Instruction-set architecture synthesis for VLIW processors

Instruction-set Architecture Synthesis for VLIW Processors

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op dinsdag 1 december 2015 om 14.00 uur

door

Roel Jordans

geboren te Roosendaal en Nispen

Dit proefschrift is goedgekeurd door de promotor en de samenstelling van de commissie is als volgt:

voorzitter:	prof.dr.ir. A.C.P.M. Backx
1^e promotor:	prof.dr. H. Corporaal
copromotor:	dr. L. Jóźwiak
leden:	prof.dr.Tech. J.H. Takala MSc (Tampere University of Technology)
	prof.dr. K.L.M. Bertels (Technische Universiteit Delft)
	prof.dr.ir. P.H.N. de With
adviseurs:	dr.ir. J.A.J. Leijten (Intel Benelux)
	dr.ir. B. Mesman

Instruction-set Architecture Synthesis for VLIW Processors

Roel Jordans

This work is supported in part by the Artemis Joint Undertaking, project ASAM 100265.

Cover design by Roel Jordans

Printed by CPI-Koninklijke Wöhrmann – The Netherlands

A catologue record is available from the Eindhoven University of Technology Library.
ISBN: 978-90-386-3963-5

Summary

Instruction-set Architecture Synthesis for VLIW Processors

The high energy efficiency and performance demands of image and signal processing components of modern mobile and autonomous applications have resulted in a situation where it is no longer feasible to only use general purpose processing systems to serve those applications. This has caused a strong shift to heterogeneous systems containing multiple highly specialized processors. While some tool support for the design process of such specialized processor architectures exists, key decisions are still made by human designers, usually based on incomplete and imprecise information. Combining this with the short interval between different product generations and limited design times strongly reduces the number of design alternatives that can be considered, and results in a sub-optimal design quality.

Current state-of-the-art technologies offer design automation for several steps of the design process by automating key activities such as the construction of a processor architecture from a high level description, the evaluation of candidate designs through simulation or emulation, or proposing extensions to an existing processor architecture. While these tools already substantially improve the design times over a completely manual design, further significant improvements can still be obtained, specifically through automation of the design analysis and decision making process. This dissertation proposes several significant improvements of the design effectiveness and efficiency through automation of several stages of the design process.

A three step approach to processor architecture design is presented which starts by using our new application analysis methods to obtain parallelism and performance estimates for the various compute intensive parts of the target application. These estimates are then used during an application restructuring phase which aims at improving the available parallelism and decides upon the mapping of application data into the processors internal memories. Taking this transformed version of the target application, its memory hierarchy as defined by the memory mapping, and the parallelism estimates allows us to propose an initial processor architecture which completes the second step. The third step is then to further refine the processor architecture and results in a highly specialized processor architecture description.

The research presented in this dissertation focusses on improving the following steps in the design process.

- A parallelism estimation method for estimating the instruction-level parallelism exposed by the application is presented. This method provides parallelism feedback used during the exploration of the application restructuring, but is also used for determining the appropriate number of issue-slots in the initial architecture. As a result, we are able to construct an initial processor architecture that both meets the performance requirements for the target application yet still is reasonably close to the final refined processor design.
- A processor architecture refinement method which allows us to avoid the (time consuming) construction of intermediate candidate processor architectures. Our approach only needs to construct both the initial and refined designs, all other considered candidate architectures need not be constructed.
- A rapid energy consumption methodology which combines the block execution profile of a simulation of the target application with its scheduled assembly listing. This makes our energy estimation method independent of the number of simulated processor clock cycles and enables the use of larger, more representative, input data sets, thus allowing for a both a faster and more realistic evaluation of the candidate designs.
- An architecture exploration framework called BuildMaster, which simplifies the implementation of our architecture refinement exploration strategies. This framework automatically detects when compilation and simulation results obtained for previously considered candidates can be re-used for the evaluation of newly proposed candidate architectures. This intermediate result caching system allows us, for example, to avoid on average over 90% of the originally required simulation time by re-using previously obtained profile information for the energy estimation.
- A set of exploration strategies which effectively refine the processor architecture and a comparison between these strategies on both the quality of the obtained result, as well as, the required exploration time. We show that the proposed exploration heuristics find results whose quality is comparable to the results found using a genetic algorithm while requiring an order of magnitude less exploration time.

Combining the presented techniques results in a highly efficient and extensible instruction-set architecture exploration methodology. In our experiments we show that our framework is able to explore hundreds of processor architecture variations per hour while consistently producing compact results that meet the expected performance.

Contents

The last thing one settles in writing a book is what one should put in first.

Blaise Pascal, “Pensées”, 1670

1 Introduction

We live in an era of electronic systems that can be found everywhere around us and, in some cases, even inside us. Obvious ones are the computers we have on our desks and in our pockets. Less obvious ones are embedded inside bigger systems, often without being noticeable separate from the whole product containing them. It is very difficult to make an accurate estimate of how many processors are currently in the world as parts of these *embedded systems*. It is however safe to say that embedded processors outnumber those present in the more conventional stand-alone computers by a large margin. For example, a contemporary smartphone contains approximately 10 processors (e.g., baseband (radio) processing, real-time video encoding/decoding, audio processing, encryption, several general purpose processors, etc.), while modern cars such as the Mercedes S-class and BMW 7 have over 60 processors (e.g., fuel injection, navigation, anti-lock breaking (ABS), in vehicle entertainment, etc.)[1].

Nowadays, such a combination of one or more embedded computing systems with mechatronics or other physical systems is often referred to as a cyber physical system. This combination of diverse yet combined systems presents complex demands on the communication and computation capabilities. A complex heterogeneous cyber physical system usually includes various kinds of information processing and involves several types of parallelism. It is therefore usually best served using a heterogeneous computing system composed of several different parallel processors. For many of the applications standard off-the-shelf embedded

[1] “The Dozens of Computers That Make Modern Cars Go (and Stop)” – `http://www.nytimes.com/2010/02/05/technology/05electronics.html`

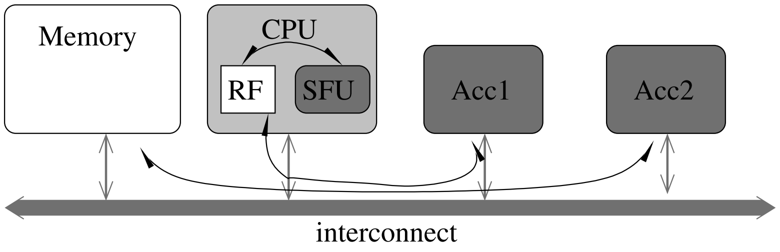


Figure 1.1: *Different types of accelerators illustrating data movement to and from the accelerator*

processors suffice. However, when performance or battery life becomes critical, these standard processors usually do not provide the satisfactory performance levels and/or performance/power trade-offs. Application specific instruction-set processors (ASIPs) and hardware accelerators provide much more freedom and can be used in such cases. As with all customizations, different specific application requirements result in different systems. In many cases, a hardware accelerator can be added to an existing off-the-shelf processor to achieve the required performance. This allows the designer of the system to increase the efficiency of the system by executing a part of the application in hardware. It leads to a highly efficient implementation, but limits the flexibility and re-use possibilities of the system. Later generations of the same product commonly contain variations of the same application which might require a re-design of the system, because its hardware accelerator part (if not reconfigurable) does not provide any possibility for adaption to new requirements.

Implementing accelerator hardware into a system design can be achieved in various ways. Figure 1.1 illustrates the three most commonly used methods.

1. Small accelerators can be implemented as instruction-set extensions of an existing processor (CPU) through the addition of a specialized function unit (SFU). A key advantage of this method is the close connection between the accelerator and the existing data-path of the processor. This results in a low energy overhead from transferring input values to the accelerator, and makes it possible to efficiently accelerate smaller parts of the code. A common constraint for accelerators of this type is that they usually do not allow any form of control-flow within the accelerated application part.

2. Larger accelerators are usually constructed outside of the processor. This allows for more complex functionality which may include more irregular control-flow within the accelerated part. In the case of *Acc1* in our example, the input values of the accelerated application part are programmed directly from the central processor. After that, the accelerated program is executed and the results are copied back again by the processor. This type of

accelerator requires active control from the processor which makes large data transfers relatively costly in both the required transfer time and energy consumption.

3. Incorporating direct memory access (DMA) into the accelerator is commonly the preferred method when designing accelerators capable of handling larger amounts of data, *Acc2* is an example of such an accelerator. This removes the CPU from the main data transfer path, which may improve the available data bandwidth if the required data rate was not supported by the processor, but adds further complexity to the accelerator.

In general, hardware accelerators such as *Acc1* and *Acc2* are mostly used when a larger non-changing part of an application can be offloaded onto the accelerator. Typical examples of such applications include the encryption and compression algorithms that are parts of communication standards. However, for most of the other applications, some form of reconfiguration of the accelerator may be required in order to keep up with evolving standards and new similar applications. Such reconfiguration can either be achieved by a tighter integration between smaller hardware accelerators and the processor (i.e. using one or more SFUs), or by adding programming capabilities to larger accelerators (which changes them into highly-specialized application specific processors themselves). The smaller size of the extensions and the programmable nature of the processor make it easier to re-combine the accelerator functionality when new versions of the target application need to be supported.

In parallel to the inclusion of specialized hardware, be it realized using hardware accelerators or instruction-set extensions, both the temporal performance and energy consumption can usually be much improved by increasing the parallelism with which the application is executed. Usually, a more parallel execution of an application enables a more substantial decrease of the frequency at which the processor system needs to work, without breaking any of the temporal constraints of the application. In turn, lowering the frequency of the processor allows for a lower supply voltage which leads to a lower energy consumption.

1.1 Parallelism in processor architectures

The maximal amount of parallelism that can effectively be exploited for a given application is determined by the structure of the application itself. For instance, there is a limit on the effectiveness of the parallelism increase of the processor architecture which has been described by Amdahl's Law. Gene Amdahl argued that the speedup of a program using multiple parallel processors (or processing elements) is limited by the processing time of the sequential part of the application

[2]. Amdahl's Law can be generalized as Equation 1.1

$$Speedup(N) = \frac{1}{S + \frac{1-S}{N}} - O_N \tag{1.1}$$

with S the serial percentage of the workload (expressed as a decimal between 0 and 1), N the number of processor cores, and O_N the parallelization overhead for N threads.

A simplified form case of Equation 1.1 can be formulated to estimate an upper limit on the speedup when ignoring the parallelization overhead and assuming an unlimited number of processor cores:

$$Speedup(upper\ limit) = \frac{1}{S} \tag{1.2}$$

For example, if 95% of an application can be parallelized and the remaining 5% can not, then the execution time of the parallelized application is limited to be at least 5% of the original execution time, which limits the maximal speedup that can be obtained to 20x.

