Algorithm Parallelism Estimation for Constraining Instruction-Set Synthesis for VLIW Processors

Roel Jordans, Rosilde Corvino, Lech Jóźwiak
Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands
r.jordans@tue.nl www.asam-project.org

***Abstract*—Customization of a (generic) processor to a particular application makes it possible to achieve high performance within a tight energy budget. Most of the published research works on processor customization extend a simple base processor with custom instructions. Only few works have considered a full instruction-set customization for complex highly parallel Very Long Instruction Word (VLIW) architectures. This paper discusses the parallelism estimation for a full instruction-set synthesis for VLIW processors and evaluates four methods to compute the maximum parallelism of a given application. We explain important reasons for computing and using such parallelism bounds, discuss the implementation of several methods, and our experimental research performed to evaluate the efficiency of each method.**

I. Introduction

Embedded applications with stringent real-time and power consumption constraints can often be well served using ASIP architectures with a high level of parallelism. However, providing more parallelism results in increased computational resources needed, and thereby, increased chip area and cost. Consequently, for a given application with its requirements, it is important to provide an adequate (not too high) amount of parallelism to keep the total silicon area and static power consumption to a possible minimum. The parallelism of an ASIP is decided to a high degree through its instruction-set architecture. The instruction-set architecture determines which operations can be executed in the processor, which of these operations can be executed in parallel, and how information is transfered from memories and register files to processing elements and vice versa. The research reported in this paper targets adaptable generic VLIW ASIPs that can be customized to a particular application through instantiation and extension of their generic architecture. Many of the existing ASIP customization and synthesis methods [1], [2] have focused on extending existing processors by proposing and implementing custom operations [3]–[7]. Several others have focused on reducing the instruction memory [8], register file sizes [9], and interconnects [10], and on compilation techniques [11], [12]. Finding an adequate combination of scheduling and instruction-set architecture decisions is of critical importance to obtain the required trade-off between the resulting timing, silicon area, and power consumption.

The high performance of VLIW processors results from an extensive exploitation of the parallelism available in the target application. This parallelism is exploited by their capability of executing (different) operations as a part of a single complex instruction distributed over multiple parallel issue-slots. The number of issue-slots should match the operation level parallelism available in the target application and is fixed for a specific architecture instance. The correct determination of a tight bound on the operation level parallelism exposed by an application is, therefore, critical to VLIW ASIP design.

This paper presents and evaluates four methods to estimate the amount of parallelism available in an application and illustrates how such methods can be utilized to provide a bound on the number of issue-slots for the instruction-set synthesis process. The parallelism estimation has been implemented as an analysis pass in the LLVM compiler framework [13]. It provides basic information about the average and maximal parallelism, and expected latency of a given application part.

The remainder of this work is structured as follows. Section II provides an introduction to VLIW instruction-set synthesis. Section III presents the application model and introduces four methods for estimating the amount of operation level parallelism. Section IV discusses the implementation of the methods and shows the results of our experiments with the presented methods. Section V compares our work to several previously published methods. Section VI concludes the paper.

II. VLIW Instruction-Set Synthesis

Our research aims at automatic synthesis of a VLIW ASIP architecture capable of satisfying the timing constraints, and efficient execution of a given (part of an) application. We try to achieve this through careful construction of an instruction-set for the ASIP in a process called instruction-set synthesis. This process includes, but is not limited to, proposing custom operations based on recurring operation patterns of the application and removing unused operations and related hardware from the architecture template of a generic ASIP. The ASIP architecture template involves several issue-slots. Each issue-slot can be connected to one or more register files and can execute a single (pipelined) operation every clock cycle.

Three main sub-problems can be recognized in the instruction-set synthesis for VLIW processors: *identifying* relevant recurring operation *patterns* in the target application and proposing possible custom operations, *partitioning* the application (part) to be executed on a set of parallel *issue-slots*,

and *selecting* an *instruction-set* for each issue-slot such that the entire application is covered, while satisfying the application timing constraints and optimizing the required area and power consumption trade-off.

