# BuildMaster: Efficient ASIP Architecture Exploration Through Compilation and Simulation Result Caching

Roel Jordans, Erkan Diken, Lech Jóźwiak, Henk Corporaal
Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands
{r.jordans, e.diken, l.jozwiak, h.corporaal}@tue.nl

***Abstract*—In this paper we introduce and discuss the BuildMaster framework. This framework supports the design space exploration of application specific VLIW processors and offers automated caching of intermediate compilation and simulation results. Both the compilation and the simulation cache can greatly help to shorten the exploration time and make it possible to use more realistic data for the evaluation of selected designs. In each of the experiments we performed, we were able to reduce the number of required simulations with over 90% and save up to 50% on the required compilation time.**

## I. Introduction

Application-Specific Instruction-set Processors (ASIPs) are commonly used in the newest generation smartphones and tablets as well as in various cyber-physical devices. Their design is a problem of primary relevance for the top segment of the high-tech industry. Design space exploration for instruction-set design of an ASIP is a very complex problem, involving a large set of possible architectural choices. Existing methods are usually handcrafted and time-consuming. In our previous research we introduced and investigated a rapid method to estimate the energy consumption of candidate architectures for VLIW ASIP processors [1]. The proposed method avoided the time-consuming simulation of the candidate architecture prototypes, without any loss of accuracy in the predicted energy consumption, as long as changes in the application profile were recognized correctly.

In this paper, we present an automated framework called *BuildMaster*, which attempts to detect when changes in the application profile happen and automatically decide when the profile information is outdated. We call this technique *simulation caching*, as it effectively caches simulation results and relates them to variations in the code transformations applied to the target application. The cached simulation results are then used to accurately predict both the energy consumption and cycle-count of the target application on the proposed VLIW ASIP architecture. The BuildMaster framework also recognizes when a newly proposed architecture is a variant of an architecture which was investigated earlier in the exploration and can decide when the former compilation results (and their corresponding performance metrics) can be reused. We call this second technique *compilation caching*. Our experiments show that both these techniques are very effective in reducing the total architecture exploration time, and they both become more effective for more and more complex problems using larger, more realistic, test sequences.

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. In section II, we discuss some of the related work on both the simulation and compilation caching. In section III, we briefly re-introduce our architecture exploration techniques and efficient cost estimation methods from [1]. Section IV introduces our new caching methods and section V experimentally evaluates their effectiveness. Section VI concludes the paper.

## II. Related work

Our ASIP architecture exploration method builds upon our efficient method for energy estimation of ASIPs presented in [1]. In our previous research we have shown that accurate estimations of both the energy consumption and the applications cycle count can be obtained using only the assembly listing and the application profile. Furthermore, we made the observation that small changes in the processor architecture usually result in only small changes, if any, of the application profile. Combining these observations together with our efficient cost estimation technique allowed us to develop an efficient ASIP architecture exploration method which avoiding many simulation runs. In [1], we presented two variations of the cost estimation method: *a)* a completely static *profile-based estimation*, which performed a single simulation run and which handled only small changes in the profile, and *b)*, a *hybrid estimation* which performed a separate simulation for each design but was still a more efficient method than the traditional trace-based cost analysis. However, our previous paper did not present any technique capable of automatically detecting when changes happen to the application profile, making it difficult to use the more efficient profile-based estimation in a reliable way. This paper extends our previous research by proposing and discussing an automatic framework which detects when changes in the profile can be expected and automatically extracts the updated profile information from a new simulation run.

We have analyzed several other processor architecture exploration frameworks capable of automatic design space exploration (e.g. [2]–[6]). Most of these frameworks use the traditional time-consuming approach of finding performance

estimates by simulating different architecture candidates and analyzing the resulting application traces. To our knowledge, [4] is the only work using profile-based estimation. However, [4] only used this technique to compute the expected cycle-count for the proposed processor architectures and did not consider the energy consumption. Although, [6] uses a technique which stores the simulation trace of the application in a database and uses the database to efficiently estimate the energy-consumption of proposed processor architectures, this technique is still very similar to the traditional time-consuming trace-based estimation.

