Exploring Processor Parallelism: Estimation Methods and Optimization Strategies

Roel Jordans, *Student Member, IEEE*, Rosilde Corvino, Lech Jóźwiak, and Henk Corporaal, *Member, IEEE*

***Abstract*—Automatic optimization of application-specific instruction-set processor (ASIP) architectures mostly focuses on the internal memory hierarchy design, or the extension of reduced instruction-set architectures with complex custom operations. This paper focuses on very long instruction word (VLIW) architectures and, more specifically, on automating the selection of an application specific VLIW issue-width. The issue-width selection strongly influences all the important processor properties (e.g. processing speed, silicon area, and power consumption). Therefore, an accurate and efficient issue-width estimation and optimization are some of the most important aspects of VLIW ASIP design. In this paper, we first compare different methods for the estimation of required the issue-width, and subsequently introduce a new force-based parallelism estimation method which is capable of estimating the required issue-width with only 3% error on average. Furthermore, we present and compare two techniques for estimating the required issue-width of software pipelined loop kernels and show that a simple utilization-based measure provides an error margin of less than 1% on average.**

***Index Terms*—design automation, parallelism estimation, very-long instruction word**

I. Introduction

HIGHLY customized application specific instruction-set processors (ASIPs) are increasingly used in advanced products requiring programmability, high-performance, and/or a limited energy consumption. Several industrial strength tool-flows, e.g. [2]–[6], are available to specify, simulate, and synthesize such processors. However, the optimization of the ASIP-architecture trade-offs is generally left to human designers. Only a few approaches have been presented to automate the design space exploration. They all assume that an existing architecture is used as a starting point of the exploration. After selecting this starting point, two strategies are generally considered, *growing* [4]–[9], i.e. extending the initial architecture until no further performance is gained, or *shrinking* [4]–[7], [10], [11], i.e. removing the (almost) unused components from the architecture until performance is lost. Both approaches show substantial area, energy, and temporal performance improvements over the starting-point designs. Both shrinking and growing can be time-consuming processes, depending on the method used to evaluate the different architectural solutions. Selection of good estimation and search strategies is therefore important for reducing the required design-time, without compromising the result quality.

This work was performed as part of the European project ASAM that has been partially funded by ARTEMIS Joint Undertaking, grant no. 100265

R. Jordans, R. Corvino, L. Jóźwiak, and H. Corporaal are with the Electronic Systems group at the Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands

This paper focusses on very long instruction word (VLIW) processor architectures. In VLIW architectures (c.f. figure 1) instruction level parallelism is provided by explicitly encoding the parallel execution of operations executed in different issue-slots in the processor instruction word. The number of issue slots in a processor, also known as the *issue width*, largely determines the available processor parallelism. Providing too much parallelism will result in an architecture with too many (often unused) resources and a very large, and thus costly, program memory. Providing too less parallelism will result in an architecture which will not be able to meet the performance requirements of the target application. Therefore, to create an efficient solution, the processor parallelism should match the parallelism exposed by the application.

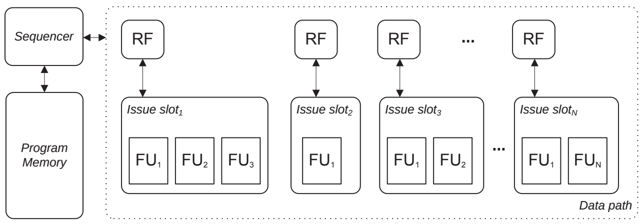


Fig. 1. The VLIW processor template

Through analysis of different kinds of hand crafted designs utilizing the target VLIW technology [12]–[15], we found that the total processor area is distributed on average as follows: 30–40% of the area is occupied by the VLIW data-path, 25% by program memory, 25% by data memories, and 10–20% by the register files. Moreover, the data memory and register file sizes (35–45%) are largely determined by the application mapping, while the program memory size (25%) is largely determined by the VLIW issue-width, and specifically the number of operations that can be executed in parallel by the VLIW processor. The data-path size is substantially influenced by the issue-width, but also by some other design decisions related to the instruction-set selection and custom operations. Moreover, the issue-width influences not only the memory and data-path sizes, but also their speed and power consumption. This makes the correct issue-width selection a critical decision in the VLIW ASIP design.

This paper focuses on the estimation of the minimal VLIW issue-width that guarantees the required performance of the target application. In the past, several methods have been presented for the issue-width estimation of a VLIW ASIP [16]–[22]. They estimate the issue-width required for obtaining a specific maximum latency when executing a specific part of the application. Similarly to most of the previously proposed methods, we also consider basic blocks, i.e. application parts with single entry and exit points. This paper extends our results presented in [22]. We extend the previously presented parallelism estimation techniques by considering the control-flow of the application and adding estimations for *software pipelined* loops. Software pipelining [23]–[25] is an important loop scheduling technique for parallel architectures. It enables the exploitation of parallelism by scheduling operations of different loop iterations in parallel.

When building on our previous work presented in [22], this paper addresses two very important aspects of VLIW ASIP design. Firstly, we investigate various methods, including a new one, for *estimating* the required issue-width, without having to completely schedule the application, and provide a quantitative comparison of these methods. Secondly, we assess the advantage of the newly proposed method over the existing ones when considering the ASIP development environment as a black-box. The previously proposed methods of growing and shrinking can require many runs of a time-consuming scheduling and/or synthesis processes. Scheduler runs are especially costly in our design framework, because we can only use the compiler of our ASIP development environment as a black-box. Compilation of a relatively small target application for a given processor configuration and simulation of the resulting application mapping can already take 1–2 minutes. This makes the optimization time strongly dependent on the selection of an appropriate issue-width optimization strategy and its starting-point. This paper compares several existing, as well as, newly proposed strategies for *finding* the required issue-width. In particular, we investigate the required number of scheduler runs for each of the strategies.

The paper is organized as follows. Section II briefly introduces the existing methods for VLIW issue-width estimation and discusses the related research. Section III provides more detail on the considered parallelism estimation methods and presents their quantitative comparison. Section IV presents different optimization strategies to find the required issue-width. Section V introduces and compares our two methods for parallelism estimation when software pipelining is used. Section VI concludes the paper.

