the
        L_t = L_t + l(v);                                             // last bin
    }
    return P;
}
```

Figure 3.3: Packing the G_{LR} for the given bins' capacity L

3.5 Speedup estimation

The estimated latency of the pipelined version of the loop can be calculated using the following equation:

$$L_{fp} = L \cdot (I_{tot} + N_{proc} - 1) + L_{stall}$$

where L is the *initiation interval* of the pipeline, I_{tot} total number of iterations in the loop and L_{stall} the upper bound on the total stall time of the pipeline. The pipeline may stall when some vertices $v \notin V_{MCC}$ are executed (recall section 3.2). The upper bound on total expected stall time is:

$$L_{stall} = \sum_{v \notin V_{MCC}} l(v) \cdot \rho(v) \cdot I_{tot}.$$

Then the estimated speedup which can be obtained by the *functional pipelining* of a given loop nest is then given by:

$$SPEEDUP_{fp} = \frac{I_{tot} \cdot L_{max}}{L_{fp}}, \tag{3.6}$$

where L_{max} is given by the equation 3.5.

The value of $SPEEDUP_{fp}$ represents only an estimate on the speedup which can be obtained. This for several reasons:

1. Obviously by the heuristic nature of the **FP-Map** algorithm.

```
SelectVertex (L_left, l_eff)
{
    forall (v ∈ Ready, highest priority (func. Priority()) node first) {
      l_eff = ((v ∈ s|s ∈ S && part(s) == 1) ? L_scc(s) : l(v) );
       if (c(v) ≠ ⊥)                                          // it's a conditional
           l_eff = FitInIfBins (v, l_eff) ;                   // calc. effective latency
       if (first vertex of unfinished conditional)
           l_eff = l_eff + η;                                 // add the branch overhead
       if (l_eff ≤ L_left)                                    // fits in the bin
          return v;
    }
    return ⊥;                                                 // no fitting vertex found
}

Priority (v)                                                  // calculate the priority
{                                                             //    of a vertex
    return ((v ∈ s|s ∈ S && part(s) == 1 && sFlag(s) == ⊤) ? ∞ :
                  ρ(v) · lp(v) · Σ_{u∈ s} l(u)/pc(s) );
}
```

Figure 3.4: Selecting vertices. The procedure selects the highest priority vertex which has effective latency (eq.3.3) smaller or equal than the space left in the current bin.

2. Not every execution of a vertex $v \notin V_{MCC}$ will cause a pipeline stall. If this happens on a processor which is less loaded then the maximally loaded one, a stall may be not necessary.

3. Actual time which a partition will take to execute may be different due to the local ILP exploitation. Once the real code for every partition will be generated it will be optimized by the a code scheduler. Nothing would stop it also from local software pipelining. We try to take these effects into account by using the profiling information for already scheduled and heavily optimized executable. After the **FP-Map** mapping the location of some operations may however completely change, leading to difficult to predict effects. In addition in the model we did not account for the communication overhead.

```
InitIfBins()                                          // initialize
{
    forall (c ∈ C) forall (g ∈ G_c)  L_bin(c, g) = 0;   // initially all branches empty
}

FitInIfBins(v, l_eff)                                 // calculate effective latency
{
    m = max_{g∈ G_c}  L_bin(c(v), g);                 // maximum load in all branches
    r = L_bin(c(v), b(v)) + l_eff;                    // new branch load
    if (r > m) return r − m;                          // this branch becomes now most loaded
    else return 0;                                    // zero effective latency
}

UpdateIfBins(v)                                       // update
{
    L_bin(c(v), b(v)) = L_bin(c(v), b(v)) + l(v);     // add vertex latency to the branch
}
```

Figure 3.5: Conditional code handling routines

Chapter 4

Mapping example

In this chapter we will present an example application of our algorithm. Consider the loop nest in figure 4.1 and $N_{proc} = 3$. There are two SCCs with $|s| > 1$ in the graph, $SCC_1 = \{c, C\}$ with $\sum p = 2$ and