Several works consider caching for re-compilation (e.g. [7]–[9]) but, no one of them the kind of compilation caching proposed in this paper. We also analyzed related work in the field of iterative compilation (e.g. [10]–[12]) as these works consider an exploration somewhat similar to the architecture exploration that we are performing. However, the only form of cache mentioned in these works seems to refer to the instruction and data caches physically inside the target processor. No references to the caching of intermediate compilation results such as considered in this work was found.

From the above one can conclude that our ASIP architecture exploration method represents a substantial novelty.

## III. Architecture exploration and efficient cost estimation[1]

The ASIP architecture exploration process proposed in [1] and [13], evaluates and selects VLIW ASIP architectures according to a predefined template. Exploring the instruction-set architecture of VLIW ASIPs is however much more complex than for simple sequential (e.g. RISC) processors. It involves a.o. decisions on the number of parallel issue-slots the distribution of operations among the issue-slots.

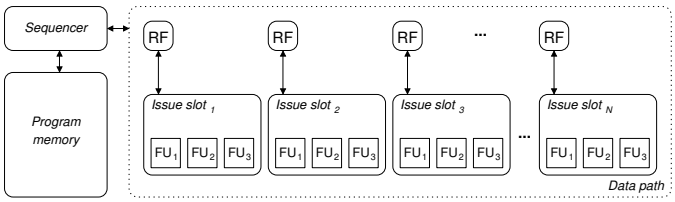



Fig. 1. Architecture template of a VLIW ASIP data-path and sequencer

Figure 1 shows the general structure of the processor template. A VLIW ASIP data-path is composed of a set 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 operations and can require pipelining. Each issue-slot is capable of starting a new operation per cycle. The inputs of the operations are taken from the register file (RF) connected to the issue-slot. The output of the issue-slot can be connected 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.

Our ASIP architecture exploration method uses a shrinking technique and assumes that an oversized ASIP architecture specialized for a coarsely optimized version of the target application (an initial prototype) is provided. This initial prototype can either be the product of a coarse, high-level, ASIP architecture exploration [14] or be hand-crafted by a human designer. During our ASIP architecture exploration, we are trying to compile the target application using only a sub-set of the resources of the initial prototype. Any resource that is not used in the compiled application is expected to be removed in the final construction of the ASIP architecture. Our architecture cost estimation model takes this into account by ignoring any unused resources and correcting for partially used resources. For example, it recomputes *a)* the required width of the instruction memory, *b)* the required sizes of register files, and *c)* the complexity of the remaining result select network. Doing so, we have succeeded in producing exactly the same cost estimates as when we apply our cost model on the actually reduced architecture (only containing the required elements). This allows us to accurately predict the cost of the proposed architecture sub-sets without actually constructing them, and saves a lot of time in the architecture exploration.

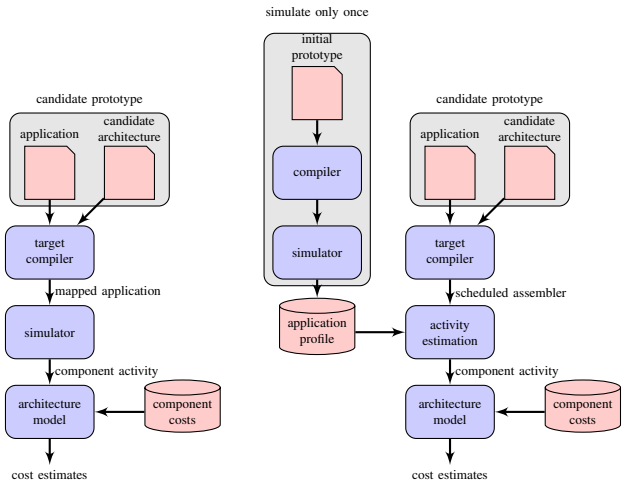
### A. Efficient cost estimation

The selection of the design cost estimation method has been shown [1] to have a large impact on the architecture exploration time. Figure 2 illustrates the cost estimation methods that were considered in [1]: the commonly used trace-based method (figure 2a), our static profile-based method (figure 2b), and the hybrid method (figure 2c).