II. Issue-width Estimation

Traditionally, the issue-width decision for a VLIW processor has been based on an analysis of the available instruction-level parallelism in the target application. Previous research [16]–[18], [26] mostly focused on estimating the *average* parallelism that can be obtained for a specific application on an unconstrained platform, only considering the *true dependencies* imposed by the target application. Wall [17] being a notable exception, focussed on the upper-bound of parallelism over traces of a complete application. More recently, Cabezas and Stanley-Marbell [20] published a method for estimating the *distribution* of parallelism across a program's execution. They showed that, in some cases, over 80% of the program's execution stream has a parallelism that is an order of magnitude smaller than the mean value. Our goal is to provide the required real-time performance. It is therefore important that enough parallelism is provided for the high-performance parts of the application, even when these parts constitute only a small portion of the application. In order to better quantify the high variation in application parallelism Theobald *et. al* [19] defined their *smoothability* metric. This metric provides a score in the range of 0–100%. A program which exhibits short bursts of high parallelism separated by long sequential sections will get a low score, while a program that has a more evenly distributed parallelism will obtain a higher score.

While both the parallelism distribution and the smoothability metric do provide insight in the parallelism variability of a whole program, they only provide a lower-bound on the parallelism required for obtaining a specific performance. Our method attempts to estimate the exact parallelism required for obtaining a specific performance for a given program part with real-time constraints. The estimated required parallelism can be directly translated into an issue-width requirement for a VLIW ASIP, or can be explored as part of a high-level design space exploration, such as the data-memory organization exploration.

In this paper, we will compare several methods to estimate instruction-level parallelism based on their suitability for issue-width estimation and their computational complexity. The following methods are considered:

1) *Average parallelism (AP)* [16]–[18], [21], [22], [26], estimated by dividing the number of operations in the program (part) by the expected latency of the program (part).
2) *Force based parallelism (FBP)* a contribution of this paper introduced in section III-A2.
3) *Maximum parallelism (MP)* [20]–[22], estimated by finding the maximal number of operations which can be scheduled in parallel.
4) *Required parallelism (RP)* [17], [21], [22], estimating the minimal upper-bound on the parallelism as required for scheduling of an application part within a given latency bound.

We also consider the effect of software pipelining [23]–[25], a commonly used technique for increasing the throughput of a loop based code, and present two methods for estimating the parallelism of software pipelined loops.

In order to ensure the practical relevance of our solutions and provide more control on the issue-width estimation by the end-user, we have added the option of explicitly constraining some specific types of hardware resources in the optimization. Common uses of this option are constraining the number of ports of the data memories and/or constraining the number of instances of specific (costly) resources (e.g. a maximum number of dividers).

III. Parallelism Estimation of Straight-line Code

This section introduces three different methods for a rapid application parallelism estimation, including our novel force based parallelism estimation, and presents the results of our experimental research performed with these methods.

A. Methods

1) Average parallelism: Perhaps the most commonly used measure to estimate the parallelism of an application is the average parallelism. It is estimated by dividing the number of operations by the required latency, and provides a lower bound on the required issue-width.

$$\Phi_{AP} = \frac{|V|}{\lambda}$$

2) Force based parallelism: Another estimate of the required issue-width can be obtained using a concept found in Force Directed Scheduling [27] in a novel way.

During force directed scheduling, the values in the *distribution graph* are computed from ASAP-ALAP schedule intervals as the sum of the probabilities of all operations which may be executed for each given cycle. An example is shown in figure 2. Both operations v_1 and v_3 can be scheduled at 3 different moments, as shown by their ASAP-ALAP schedule interval in figure 2b. Their scheduling probability is therefore $1/3$ for each cycle. The distribution graph of the DFG example is shown in figure 2c. For cycle 1, for example, the summed probability was computed by adding $p(v1) = 1/3$ and $p(v_2) = 1$ which results in a bar height of $1 1/3$.

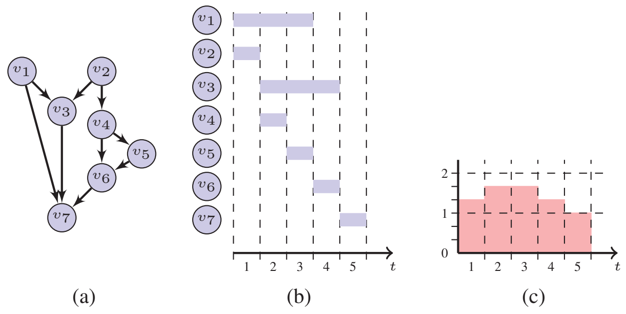


Fig. 2. Example DFG (a) with ASAP-ALAP schedule intervals (b), and the corresponding distribution graph used in estimating the force based parallelism (c).

Force Directed Scheduling selects the next operation to be scheduled based on a force calculated from this distribution graph. However, we observe that the distribution graph itself is a good predictor for the required parallelism of an application part. We therefore define the force based parallelism estimate as the maximum value of the summed probabilities in the distribution graph. For example, from figure 2c one will find the value of $1 2/3$, which could lead to the conclusion that a parallelism of 2 is an appropriate solution. Estimating the force based parallelism for the 8 point IDCT algorithm results in a value of 7.85, closely corresponding to the required parallelism of 8.

It should be noted that the force based parallelism does not provide an upper nor lower bound on the parallelism, but a value close to the actually required value. Figure 2 shows an under-estimation while figure 3 shows a graph that results in an over-estimation. In this example, the operation v_x can be scheduled in parallel to operations v_1 and v_2. This results in a FBP of 2.5 whereas the required parallelism for this graph is only 2. More extreme cases, resulting in larger overestimations, can be constructed in a similar fashion.

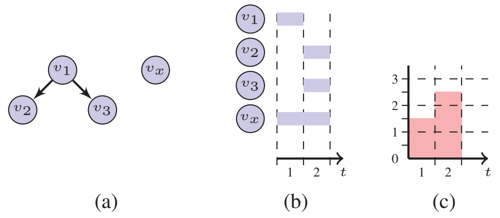


Fig. 3. An example DFG resulting in an over-estimation of the required parallelism by the FBP method.