At a first glance, Amdahl's Law seems to put a strict bound on the usefulness of increasing parallelism in processor architectures. However, a later observation by Gustafson [30] counteracts this. Gustafson observed that the parallel portion of an application is not a constant, but grows proportionally to the increasing processing power of the system as described by Equation 1.3.

$$Speedup(N) = S + N(1 - S) - O_N \tag{1.3}$$

This relation, known as Gustafson's Trend, can easily be observed in the evolution of computer games over the last decades; as computational resources increased, so did the sophistication of computer games, both in terms of higher resolution graphics and more detailed physics modeling [27]. Similar trends can be observed in different fields as well, for example, Gustafson did his observations when working with large scale fluid dynamics simulation on a 1024-processor system. In his case, increased processor capacity generally resulted in simulations with higher grid resolution, more time steps, and increased difference operator complexity [30].

1.1.1 Different kinds of parallelism

Several different kinds of parallelism can be recognized within an application depending on the granularity of the parallelism.

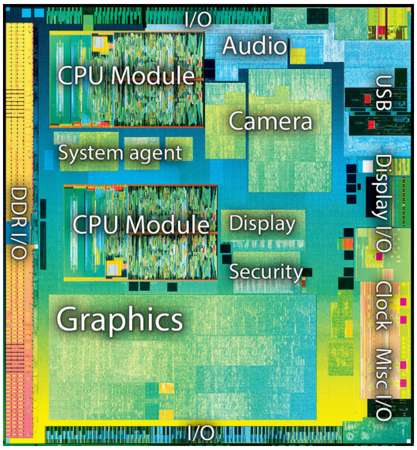
Task-level parallelism (TLP) An application may be composed of a system of processing (communicating) application parts which can be executed in parallel on different processors of a multi-processor system. Such an application part is usually referred to as a task. Different tasks may have different processing requirements and specialized hardware may be provided for an efficient execution of each specific task. Key to task-level parallelism is the fact that different tasks are executed using independent instruction streams, either through running tasks in a multi-processor system or using multi-threading on a shared processor. This form of parallelism is sometimes also referred to as thread level parallelism.

Instruction-level parallelism (ILP) Instructions, the basic steps of the program execution, can also be executed in parallel when they do not depend on each other's result. Superscalar processors contain multiple operation execution pipelines and determine at runtime which instructions can be executed on which execution unit. Many modern general purpose processors use a superscalar design internally, however, this comes at a price. The additional hardware for the instruction scheduling logic can have a significant impact on the overall area and energy consumption of the processor. Explicitly programmed instruction-set processors partially avoid this hardware overhead by moving the scheduling decisions to the compiler and explicitly encode which operations get executed into the program memory of a processor. This simplifies the processor design at the cost of extra program memory and a highly complex compilation process. Very Long Instruction Word (VLIW) processors, as considered in this dissertation, are an example of such explicitly programmed processors that can efficiently exploit ILP.

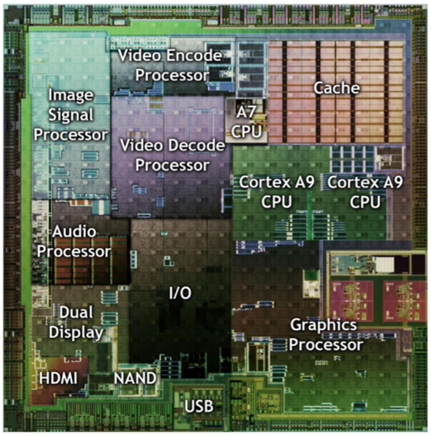
Operation-level parallelism (OLP) Frequently occurring patterns of basic operations can be combined into complex operations and implemented as instruction-set extensions. A common example of a complex operation is the multiply-and-add operation that can be found in many digital signal processing (DSP) designs. However, more complex operations, for example, implementing a partial Fourier transform or a single step of an encryption program, can also be provided. Such complex operations are closely related to hardware accelerators, the main difference being the tight coupling with the processor, which makes it possible to accelerate smaller operation sequences and reduces the communication overhead compared to an external accelerator.

Data-level parallelism (DLP) The same operations may have to be executed on several parallel data items. Specifically, the computations within an application may sometimes be written as mathematical vector operations where the same basic operation gets applied to several (preferably many) data elements. Image and signal processing applications commonly have large parts which exhibit data-level parallelism.

It is important to observe that some of these kinds of parallelism strongly overlap on which algorithms and applications they can be applied. However, their implementations do differ significantly and the selection of one or more



(a) Intel Atom Z3770*



(b) Nvidia Tegra 2†

Figure 1.2: *Two competing MPSoC's commonly found in current smartphones (no relative scaling of die sizes implied)*

*Source: `http://tweakers.net/reviews/3162/2/intels-atom-bay-trail-de-eerste-nieuwe-atom-in-vijf-jaar-zes-verschillende-bay-trails.html`

†Source: `http://www.anandtech.com/show/4144/lg-optimus-2x-nvidia-tegra-2-review-the-first-dual-core-smartphone/3`

kinds of parallelism to implement for a specific processor architecture will depend on the combination of algorithms and applications executed on it, as well as, the flexibility (programability) demands for its future uses. For example, while it may be possible to distribute an application that presents a high level of data-level parallelism across different tasks in a multi-processor system, doing so might not result in the most efficient overall system.

1.1.2 Real life examples

Task-level parallelism is commonly supported using several processors, e.g. a Multi-Processor System-on-Chip (MPSoC). Such an MPSoC usually contains one or more standard processors, together with several specialized (programmable) accelerators which efficiently handle various high-performance tasks. Figure 1.2 shows the chip die photographs of two common MPSoCs from competing manufacturers; the different processor blocks are marked in the figure. It can be observed that the general purpose processor part (marked CPU) represents only a fraction of the total chip area. The remaining marked blocks represent special purpose accelerators. Such special purpose accelerators are often programmable processors by themselves, specially designed for the type of tasks that they are supposed to execute. The tailoring of such an accelerator to a certain application includes providing the processor with the ability to execute multiple operations of the application in parallel in a VLIW instruction, as well as, the addition of function units implementing complex operation patterns that can be executed as parts of an even more complex (VLIW) instruction. Such a specialized processor is usually called an Application Specific Instruction-Set Processor (ASIP). Next to the ASIP blocks, a MPSoC often also contains one or more non-programmable accelerators. This non-programmable hardware provides a very efficient implementation of a set of fixed algorithms. The non-programmable nature makes this logic much less flexible in the face of evolving standards and the introduction of novel algorithms, but it increases efficiency and security, because the fixed implementation makes malicious modification of the implemented algorithm extremely difficult. In general, all non-safety-critical and non-performance-critical, but still high-performance, application parts that still require acceleration or improved energy efficiency, are nowadays implemented as programmable ASIPs to enable software updates for the system, so that the system can efficiently support future standards and late design modifications. The added complexity required for making an ASIP programmable can often be kept within reason, making the energy efficiency of an ASIP much more close to that of a non-programmable hardware accelerator than to that of a general purpose processor.

When creating a new ASIP, the designer usually starts with an existing (general purpose) processor and either *a)* extends this processor with complex custom operations to increase efficiency of specific algorithm parts (increasing the OLP), or *b)* starts by adding parallel execution units which increases the processor's ability to execute more operations in parallel (increasing the DLP and/or ILP). Both approaches can result in efficiently programmable ASIPs and are often combined when very tight performance constraints need to be met.

1.2 Context of this work

The work presented in this dissertation was performed as part of the ASAM project[2]. In brief, the goal of the ASAM project was to automate the process of designing a new MPSoCs based on ASIP blocks which are designed automatically and concurrently with the MPSoC. For this purpose, tightly cooperating *macro-architecture* and *micro-architecture* exploration stages are envisioned [38], as shown in Figure 1.3. The macro-architecture exploration is responsible for designing the MPSoC containing several ASIPs providing TLP, whereas the micro-architecture exploration designs single ASIP blocks and implements DLP, OLP, and ILP. This directly illustrates both the necessity and difficulty of such an undertaking; the macro-architecture exploration will require information about the performance of the ASIPs that will be designed in order to decide which

[2]Automatic Architecture Synthesis and Application Mapping – `http://www.asam-project.org`

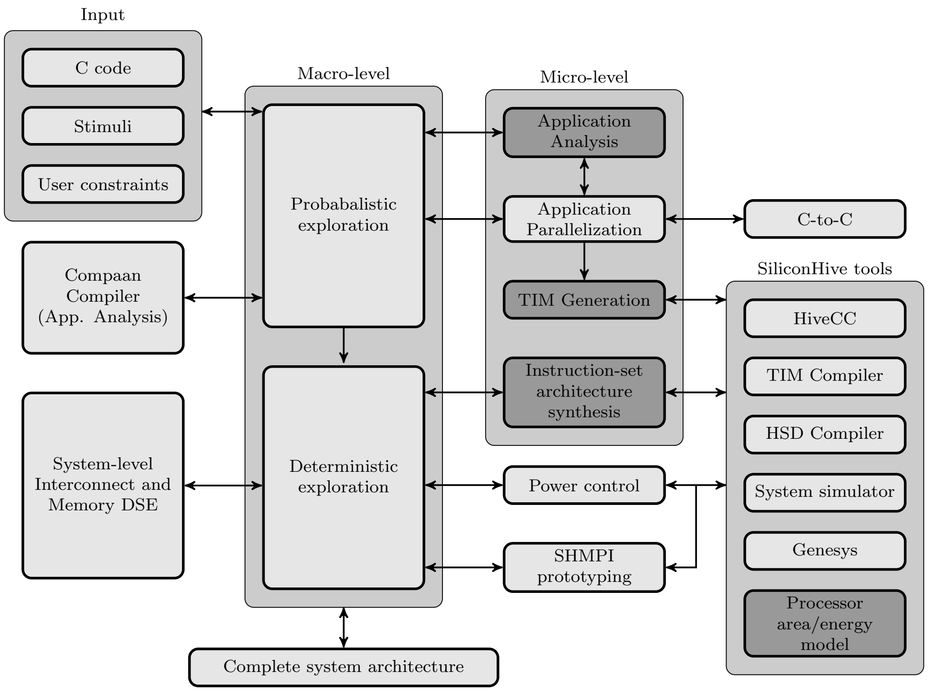

Input
C code
Stimuli
User constraints
Compaan Compiler (App. Analysis)
System-level Interconnect and Memory DSE
Macro-level
Probabalistic exploration
Deterministic exploration
Complete system architecture
Micro-level
Application Analysis
Application Parallelization
TIM Generation
Instruction-set architecture synthesis
Power control
SHMPI prototyping
C-to-C
SiliconHive tools
HiveCC
TIM Compiler
HSD Compiler
System simulator
Genesys
Processor area/energy model

Figure 1.3: *An overview of the MPSoC design flow developed for the ASAM project illustrating the macro- and micro-level architecture exploration showing the contributions of this thesis in a darker shade. A detailed description of this flow is presented in Chapter 3.*

parts of the application to execute where, while the micro-architecture exploration needs to know the tasks that will be mapped onto a particular ASIP in order to propose its architecture which will determine its performance. It is a circular dependence between the marco- and micro-architecture design space exploration. In the ASAM project, the design phase ordering problem is solved within the macro-architecture exploration using early best-case and worst-case performance estimates for executing separate tasks on an ASIP, and solving the more detailed architectural decisions during the design of each individual ASIP in the later micro-architecture exploration phases. This way the macro-architecture design space exploration produces a MPSoC proposal and the micro-architecture design space exploration elaborates the proposal and provides feedback on its performance characteristics to the macro-architecture exploration. The process is repeated until a satisfactory MPSoC design is obtained.