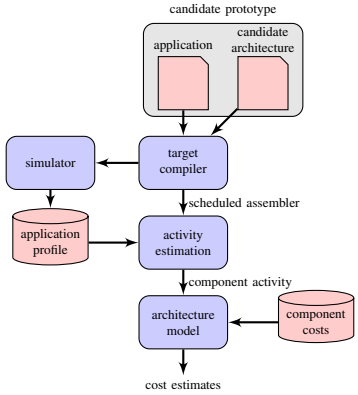
In the *trace-based* approach (cf. figure 2a), the application is compiled for the candidate processor architecture and the resulting mapped application is simulated. Activity counts for the components of the candidate architecture are collected from a simulation trace, and the dynamic power cost of the candidate is computed using these activity counts. This approach has two disadvantages: *1)* it requires a complete simulation of each candidate prototype, and *2)* the complete simulation trace of the candidate prototype can be very large. They both make the estimation time highly dependent on the size and complexity of the target application and its test-data. In our experiments, we found that trace-based estimation easily becomes impractical, even for small applications, especially when the cost estimation needs to be repeated for each considered design point when exploring the ASIP architecture.

Our *static profile-based* method (cf. figure 2b) only needs a single simulation run of the initial prototype to extract its application profile consisting of the execution count of each basic block of the application code. Based on this information, we can compute the activity counts of each

[1] [1] presents a more verbose introduction on the topics presented in this section




(a) trace-based estimation (b) static profile-based estimation




(c) hybrid

Fig. 2. Cost estimation methods for ASIP processors presented in [1]

architecture component by analyzing the activity within basic blocks in the scheduled assembler output of the compiler, and multiplying these activities with the execution count from the application profile. This way, we avoid the problems associated with a trace-based approach and can accurately predict the application power consumption within milliseconds instead of minutes or even hours. The application execution time can also be predicted in a strictly analogous way within milliseconds. However, the application profile may not be the same for different compilations of the candidate prototype and it may need to be adjusted to reflect performed transformations. This is especially true when the code transformations affect the control-flow structure of the candidate prototype, as such transformations can influence the application's profile. As mentioned in [1], this problem can be trivially handled when the compiler is capable of producing an updated profile during the compilation process. However, many compilers do not have this feature which forces our framework to detect changes in the application profile through both the (debug) output of the compilation and the structure of the produced assembly code.

A *hybrid* (cf. figure 2c) approach is also possible. The hybrid approach uses the fast component activity computation from the static profile-based estimation but extracts a new application profile of the candidate prototype from a simulation run. This method is more robust when it comes to code transformations in the compilation process but does require a simulation of each candidate prototype.

## IV. Compilation and Simulation Result Caching

The BuildMaster framework provides an easy interface to the compiler and simulator for our exploration routines [13] and automates the caching of intermediate results. This significantly simplifies the implementation of the architecture exploration strategies and allows the developer of such strategies to focus on the strategy itself and to ignore possible negative effects on the exploration time from considering equivalent architectures.

### A. The compilation cache

The BuildMaster framework automatically recognizes when two different architecture prototypes should result in the same optimized design. The decision on ignoring the unused resources in our cost model plays a critical role in this process, as it allows different initial architectures to end up as the same design point. Figure 3 illustrates this with an example. While exploring a 3-issue VLIW processor we create a candidate prototype which removes function unit $FU_3$ from issue-slot 2 and $FU_2$ from issue-slot 3. We notice that the compiler did not require $FU_2$ from issue-slot 1 and $FU_1$ from issue-slot 3 (cf. figure 3a). When, later in the exploration, we try a similar architecture but now also disable $FU_2$ from issue-slot 1 we expect that the result will be as shown in figure 3b and has the same performance metrics as we found for the first situation.