3) Maximum parallelism: The maximum parallelism [21], [22] can be estimated in a way that is similar to the estimation of the force based parallelism. The only difference is that all nodes are counted with the same weight and that the length of the schedule interval is not taken into account as shown in figure 4. Estimating the maximum parallelism for the example DFG shown in figure 2 results in a parallelism of 3, a parallelism which cannot be obtained in any valid schedule of the DFG, but which, when provided, does guarantee the required latency.

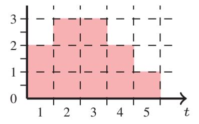


Fig. 4. Potential parallelism graph used in estimating the maximum parallelism for the example DFG given in figure 2a.

Much care should be taken though when estimating the maximum parallelism under resource constraints, as the minimal schedule latency may increase due to the added constraints.

B. Experimental results

All three presented methods and the reference method for VLIW issue-width estimation have been implemented using the intermediate representation (IR) of the LLVM compiler framework [28], and used to compare their respective quality in the approximation of the required issue-width. The experimental results have been analyzed using the R environment for statistical computing [29].

The experiments reported in this paper have been performed on a set of 3667 basic blocks taken from an MPEG4-SP encoder application. This application contains a representative set of basic blocks showing different kinds of processing. Each of these basic blocks was taken as a separate experiment and the parallelism was estimated for it's ASAP schedule. Almost all these basic blocks fall within the range of 1–150 operations but there are several larger blocks with sizes up to 1279 operations (e.g. `fdct`). In the experiments, all three parallelism estimation methods have been applied to each of the basic blocks, with and without adding a constraint on the number of parallel memory accesses. The memory constraint was selected as a common example of an explicit resource constraint. Any other resource constraints (e.g. constraining costly function units) can be added in a similar fashion.

The experiments have been grouped as *unconstrained* and *constrained* cases, referring respectively to the experiments without and those with the added resource constraint. Figure 5 shows a box-plot of the results obtained from our experiments normalized to the required parallelism (RP) of each basic block found through an exhaustive search.

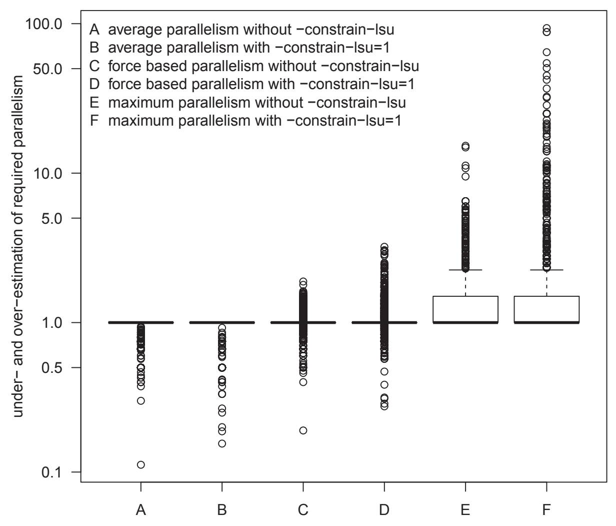


Fig. 5. The deviation of various parallelism estimation methods from the required parallelism. Normalized to the required parallelism.

From the experimental results shown in figure 5 we conclude, as expected, that the AP provides a *lower-bound* on the VLIW issue-width required for executing the application, while the MP provides an *upper-bound*. The AP underestimates the RP, on average by 7% independent of the presence of extra resource constraints. However, this under-estimation of issue-width can be up to a factor of 8.9x as shown in our experiments. The MP provides an overestimation of up to two orders of magnitude and, on average, 31% for the unconstrained and 72% for the constrained experiments. The FBP delivers the most accurate estimation, on average resulting in a 3% overestimation for the unconstrained and a 6% overestimation for the constrained experiments. The worst-case result for FBP is an underestimation of the required parallelism by 5.2x.

C. *Conclusion on parallelism estimation*

From our experiments, it follows that the average parallelism measure usually provides a quite accurate view of the issue-width requirement of an application. However, in the worst case, it underestimated the required issue-width by a factor of 8.9x.

We also conclude that, the maximum parallelism provides an upper bound with a large error margin. We therefore consider the maximum parallelism to be less useful for a direct issue-width estimation. However, as it will be shown in the next section, the maximum parallelism can be used to create an improved search strategy for finding the required parallelism.

Finally, we have shown that the force-based parallelism estimation is more precise than the average parallelism estimation and has a much smaller worst-case deviation. Our force-based parallelism measure should therefore be the preferred method for making initial estimates of the required parallelism.

IV. VLIW Issue-width Optimization

The required issue-width can be computed both using a *growing* or *shrinking* strategy, where growing is the most common [4], [7]–[9] strategy for exploring parallelism. However, applying a linear search for finding the optimal issue-width that guarantees a given latency may not be the best choice since this requires Φ_{RP} iterations of the scheduling algorithm, with Φ_{RP} equal to the required parallelism.

Another possibility, when an upper-bound to the parallelism is known, for example through estimating the maximum parallelism, is to perform a binary search, which requires a number of scheduler iterations logarithmic to the size of the considered parallelism range.

Both the previous work (e.g. [17]) and our initial experiments have shown that the required issue-width Φ_{RP} usually has a relatively small value in comparison to the maximum parallelism Φ_{MP}, often even smaller than $\log \Phi_{MP}$. For example (cf. figure 6), a naive growing technique would find the Φ_{RP} in 8 scheduling steps in this case. A binary search strategy starting on the range 1–Φ_{MP} would also require 8 scheduling steps. Starting the growing technique at the average parallelism Φ_{AP} improves the performance of the growing strategy by reducing the number of the required scheduling steps to 4. Similarly, changing the range partitioning within the binary search algorithm can help in improving the average performance of the binary search strategy. For example, dividing the solution range into the lower-third and upper-two-thirds partitions results in 5 scheduling steps. Furthermore, it is also possible to select the first pivot independent of the division strategy of the remaining ranges. Using the force-based parallelism Φ_{FBP} as first pivot, and to continue from there with a balanced binary search, results in 4 scheduling steps. Finding the best starting-point and search strategy are therefore critical to an optimal performance of the required parallelism estimation method.