In order to formulate the instruction-set synthesis problem, bounds on the design space are required, and, in particular, a bound on the number of issue-slots that can be utilized to a reasonable degree by a given application part. This bound can be used to constrain the size of the instruction-set synthesis design space as well as to predict the size and performance of the resulting ASIP architecture. The main goal of the research reported in this paper is, therefore, to provide an upper-bound on the amount of parallelism of a given application and to propose how to use this bound to constrain the number of issue-slots considered in our formulation of the application partitioning and ASIP architecture synthesis problem.

III. Parallelism Estimation Method

The method presented in this paper is limited to predicting the parallelism of a simple straight-line (sequential) code, called *basic blocks*. The first part of this section presents the application model and the second part, the four parallelism estimation methods.

A. Application Model

As in [14], the application is represented by a Directed Acyclic Graph (DAG) $G(V, E)$ with operations $V = \{v_i : i = 0, \ldots, N\}$ and operation dependencies $E = \{(v_i, v_j) : i, j = 0, \ldots, N\}$ (e.g. Figure 1a). Nodes v_0 and v_N are special nodes, which are added to the original graph to make the graph polar. They do not represent any operation and, therefore, have an execution time of 0. These nodes provide a clearly defined source (v_0) and sink (v_N). We also use the algorithms for As Soon As Possible (ASAP) scheduling and As Late As Possible (ALAP) scheduling described in [14]. The results of scheduling are represented by vectors of operation start-times. The vector of ASAP scheduled operation start-times is defined as $\mathbf{t}^S$, where t_i^S denotes the start-time of operation v_i. The latency λ of a DAG is defined as the number of time-steps required for executing the scheduled DAG, i.e. the difference between start times of the source and sink node (e.g. $\lambda_{ASAP} = t_N^S - t_0^S$). The vector of ALAP scheduled operation start-times is similarly defined as $\mathbf{t}^L$. The application model assumes that all operations have an execution time of a single clock cycle.

The parallelism level Φ_t of a DAG G at an instant $t \in [0, \bar{\lambda}]$, with $\bar{\lambda} \in \mathbb{N}^*$ and $\bar{\lambda}$ denoting the upper bound of the scheduled DAG execution latency, is defined as the number of operations that can be scheduled at the same instant t to be executed in parallel. An estimation of the parallelism level over the interval $[0, \bar{\lambda}]$ can be used to decide the parallelism level required to optimally execute the DAG.

B. Average Parallelism

A frequently used metric for estimating the parallelism of an application is the average parallelism, which is calculated by dividing the number of operations $|V|$ over the latency bound $\bar{\lambda}$ of the graph:

$$\Phi_{avg} = \frac{|V|}{\bar{\lambda}} \tag{1}$$

However, the calculation of the average parallelism does not take operation dependencies into account and, therefore, is not appropriate for an accurate estimation of the parallelism available in an application.

C. Maximum Parallelism

Another possible metric to decide the required number of parallel resources is the maximum parallelism. An upper bound on the maximum parallelism of a DAG can be computed by finding the maximum number of operations v_i which actually can be scheduled in a time-slot t due to the dependencies of the graph:

$$\Phi_{max} = \max_t\{\Phi_t\} = \max_{t:(0,\ldots,\bar{\lambda})} \left|v_i \in V : t_i^S \leq t \leq t_i^L\right| \tag{2}$$

This formulation uses the ASAP t_i^S and ALAP t_i^L scheduled start times of the operations v_i, where the time interval $[t_i^S, t_i^L]$ is called the schedule mobility of v_i.

Figure 1 illustrates the maximum parallelism with an example. Figure 1b shows the schedule mobility as horizontal blue bars using ASAP and ALAP scheduling times calculated for $\bar{\lambda} = 5$. The maximum overlap in the schedule mobility provides the first parallelism estimate $\Phi = 3$ (c.f. Figure 1b). However, in this example, Φ is overestimated because data dependencies limit the scheduling choices highlighted by the schedule mobilities (e.g. v_3 depends on v_1).