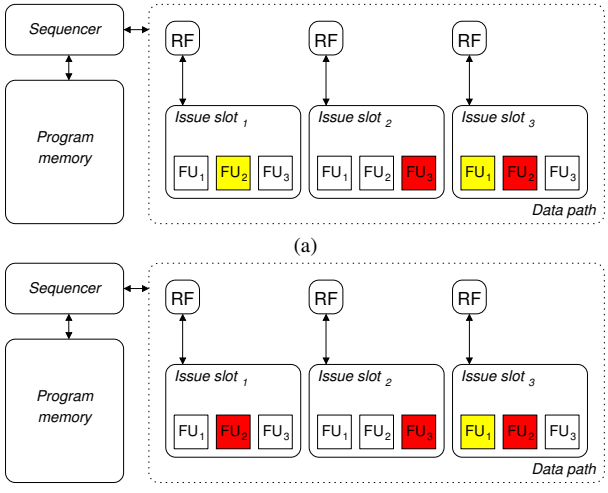



Fig. 3. Two equivalent architectures showing resources removed during the selection of candidates (marked red) and unused resources after compilation (marked yellow)

The compilation cache detects such cases by registering which resources are unused for previous prototypes in correspondence to the list of resources which were explicitly disabled in that prototype. Any candidate architecture which explicitly disables all resources that were also disabled in the cached prototype plus a subset of its unused resources

is considered a hit on the compilation cache. The cost metrics (energy, area, and cycle count) of the previous prototype are returned immediately and no cost estimation is performed on the new candidate. Candidate architectures which do not provide a hit on the compilation cache will be added as new entries after their cost has been estimated.

### B. The simulation cache

The simulation cache builds upon our efficient cost estimation method and is responsible for automatically switching between the hybrid and profile-based cost estimation methods. It keeps track of changes in key loop transformations (we currently track changes in software pipelining) and will use the hybrid method to update the application profile when changes are detected. The detection of if-conversion (and other transformations which remove basic-blocks) is handled by the estimator itself and do not require an updated profile. The BuildMaster framework is currently not able to properly detect transformations such as loop unrolling or loop peeling as the used compiler does not provide information about the effects of these transformations in a useful way. Properly detecting these transformations from the compiler output would either require changes to the compiler output or an extensive analysis of the generated assembly code and is considered to be outside the scope of this work.

Previously extracted profiles are cached and indexed based on a hash-table storing hashes of a string representation of the loop transformations applied during their corresponding compilation. This hash-table allows us to efficiently detect when an applicable profile exists. When a matching profile is found, we use the profile-based cost-estimation method. If such a profile does not exist, we fall-back onto the hybrid method and add the profile to the simulation cache for later use. This simple but effective method allows us to reliably find applicable profiles and can easily be extended when information regarding other code-structure changing transformations becomes available.

## V. Experiments

We have implemented the BuildMaster framework and integrated it into our design space exploration framework [13]. This allowed us to test various cache configurations under different exploration runs. In this section we compare the differences between enabling and disabling either one or both the compilation cache and the simulation cache. Our experiments show the speedup of the total design space exploration (cf. figure 4 and figure 5) as well as the cache hit-rates when both caches are enabled (cf. figure 6).

The experiments have been performed using six applications from different application domains and having different characteristics which have been prepared for usage with our exploration framework. Two ECG hart-beat detection applications ECG-A and ECG-B (referred to as A and B in the figures), two AES encryption and decryption applications, one using a small test sequence (C) and one using a large test sequence (D), 2D down sampling (E), and a low-pass spatial filter (LPSF, F) were selected for the experiments.

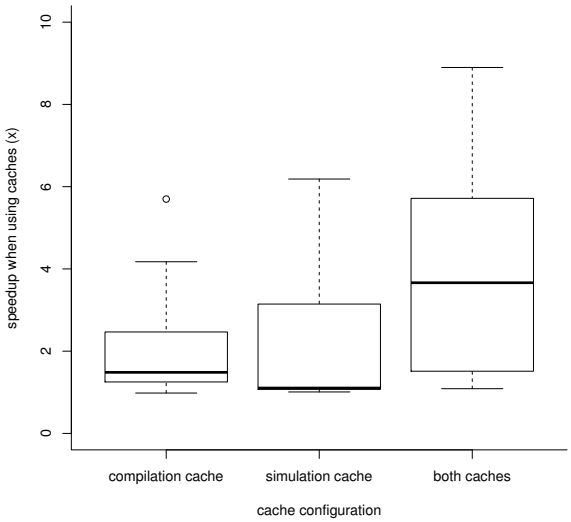



Fig. 4. A boxplot showing the exploration-time speedup ranges using different caching strategies

Each application was explored using two different exploration strategies (taken from [13]) referred to as *best-match* and *first-match*. Both strategies perform a greedy search through the processor architecture design space, first-match removing each resource which brings an improvement on the design cost, while best-match is considering all remaining alternatives for the removal of a single resource and selecting the resource that provides the largest gain. The experiments used the energy-delay product as a criterion to guide the selection process but other cost-functions should yield similar results with regard to the cache performance.