The micro-level architecture exploration is subdivided into three phases to further split the VLIW architecture synthesis problem into more manageable steps and to enable an efficient structured interaction between the macro- and micro-level exploration. These three phases are as follows:

- Application analysis
- Application parallelization and coarse ASIP synthesis
- ASIP instruction-set architecture synthesis

The PhD project presented in this dissertation is focused on the last phase of the instruction-set architecture synthesis, but also contributed to the two earlier phases.

1.3 Problem statement

This dissertation presents the work performed as part of the ASAM micro-architecture exploration and synthesis phase. A three step approach to VLIW ASIP architecture design is proposed which starts by using our application analysis methods to obtain parallelism and performance estimates for the various compute intensive parts of the target application. These estimates are then used during an application restructuring step. This restructuring improves the exploitation of available parallelism, performs the actual application parallelization, and decides the mapping of application data into the processor's internal memories. Taking this transformed version of the target application, its memory hierarchy as defined by the memory mapping, and the predicted parallelism (based on the earlier estimates) allows us to propose a coarse initial processor architecture. The application's parallel execution structure and a corresponding coarse ASIP architecture, including the number of parallel ASIP memories, is then constructed based on the generated mapping. This proposed ASIP architecture defines the internal memory hierarchy, an initial internal communication structure, and a preliminary set of issue-slots and register files. The goals of this second step are to provide an initial ASIP and application pair which already approximates the required temporal performance, but still is composed of (possibly over-dimensioned) ASIP building blocks from a standard library. The third step, instruction-set architecture synthesis, is then to further refine this coarse initial processor architecture through specialization of the issue-slots and optimization of the register files and interconnect. Completing this third step results in a highly specialized processor architecture with a highly specialized application specific instruction-set, capable of efficiently running the target application.

The research presented in this dissertation focusses on automatic ASIP instruction-set architecture synthesis, as well as, the closely related performance estimation of an application specific hardware/software (sub-)system implemented on a single ASIP. In the scope of this research, a set of effective and efficient methods and automatic tool prototypes had to be researched, developed, and experimentally validated, in order to enable such an instruction-set architecture synthesis as was required within the ASAM project. Several key problems have been identified in this process which are limiting factors for implementing an efficient and effective processor architecture exploration. The main identified problems are as follows:

1. Both the distribution of tasks on a, yet to be constructed, MPSoC platform, as well as, the application restructuring step, require early estimates on the kinds of parallelism available in a particular application part and their expected performance. High quality parallelism and execution time estimates help by both improving the selection of the proper task distribution among the processors in a MPSoC, but also aid the construction of initial ASIP and MPSoC architecture proposals that, through this, have a chance to be closer to the final design.

2. The current state-of-the-art implementations for the evaluation of proposed candidate architectures commonly depend on an activity trace of (part of) the target application. Both obtaining and processing such a trace can be very time consuming, which limits the effectiveness of the architecture exploration by forcing the use of (less representative) shorter execution traces.

3. Implementing different exploration strategies efficiently implies thorough tracking of previously explored design points. When getting closer to a final architecture, many design points will differ only slightly. Recognizing when previously obtained results are available for re-use offers an opportunity for a substantial exploration efficiency improvement. This intermediate result tracking is, to a large degree, independent of the exploration strategy.

4. State-of-the-art processor architecture exploration methods need to construct and analyze each proposed candidate processor architecture. This is a very time consuming process which significantly impacts the exploration efficiency and should be avoided whenever possible.

5. Refining the instruction-set architecture of an initially proposed ASIP architecture is a process that involves proposing and comparing many different candidate architectures. A smart candidate construction and selection strategy is key to an efficient exploration.

The aim of the work presented in this dissertation is to address the above problems and provide satisfactory solutions.

1.4 Contributions

The research presented in this dissertation contributes substantial improvements to the following steps in the ASIP architecture design process.

1. *A method for estimation of the instruction-level parallelism* exposed by the application is presented. This method provides a measurement of available parallelism used during the exploration of the application restructuring, as well as, for determining the appropriate number of issue-slots in the initial ASIP architecture. In result of using the parallelism estimates, we are able to construct an initial ASIP architecture that both meets the performance requirements for the target application and is reasonably close to the final refined ASIP architecture, which both accelerates the final architecture design and enables reasonably accurate early feedback on ASIP performance characteristics (Chapter 4).

2. *A rapid energy consumption estimation methodology* which combines the block execution profile from a simulation of the target application with its scheduled assembly listing. This makes our energy estimation method independent of the number of simulated processor clock cycles, and in consequence, enables an efficient use of larger more representative input data sets, allowing for both a faster and more realistic evaluation of the candidate designs (Chapter 5).

3. *An automatic architecture exploration framework called BuildMaster*, which simplifies the implementation of our architecture refinement exploration strategies. This framework automatically detects when the compilation and/or simulation results obtained for previously considered candidate architectures can be re-used for the evaluation of newly proposed candidate architectures. Doing so allows us to avoid many of the time-consuming compilation and simulation steps. This intermediate result caching system allows us, for example, to avoid on average over 90% of the originally required simulation time by re-using the previously obtained profile information for the energy estimation (Chapter 6).

4. *A generic processor architecture refinement method* which allows us to avoid the (time consuming) construction of intermediate candidate processor architectures. Our approach only needs to construct both the initial and refined designs; all other considered candidate architectures need not to actually be constructed (Chapter 7.1).

5. *A set of VLIW ASIP exploration strategies* which effectively refine the processor architecture and a comparison between these strategies in relation to both the quality of the obtained result, as well as, the required exploration time. We show that the proposed exploration heuristics find results of quality comparable to those found using a genetic algorithm, while requiring an order of magnitude less exploration time (Chapter 7.2-4).

Combining the above mentioned techniques results in a highly efficient automated instruction-set architecture exploration technology and provides an extensible framework for experimenting with different exploration strategies. In the experiments reported in this dissertation we show that our framework is able to explore hundreds of processor architecture variations per hour while consistently producing compact instruction-set architecture designs that meet the expected performance.

1.5 Dissertation outline

This dissertation is organized as follows:

Chapter 2 ***"Related work"***, presents a selection of recent work related to the automatic construction of VLIW ASIPs, including an introduction of the SiliconHive design flow and VLIW processor architecture template which was used as part of the ASAM project.

Chapter 3 ***"VLIW processor design in the ASAM project"***, introduces the three step ASIP design flow that was developed by the TU/e team of the ASAM project and discusses the proposed VLIW processor design methodology. It describes which tasks need to be performed during the various stages of the design process and how this is achieved using the methods presented in this dissertation.

Chapter 4 ***"Early performance estimation"***, demonstrates our methods for early best-case and worst-case performance estimation of an application part for a not-yet-designed VLIW processor architecture and evaluates the fitness of the presented methods for the ASAM design methodology.

Chapter 5 ***"Energy and area modeling"***, continues with a more in-depth discussion of our specific VLIW architecture template and discusses the architecture modeling (energy and area) that has been used in our architecture exploration tools and experiments.

Chapter 6 ***"Intermediate result caching"***, discusses our BuildMaster framework for effective processor architecture exploration. Many time-consuming steps are involved in an automated ASIP architecture exploration. Good management and reuse of previously obtained information can significantly help in avoiding many of these time-consuming steps which can significantly reduce the exploration time.

Chapter 7 ***"Automated design space exploration"***, proposes three methods for automated instruction-set architecture exploration and synthesis for VLIW processors and discusses their limitations and effectiveness.

Chapter 8 ***"Conclusions and future work"***, finalizes this dissertation with our conclusions and a discussion of the possible future work.

“The Guide says there is an art to flying”, said Ford, “or rather a knack. The knack lies in learning how to throw yourself at the ground and miss.”

Douglas Adams, “Life, the Universe and Everything”, 1982

2 Related work

The development of contemporary digital systems heavily relies on electronic design automation (EDA) tools. Placing, sizing, and connecting the 1 billion transistors of a contemporary MPSoC simply is not possible without a huge amount of fully automated design assistance. Historically, EDA tools focussed solely at placement and routing of transistors. However, over time this limited approach became infeasible as circuit complexity increased. As a result, EDA tools adapted libraries of higher-level standard components. Initially these components were simple logic gates (and, or, etc.), but later usage of only these small blocks also proved insufficient and larger so called Intellectual Property (IP) blocks were added to the libraries. These IP blocks can be as simple as a memory controller, but may also contain complete processors including local cache memories. Nowadays the design and support of such IP libraries has become an important part of the digital electronics design industry and the sole reason for the existence of companies such as ARM and Imagination Technologies.

Managing a system-level design containing several such complex IP blocks is a very complex task which requires highly specialized tools. Currently three major EDA tool vendors deliver such tools (Synopsys[1], Cadence[2], and Mentor Graphics[3]), and virtually everyone designing or using IP blocks will be using the EDA tools of one or more of these companies. Mentor Graphics, the smallest of the three, focusses mostly on the realization of designs provided by human experts and doesn’t (by itself) provide much support for choosing between alternative high-

[1]`http://www.synopsys.com`

[2]`http://www.cadence.com`

[3]`http://www.mentor.com`

Table 2.1: *Key features of related tools and projects*

Vendor/Toolflow	**Style**	**Language**	**Template**	**Origin**	**ISA Exploration**	**Section**
Cadence						
XTensa	Architecture extension	TIE	VLIW + extensions	Hardware synthesis	manual	2.1.1
Synopsys						
Processor designer	Structural description	LISA 2.0	ADL	Simulator construction	manual	2.1.2
ASIP designer	ISA description	nML	ADL	Compiler synthesis	manual	2.1.2
Research projects						
ArchC	ISA description	ArchC	ADL	Simulator construction	manual	2.2.1
Codasip	ISA description	Codal	ADL	Simulator construction	manual	2.2.1
LISA 3.0	Structural description	LISA 3.0	ADL	Simulator construction	manual	2.2.1
TCE	Structural description	ADF	TTA	Hardware synthesis	automated	2.2.2
PICO	Architecture extension	configuration	VLIW + accelerator	Hardware synthesis	automated	2.2.3
SiliconHive/Intel						
HiveCores	Structural description	TIM	VLIW + extensions	Hardware synthesis	manual	2.3.2
This work	Structural description	TIM	VLIW + extensions	Hardware synthesis	automated	3.1

level designs. The tool that comes closest to providing an automated application-to-design path is Calypto Design Systems' Catapult-C, which started off as a product from Mentor Graphics. Catapult-C, however, is mostly aimed at the high-level synthesis of hardware accelerators only and has no special advantage when used to design application specific processor architectures. However, it can be useful when creating SFUs. In contrast, Cadence and Synopsys do provide tools which allow for more automatic design of both hardware accelerators and application specific processor architectures.

