Instruction-set Architecture Exploration of VLIW ASIPs Using a Genetic Algorithm

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

***Abstract*—Genetic algorithms are commonly used for automatically solving complex design problem because exploration using genetic algorithms can consistently deliver good results when the algorithm is given a long enough run-time. However, the exploration time for problems with huge design spaces can be very long, often making exploration using a genetic algorithm practically infeasible. In this work, we present a genetic algorithm for exploring the instruction-set architecture of VLIW ASIPs and demonstrate its effectiveness by comparing it to two heuristic algorithms. We present several optimizations to the genetic algorithm configuration, and demonstrate how caching of intermediate compilation and simulation results can reduce the exploration time by an order of magnitude.**

***Index Terms*—instruction-set, very-long instruction word, genetic algorithm, design space exploration**

I. Introduction

Very-long instruction word (VLIW) processors constitute a highly energy efficient architecture base for creating application specific instruction-set processors (ASIPs). The design space exploration of such VLIW ASIP based systems is a complex optimization problem and several frameworks exist [1]–[4] that help to automate this exploration. However, automated tools are often not trusted yet and many companies still design VLIW ASIPs by hand.

In this paper, we present a method for VLIW ASIP instruction-set exploration using a genetic algorithm. Genetic algorithms are commonly used to solve complex optimization problems [5], [6] but can take a very long time to complete their search with satisfactory results. Their long run-time especially becomes a problem when exploring processor architectures, as a very large number of time-consuming simulations of the proposed design are required to evaluate the candidate designs. This can often reduce the effectiveness of the genetic algorithm and a shortened design-time when compared to a manual design may not always be guaranteed. However, we recently improved upon the current state-of-the-art by introducing compilation and simulation result caching [7] for evaluating the energy and area consumption of proposed VLIW ASIPs. Our caching approach works by efficiently re-using compilation and simulation results when considered architectures sufficiently similar. This allows us to greatly reduce the required number of compilation and simulation runs, making the exploration using a genetic algorithm feasible.

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

This paper is organized as follows. Section II introduces the VLIW ASIP architecture template and the overall exploration approach. Section III presents the configuration of the genetic algorithm, as well as, some optimizations to the genetic algorithm which allowed us to reduce the total exploration time. Section IV presents an investigation of the effectiveness of the exploration through comparison of the presented genetic algorithm to the heuristic exploration presented in [4]. Section V concludes the paper.

II. Architecture exploration

Figure 1 shows the general structure of our VLIW processor template. The VLIW data-path is composed of a set of issue-slots (IS), each containing one or more function-units (FU). The data-path is controlled by a sequencer which executes instructions from the program memory. The function-units in the issue-slots implement the (optionally pipelined) operations. Each issue-slot is capable of starting a new operation per cycle. The input data of the operations are taken from the register file (RF) connected to the issue-slot. The output results of the issue-slot can be written to one or more register files through the result select network (not shown in figure 1). Each register file can have one or more input ports to allow parallel writes to the register file. One or more local memories (not shown in figure 1) can be present in the processor. These local memories are accessed through a special load/store function-unit (LSU). Only one LSU can be connected to a single local memory.

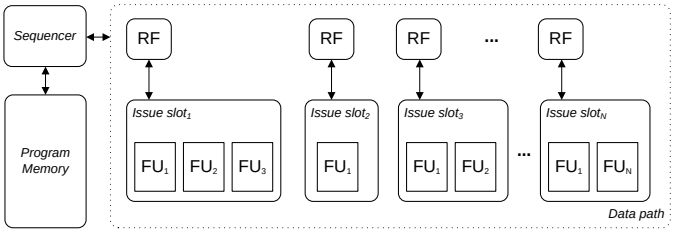

Fig. 1: Architecture template of a VLIW ASIP data-path and sequencer

Our ASIP architecture exploration method uses a shrinking technique and assumes that an *initial prototype* is provided. This initial prototype consists of an oversized ASIP architecture accompanied by a coarsely optimized version of the target application. In our ASIP design flow, this initial prototype is the product of a coarse, high-level, ASIP architecture exploration [8] which aims at finding the best parallel version of the

application code when considering loop-transformations (e.g. loop fusion, tiling, and vectorization) and corresponding ASIP architecture. However, initial prototypes from other sources (e.g. a human designer) can also be used.