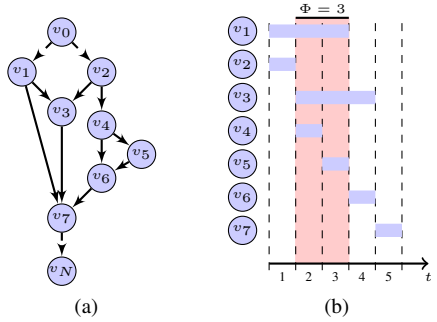

Fig. 1. Using the schedule mobility to provide an upper-bound on parallelism: (a) An example DAG, and (b) ASAP–ALAP schedule mobility for $\bar{\lambda} = 5$ highlighting maximum parallelism at $t = 2$ and $t = 3$

D. Maximum Required Parallelism

As shown in the previous example, the maximum parallelism provides an upper-bound on the operation-level parallelism provided in an application part. However, a more efficient schedule providing the same latency while using fewer resources is possible with graphs showing a high level of parallelism in a limited zone of the DAG, as illustrated in Figure 2. When determining the maximum parallelism for this second example we find that both partitions (v_m, ..., v_n) and

$(v_p, \ldots, v_q)$ can be scheduled in parallel. However, if the group of operations $(v_m, \ldots, v_n)$ is rescheduled in parallel with the operation v_x, the parallelism of the DFG is tremendously reduced without increasing the latency of the final schedule.

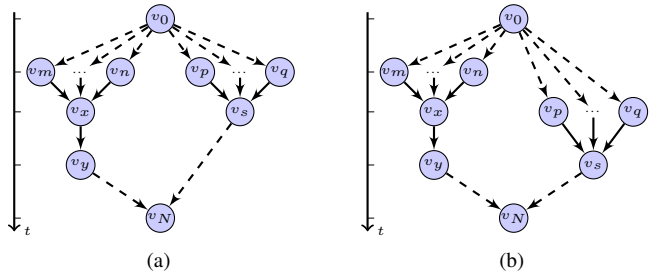

Fig. 2. A very wide DAG with multiple paths scheduled for (a) maximum parallelism, and (b) maximum required parallelism

Such an over-estimation of the required operation-level parallelism can result in a significant increase of the design space for our VLIW instruction-set synthesis problem. Therefore we define a more tight bound on the parallelism as follows: the *maximum required parallelism* is defined as the minimal upper-bound on the operation-level parallelism as required for scheduling the application part within a given latency-bound.

We developed two methods for calculating the maximum required parallelism. A fast but approximate solution, which is calculated using a heuristic scheduler, and a slower exact solution, which uses constraint programming.

E. Maximum Required Parallelism Using a Heuristic

For a given value of Φ_{max}, it is possible to use a resource-constrained scheduler (e.g. implementing list scheduling) to determine the scheduled latency of a DAG given a constraint on the parallelism. Using this scheduler, we can perform a search for the maximum required parallelism to schedule the DAG within the range $\{1, \ldots, \Phi_{max}\}$. The efficiency of this search can be enhanced by using a binary search strategy due to the monotonic nature of the scheduling problem. Algorithm 1 shows a procedure based on binary search which estimates the maximum required parallelism of a DAG for a given upper bound of the DAG execution latency $\bar{\lambda}$. Since list scheduling can be performed in $O(|V|)$, and the binary search across $\{1, \ldots, \Phi_{max}\}$ is performed in $O(\log|\Phi_{max}|)$, with $\Phi_{max} \leq |V|$, Algorithm 1 can be implemented with a complexity of $O(|V|\log|V|)$.

F. Maximum Required Parallelism Using Constraint Programming

Another way of finding the maximum required parallelism is through constraint programming. Algorithm 2 lists the constraint programming formulation used for describing the maximum required parallelism. This formulation first defines the range of valid schedule times for all operations to be between 0 and the latency bound $\bar{\lambda}-1$. Secondly, it constrains the set of possible solutions to adhere to the operation dependencies as required by the input DAG. Thirdly, the maximum parallelism is defined as the maximum number of operations scheduled at any instant t. Finally, this maximum parallelism is minimized, thus resulting in a maximum required parallelism when the search is completed.