This chapter presents an overview of several of the currently available methods for (automated) design of customized, application specific, processor architectures, which are in a quite close relation to the research of this dissertation. Table 2.1 gives an overview of these methods and shows the sections were each toolflow is presented in more detail. The *style* and *origin* columns of Table 2.1 are indications of the architecture granularity and original purpose of the toolflows. Each of the presented tool flows nowadays has full support for generating hardware with an instruction-set simulator and compiler. However, the original design choices often do have a lasting impact on the abilities and strengths of each of these toolflows as will be discussed below. The interpretation of the *language* and *template* columns is explained for each of the toolflows in their respective section within this chapter.

This chapter first presents the commercially available tools from both Cadence and Synopsys, and then continues with a presentation of the recent research on the topic. We finalize the related work chapter with a discussion of the SiliconHive/Intel design framework that was used within the ASAM project, of which the research presented in this dissertation is a part.

2.1 Commercial EDA tools

Both Cadence and Synopsys provide a large portfolio of EDA tools. These various tools are aimed at different phases of the design process, and can often be used in combination with each-other in a semi-integrated fashion to offer a complete design flow from a high-level design problem specification to a detailed circuit design. In the last decade, through a series of external acquisitions both vendors have been moving to include more high-level design tools in their tool frameworks. This section will present the tools of both vendors which are relevant in relation to automated instruction-set architecture synthesis of VLIW processors, the topic of this dissertation.

2.1.1 Cadence

Similar to Mentor Graphics, Cadence traditionally focussed on providing tools that take a complete design and implement it in the latest technology. As such, Cadence mostly provides EDA tools that take a high-level system description and

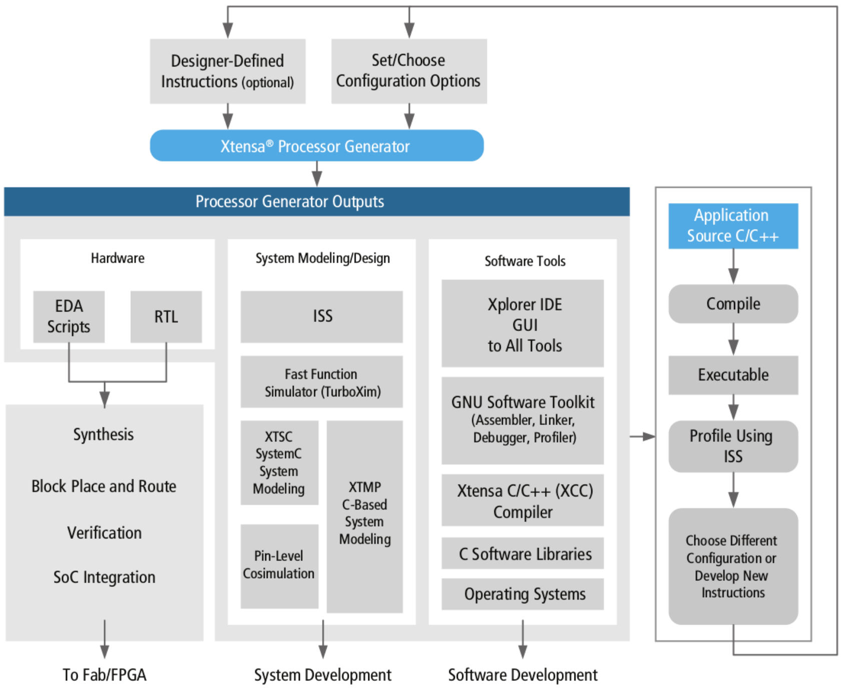

Designer-Defined Instructions (optional)
Set/Choose Configuration Options
Xtensa® Processor Generator
Processor Generator Outputs
Hardware
EDA Scripts
RTL
Synthesis
Block Place and Route
Verification
SoC Integration
To Fab/FPGA
System Modeling/Design
ISS
Fast Function Simulator (TurboXim)
XTSC SystemC System Modeling
XTMP C-Based System Modeling
Pin-Level Cosimulation
System Development
Software Tools
Xplorer IDE GUI to All Tools
GNU Software Toolkit (Assembler, Linker, Debugger, Profiler)
Xtensa C/C++ (XCC) Compiler
C Software Libraries
Operating Systems
Software Development
Application Source C/C++
Compile
Executable
Profile Using ISS
Choose Different Configuration or Develop New Instructions

Figure 2.1: *Processor Customization with Cadence XTensa*[*]

[*]Source: `http://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable`

iteratively translate the design into a more detailed lower-level design until the final circuit is realized. However, more recently, Cadence has strengthened its position in the automated high-level design market, first by acquiring Tensilica in 2013, and thereafter by the acquisition of Forte in 2014.

Forte's Cynthesizer tool together with the Cadence C-to-Silicon design-flow provided Cadence with a high-level synthesis design-flow similar to that of Mentor Graphics. However, as with Mentor Graphics, Forte's tools and Cadence's C-to-Silocon design-flows mostly focus on high-level synthesis of hardware accelerators and less at the automatic synthesis of application-specific processor architectures. The acquisition of Tensilica, however, changed this.

Tensilica was a company that specialized in programmable IP solutions and their tools include a language that allows a designer to describe a new processor architecture at the instruction-set architecture level. Based on this architecture description, the Tensilica tools automatically generate the processor architecture hardware-design together with the required software to program the newly designed processor architecture. These tools now live on as part of Cadence's XTensa tool-suite.

The Cadence XTensa design-flow, illustrated in Figure 2.1, automates the construction of new processor architectures and their corresponding support software. The designer is presented with a configurable base processor architecture which can be extended with extra operations. These operations are specified manually by the expert designer and are included directly in the processor datapath as *designer defined instructions*. Both *hardware* description (RTL) and supporting *system modeling and software tools* (simulator/compiler) are then generated for the revised architecture in minutes. This provides an expert user with a methodology to quickly evaluate the effect of different processor architecture variations on the performance of the target application. This design-flow helps a lot when designing an application specific processor architecture, but still relies on design exploration and decisions of a human designer. Identifying customization possibilities and other extensions, such as the addition of custom operation patterns, require either the usage of external tools or the presence of an expert user.

2.1.2 Synopsys

Synopsys, currently the largest of the three main EDA companies, has been involved in electronic system-level design a bit longer than the other two but, like Cadence, has also recently been expanding its interest in processor architecture synthesis tools. These tools include Synopsys Processor Designer, shown in Figure 2.2, which features the design-flow that was acquired from Coware in 2010, and the Synopsys ASIP Designer tools (formerly IP Designer) shown in Figure 2.3, which were acquired from Target in 2014.

The Processor Designer toolflow allows a user to describe a processor in the LISA architecture description language and automatically creates both the hardware description and software support tools. The LISA language provides a high flexibility to describe the instruction-set of various processors, such as SIMD, MIMD and VLIW-type architectures [66, 73]. Moreover, processors with complex pipelines can be easily modeled. The original purpose of LISA was to automatically generate instruction-set simulators and assemblers [66]. It was later extended to also include hardware synthesis [73].

Synopsys ASIP Designer provides a different architecture description language (nML [25, 28]) which is also aimed at the description and synthesis of application specific processor architectures and their support software. The nML language is very similar in intents and purpose to the LISA language but several subtle differences exist. For example, nML aims more directly at describing the instruction-set architecture of the processor, including the semantics and encoding of instructions, which makes it slightly more suited for generating a retargetable C compiler together with the simulator and processor hardware [25]. This stronger focus on generating a full compiler does however restrict possible hardware optimizations compared to the LISA based flow. For example, sharing hardware resources within a function unit is more limited for nML based architectures than it is for those described using LISA [73].

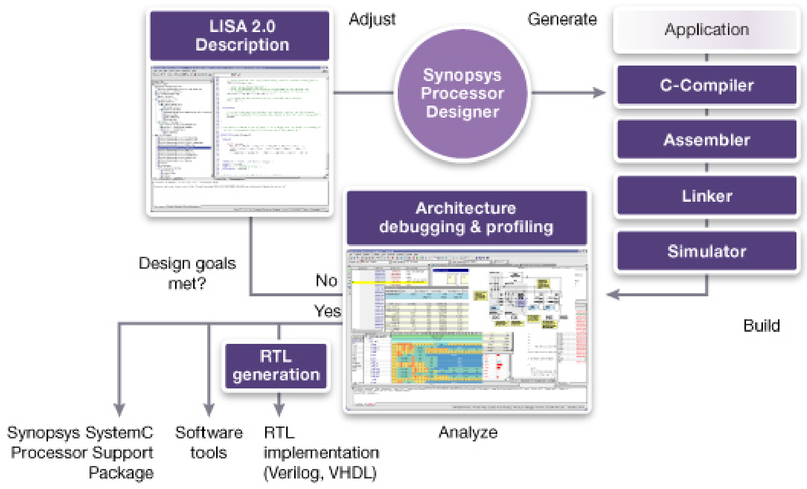


Figure 2.2: *Synopsys Processor Designer**

*Source: `http://www.synopsys.com/systems/blockdesign/processordev`

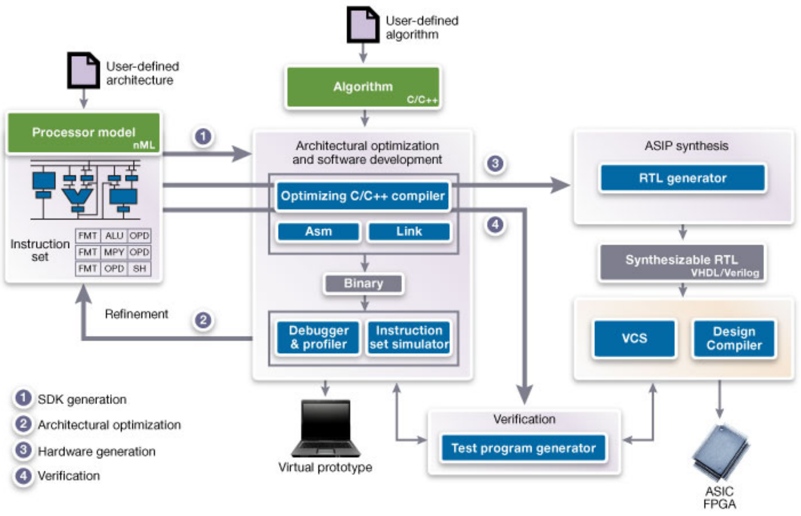


Figure 2.3: *Synopsys ASIP Designer**

*Source: `http://www.synopsys.com/dw/ipdir.php?ds=asip-designer`

Outside of these small differences both Processor Designer and ASIP designer remain very similar. In both cases an expert user has to provide an architecture description from which the tools are then able to generate a compiler and simulator. Using these generated tools, the user can then compile and simulate the target application on the proposed architecture. Cycle count and resource usage statistics are then gathered by the user upon which further alterations to the processor architecture can be proposed. The selection and implementation of these extensions is done manually by the user. Hardware (RTL) generation is usually only performed after the user is satisfied with the performance of a simulated version of the processor because of the time consuming nature of running the actual hardware synthesis and the extremely slow speed of RTL simulation.