A. *Possible search strategies*

As stated above, two main search strategies are possible, linear search and binary search. Several starting points are possible for both of them. This section will further explain the different possibilities for both strategies.

1) Linear search: Both growing and shrinking strategies can be combined with linear search depending on the selected starting point. The simplest approach is to start at a parallelism of 1 and increase the parallelism until a satisfactory latency is obtained.

A faster way to obtain a single design point satisfying the latency requirements is to start from an estimated parallelism value which is closer to the final result. Both the average parallelism and the force based parallelism are good candidates for this. However, both Φ_{AP} and Φ_{FBP} are fractional numbers, which makes the selection of the rounding strategy important. Since the average parallelism provides a lower-bound, it can be rounded up to the next integer value. However, deciding upon the rounding for the force based parallelism estimation is not so straightforward as it can either over- or under-estimate the required parallelism. We therefore provide the results for

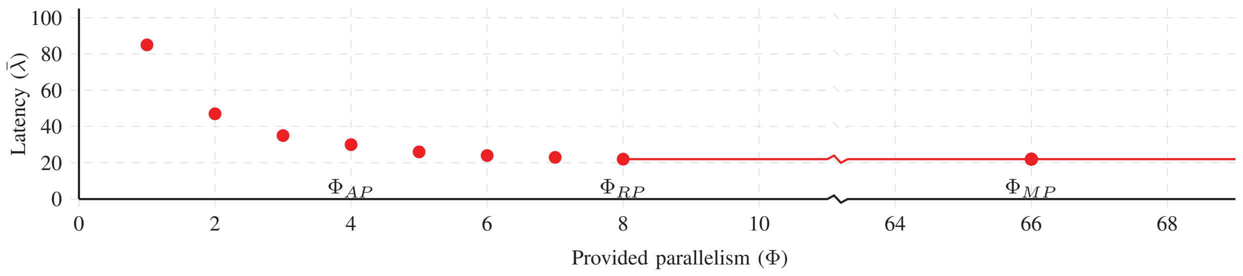


Fig. 6. Latency versus parallelism plot for an 8-point IDCT function with average parallelism (Φ_{AP}), required parallelism for its ASAP latency (Φ_{RP}), and maximum parallelism (Φ_{MP}) marked

separate experiments using either the rounded up (ceil Φ_{FBP}) or the rounded down (floor Φ_{FBP}) values as starting points.

A downside of starting at Φ_{FBP} is that, if the initial estimate provides us with a satisfying result, it is required to verify that this is the optimal result, which requires an extra run of the scheduler with a parallelism of one less. This extra scheduler run can be avoided if Φ_{AP} and Φ_{FBP} are equal, because this implicitly verifies optimality by proving that Φ_{FBP} is in fact the minimal value in such cases.

2) Binary search: This method requires a starting range and a rule for selecting the pivot. Only one reasonable starting range is available from our parallelism estimation methods which can provide both upper- and lower-bounds for the required parallelism. However, the balance of the search and selection of an initial pivot are critical to the performance of the binary search, as shown in the example accompanying figure 6. Both the balance of the search and the selection of the initial pivot have therefore been explored in our experiments. The first set of experiments with the binary search strategy varies the search balance, through a parameter α of the algorithm. The second set of experiments with the binary search strategy used Φ_{FBP} as the initial pivot and was performed for the same values of α. Algorithm 1 shows how such an unbalanced binary search can be implemented using a balancing parameter α, while using Φ_{FBP} as the first pivot.

Algorithm 1 Computing required parallelism using an unbalanced binary search where the balance is controlled by parameter α

```
Require: Basic block $BB$ and latency bound $\bar{\lambda}$
Ensure: Calculate the issue-width $\Phi$ of $BB$ such that the scheduled
    latency $\lambda$ is the greatest integer inferior to $\bar{\lambda}$
 1: $\Phi_{max} \leftarrow \Phi_{MP}$
 2: $\Phi_{min} \leftarrow \Phi_{AP}$
 3: $\Phi_{pivot} \leftarrow$ floor $(\Phi_{FBP})$
 4: while $\Phi_{max} > \Phi_{min}$ do
 5:     $\lambda \leftarrow$ Schedule $(BB, \Phi_{pivot})$
 6:     if $\lambda > \bar{\lambda}$ then
 7:         $\Phi_{min} \leftarrow \Phi_{pivot} + 1$
 8:     else
 9:         $\Phi_{max} \leftarrow \Phi_{pivot}$
10:     end if
11:     $\Phi_{pivot} \leftarrow \Phi_{min} + \lfloor(\Phi_{max} - \Phi_{min})/\alpha\rfloor$
12: end while
13: return $\Phi_{RP} \leftarrow \Phi_{min}$
```

B. Experimental results

For this set of our experiments we used the same framework and benchmark set as were used for the comparison experiments presented in the previous section. We have again grouped the experiments as *unconstrained* and *constrained* cases, referring respectively to the experiments without and those with the added resource constraint. This time we focused on the number of scheduler runs required for finding the required parallelism for obtaining an ASAP schedule. We did not count the extra scheduler run required for determining the ASAP latency of the blocks. The results of our experiments are presented in table I. The results of the binary search strategy with Φ_{FBP} as the initial pivot are only shown for an α of 2 as variations in α had negligible influence on the number of scheduler runs.

TABLE I
TOTAL NUMBER OF SCHEDULER ITERATIONS DURING RP SEARCH OVER ALL 3667 BLOCKS OF AN MPEG4-SP ENCODER FOR BOTH LINEAR AND BINARY SEARCH STRATEGY

method	start	α	unconstrained	constrained
linear	1		8523	6797
	AP		4705	4457
	ceil(FBP)		7787	7671
	floor(FBP)		7238	7091
binary search	AP–MP	2	6134	6463
		5	5560	5690
		10	5521	5565
		25	5510	5513
binary search	AP–FBP–MP	2	4149	4198

It should be noted that the quality of our results is strongly dependent on the quality of the internal scheduler. Our implementation uses a list scheduler but other schedulers can be used as long as they provide a deterministic result. From the many available list-scheduler heuristics [30], we selected the *dependency height* as the main criterion and we *prioritize* load-operations in order to increase the scheduler's freedom for scheduling shorter sequences. We found that using this combination of instruction selection criteria we can obtain a high quality result[1] without increasing the computational complexity. Observe that it is possible to achieve even higher quality results when using more effective scheduling algorithms, but at the cost of their higher computational complexity. Our RP

[1]On average within 3% of the actual required parallelism as computed using an optimal scheduler based on constraint programming [31]

method only requires that the scheduler is deterministic, but is otherwise independent of the specific scheduling algorithm used. This means that using a more effective (but slower) scheduling algorithm we will be able to achieve an even higher result quality.