The automatic ASIP instruction-set architecture exploration method explores the following properties:

- The number of issue-slots
- The types of function units available in each issue-slot
- The operations available in each function-unit
- The sizes of register files
- The size of the program memory

Our automatic ASIP instruction-set architecture exploration does not explore the number and sizes of local memories. In our total ASIP design flow, these have been explored as part of the application transformation (parallelization) and coarse ASIP architecture synthesis performed at a higher level, and are already decided by [8] when constructing the initial prototype. Previous research (e.g. [9], [10]) has shown that a large part (50–80%) of the architecture cost is due to the data storage and transfer. The memory and communication exploration is therefore commonly separated from and performed before the precise data-path exploration.

A. Architecture cost evaluation model

The energy and area costs of each proposed processor architecture are evaluated using our architecture cost model. This model uses the scheduled assembler code from the target compiler, which is configured to only use resources available in the proposed processor, and combines it with the application's profile to provide activity counts for the internal components of the proposed processor. These activity counts are then used to estimate the energy consumption of the proposed processor architecture.

We have redirected some of the architecture optimization work to the architecture model in order to keep the exploration time within reasonable bounds. In our architecture model we have decided to ignore any unused resources that are still remaining from the platform instantiation of the initial prototype. Any resource that is unused in a candidate prototype will not be counted towards the area and energy estimation of that candidate prototype. The architecture model recomputes the required size of the program memory, based on the set of the actually used resources and the length of the compiled target application.

Currently, our architecture model is capable of ignoring the following unused ASIP components:

- *Complete function units* the area and energy estimation, and the program memory size computation.
- *Partial function units* the program memory size computation based on their unused operations.
- *Partial and complete register files* the area and energy estimation, and program memory size computation based on the register file pressure of the compiled application.

The architecture model also takes changes in the interconnect of the ASIP resulting from the removal of components into account for both the area and energy estimation. This allows us to quickly see which resources are required and which are not, and what the effect is of the resources' removal on the resulting ASIP. It also gives us a clear view on which resources may form bottlenecks in the ASIP, and which are good candidates for further exploration.

III. Genetic Algorithm

We have implemented the genetic search algorithm in our architecture exploration framework using the AI::Genetic::Pro library [11]. Usage of such a library makes it very easy to implement complex optimizations in a genetic algorithm. Two support functions (`fitness` and `terminate`) and configuration settings need to be provided to customize the genetic algorithm for our purpose. These customizations are presented in the section below.

A. Genome construction

A genome is used by the genetic algorithm to identify candidate solutions. For our exploration we use a bit-vector as genome. Each bit in the genome represents the presence (or absence) of a function unit in the processor.

B. Genetic algorithm configuration

There are several configuration options for the AI::Genetic::Pro library. Most of these are options that are general to genetic algorithms.

1) Population size: The size of the population is selected based on the size of the genome. The initial population needs to be large enough to provide sufficient variation in the population but using a very large population will increase the exploration time. We initially found that using a population which is 3 times the number of bits in the genome gives both reasonable results and an acceptable exploration time. Furthermore, there is much less need for a large population in the later stages of the evolution. We therefore shrink the size of the population after each round of evolution by 10%.

2) Mutation and cross-over: A new set of candidate architectures is generated after each round of evolution. New genomes are generated as combinations of old ones. These combinations are formed by cross-over and mutation. Cross-over takes parts of the best genomes and combines them. The best genomes are selected randomly in a roulette selection where each genome has a chance of being selected proportional to its fitness. Each selected pair of genomes is then combined in the cross-over stage and results in a new individual. Random mutations are applied to these new individuals in order to increase the variation in the population. These mutations greatly help the genetic algorithm to avoid getting stuck in a local optimum. We selected a relatively high mutation rate (15%) for our experiments.

3) Preservation of candidates: To make sure that the best solutions do not get lost during evolution, we copy the 3 best solutions into the next generation.

4) Caching: The AI::Genetic::Pro library also provides it's own caching. It remembers previously considered genomes and doesn't evaluate their performance when they appear again during later stages in the evolution. Using this kind of caching greatly improves the run-time of the exploration without influencing the results.

C. Fitness function

The fitness function constructs new candidates and evaluates their fitness. It uses the BuildMaster framework [7] for this purpose. The BuildMaster framework will then decide if compilation and or simulation is required for this specific instance and will perform the necessary steps to obtain the performance metrics for the proposed candidate. Based on the returned metrics and a user selected cost-function, the fitness function then decides on the fitness of the proposed candidate.

D. Terminate function and number of generations

The terminate function is called after the evaluation of each generation. It is used to detect if the best solution has stabilized. We detect this by detection when no better solution has been found for a given number of iterations. Our architecture exploration tool allows the user to set this length and provides a default value of 10 generations.