Synopsys also offers a high-level synthesis tool called Synphony C Compiler. Again, this high-level synthesis tool is aimed more at non-programmable hardware accelerators and less at application specific processor architecture design. However, this hasn't always been the case. The Synphony C Compiler was the product of Synfora which originated in 2003 from the PICO project as a start-up company. PICO, which stands for Program-In, Chip-Out, did much more than the synthesis of hardware accelerators and will be discussed in more detail in Section 2.2.3.

2.2 Research projects

In parallel to the commercial offerings discussed above, several research projects have recently been performed in relation to the high-level design of application specific processor architectures. Most of these research projects focus on providing or improving architectural description languages for the construction of application specific processor architectures (see Section 2.2.1. Two projects were found to differentiate themselves from the others in that they provide support for automated architecture exploration. These projects, the TCE framework and the PICO project, will be discussed separately and in more detail in Sections 2.2.2 and 2.2.3 respectively.

2.2.1 Architecture description languages

Several domain specific languages have been developed for the description of both functionality and structure of application specific processor architectures. Using such an architecture description language (ADL), and its related EDA tools, allows a designer to quickly make variations of a processor architecture and consider the effects of design choices on the cost and performance of the final product. Various design analysis tools, including simulation, are commonly provided with the ADL tools for this purpose. Examples of such languages are the nML and LISA languages as used by Synopsys ASIP Designer and Synopsys

Processor Designer, respectively. More variations exist in research: ArchC [4] is still being developed at the University of Campinas in Brazil[4], Codal [13] is being researched by both Codasip[5] and the Technical University of Brno in the Czech Republic, and a new version (3.0) of LISA [14,41,73] is in development at RWTH Aachen University[6].

The current research on these languages and their respective frameworks focusses mostly on the translation of a high-level processor architecture description into a corresponding structural description in a hardware description language, such as VHDL or Verilog, as well as, the generation of programming tools such as a C/C++ compiler or assembler, debugging and instruction-set simulation tools, and application analysis and profiling tools. In some cases (e.g. LISA 3.0 and ArchC), support for system-level or multi-processor integration is also being added. LISA 3.0 also improves on its previous incarnation by the addition of support for reconfigurable computing [14,41]. A reconfigurable processor introduces a reconfigurable logic component in or near the datapath. Such a component can have either a fine grained, similar to a small field programmable gate array (FPGA), or coarse grained, more like a coarse grained reconfigurable array (CGRA), reconfigurability. This addition allows for further customization of the instruction-set even after the finalization of the processor silicon, for example during the processor initialization or possibly even at runtime.

In general, the process of using these ADL based tools is very similar to that of the Synopsys tools described above. Support is provided for constructing both development tools such as a compiler and simulator, as well as, a RTL description of the hardware of a processor described using in the ADL. Application analysis tools focussing at highlighting hot spots and candidate instruction-set extensions can also be provided to the designer. However, like with the Cadence and Synopsys tools, the final decision making on which processor variation to consider for a next design iteration is left to the expert designer.

Many more ADL frameworks exist and this section named only a few which were relevant to the research of this dissertation; for more information on this topic see the book “Processor Description Languages” by Mishra and Dutt [57].

2.2.2 TCE: TTA-based Co-design Environment

The TTA-based Co-design Environment[7] is a set of tools aimed at designing processor architectures according to the Transport Triggered Architecture (TTA) template. TTA processors are, like VLIW processors, a sub-set of the explicitly programmed processor architectures and can be seen as exposed datapath VLIW processors. Unlike VLIW processors, TTA processors do not directly encode which operations are to be executed, but are programmed by specifying data

[4]`http://www.archc.org`
[5]`http://www.codasip.com`
[6]`http://www.ice.rwth-aachen.de/research/tools-projects/lisa/lisa`
[7]`http://tce.cs.tut.fi/`

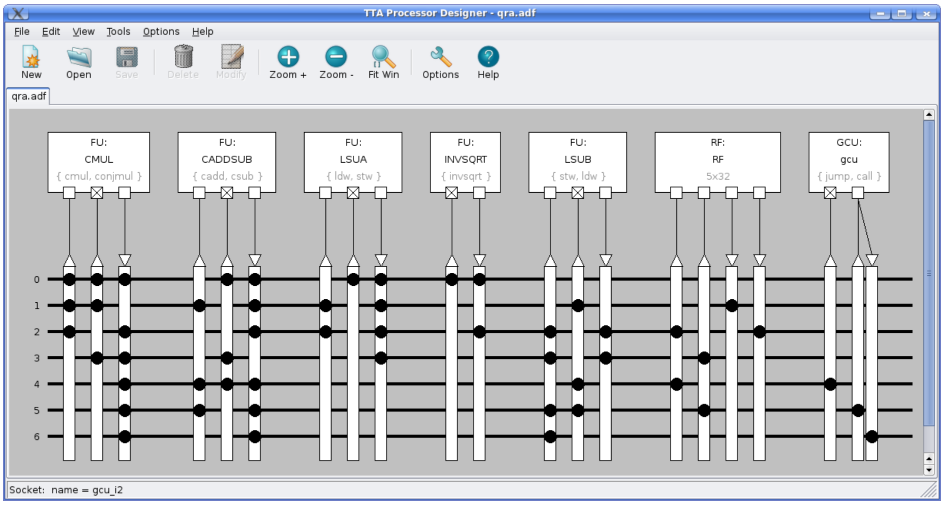

TTA Processor Designer - qra.adf
File Edit View Tools Options Help
New Open Save Delete Modify Zoom + Zoom - Fit Win Options Help
qra.adf
FU: CMUL { cmul, conjmul }
FU: CADDSUB { cadd, csub }
FU: LSUA { ldw, stw }
FU: INVSQRT { invsqrt }
FU: LSUB { stw, ldw }
RF: RF 5x32
GCU: gcu { jump, call }
0
1
2
3
4
5
6
Socket: name = gcu_i2

Figure 2.4: *Transport Triggered Architecture processor template**

*Source: `http://tce.cs.tut.fi/screenshots/designing_the_architecture.png`

movements. As a result, all register file bypassing from function units and register files is fully exposed to the compiler. The TTA programming model has the benefit of enabling software bypassing, a technique where short-lived intermediate results of computations are directly forwarded to the function-unit consuming the data, while completely bypassing the register file. This reduces both the register file size and port requirements, as well as, energy consumption for the TTA architecture compared to more traditional VLIW architectures. A similar reduction of the register file energy consumption can be obtained using hardware bypassing [64, 75], but that technique generally has a larger hardware overhead as it requires run-time detection of bypassing opportunities. Figure 2.4 illustrates the TTA processor architecture template. It shows how the function units, register file, and control unit, are connected through sockets to the transfer buses. Programming is achieved by controlling the connections in of the sockets with the buses. From this figure it is also clear that register file bypassing can be implemented in software simply by forwarding a result from one function unit directly to the input of another.

Research on Transport Triggered Architectures started with the MOVE project [15, 31] at Delft University of Technology during the 90s. Later, when the Delft MOVE project was discontinued, Tampere University of Technology continued the research, and created the next generation of the MOVE framework which they named the TTA-based Co-design Environment. Hoogerbrugge and Corporaal [15, 31, 32] investigated automatic synthesis of TTA processor architectures as part of the MOVE project and a derivative of this work is still available within the TCE. Chapter 6 of the TCE manual [78], titled “Co-design tools”, is dedicated to the tools available for supporting automatic processor architecture exploration. These tools are somewhat similar to the tools and techniques presented in this dissertation. However, this dissertation presents several techniques and tools that target another processor architecture style (VLIW).

In general, the work described in this dissertation is to some degree similar to that of the TCE, but has a strong focus on optimizing the exploration efficiency. The methods presented within this dissertation improve upon those related to the TCE as follows:

- The techniques presented in Chapter 4 can be used to give early estimates on the number of buses and function units to create a good initial architecture. The TCE expects that an initial processor is designed and constructed by the user and is then iteratively adapted to better suit its purpose. Starting with a better architecture reduces the number of iterations in the adaptation process and, as a result, substantially speeds-up the exploration.

- As part of the exploration, the TCE framework provides the possibility of performing a compiled simulation. As preparation for the compiled simulation, the processor simulator (`ttasim`) is compiled to include a compiled form of the target application. This avoids the instruction-set interpretation step traditionally needed for simulation and significantly speeds up simulation but does require the compilation of a specialized simulator program [78]. The `ttasim` documentation suggests that it can be combined with ccache [82] to drastically reduce compilation times before simulation. Ccache works by saving compiled binary files into a cache. When ccache notices that a file about to be compiled matches a previously compiled (and cached) file, it simply reloads file from the cache, thus eliminating recompilation of unmodified files and saving time [78]. This can be very useful when running the same simulation program again, due to drastically reduced compilation times. Our BuildMaster framework, presented in Chapter 6 works similarly as ccache, but takes more architectural knowledge into account. This allows our BuildMaster to also recognize when slightly different hardware configurations will result in exactly the same compiled binary code. Our approach therefore recognizes more opportunities than only the trivial ones observed using ccache, as a result it achieves higher compilation cache hit-rates and delivers a higher compilation time reduction.

- The BuildMaster framework also manages our energy and area estimation, combined with our profile-based energy estimation presented in Chapter 5. TCE uses a simulation-trace based energy estimation, and therefore requires a simulation run for each considered design-point. Our approach only requires a new simulation run when the application’s execution profile changes, which only happens after significant changes to the processor architecture. Our BuildMaster framework is capable of predicting when these changes will happen and will only re-run a simulation when predicts that this is actually required. In our experiments (see Chapter 6) we found that we can avoid on average over 90% of the simulation runs using this technique. This can greatly reduce the total simulation time, especially when an architecture supporting a large application or benchmark is to be explored.

- Our instruction-set architecture exploration algorithms, presented in Chapter 7, differ from those for the TCE also in relation to the fact that we target VLIW-based processors and not TTA-based ones. This, combined with our careful construction and selection of an initial architecture for the exploration, especially when combined with our thorough caching of intermediate results, allows us to obtain a highly efficient processor architecture in a very short time.

Most of the presented techniques, after small modifications, could also apply to the TCE and could help to further reduce TTA-based processor architecture exploration times.

2.2.3 PICO: Program-In Chip-Out

As was mentioned above, the PICO project, grandparent to parts of Synopsis' current design-flow, offered more than the automatic synthesis of hardware accelerators which was incorporated into Synphony C compiler. In its original form PICO covered the automatic synthesis of a processor system containing a set of non-programmable hardware accelerators combined with a single VLIW processor [1, 42]. Figure 2.5 illustrates the PICO system architecture template.

In essence the goals of PICO were very similar to those of the ASAM project. Both projects aimed to automatically develop an application specific multi-processor system. However, there are also several key differences. PICO approaches the problem by synthesizing a system with a single VLIW processor and a set of hardware accelerators, whereas the ASAM project utilizes one or more heavily specialized highly parallel VLIW processors and no hardware accelerators. The differences between these two approaches stem mostly from the differences in their VLIW processor templates. For example, the PICO VLIW processor template (shown in Figure 2.6) uses a single register file for each data-type (integer, floating point, etc.), this severely limits the number of operations which can be executed in parallel. Many read ports need to be available to provide the operands to each operation executed in parallel. Large many ported register files quickly become very expensive regarding both area and energy, and limit the maximum operating frequency of the VLIW processor [53, 79].