C. Conclusion on the issue-width optimization

Usage of a binary search strategy with the FBP estimate as the initial search point to find a single design point for the parallelism-latency trade-off optimization results in the fewest required search steps. In our experiments this resulted in a 11% reduction from the currently used method of linear search from the average parallelism for the unconstrained experiments, and in a 6% reduction for the constrained experiments. We therefore recommend to use a combination of our FBP metric with binary search when looking for a single design point. However, we recognize that the full Pareto-front of solutions can be more interesting in many cases. Computing the Pareto-front requires the exhaustive linear search.

V. Parallelism Estimation of Pipelined Loops

Software pipelining [23] is an important throughput enhancement technique used when scheduling the application code for execution on parallel architectures. Increased utilization of parallel resources is achieved by overlapping the execution of multiple iterations of a loop core. Figure 7 for example, shows how the overlapping of multiple iterations of a loop kernel (distinguished by their different background color and texture) increases the parallelism exposed by a loop, and, in consequence, the parallelism exploitation. Section V-A provides a more in-depth explanation of this example.

Two main techniques are used for creating software pipelined schedules: *modulo scheduling* [23], [24], and *unroll-and-jam* [25]. Both techniques aim at creating a software pipelined schedule, but use different approaches. Our estimation methods build upon their common concepts and are independent of the used software pipelining method.

A. Determining the minimum initiation interval

The initiation interval (II) of a software pipelined loop is the distance, in cycles, between the start of two consecutive loop iterations. The initiation interval can be constrained by two factors: *1)* the available resources, and *2)* the inter-iteration dependencies of the loop core.

1) Resource constraints: In our case resources are usually unconstrained, because we are constructing new architectures. We may however impose constraints on some especially costly resources. Only the resources which have explicit constraints assigned are therefore taken into account when estimating the minimal initiation-interval. In our architectures, the main resource constraint influencing the minimal initiation interval is the number of single-ported memories used. Only a single load/store operation can be performed per cycle and per memory. As we do allow the existence of multiple memories in our processor, multiple arrays (or data sets) can be accessed in parallel, as long as they are mapped onto different memories.

To illustrate this we refer to the down-sampling example shown in listing 1. Figure 7a shows a compact representation of the schedule of the loop operations without software pipelining, horizontal black lines are used to show the repeated part of the loop core. In the loop shown in listing 1, two elements are read from array A and one element is written to array B. Considering the resource constraint of the load/store unit(s) in the architecture, we find two possible solutions for the minimal initiation interval of this loop.

1) When both arrays are mapped onto the same memory the minimal II is 3 (cf. figure 7b)
2) When both arrays are mapped onto different memories the minimal II is 2 (cf. figure 7c)

```
for(int i = 0; i < N; i++) {
    B[i] = (A[2*i] + A[2*i+1]) / 2;
}
```

Listing 1. Example loop nest showing an initiation interval constrained by the number of available load/store unit(s).

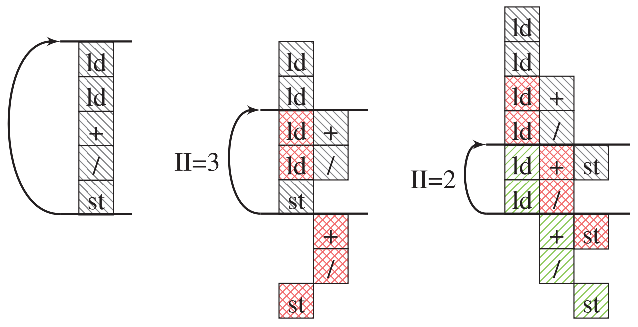


Fig. 7. Simplified schedules of the loop shown in listing 1 showing the original sequential schedule and two software pipelined versions demonstrating the influence of different memory mappings. Operations from different loop iterations are distinguished by their background color and texture. Only the kernel operations are shown in these schedules, address calculation and control-flow operations are hidden for brevity.

In our architecture, loading N data elements from a single memory requires N cycles. For software pipelined loops, this results in the minimal initiation interval being at least equal to the maximum number of elements accessed in a single memory.

2) Inter-iteration dependencies: Inter-iteration dependencies appear when a loop iteration requires a result that was produced by an earlier loop iteration. So called *reduction loops* are a frequently occurring example of this kind of behaviour. Listing 2 shows an example of such a loop where B_i contains the sum of all elements A_j with $0 \leq j \leq i$.

```
B[0] = A[0];
for(int i = 1; i < N; i++) {
    B[i] = B[i-1] + A[i];
}
```

Listing 2. Example loop nest showing an initiation interval constrained by a loop caried dependency.

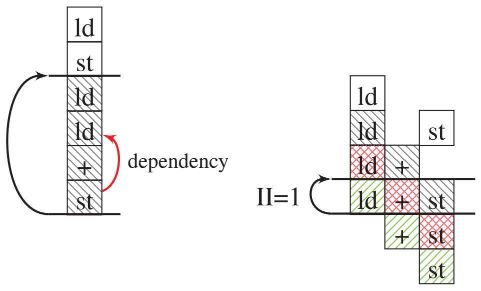
The problem with this kind of loop is that a new iteration can only be started after the previous B_i has been calculated.

However, this kind of inter-iteration dependency can be broken by introducing a local variable which values are stored in a register. Listing 3 shows how this can be achieved for the code shown in listing 2.

```
register int r = A[0];
B[0] = r;
for(int i = 1; i < N; i++) {
    r = r + A[i];
    B[i] = r;
}
```

Listing 3. Restructured version of the code shown in listing 2, breaking the loop caried dependency by storing the intermediate result into a register.