```
for(i=0;i<10;i++)
  for(j=0;j<20;j++){
    int a,b,c,d,e,f,g,h,k;
    a=A[i][j];
    b=bkx(B[i][j]);
    U[i][j]=a;
    if(a>2){
      c=C[i][j-2]%3+rrl(0x0234);
      C[i][j]=sin(c);
      d=C[i][j]-2;
    } else {
      E[i][j]=^rll(E[i][j]+0x01);;
      d=E[i][j]%5;
    }
    D[i][j]=tew(d,4);
    if(a>b){
      e=R[i][j]-A[i][j];
      f=sqrt(e)+wt(A[i][j]);
    } else {
      g=L[i][j-1]-A[i][j];
      k=h*1.28+wt(g);
      I[i][j]=sqrt(k);
      L[i][j]=sin(k);
      f=sqrt(L[i][j])*0.2;
    }
    s=f/L[i][j];
  }
```

Figure 4.1: An example C source code. For readability reasons all the variables were renamed and privatized to avoid any *anti* and *output* dependencies

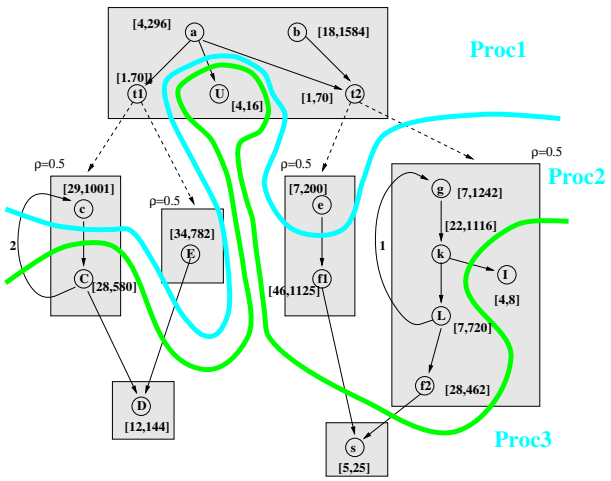

Figure 4.2: The G_{LR} for the example source code from figure 4.1. Only some coefficients are presented: $[l(v), priority(v)]$ for vertices, and p's for only backward edges. Dashed arrows represent control flow.

$SCC_2 = \{g, k, L\}$ with $\sum p = 1$. Also two conditional statements are present with *test* vertices $t1$ and $t2$. During profiling we observed the branch probabilities close to 0.5 for both conditionals. Therefore no vertices were excluded from the graph (i.e. $V = V_{MCC}$). The longest branches of the conditionals are: $br_1 = \{c, C\}$ with total latency of 57 and $br_2 = \{g, k, I, L, f\}$ with total latency of 68. The range of the search interval was calculated as follows:

$$L_{min,Op} = 28, \quad L_{min,Res} = \lceil 170/3 \rceil = 57, \quad L_{min,Rec} = \max\{\lceil 57/2 \rceil, 36\} = 36,$$

$$\hookrightarrow \quad L_{min} = 57, \quad L_{max} = 170.$$

First run of the **Packing** for the value of $L = 57$ resulted in a failure, but it produced a new upper bound value $L_{max} = 82$. After a number of steps of the search we arrived at the final value of $L = 65$. The steps of the **Packing** for this value of L are presented in table 4.1. The allocation process for each processor is shown separately. Eg:, on the second processor, node L is allocated as fourth and its effective load is 0 because its branch was still shorter than the other one, containing node f. The thick lines in figure 4.2 denote the partitioning boundaries.

Processor 1	Allocated nodes						
v	b	a	t1	c	E	t2	d
$l_{eff}(v)$	18	4	1	29	5	1	7
$L_{bin}(n=1)$	18	22	23	52	57	58	65

Processor 2	Allocated nodes				
v	g	f1	k	L	f2
$l_{eff}(v)$	8	38	0	0	19
$L_{bin}(n=2)$	8	46	46	46	65

Processor 3	Allocated nodes				
v	C	D	s	U	I
$l_{eff}(v)$	29	12	5	4	4
$L_{bin}(n=3)$	29	41	46	50	54

Table 4.1: Packing process for $L = 65$

Now the speedup can be calculated using the formula 3.6:

$$SPEEDUP_{fp} = \frac{200 * 170}{65(200 + 3 - 1)} = 2.59$$

Chapter 5

Extensions

The *functional pipelining* technique is expected to be successful in parallelization of the large loop nests, lying high in the call hierarchy of the program. Unfortunately in majority of the embedded programs written in ANSI C such loop nests do not occur often. Some special measures have to be taken to extend the application scope of the algorithm. In the discussion below we will use the concepts of the code transformations as described in [16]. The following optimizations can increase the application scope of the **FP-Map** algorithm:

- A loop nest may be not tightly nested (i.e. a loop is not the only statement in the body of another loop). The problem can be handled by applying the code transformation called *code sinking*.
- Often nested loops are distributed among calling procedures. *Call inlining* allows the increase of the depth of the loop nest.