E. Further optimizations

In order to get off to a good start we insert a single known-to-work architecture (the initial prototype) into the initial population. This ensures that there is at least one working architecture in the initial population. However, even with this the exploration times were still very high in our initial experiments. We improved upon this by inserting more known-to-work architectures. These architectures were obtained from a quick exploration of the VLIW issue-width using the first-match search strategy presented in [4]. This allowed us to get a much better initial population and greatly reduced the exploration time. It also allowed us to reduce the required population size to be equal to the number of bits in our genome (one third of the size originally used), without significantly impacting the result quality. This led to a substantial further reduction of the exploration time.

We also applied a second, more aggressive, optimization. It is possible to obtain a list of the resources that were actually used after the evaluation of a candidate solution. In order to aggressively reduce the available resources, we rewrite the genome of the candidate to reflect which resources were actually used. This way, the genetic algorithm is forced to quickly learn which resources can be removed, but it also increases the risk of running into a locally optimal solution.

F. Cost metrics

As briefly mentioned above, the genetic algorithm searches for the best processor architecture. For this purpose, our ASIP exploration framework offers several different, commonly used, cost metrics such as the energy-delay (ED) product and the energy-delay-squared (EDD) product. The energy-delay product puts more emphasis on the energy consumption, the energy-delay-squared product puts more emphasis on the delay. In both cases, a design is better when it has a lower score. However, the genetic algorithm library [11] used in our experiments, requires a fitness function (i.e. higher is better) and does not allow negative fitness values. We therefore translated each cost metric into a corresponding fitness function by using the reciprocal value of the cost function as a fitness measure.

IV. Experiments

For the experimental evaluation in this paper, the same test-cases were used as in [7]. These test-cases are two heart-rate detection (ECG) applications (`ecg-1` and `ecg-2`), one AES encryption/decription application (`aes`), and two image processing kernels (`down` performing down-sampling, and `lpsf` performing spatial filtering). Both the ECG applications, `ecg-1` and `ecg-2`, implement a combination of a filtering and decision process and are mainly constrained by the decision process. This causes loop optimizations have less effect, which results in a relatively straight-forward exploration. The `aes` benchmark has a much more computationally complex kernel, making this benchmark a difficult problem for the compiler. However, the `aes` benchmark doesn't provide much opportunities for loop optimizations. The image processing kernels `down` and `lpsf` do provide substantial opportunities for loop transformations and software-pipelining. This makes the exploration space much more irregular which results in a much more difficult exploration problem, as can be observed from the experimental results.

In our experiments we have used the energy-delay product as cost metric, similar results and observations are expected when using a different cost metric. Final scores are presented as the improvement factor compared to a hand-crafted initial architecture, higher final scores therefore represent better results.

A. Comparing exploration strategies

Firstly, we compare our genetic algorithm to our heuristic exploration methods [4], [7] for both the quality of the final results and the time required for the exploration.

Figure 2 shows the final scores obtained using the two heuristic approaches (`first-match` and `best-match`) from [4], and compares them against the results obtained by the presented genetic algorithm. The heuristic exploration algorithm is a deterministic process which will produce the exact same configuration given the same inputs. However, the genetic algorithm is non-deterministic by nature and may not always stabilize on the best solution. Figure 2 therefore presents the average final score through height of the bar itself, while the error-bars illustrate the minimum and maximum scores.

As can be seen from figure 2, the final exploration scores are quite stable across the different methods for all our experiments except for the `lpsf` kernel. The design-space for the `lpsf` application is very irregular and presents many opportunities to the design-space exploration tools to get stuck

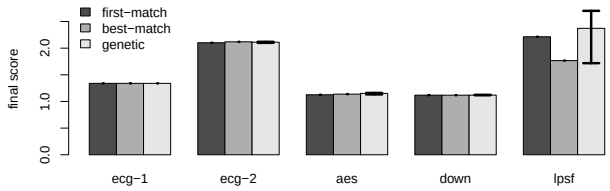

Fig. 2: Final optimization score for each benchmark

at a locally optimal solution. Both heuristic approaches easily end up in such a locally optimal solution, while the genetic algorithm has a chance of finding a better solution. However, the genetic algorithm also does get stuck at less efficient solutions and results in only slightly better solutions than the `first-match` heuristic on average.

Figure 3 compares the time required for exploring the processor architecture using each method. Again, the height of the bars shows the average time spent over 6 executions of an exploration while the error bars show the minimal and maximal execution times encountered during the experiments.