Figure 8 shows the effect of this transformation on the software pipelined schedule. A side effect of this transformation is that the number of memory accesses is reduced which, in turn, leads to a decreased minimum initiation-interval.



(a) Original (b) Transformed

Fig. 8. Simplified schedules for the original (listing 2) and transformed (listing 3) version of a loop showing an inter-iteration dependency. The original schedule shows the inter-iteration dependency which constrains software pipelining. The transformed schedule has been software pipelined and assumes that A and B are stored in different memories.

In general, all inter-iteration dependencies can be removed by inserting one or more temporary variables into the code. Adding many temporary variables increases the register file size requirements which may decrease the quality of the resulting processor design. However, usually applications only require a limited number of temporary variables. Furthermore, loops requiring a very large sliding window (and therefore many temporary variables) are usually good candidates for vectorization, which helps to decrease the number of required temporary variables to a better manageable number. In our work on VLIW ASIP design, we therefore do not consider inter-iteration dependencies as a limiting factor for the initiation interval. As a result, the minimal initiation interval estimates proposed in this paper are solely based on the resource constraints of the architecture, and in particular on the memory access constraints described above.

B. Methods

In the last set of experiments, we compared two methods for the parallelism estimation when software pipelining is applied. Both methods compute the minimal initiation interval from the memory access counts.

1) Utilization-based estimation: This method assumes that the final schedule efficiently utilizes the resources available in the processor architecture. This means that the overlapping operations of the different loop core iterations are distributed in such a way that the obtained initiation interval becomes equal to the minimal initiation interval. Dividing the number of operations in a single copy of the loop core ($|V|$) by the initiation interval (II) gives a lower bound on the required number of issue-slots, quite similar to the average parallelism method for a straight-line code. The only way to achieve this parallelism is when the software-pipelined schedule efficiently utilizes the provided issue-slots.

$$\Phi_{SWP_1} = \left\lceil \frac{|V|}{II} \right\rceil$$

The schedule shown in figure 8b serves to illustrate this method. The transformed kernel has 3 operations ($|V|$) and its initiation interval (II) is 1 cycle. Using the utilization-based method, the estimated parallelism is 3, which matches with the observed software pipelined parallelism shown in figure 8b.

2) Duplication-based estimation: Our second method computes the number of parallel copies of the loop core in the software pipelined schedule. We compute this number by dividing the latency of a single execution of the loop core (λ) by the minimal initiation interval (II).

$$N_{copies} = \left\lceil \frac{\lambda}{II} \right\rceil$$

The software pipelined parallelism can then be estimated by multiplying the parallelism of the original loop core with the number of parallel copies.

$$\Phi_{SWP_2} = \Phi_{orig} N_{copies}$$

Re-using the schedule shown in figure 8b, we see that a single execution of the transformed loop core has a latency (λ) of 3 cycles, an initiation interval (II) of 1 cycle, and a non-pipelined parallelism (Φ_{orig}) of 1. Using the duplication-based method, we find that 3 copies of the loop will run in parallel, resulting in the total parallelism of the software pipelined loop being 3.

In our experiments we have used the required parallelism Φ_{RP}, obtained from algorithm 1. Other parallelism estimates, such as found using the AP or FBP methods, are also usable. However, using the parallelism estimates obtained with the AP method will produce a result that is very similar to the results provided by the utilization-based parallelism estimation Φ_{SWP_1}. Using parallelism estimates obtained with the FBP method will produce very similar results compared to our choice of using the estimates from the RP method.

C. Experimental results

We mapped several kernels (partially) from the Polybench benchmark [32] onto different customized instances of our target VLIW ASIP architecture. This resulted in a total of 14 different software pipelined loops for which we could compare the estimated software pipelined parallelism with the actually obtained parallelism. In this section, we compare the results of our parallelism estimation methods with the actually obtained parallelism to quantify the respective quality of our estimation methods.

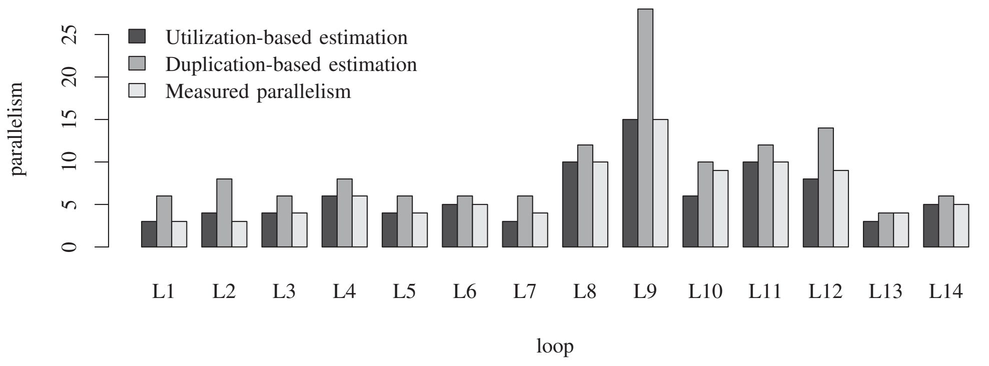


Fig. 9. Comparison of estimated parallelism versus observed parallelism after software pipelining in custom built architectures.

Figure 9 shows the estimated parallelism using our two methods together with the actually obtained parallelism when running the loop code on an architecture which was manually customized for that particular loop. We can see that the utilization-based estimation method performs the best. Its average error is less than 1% in our experiments. Our duplication-based estimation method performs worse, it provides a, quite large, over-estimation (54% on average).

Further investigation into the estimation errors of our utilization-based method shows us that their source is outside of our method, but in the abstraction of LLVM's IR over the actual instruction-set of our target architecture. The effect of these estimation errors is especially visible for loops with a very small II such as L7 and L10 (which both have an II of 1). Implementing this estimation at a lower, more accurate, level of the compiler will therefore result in even better estimations.