- Several loop nets with similar iteration space may follow each other in the program. By performing *normalization* and *loop fusion* (also *loop skewing* may be necessary) they can be merged together to form a single large loop.

In the presentation of the algorithm we used a user given constant search step value ϵ. This value can be adjusted after every step by observing the extra capacity ΔL which would be required to put an extra vertex into each bin. Then the value of ϵ can be calculated as:

$$\epsilon = \min_{n=1...N_{proc}} \Delta L_n$$

Chapter 6

Conclusions

In this report we proposed an approach to the *functional pipelining* of the embedded programs written in ANSI C. The problem of increasing the parallelism exploitation ratio in the sequential programs has been discussed. We concluded that this goal can only the reached by a combined fine- and coarse-grain parallelism exploitation. A multi-processor embedded system framework has been recognized as a very interesting and cost-effective extension to a single processor embedded system. We proposed an algorithm for mapping the embedded programs written in ANSI C onto a pipeline of ASIP processors. The algorithm has computational complexity of $\mathcal{O}(|V||E|)$ and was especially developed for the parallelization of the loop nests laying very high in the call hierarchy of a program and containing all kinds of control statements.

Bibliography

[1] V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[2] Saman P. Amarasinghe, Jennifer M. Anderson, Christopher S. Wilson, Shin-Wei Liao, Brian R. Murphy, Robert S. French, Monica S. Lam, and Mary W. Hall. Multiprocessors From a Software Perspective. *IEEE micro*, pages 52–61, June 1996.

[3] T.H. Corman, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, MIT-Press, 1990.

[4] Henk Corporaal. *Transport Triggered Architectures; Design and Evaluation*. PhD thesis, Delft Univ. of Technology, September 1995. ISBN 90-9008662-5.

[5] Henk Corporaal and Hans Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.

[6] Kemal Ebcioğlu. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, December 1987.

[7] Kemal Ebcioğlu and Toshio Nakatani. A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture. In *Proceedings of the Second Workshop on Programming Languages and Compilers for Parallel Computing*, University of Illinois at Urbana-Champaign, 1989.

[8] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of the NP-Completeness.* Freeman, San Francisco,CA, 1979.

[9] Jan Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft Univ. of Technology, February 1996.

[10] Jeroen Hordijk and Henk Corporaal. The Impact of Data Communication and Control Synchronization on Coarse-Grain Task Parallelism. In *Second Annual Conf. of ASCI*, Lommel, Belgium, June 1996.

[11] Ireneusz Karkowski and Henk Corporaal. Overcoming the limitations of the traditional loop parallelization. In *Proceedings of the HPCN'97*, April 1997. (to appear).

[12] Monica Lam. *A Systolic Array Optimizing Compiler*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Norwell, Massachusetts, 1989.

[13] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, October 1981.

[14] Mark G. Stoodley and Corinna G. Lee. Software pipelining loops with conditional branches. In *IEEE micro*, pages 262–273, 1996.

[15] David W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of ASPLOS-IV*, pages 176–188, Santa Clara, California, April 1991. ACM.

[16] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.