The PICO design-flow, illustrated in Figure 2.7, provides a fully automated design flow for developing the non-programmable accelerator (NPA) subsystems, the VLIW control processor, and the cache memory hierarchy. To limit the size of the design space, each of these three components (NPAs, VLIW architecture, and cache hierarchy) is explored independently from the others. Considering all three

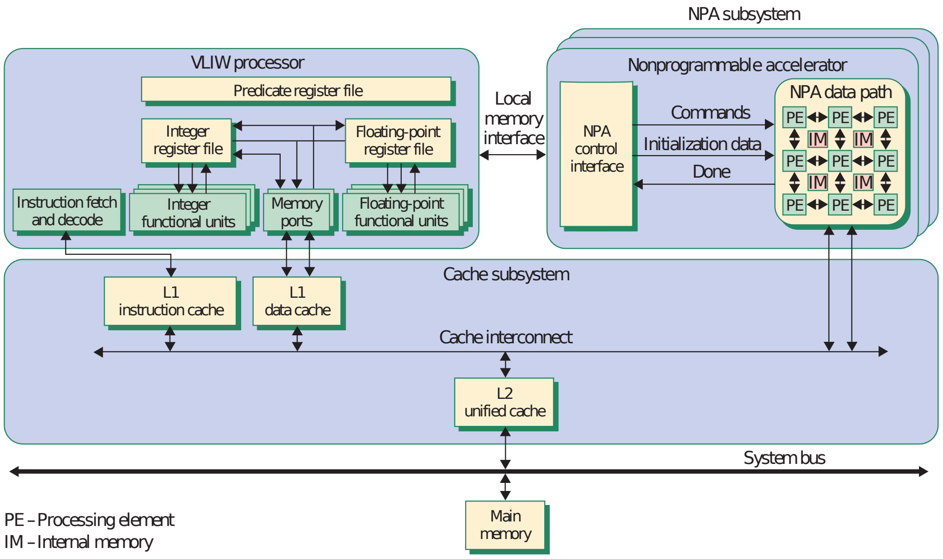


Figure 2.5: *PICO system architecture template [42]*

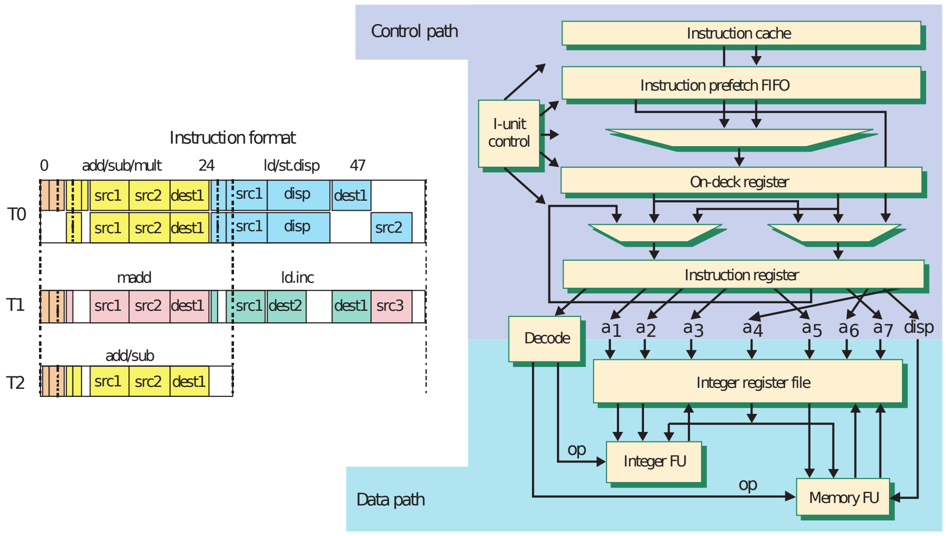


Figure 2.6: *PICO VLIW processor architecture template [42]*

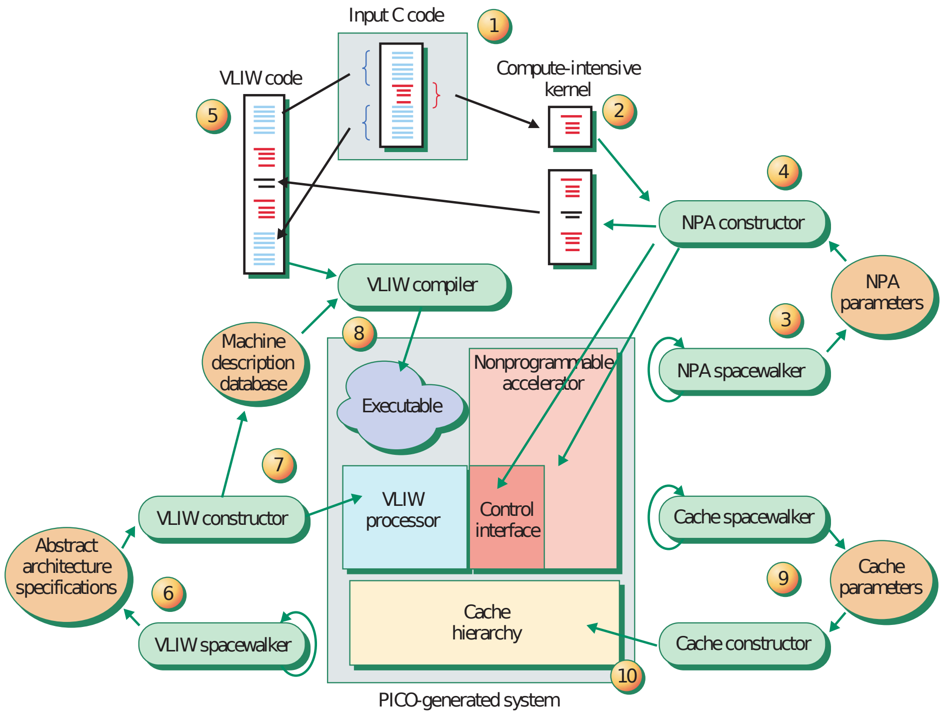


Figure 2.7: *PICO design-flow organization [42]*

components combined results in a too large design space which severely limits the effectiveness of any automated exploration [1, 42]. During the exploration, a Pareto-optimal set of solutions is obtained for each of the three system components. First compute-intensive kernels are identified (2) (see step 2 in Figure 2.7) in the input C code (1) and NPA architectures are explored for each of these kernels (3), (4). The compute-intensive parts are then replaced with calls to the hardware accelerators in the original C code (5) and a set of alternative VLIW processor architectures is then designed (6), (7), (8). Finally, the cache hierarchy is tuned for the memory requirements of the application (9) and compatible VLIW, cache, and NPA designs are combined to form a set of Pareto-optimal designs (10). The focus for the system architecture exploration by PICO is on the trade-off between the area and timing. Area is measured in either physical chip area or gate count, whereas an estimation of the application's processor runtime is used for the timing. Similar to our approach, the timing estimate is computed as the total sum of each basic block's schedule length multiplied by its profiled execution count.

The aims of the PICO project were quite similar to those of the ASAM project, of which this dissertation is a part. However, the architecture template for the

PICO project was substantially more restricted than that of the ASAM project. The research reported in this dissertation much improved upon the PICO project results, a.o. as follows:

- The VLIW processor architecture template used within the ASAM project has substantially more design freedom. This allows us to construct VLIW processor architectures which perform like hardware accelerators but are still fully programmable. This comes at the cost of a much larger design space, and thus, a more complex design space exploration. The ASAM project developed methods and tools to explore this much larger design space effectively and efficiently.

- While PICO focusses on a single VLIW processor with the addition of hardware accelerators, the ASAM project replaces these hardware accelerators with several heavily customized VLIW processors. This results in a fully programmable heterogeneous multi-processor system which improves the customization and adaption possibilities of the platform. The ASAM approach also enables a much more heavy use of code transformations to optimize and combine compute intensive kernels in an efficient way. The ASAM project considers loop tiling, kernel fusion, and vectorization whereas the PICO NPA Spacewalker only considers loop tiling [74]. Adding these extra code transformations drastically increases the size of the design space. The ASAM project handles this increased design space complexity by incorporating a high-level exploration of these code transformations using early performance estimates of both single kernels and kernel combinations. Based on this exploration we are able to find promising kernel combinations while designing each single VLIW processor node. Chapter 4 presents our effective and efficient techniques for early performance estimation and gives bounds on the best-case and worst-case timing behaviour for single kernels.

- PICO considers the trade-off in design area and performance (cycle count) during its exploration, the ASAM project considers the same, but adds the energy consumption to the considered design quality metrics. Like PICO, we use the profiled execution count of the basic blocks of an application to estimate the cycle count of kernels mapped onto VLIW processor candidates. However, we have extended this method to provide sufficient information to also perform energy estimation. Building further on the profile-based energy estimation, we recognize that the execution profile of different architecture candidates changes relatively infrequently. In chapters 5 and 6, we therefore present techniques which cache previous profiles obtained through simulation and recognize when the execution profile is likely to change. These techniques allow us to avoid over 90% of the otherwise required simulation time and provide a significant exploration time reduction, making the exploration of large design spaces feasible.

The work presented in this dissertation focusses on techniques and tools to improve the architecture exploration speed. The developed techniques and tools can be integrated into PICO to allow for a much faster and more thorough design space exploration.

2.3 The SiliconHive tools

As with all large high-tech companies, public information about the Intel Benelux VLIW ASIP development technology is mainly available through cooperation in research projects such as the ASAM project or with the added knowledge of its ancestry. Through its previous incarnations at both SiliconHive and Philips, a veritable treasure trove of information on this technology can be uncovered. The initial research and development was carried over into SiliconHive when it was spun-out of Philips Research as part of the Philips Technology Incubator program in 2003. Intel then acquired SiliconHive in 2011 after a period of growth and further development.

This section uses a selection of the information made available through the ASAM project and from publications from both the earlier Philips and SiliconHive periods, to introduce the SiliconHive VLIW ASIP architecture template and the pre-existing SiliconHive development framework. The SiliconHive architecture template is introduced in this section. Key features of the related retargetable compiler are introduced in Section 2.4.

2.3.1 Overview

Similar to the other tool flows discussed in this chapter, the SiliconHive tool flow, illustrated in Figure 2.8, offers an architecture description language (called TIM) and a set of tools to generate hardware RTL descriptions, an instruction-set simulator, a retargetable C compiler, and various other debugging and software development tools. In parallel, the SiliconHive tools also offer a second language (called HSD) that allows a user to construct a multi-processor system consisting of one or more VLIW processors (specified using the TIM language) and other hardware components (such as hardware accelerators, memory units, and peripherals) taken from a library or imported from external sources.