Investigating the estimation errors of the duplication-based method shows a more fundamental problem. The duplication-based method estimates the number of copies of the loop core executed in parallel. However, the operations of a loop core are usually not uniformly distributed in time and the resulting non-pipelined schedule will only utilize the required parallelism for a small portion of the time. The over-estimation of the software pipelined parallelism is a direct effect of the limited utilization within the original schedule. One of the key benefits of software pipelining is that it enables the scheduler to fill the gaps in the schedule of one loop iteration by executing operations of another iteration, resulting this way in a much better overall utilization. A variation of the duplication-based method is possible by actually unrolling the input code N_{copies} times and directly estimating the parallelism of the unrolled loop. However, our experiments show that this gives results which are equivalent to the results obtained using the utilization-based method, which is much simpler to apply.

In this section, we have shown how a simple utilization-based method can be used to efficiently obtain very good estimates for the parallelism of software pipelined loops. The average error margin of the utilization-based method was shown to be less then 1%. Furthermore, all the observed errors were not caused by our method, but by abstractions of the LLVM-IR on which our analysis was performed. Implementing the method in a later stage of compilation providing less abstract information will result in even more accurate estimations.

VI. Conclusions and Future Work

In this paper, we presented and compared three methods for estimating the required VLIW issue-width of an ASIP for a given target application. We experimentally demonstrated that our force based parallelism estimation proposed in this paper delivers results with a 3% over-estimation on average, substantially outperforming the commonly used average parallelism estimation regarding both the average and maximum error. Moreover, our algorithms can be controlled by the ASIP designer to account for resource constraints, such as the maximum number of instances for a specific type of function unit (e.g. a divider), etc. We have also presented several different strategies for obtaining the required VLIW issue-width for a specific latency and were able to reduce the number of required scheduler runs by 11% on average. We also found that the parallelism–latency trade-off is often more important. Furthermore, we investigated two methods of estimating the required VLIW issue-width for software pipelined loop bodies. We found that our simple and very efficient utilization-based method was capable of estimating the required parallelism with less than 1% error on average. Finally, we found that the remaining estimation errors were only caused by the application code abstractions of the LLVM IR on which we based our estimations.

Our future work includes two parts. Firstly, we want to implement the estimation algorithm as a part of the optimization heuristics when exploring the effects of loop transformations during an automatic VLIW architecture optimization. Secondly, we want to improve the accuracy of our estimation algorithms by implementing them at a lower level of the compiler abstraction.

REFERENCES

[1] ASAM, "Project website." [Online]. Available: http://www.asam-project.org

[2] Synopsys, "Synopsys Processor Designer." [Online]. Available: http://www.synopsys.com

[3] Target, "Target Compiler Technologies: IP Designer." [Online]. Available: http://www.retarget.com/

[4] FlexASP project, "TTA-based co-design environment." [Online]. Available: http://tce.cs.tut.fi/

[5] S. Aditya, B. Rau, and V. Kathail, "Automatic architecture synthesis and compiler retargeting for VLIW and EPIC processors," in *ISSS 1999 — 12th International Symposium on System Synthesis*. IEEE, November 1999, pp. 107–113.

[6] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D. Cronquist, and M. Sivaraman, "PICO: automatically designing custom computers," *IEEE Computer*, vol. 35, no. 9, pp. 39–47, September 2002.

[7] H. Corporaal and J. Hoogerbrugge, "Cosynthesis with the MOVE framework," in *CESA 1996 — Multiconference on Computational Engineering in Systems Applications — Symposium on Modeling, Analysis, and Simulation*. IEEE, July 1996, pp. 184–189.

[8] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1209–1229, July 2006.

[9] C. Wolinski and K. Kuchcinski, "Automatic selection of application-specific reconfigurable processor extensions," in *DATE 2008 — Design, Automation & Test in Europe Conference & Exhibition*. IEEE, March 2008, pp. 1214–1219.

[10] J. Matai, J. Oberg, A. Irturk, T. Kim, and R. Kastner, "Trimmed VLIW: Moving application specific processors towards high level synthesis," in *ESLsyn 2012 — The Electronic System Level Synthesis Conference*. IEEE, June 2012, pp. 11–16.

[11] A. Irturk, J. Matai, J. Oberg, J. Su, and R. Kastner, "Simulate and eliminate: A top-to-bottom design methodology for automatic generation of application specific architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 8, pp. 1173–1183, August 2011.

[12] P. Qiao, "Design and optimization of digital hearing aid system based on Silicon Hive technology," Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, August 2010. [Online]. Available: http://alexandria.tue.nl/repository/books/709025.pdf

[13] P. Qiao, H. Corporaal, and M. Lindwer, "A 0.964 mW digital hearing aid system," in *DATE 2011 — Design, Automation & Test in Europe Conference & Exhibition*. IEEE, March 2011, pp. 1–4.

[14] Y. Ökmen, "SIMD floating point processor and efficient implementation of ray tracing algorithm," Master's thesis, TU Delft, Delft, The Netherlands, October 2011. [Online]. Available: http://repository.tudelft.nl/assets/uuid:b0a8ae03-18b9-4a0e-9761-64ffd2851074/Yunus_Okmen_MSc_Thesis.pdf

[15] E. Diken, R. Jordans, R. Corvino, and L. Jóźwiak, "Application analysis driven ASIP-based system synthesis for ECG," in *Embedded World Conference*, February 2012, pp. 1–8.

[16] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions," *IEEE Transactions on Computers*, vol. 19, no. 10, pp. 889–895, October 1970.

[17] D. W. Wall, "Limits of instruction-level parallelism," in *ASPLOS 1991 — 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, April 1991, pp. 176–188.

[18] T. M. Austin and G. S. Sohi, "Dynamic dependency analysis of ordinary programs," in *ISCA 1992 — 19th annual International Symposium on Computer Architecture*. ACM, May 1992, pp. 342–351.

[19] K. B. Theobald, G. R. Gao, and L. J. Hendren, "On the limits of program parallelism and its smoothability," in *MICRO 1992 — 25th Annual International Symposium on Microarchitecture*. ACM, December 1992, pp. 10–19.