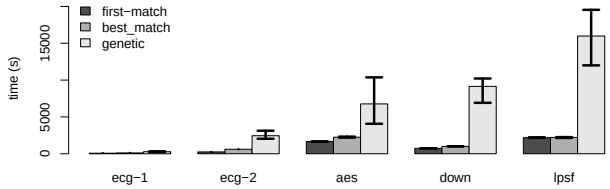

Fig. 3: Exploration times using different strategies

From figure 3, one can clearly see that the heuristic exploration takes much less time (at most 37 minutes for `lpsf`) than the genetic algorithm does (over 4 hours on average for `lpsf`) and that this difference is consistent over the experiments. The experiments also show the effect of the variable number of generations used in the genetic algorithm due to the early termination as presented in section III-D. This optimization allowed us to significantly shorten the exploration time, in several cases by reducing the required number of generations by half.

B. Effectiveness of intermediate result caching

A key to making the genetic exploration practically feasible was our usage of caching [7] for both compilation and simulation results. Especially the simulation cache proved effective, consistently scoring over 90% hit-rates (95% geomean). The compilation cache proved less effective with a geomean hit-rate of 15%. However, the compilation cache hit-rate statistics were heavily influenced by the genetic algorithm for which it has a significantly lower hit-rate (geomean 6%) due to the unstructured behaviour of the genetic algorithm. The lack of ordering in the consideration of candidate architectures by the genetic algorithm strongly reduces the probability of compiler cache hits. The heuristic exploration strategies have a much better hit-rate (geomean 25%) and especially the more difficult explorations (`down` and `lpsf`) benefit greatly from the compiler cache with hit-rates up to 70%.

Overall, we found that using both compilation and simulation caching we could efficiently explore up to 1500 architecture candidates within 6 hours for the `lpsf` benchmark (the most difficult benchmark) which translates to an average of 4 considered candidate architectures per minute. Without caching the average time required per considered architecture is much higher. A single simulation of the `lpsf` benchmark alone takes up to 1 minute and compilation of the `lpsf` benchmark can take close to 1 minute as well. Considering these numbers, we estimate that exploring the `lpsf` benchmark without the use of caching would take approximately 50 hours.

V. Conclusion

In this work we have proposed and discussed a practically feasible instruction-set architecture exploration method for application specific instruction-set processors using a genetic algorithm, and compared its result quality and exploration time to two previously published heuristic exploration approaches. We also investigated the effect of caching of intermediate compilation and simulation results on the exploration time. Our experiments show that both the genetic algorithm and the heuristics produce similar final solutions, but the genetic algorithm is more tolerant to highly irregular design spaces. We also find that using our automated intermediate result caching methodology resulted in a reduction the exploration time of the genetic algorithm by an order of magnitude.

This work presented several effective methods for speeding up the genetic algorithm. However, further tuning of the genetic algorithms parameters (e.g. population size, mutation rate) is possible and is planned for future work.

References

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

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

[3] J. Hoogerbrugge and H. Corporaal, "Automatic synthesis of transport triggered processors," in *Proc. of ASCI*, 1995, pp. 1–10.

[4] R. Jordans, R. Corvino, L. Jóźwiak, and H. Corporaal, "Instruction-set architecture exploration strategies for deeply clustered vliw asips," in *ECyPS 2013*, Montenegro, June 2013, pp. 38–41.

[5] M. Gen and R. Cheng, *Genetic algorithms and engineering optimization*. John Wiley & Sons, 2000, vol. 7.

[6] 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.

[7] R. Jordans, E. Diken, L. Jóźwiak, and H. Corporaal, "Buildmaster: Efficient asip architecture exploration through compilation and simulation result caching," in *DDECS 2014*, April 2014.

[8] R. Corvino, A. Gamatie, M. Geilen, and L. Jozwiak, "Design space exploration in application-specific hardware synthesis for multiple communicating nested loops," in *SAMOS XII*, Samos, Greece, July 2012.

[9] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, "Data and memory optimization techniques for embedded systems," *ACM TODAES*, vol. 6, no. 2, pp. 149–206, 2001.

[10] K. Danckaert, K. Masselos, F. Cathoor, H. J. De Man, and C. Goutis, "Strategy for power-efficient design of parallel systems," *IEEE Tran. on VLSI Systems*, vol. 7, no. 2, pp. 258–265, 1999.

[11] S. Lukasz, "AI::Genetic::Pro - Efficient genetic algorithms for professional purpose." [Online]. Available: http://search.cpan.org/~strzelec/AI-Genetic-Pro/lib/AI/Genetic/Pro.pm