Algorithm 1 Parallelism estimation

Require: DAG $G(V, E)$ and latency bound $\bar{\lambda}$
Ensure: Calculate the parallelism Φ of G such that the scheduled latency λ is the gratest integer inferior to $\bar{\lambda}$

$\mathbf{t}^S \leftarrow \text{ASAP}\ (G)$
$\lambda_{min} \leftarrow t_N^S - t_0^S$
if $\bar{\lambda} < \lambda_{min}$ **then**
  return error {infeasible latency bound}
end if
$\mathbf{t}^L \leftarrow \text{ALAP}\ (G, \bar{\lambda})$
$\Phi_{max} \leftarrow \max_{t:(0,\ldots,\bar{\lambda})} \left|v_i \in V : t_i^S \leq t \leq t_i^L\right|$
$\Phi_{min} \leftarrow 1$
while $\Phi_{max} > \Phi_{min}$ **do**
  $\Phi_{pivot} \leftarrow \lfloor(\Phi_{max} + \Phi_{min})/2\rfloor$
  $\lambda \leftarrow \text{Schedule}\ (G, \Phi_{pivot})$
  if $\lambda > \bar{\lambda}$ **then**
    $\Phi_{min} \leftarrow \Phi_{pivot} + 1$
  else
    $\Phi_{max} \leftarrow \Phi_{pivot}$
  end if
end while
return $\Phi \leftarrow \Phi_{min}$

Algorithm 2 Constraint-set for solving maximum required parallelism

Require: DAG $G(V, E)$ and latency bound $\bar{\lambda}$
Ensure: Calculate the maximum parallelism Φ_{max} of G such that the scheduled latency λ is inferior to $\bar{\lambda}$

$t_i \in (0, \ldots, \bar{\lambda} - 1)$
for all $(v_i, v_j) \in E$ **do**
  impose $t_i + 1 \leq t_j$
end for
impose $\Phi_{max} = \max_{t:(0,\ldots,\bar{\lambda})} |v_i \in V : t_i = t|$
Require: A lower bound on the parallelism Φ_{max}
Ensure: Minimize Φ_{max}

IV. Experimental Research and Evaluation

To perform the experimental research and evaluation of the methods. The parallelism estimation algorithms presented in the previous section have been implemented as analysis passes of the LLVM compiler framework [13], working on the platform independent LLVM intermediate representation. The maximum parallelism estimation method using constraint programming was implemented by combining the LLVM framework with the Gecode [15] constraint programming framework. All parallelism estimation methods have been evaluated using the IDCT and IQZZ functions taken from a JPEG decoding application, a five point FIR filter, matrix-matrix and matrix-vector multiplication, and DES encryption. These functions have been optimized using the existing LLVM

passes at optimization level `-O3` and a fully unrolled version of both the IDCT and IQZZ benchmarks was also considered. The experiments were performed on the 10 largest basic blocks available in our benchmark collection. The sizes of these blocks ranged from 14 to 802 operations.

The experimental results show that, though Φ_{avg} does approach Φ_{req}, Φ_{avg} is always an underestimation of Φ_{req} but, after rounding up, diverges by an average of 19% (0.81x). The maximum parallelism Φ_{max} however, overestimates Φ_{req} by up to 16x (`des`) and on average by 6x. The heuristic method Φ_{heur} overestimates Φ_{req} by 1.45x on average (and up to 2.7x). The constraint programming method gives the best results. Indeed, it finds a proven minimal Φ_{req}, but takes an order of magnitude more time to compute when compared to the heuristic method.

When comparing the four parallelism estimation methods, we find that the constraint programming method provides the best, proven, upper bound on the required parallelism while requiring more time to compute than the heuristic method. Moreover, we also want to note the quality of the estimation provided by the average parallelism, which, even though it does not provide an upper-bound on parallelism, is capable of estimating the required parallelism in our benchmark set with enough accuracy for manual engineering purposes.

V. Previous Research on Parallelism Estimation

Previous research proposes many techniques to schedule applications onto a hardware platform. However, such algorithms usually assume a fixed set of resource constraints [16]–[20], while time-constrained variations [21] are much less frequently used. This is especially true when concerning scheduling for VLIW processors, where scheduling and partitioning are only considered after the hardware has been determined. Therefore, some scheduling and mapping techniques have been borrowed from the high level synthesis field, where time-constrained scheduling is more common. However, such techniques only incorporate either scheduling or partitioning, but not both together as required for VLIW processors.