[20] V. C. Cabezas and P. Stanley-Marbell, "Parallelism and data movement characterization of contemporary application classes," in *SPAA 2011 — 23rd Symposium on Parallelism in Algorithms and Architectures*. ACM, June 2011, pp. 95–104.

[21] R. Jordans, R. Corvino, and L. Jóźwiak, "Algorithm parallelism estimation for constraining instruction-set synthesis for VLIW processors," in *DSD 2012 - 15th Euromicro Conference on Digital System Design*. IEEE, September 2012, pp. 152–155.

[22] R. Jordans, R. Corvino, L. Jóźwiak, and H. Corporaal, "Exploring processor parallelism: Estimation methods and optimization strategies," in *DDECS 2013 - 16th Symposium on Design and Diagnostics of Electronic Circuits and Systems*. IEEE, April 2013, pp. 18–23.

[23] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," *ACM SIGPLAN Notices*, vol. 23, no. 7, pp. 318–328, July 1988.

[24] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *MICRO 1994 — 27th Annual International Symposium on Microarchitecture*. ACM, December 1994, pp. 63–74.

[25] S. Carr, C. Ding, and P. Sweany, "Improving software pipelining with unroll-and-jam," in *HICSS 1996 — 29th Hawaii International Conference on System Sciences*. IEEE, January 1996, pp. 183–192.

[26] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Transactions on Computers*, vol. 21, no. 12, pp. 1405–1411, December 1972.

[27] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, June 1989.

[28] LLVM, "Project website." [Online]. Available: http://www.llvm.org

[29] The R project for statistical computing, "Project website." [Online]. Available: http://www.r-project.org/

[30] M. Smotherman, S. Krishnamurthy, P. S. Aravind, and D. Hunnicutt, "Efficient DAG construction and heuristic calculation for instruction scheduling," in *MICRO 1991 — 24th Anual International Symposium on Microarchitecture*. ACM, November 1991, pp. 93–102.

[31] A. M. Malik, J. McInnes, and P. van Beek, "Optimal basic block instruction scheduling for multiple-issue processors using constraint programming," *International Journal on Artificial Inteligence Tools*, vol. 17, no. 1, pp. 37–54, February 2008.

[32] L.-N. Pouchet, "Polybench/C 3.2," 2013. [Online]. Available: http://www.cse.ohio-state.edu/ pouchet/software/polybench/

Roel Jordans (M'13) received the MSc degree in field of Electrical Engineering from Eindhoven University of Technology in 2009. He worked within the PreMaDoNA project on the MAMPS tool flow as a researcher afterwards. As of September 2010 he continues his education as a PhD student at the Electronic Systems group of the Department of Electrical Engineering. His research interest include VLIW architectures and the automatic synthesis of application specific instruction-set processors.

Rosilde Corvino is a research scientist and project manager in the Electronic Systems group at the Department of Electrical Engineering at Eindhoven Technical University, The Netherlands. She is currently a project manager and work-package responsible in the European project ASAM - Automatic Architecture Synthesis and Application Mapping. In 2010, she was a post-doctoral research fellow in DaRT team at INRIA Lille Nord Europe and was involved in Gaspard2 project. She earned her PhD in 2009, from University Joseph Fourier of Grenoble, in micro and nanoelectronics. In 2005/06, she obtained a double Italian and French M.Sc in electronic engineer. Her research interests involve design space exploration, parallelization techniques, data transfer and storage mechanisms, high level synthesis, application specific processor design for data intensive applications. She is author of numerous research papers and a book chapter. She serves on program committees of DSD and ISQED.

Lech Jóźwiak is an Associate Professor, Head of the Section of Digital Circuits and Formal design Methods, at the Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands. He is an author of a new information driven approach to digital circuits synthesis, theories of information relationships and measures and general decomposition of discrete relations, and methodology of quality driven design that have a considerable practical importance. He is also a creator of a number of practical products in the fields of application-specific embedded systems and EDA tools. His research interests include system, circuit, information theory, artificial intelligence, embedded systems, re-configurable and parallel computing, dependable computing, multi-objective circuit and system optimization, and system analysis and validation. He is the author of more than 150 journal and conference papers, some book chapters, and several tutorials at international conferences and summer schools. He is an Editor of "Microprocessors and Microsystems", "Journal of Systems Architecture" and "International Journal of High Performance Systems Architecture". He is a Director of EUROMICRO; co-founder and Steering Committee Chair of the EUROMICRO Symposium on Digital System Design; Advisory Committee and Organizing Committee member in the IEEE International Symposium on Quality Electronic Design; and program committee member of many other conferences. He is an advisor and consultant to the industry, Ministry of Economy and Commission of the European Communities. He recently advised the European Commission in relation to Embedded and High-performance Computing Systems for the purpose of the Framework Program 7 preparation. In 2008 he was a recipient of the Honorary Fellow Award of the International Society of Quality Electronic Design for "Outstanding Achievements and Contributions to Quality of Electronic Design". His biography is listed in "The Roll of Honour of the Polish Science" of the Polish State Committee for Scientific Research and in Marquis "Who is Who in the World" and "Who is Who in Science and Technology".

Henk Corporaal (M'09) received the M.S. degree in theoretical physics from the University of Groningen, Groningen, The Netherlands, and the Ph.D. degree in electrical engineering, in the area of computer architecture, from the Delft University of Technology, Delft, The Netherland.

He has been teaching at several schools for higher education. He has been an Associate Professor with the Delft University of Technology in the field of computer architecture and code generation. He was a Joint Professor with the National University of Singapore, Singapore, and was the Scientific Director of the joint NUS-TUE Design Technology Institute. He was also the Department Head and Chief Scientist with the Design Technology for Integrated Information and Communication Systems Division, IMEC, Leuven, Belgium. Currently, he is a Professor of embedded system architectures with the Eindhoven University of Technology, Eindhoven, The Netherlands. He has co-authored over 250 journal and conference papers in the (multi)processor architecture and embedded system design area. Furthermore, he invented a new class of very long instruction word architectures, the Transport Triggered Architectures, which is used in several commercial products and by many research groups. His current research interests include single and multiprocessor architectures and the predictable design of soft and hard real-time embedded systems.