Starting from an original application design, the user starts by composing or selecting an initial MPSoC platform design and then decomposing the application into a parallelized version tuned for the initial platform. After mapping the application onto this platform, the user can then compile and simulate the application to find the remaining critical points of the design. Using this information, the user then can manually update the parallelization, mapping, and/or the platform composition in an iterative design process. Several pre-selected processor and platform designs are available for an easy start but the parallelization, mapping, and design-space exploration steps need to be performed manually.

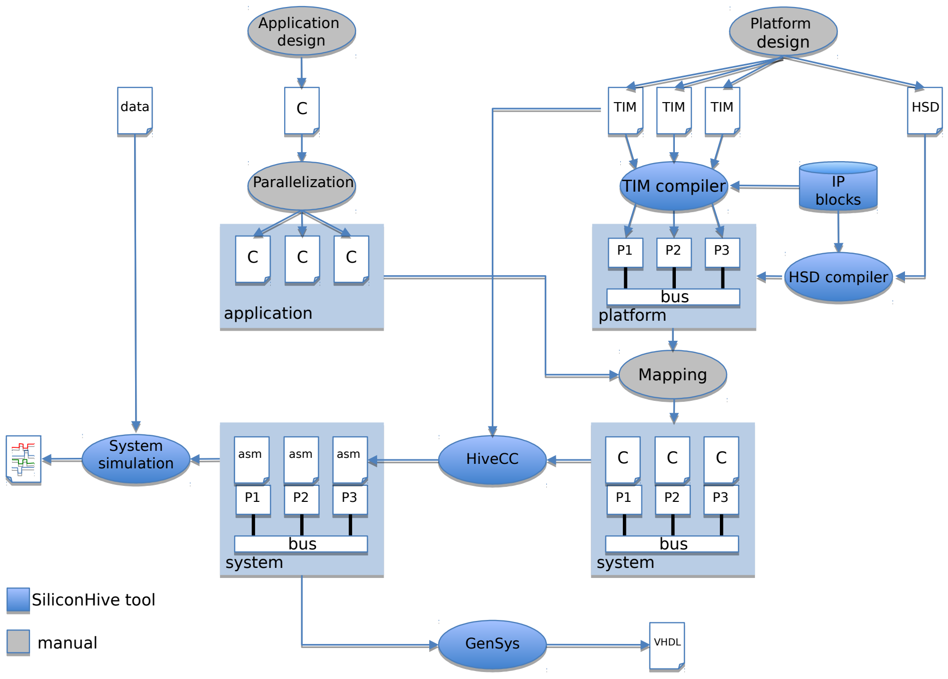


Figure 2.8: *SiliconHive flow*[*]

*Source: ASAM project

2.3.2 Architecture template

The SiliconHive ASIP design technology offers a highly flexible template of a customizable VLIW processor. Figure 2.9 illustrates this processor architecture template. A processor (*cell*) is organized in two parts, one part (*coreio*) contains the local memories and handles the interface with the external world, while the other part (*core*) performs the actual operations as described by the program in the local program memory.

Interfacing to the external world and memories

SiliconHive processors usually contain one or more local scratchpad memories which are used for low-latency storage of (intermediate) data used by the algorithm running on the processor. Several memories with different organization (e.g. different sizes and/or data-widths) and differently implemented (e.g. register-based or SRAM) can be included in a single processor as required by the target application. Local memories are also connected through a slave interface to the global MPSoC interconnect hierarchy so that the other elements of the MPSoC

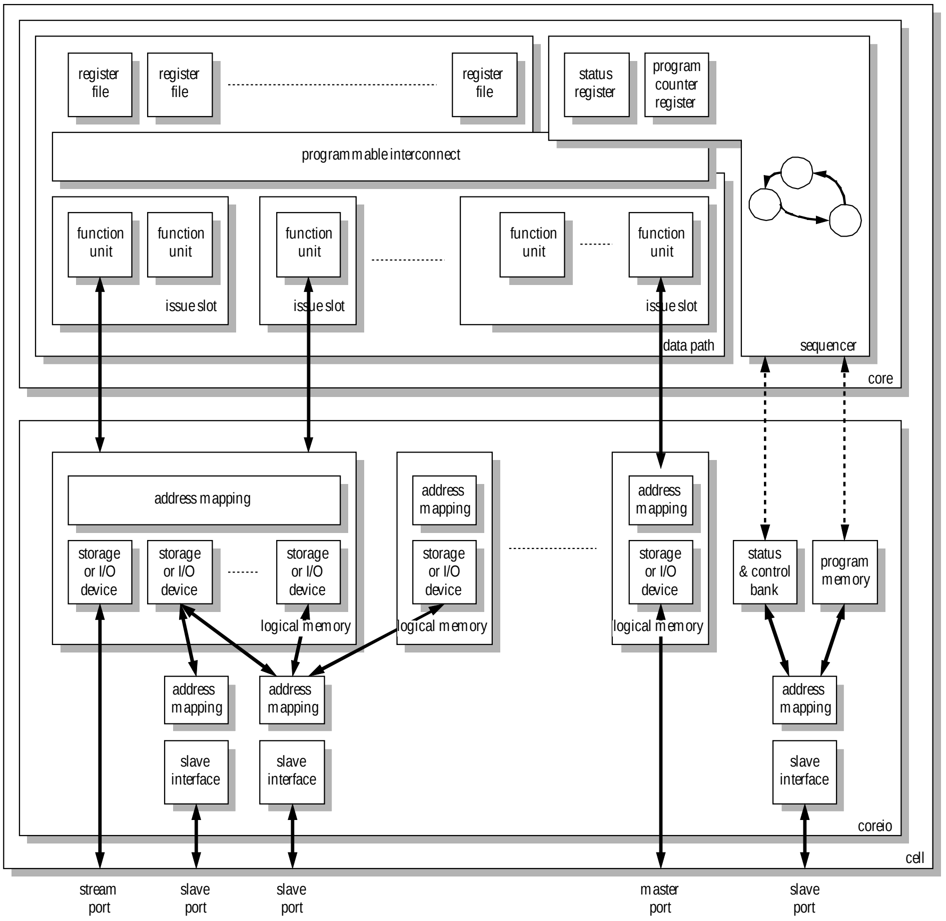


Figure 2.9: *SiliconHive processor architecture template [38]*

containing the SiliconHive processor can access these memories when providing input and/or consuming output to/from the SiliconHive processor. One or more master interfaces may also be present in the coreio. These master interfaces can either connect to an external memory, a direct memory access (DMA) controller, or the local memory of another SiliconHive processor. Stream interfaces (FIFOs) can also be added to the processor. Such FIFO interfaces are mainly suitable for small data transactions and are usually utilized by providing hand-shake signals while performing larger transactions through a master interface transfer.

In parallel to these data storage and transfer components, the coreio of a SiliconHive processor also contains the program memory, as well as, a set of status and control registers which can be used for reconfiguring the processor.

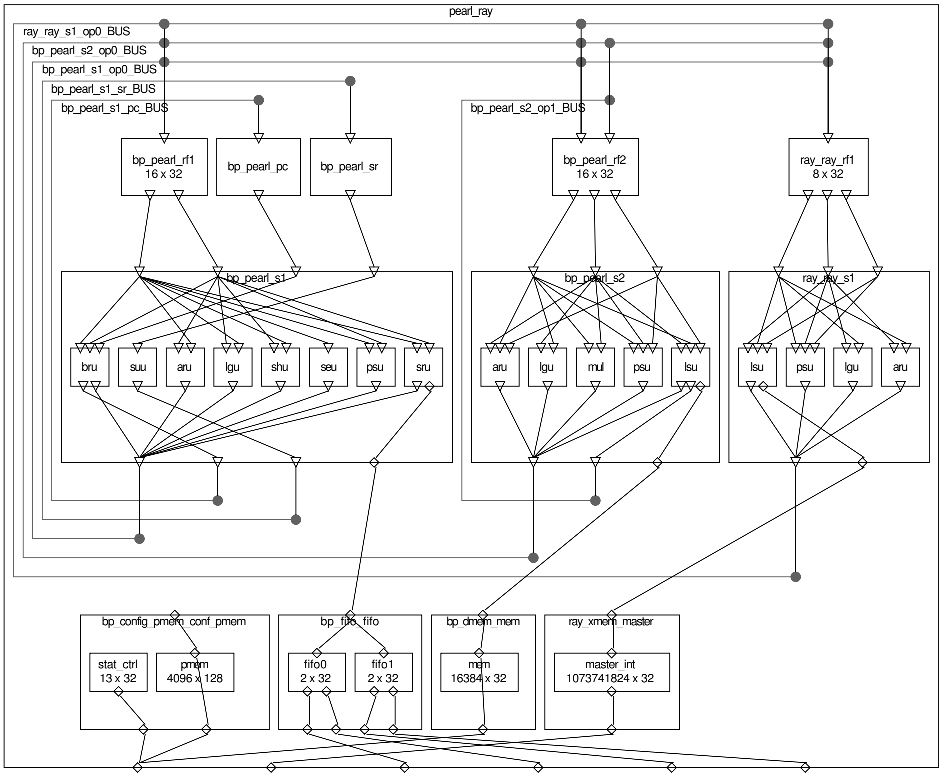


Figure 2.10: *An example SiliconHive processor, the Pearl Ray [62]*

Examples of such reconfiguration are actions like reprogramming the program memory, entering/leaving the low-power sleep mode, and starting/stopping a kernel.

The core itself contains a very small sequencer block, which task it is to interface the datapath with the status and control registers, and which fetches the appropriate instructions from the program memory. Instruction decoding is very cheap in the SiliconHive processor architecture by using a horizontally programmed processor style. Instruction bits usually correspond directly to part of the configuration bits of the processor registers and input select multiplexers.

Core structure

The actual execution of the program is performed inside the datapath. Here, operations are executed within *issue-slots*. Each issue-slot is composed of a set of *function-units* which implement the actual operations. This division of issue-slots into function-units can be somewhat confusing as most of the related work

[7,15,31,53,79] uses the term function-unit (or functional-unit) to designate what is called an issue-slot in SiliconHive terminology. However, this dissertation will use the SiliconHive terminology since it builds upon their processor architecture framework.

Within the datapath, issue-slots are connected to multiple register files using a (optionally shared) interconnect [7, 48, 62, 83]. This provides an efficient implementation for very wide VLIW processor architectures without incurring the overhead of a large, centralized, register file [53, 79].

Figure 2.10 shows the Pearl Ray processor, one of the example processors of the SiliconHive design flow [62]. It demonstrates a processor with 3 issue-slots, 5 register files (3 general purpose register files, one program counter, and one status register), a local memory, a master interface, and two bidirectional FIFOs. The leftmost issue-slot communicates with the sequencer, it contains a 'status update unit' (SUU) and is connected to both the status register and program counter.