The constraint programming method presented in this paper provides a proven bound on the maximum required parallelism. We are not aware of any other works capable of providing a proven bound on the parallelism of an application as required for an automatic VLIW processor instruction-set synthesis.

VI. Conclusion

This paper has discussed and compared several parallelism estimation methods which compute the maximum required parallelism for a VLIW type architecture based on the dependency graph of the target application. The presented methods have been evaluated. The method utilizing constraint programming provides an exact, proven, bound on the number of issue-slots required for automatic instruction-set synthesis for VLIW processors.

Acknowledgment

This work was performed as part of the European project ASAM [22] that has been partially funded by ARTEMIS Joint Undertaking, grant no. 100265.

References

[1] L. Jóźwiak, N. Nedjah, and M. Figueroa, "Modern development methods and tools for embedded reconfigurable systems: A survey," *Integrated VLSI Journal*, vol. 43, pp. 1–33, January 2010.

[2] M. Jain, M. Balakrishnan, and A. Kumar, "ASIP design methodologies: survey and issues," in *VLSI Design, 2001. Fourteenth International Conference on*. IEEE, 2001, pp. 76–81.

[3] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," *Reconfigurable Computing: Architectures, Tools and Applications*, pp. 209–220, 2008.

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

[5] N. T. Clark, H. Zhong, and S. A. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Transactions on Computers*, vol. 54, p. 2005, 2005.

[6] A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli, "Automatic instruction set extension and utilization for embedded processors," in *Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on*, june 2003, pp. 108 – 118.

[7] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 8, no. 3, pp. 355–383, July 2003.

[8] S. Haga, A. Webber, Y. Zhang, N. Nguyen, and R. Barua, "Reducing code size in VLIW instruction scheduling," *Journal of Embedded Computing*, vol. 1, no. 3, pp. 415–433, 2005.

[9] P. Brisk, A. K. Verma, and P. Ienne, "Optimal polynomial-time interprocedural register allocation for high-level synthesis and asip design," in *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '07. Piscataway, NJ, USA: IEEE Press, 2007, pp. 172–179.

[10] Z. Shao, Q. Zhuge, M. Liu, B. Xiao, and E. Sha, "Switching-activity minimization on instruction-level loop for vliw dsp applications," *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*, pp. 224–234, 2004.

[11] B. de Dinechin, "From machine scheduling to VLIW instruction scheduling," *ST Journal of Research*, vol. 1, no. 2, 2006.

[12] J. Ellis, "Bulldog: A compiler for vliw architectures," Yale Univ., New Haven, CT (USA), Tech. Rep., 1985.

[13] LLVM, "Project website," www.llvm.org.

[14] G. De Micheli, *Synthesis and optimization of digital circuits*, 1st ed. McGraw-Hill, 1994.

[15] Gecode, "Project website," www.gecode.org.

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

[17] S. Novack and A. Nicolau, "Resource-directed loop pipelining: Exposing just enough parallelism," *The Computer Journal*, vol. 40, no. 6, p. 311, 1997.

[18] N. Nethercote, D. Burger, and K. McKinley, "Convergent compilation applied to loop unrolling," *Transactions on High-Performance Embedded Architectures and Compilers I*, pp. 140–158, 2007.

[19] L. Qiao, W. Huang, and Z. Tang, "A static data dependence analysis approach for software pipelining," *Network and Parallel Computing*, pp. 213–220, 2005.

[20] J. Llosa, E. Ayguadé, A. Gonzalez, M. Valero, and J. Eckhardt, "Lifetime-sensitive modulo scheduling in a production environment," *Computers, IEEE Transactions on*, vol. 50, no. 3, pp. 234–249, 2001.

[21] F. Sánchez and J. Cortadella, "Time-constrained loop pipelining," in *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*. IEEE, 1995, pp. 592–596.

[22] ASAM, "Project website," www.asam-project.org.