### A. Exploration time speedup

Figure 4 shows the obtained speedup for different architecture exploration runs. We observed no cases where the addition of either cache resulted in a slow-down of the exploration (speedup $< 1$) and found that in most cases the exploration time was significantly reduced. Especially, the exploration time of the more complex applications seems to be strongly decreased by the presence of the caches. The geometric mean of the speedup when only using the compilation cache was 1.8, when only using the simulation cache it was 1.7, and when using both cache levels it was 3.0. From this we conclude that both caches are roughly equally effective over our set of experiments.

Looking in more depth into our experiments we see that some of them show a greater benefit from the compilation cache while some others benefit more from the simulation cache. Figure 5 provides a more detailed view. Applications C and D demonstrate the impact of the size of the input data. Traditionally it costs a large amount of time to simulate the target application with a large dataset, smaller datasets are usually considered in an attempt to keep the design space exploration time within reasonable bounds. However, we noticed in our experiments that using a too small dataset has a substantial impact on the quality of the final architecture.

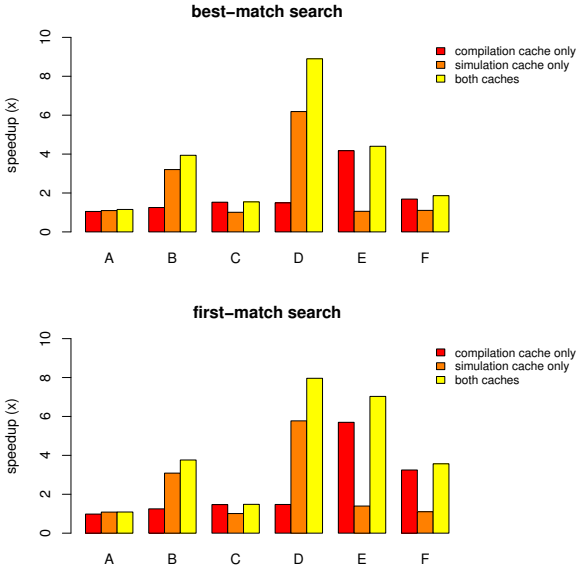



Fig. 5. Exploration-time speedup ranges using different caching strategies

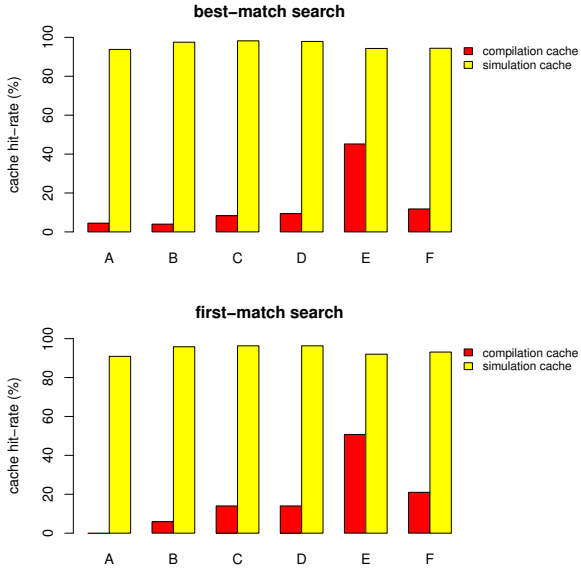



Fig. 6. Observed cache hit-rates for two different architecture exploration strategies

Application D shows us (under both exploration strategies) a that a larger architecture is found to be cost efficient for the AES encryption when realistic input data is used. Applications with a relatively small input size, and especially those with a large final architecture (such as E and F) tend to have more benefit from the compilation cache. This can be explained through the size of the final architectures (5 and 7 issue-width VLIW processors respectively) with a high specialization of each issue-slot. Exploring such highly specialized wide VLIW processors requires many small steps when exploring the function-unit composition (i.e. defining which operations have to be available in each issue-slot). These many small exploration steps are more likely to trigger hits in the compilation cache when a large architecture is considered. Furthermore, applications E and F both have several kernels which are software-pipelined, this makes the compilation and scheduling problem for these applications more difficult and thus more time consuming than for the other applications.

The influence of the input data size makes it difficult to precisely compare the proposed caching methods to the traditional exploration without caching. It is clear that a lot can be gained and that this method allows for an efficient usage of larger input datasets. This last feature substantially helps us in creating new processor designs which are better tuned to their specific usage but makes comparison based on only the exploration time incomplete.

### B. Cache hit-rates

The hit-rates of both caches may give us a better insight into the actual benefits of the caching. Figure 6 shows the hit-rates observed in our experiments for both caches. Observe that the simulation cache is very effective and consistently gets hit-rates above 90% for all of our experiments (95% on average). This can be translated directly into the observed speedup as it allows us to skip over 90% of the simulation runs when compared to the traditional methods.

The speedup related to the compilation cache is more difficult to quantify. From figure 6 we can see, for example, that application E has a compilation cache hit-rate of approximately 50%. While this does allow us to save quite some exploration time, it does not fully explain the observed 4-6x speedup in the exploration time measurements. A secondary effect, referred to as *caching induced exploration-path divergence*, is the cause of this.

### C. Caching induced exploration path divergence

Caching induced exploration-path divergence is mainly observed in the compilation cache but can also be seen in the simulation cache. This divergence happens when a cached value is returned which is different from the actual value that would have been found without the cache. Table I gives two example applications to illustrate these effects. Here we observe the typical symptom of a caching induced exploration path divergence: The total of cache hits and misses when the cache is enabled does not always equal the total misses when the cache is disabled. This implies that sometimes a different number of design points is considered depending on the fact if the compilation cache is enabled or not. This is a clear sign of the caching induced exploration path divergence. This divergence caused the exploration to find a different *final improvement* for the LPSF application (application F) when the compiler cache was enabled (cache configurations 1 and 2). We see a similar effect for the simulation cache with the down-sampling application (application E) where we also find that enabling the cache yields slightly different improvement of the considered cost function (cache configurations 1 and 3).

In the simulation cache, exploration path divergence can happen when the compiler uses a transformation which is not properly detected by the simulation cache. In the case of the down-sampling application (application E) the optimization to blame was the loop peeling which was performed as part of

TABLE I

DETAILED EXPERIMENTAL RESULTS SHOWING CACHING INDUCED EXPLORATION PATH DIVERGENCE ON APPLICATIONS E AND F WHEN USING FIRST-MATCH SEARCH. THE *cache configuration* COLUMN REFERS TO THE FOLLOWING CACHE CONFIGURATIONS: *1)* BOTH CACHES ENABLED, *2)* COMPILATION CACHE ONLY, *3)* SIMULATION CACHE ONLY, AND *4)* BOTH CACHES DISABLED. THE TIME COLUMN PRESENTS THE TOTAL EXPLORATION TIME IN SECONDS.

| | Compile cache | | | Simulator cache | | | final exploration result | | cache |
|---|---|---|---|---|---|---|---|---|---|
| benchmark | hit-rate (%) | hits | misses | hit-rate (%) | hits | misses | improvement | time | configuration |
| Down-sampling | 50.7 | 38 | 37 | 92.0 | 23 | 2 | 1.112918 | 801 | 1 |
| | 48.1 | 38 | 41 | — | 0 | 29 | 1.122509 | 988 | 2 |
| | — | 0 | 143 | 98.5 | 129 | 2 | 1.112918 | 4038 | 3 |
| | — | 0 | 184 | — | 0 | 172 | 1.122509 | 5631 | 4 |
| LPSF | 21.0 | 17 | 64 | 93.1 | 54 | 4 | 4.324151 | 2940 | 1 |
| | 21.0 | 17 | 64 | — | 0 | 58 | 4.324151 | 3229 | 2 |
| | — | 0 | 195 | 96.8 | 183 | 6 | 4.379904 | 9514 | 3 |
| | — | 0 | 195 | — | 0 | 189 | 4.379904 | 10485 | 4 |

the software-pipelining. In this application, one execution of a loop kernel was moved from the loop core into the prologue, resulting in a slightly decreased loop count. The BuildMaster framework is currently not able to properly detect the loop peeling transformation as no direct information on this kind of transformations is available from the compiler output. We observed only this single simulation cache induced divergence in our experiments.

Our experiments show that the compilation cache is much more susceptible to caching induced exploration path divergence. Only for the most simple application in our benchmark set (application A) the cache hits and misses add up to the number of points considered when compilation caching is disabled. Based on our experiments we can formulate our hypothesis that the main reason for compilation cache induced divergence is the sensitivity of the compiler heuristics to the set of available resources. This can cause the compiler to find a different schedule when a simplified version of the same problem is presented. However, the exploration path divergence was only observed in a single experiment (application F), producing a different final design. It was also only observed when the first-match heuristic was selected.

So far, we have only observed the above two cases where the caching induced exploration path divergence resulted in a different final design. However, in both these cases the cost of the final design cost improvement found without caching was approximately only 1% lower than the cost of the final design found when using caching, while caching helped to greatly reduce the total exploration time.

## VI. CONCLUSION

In this paper we have presented and discussed the BuildMaster framework. This framework offers a very effective and efficient automated caching of intermediate compilation and simulation results during the design space exploration of VLIW ASIPs. Both the compilation and the simulation cache can facilitate the reduction of the architecture exploration time and make it possible to efficiently use more realistic larger datasets for the evaluation of the proposed designs. The presented caching methods become more and more effective for larger applications using more realistic larger input datasets. This is a very useful feature. Due to this feature our framework can contribute towards the construction of higher quality VLIW ASIPs better specialized to particular applications, while at the same time strongly reducing their design time.

## REFERENCES


[1] R. Jordans, R. Corvino, L. Jóźwiak, and H. Corporaal, "An efficient method for energy estimation of application specific instruction-set processors," in *DSD 2013 - 16th Euromicro Conference on Digital System Design*, Santander, Spain, September 2013, pp. 471–474.

[2] 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.

[3] S. Aditya, B. Rau, and V. Kathail, "Automatic architecture synthesis and compiler retargeting for VLIW and EPIC processors," in *Proc. of ISSS*, vol. 9, 1999.

[4] H. Corporaal and J. Hoogerbrugge, "Cosynthesis with the MOVE framework," in *Symposium on Modelling, Analysis, and Simulation*, 1996, pp. 184–189.

[5] 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. 1–11, August 2011.

[6] T. Pitkänen, T. Rantanen, A. Cilio, and J. Takala, "Hardware cost estimation for application-specific processor design," *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 251–264, 2005.

[7] S. Ashby, L. A. Tuura, G. Eulisse, and S. Schmid, "Parallel compilation of cms software," CERN-CMS-CR-2004-051, Tech. Rep., 2004.

[8] E. Thiele, "Compilercache." [Online]. Available: http://www.erikyyy.de/compilercache

[9] A. Tridgell, "ccache." [Online]. Available: http://ccache.samba.org

[10] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, E. Rohou *et al.*, "Iterative compilation in a non-linear optimisation space," in *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[11] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*. IEEE, 2003, pp. 204–215.

[12] G. Fursin, M. F. OBoyle, and P. M. Knijnenburg, "Evaluating iterative compilation," in *Languages and Compilers for Parallel Computing*. Springer, 2005, pp. 362–376.

[13] R. Jordans, R. Corvino, L. Jóźwiak, and H. Corporaal, "Instruction-set architecture exploration strategies for deeply clustered vliw asips," in *ECyPS 2013 - EUROMICRO/IEEE Workshop on Embedded and Cyber-Physical Systems*, Budva, Montenegro, June 2013, pp. 38–41.

[14] L. Jozwiak, M. Lindwer, R. Corvino, P. Meloni, L. Micconi, J. Madsen, E. Diken, D. Gangadharan, R. Jordans, S. Pomata, P. Pop, G. Tuveri, L. Raffo, and G. Notarangelo, "Asam: Automatic architecture synthesis and application mapping," *Microprocessors and Microsystems*, vol. 37, no. 8, pp. 1002–1019, October 2013.