Programming a SiliconHive processor is achieved by configuring the appropriate connections between issue-slots, selecting appropriate register file indices for each register file port, and selecting the executed operation for each issue-slot. In this explicitly programmed processor, the processor configuration maps almost directly into bit-fields of the program word [7]. For example, the Pearl Ray processor instruction word is constructed as follows:

- Issue-slots 1, 2, and 3 (each):
  - n operation selection bits, with $n = \lceil 2log(ops) \rceil$ and ops the number of operations in the issue-slot.
  - Extra immediate bits for operations taking immediate (constant) values.
- General purpose register file 1, 2, and 3 (each):
  - Operand select bits for each output port
  - A write index for each input port
  - A bus select for each input port of a register file, which is used directly to configure the multiplexers in the result select network (illustrated as dots in Figure 2.10).
- Status register and program counter:
  - A write-enable bit.

Overlaying reduces the length of the program word by assigning multiple purposes to program word bits depending on the selected operation. For example, the immediate bits are overlapped with the select bits for one of the other operands. This causes a reduction of the program word length without causing encoding conflicts when an operation which uses an immediate (e.g. `add %reg1, %reg2,`

`#imm`) operand will use one less register file input compared to its non-immediate variant (e.g. `add %reg1, %reg2, %reg3`).

The explicitly programmed nature of the SiliconHive processors makes it relatively easy to compute the final instruction width for newly constructed processor architectures. This is especially useful when modeling the impact of architecture changes (see Chapter 5). For the example Pearl Ray processor, the program word width after overlaying is 111 bits.

2.4 Compiler support

One of the major difficulties in the development and usage of VLIW processors is the ability of the programmer to obtain a sufficiently high resource utilization of the processor. The high amount of programming freedom provided by explicitly programmed processor architectures, which enables the high performance benefits of these architectures, also increases the complexity of the scheduling process. Combining this with the presence of large, complex, custom operations and VLIW scheduling techniques such as software pipelining [46] easily results in a highly complex compiler.

This high compiler complexity, up to the point where the creation of a compiler becomes infeasible, is often one of the motivations for moving parts of the decision making process to the programmer, the user of the compiler. Annotations added to the input C code allow for a simplification of the compilers decision making process which often leads to a better final result when an experienced programmer is using the compiler. Even with a good compiler, annotations may bring a substantial profit as they allow for localized overrides of the compiler heuristics in cases where sub-optimal results are being produced.

2.4.1 Source code annotation

Source code annotation is a popular technique to enable some of the more esoteric processor features without invasive changes to the compiler itself. Classical examples are intrinsics and annotations that capture the mapping of data into one of the (possibly many) different memories of a processor. However, hints for enabling and disabling specific optimizations can also be provided as source code annotations. Complex optimizations can be very time consuming, while most of their benefit can only be observed for a (small) portion of the application code.

Complex operations

Complex operations, such as an FFT butterfly operation or those working on vector elements, are difficult to represent or efficiently detect in the C language. For this purpose, most compilers provide direct access to such operations through intrinsics. A compiler intrinsic looks like a C function call, but translates within the compiler directly into the (complex) operation it represents. This allows the programmer to force the compiler to select the intended operation and allows for a strong simplification of the custom operation selection heuristics in the compiler itself. Listing 2.1 shows an example of an explicitly selected multiply-accumulate operation. In this case the intrinsic is usually not needed as the compiler will recognize the operation automatically, but bigger or more complex operations, such as the FFT butterfly operation mentioned above, this may not be feasible and the use of the intrinsic will be required in order to generate efficient code. Another common use of intrinsics is to represent low-level functionality that has no representation in C such as a FIFO send or receive operation (as demonstrated below in Listing 2.3).

```
#if HAS_std_mac
  r = OP_std_mac (a, b, c);
#else
  r = a * b + c
#endif
```

Listing 2.1: *Example use of an intrinsic*

Memory mapping

The mapping of (global) data arrays onto one of the local memories of a processor is often controlled by added annotations. For example, OpenCL [77] recognizes `__private` as a keyword which denotes that the thus marked data should be mapped into the private memory of the processor running the kernel.

The SiliconHive compiler recognizes a `MEM(`*`memory_name`*`)` annotation, illustrated in Listing 2.2, where *memory_name* is one of the memories of the processor design, which forces the data to be mapped onto the selected memory. Any global data that is not annotated will be mapped into a default memory. Mapping all the data into a single memory can severely limit the parallelism at which the application can be executed as only a single load-store unit (LSU) is usually connected to each memory. Though connecting multiple LSUs is possible this either requires arbitration (which sequentializes access from the different LSUs) or a multi-port memory (which is expensive). As such, mapping all data into a single memory effectively restricts the number of parallel load-store operations to one per cycle, which severely impacts the overall performance of memory-intensive applications.

```
int MEM(mem1) a; // A variable in memory mem1
```

Listing 2.2: *Example use of a* `MEM` *annotation*

Optimization hints

Optimization hints can often be provided either by using standardized C keywords such as `inline`, `restrict`, and `register`, by in-line annotations such as `__builtin_expect()` in GCC, or by compiler directives using `#pragma` statements.

Typically, `#pragma` directives are used by the SiliconHive compiler to control loop optimizations (such as unrolling) and scheduling parameters (such as suggestions for the initiation interval for software pipelining). Exhaustive scheduling can also be selected for specific parts of the program through a `#pragma` statement as further discussed in Section 2.4.2.

The SiliconHive compiler also allows the programmer to force the execution of an operation onto a specific function unit using the `ON` keyword, which can help the scheduler to find a better mapping of the application. Extra dependencies between operations can be added to the C code using the `DIST`, `AFTER`, `SYNC`, and `SYNC_WITH` keywords. Such explicit synchronization statements allow, for example, the addition of a scheduling dependency between the FIFO read which receives a handshake signal, and the memory read which retrieves the actual data that was received by the processor. An example usage of such keywords is given in Listing 2.3. Adding these annotations is required when speculative execution of operations is enabled in the compiler. Failing to provide proper annotations may result in operation re-ordering which and will cause incorrect results during the execution of the program.

```
int SYNC_WITH(0) foo;
int SYNC_WITH(1) bar;
void func(void)
{
a: OP_std_snd (0, 0) SYNC(0);
   foo = 1; // can go after b, but not before a
   bar = 2; // can go before a, but not after b
b: OP_std_snd (0, 1) SYNC(1);
}
```

Listing 2.3: *Example use of SYNC and SYNC_WITH annotations*

2.4.2 Code transformations

Code transformations form a key to enabling a high ILP, but also for controlling the buffer sizes required for intermediate data. Code transformations, such as *speculation* and *unrolling*, or scheduling techniques, such as *software pipelining*, are mostly aiming at increasing the explicitly available ILP within important sections of the program. However, such ILP enhancing techniques usually also come at the cost of an increase of the program size. The SiliconHive compiler supports these ILP enabling optimizations both as automatic optimizations, but also provides the user with more control through enabling/disabling these optimizations for parts of the code using annotations. Listing 2.4 shows an example of explicitly enabled software pipelining on a loop.

```
for(i = j = 0; i < 100; i++, j++) {
  table[j] = table[i] * 3 + 2;
#pragma hivecc pipelining=0
}
```

Listing 2.4: *Example use of a code transformation annotation*

Loop transformations, a subset of the code transformations with focus on the structure of loop nests, generally aim at controlling the distribution and size of data elements communicated between consecutive loop nests, but also affect the sizes of data communicated between the processor and external inputs and outputs (such as external memories or other processing tiles). The loop transformations used within the ASAM project are mainly: loop fusion, loop tiling, and loop vectorization. These loop transformations are currently not provided by the SiliconHive compiler and are performed within the ASAM project as source-level code transformations.

For the purpose of loop transformations, formal mathematical models of loop nests, such as the Polyhedral model [6], are used. Such formal models allow for a direct analysis of the effects of loop transformations on the memory requirements of the transformed code. Although the SiliconHive compiler does not yet support such loop transformations, several tools are already available [5, 6, 9, 29, 84] which allow for translations to and from the polyhedral domain, as well as, the automatic exploration of loop transformations. Chapter 3 further illustrates how these methods are used within this disseration and the ASAM project in general.

2.4.3 Extensions for architecture exploration

Using a compiler in the context of an architecture construction or exploration framework increases the demands on the compiler. A lot of time can be saved if the same compiler can be used for several variations of a specialized processor without the need for rebuilding (parts of) the compiler. In the SiliconHive compiler, this is achieved through the addition of several compiler controls. These compiler controls allow the programmer to override the set of available processor resources that the compiler is allowed to use. Such compiler flags make it very easy to investigate the effects of removing specific function unit, or even a complete issue-slot, from a proposed processor architecture.

However, code annotations pertaining to resource allocation, such as an explicit function unit binding or the use of a complex operation through an intrinsic, are considered definitive by the SiliconHive compiler. Thus, removing an explicitly used resource from a processor will result in a conflict with source code annotations present in the target application. For example, removing a function unit may result in the removal of an intrinsic from the compiler which was used in the target application. In some cases, replacement or emulation code can be provided by the programmer. Such emulation code needs to be provided in source code form itself as pre-compiled emulation libraries will also not have the opportunity to take removed resources into account.

A similar difficulty appears when a load-store unit is removed, making one of the memories of the candidate processor inaccessible. Such a removal can invalidate the current memory mapping of the target application in a way that can not be covered easily by the use of replacement code. It is quite likely that the entire memory mapping needs to be reconsidered in such a case. As a result, either the exploration can not have the freedom of removing load-store units and their related issue-slot and memory interfacing hardware, in which case the exploration needs to be provided with a set of different initial architectures representing different memory mappings; or the compiler needs to provide an automated method for finding an appropriate distribution of data across memories so that the distribution of data across different local memories will not require annotation.

2.5 Conclusion

In this chapter we have presented several tool flows and research projects that either overlap directly with the full scope of the ASAM project, or that are used within the ASAM project itself. However, there are many other previous works that cover sub-parts of the problems investigated in this dissertation. In particular, we recognize that instruction-set architecture synthesis relies heavily on tools and techniques from various research fields, such as; instruction scheduling [7,31,46,51,53,65,70,76], instruction-set extension [59–61,69,81,86], hardware synthesis [37,74], performance and energy modeling [44,45,67,72,88], design space exploration [37, 50], and many others. Such works are further discussed in the related chapters of this dissertation. A more thorough overview of these tools and techniques, and many others, can be found in [40].

T'aint what you do, it's the way you do it.
Melvin 'Sy' Oliver & James 'Trummy' Young, 1939

3 VLIW processor design in the ASAM project

As previously mentioned, the ASAM project focusses on the automatic synthesis of a heterogeneous VLIW ASIP based multi-processor system. The ASAM design-flow is built upon the VLIW MPSoC design framework of Intel Benelux (formerly SiliconHive) that was presented in the previous chapter. The aims of the ASAM project include the automation of several of the manual steps. In particular the exploration, analysis, and decision making regarding multi-ASIP platform design, application parallelization, ASIP customization, and application mapping steps were considered for automation.

One of the key problems addressed by ASAM is that it is impossible to perform an efficient parallelization and mapping without information on the performance of application parts on specific processing elements of the platform, but it is equally impossible to construct a reasonable multi-processor platform and each of the application-specific processing elements without knowledge of the parallelization and mapping due to the cyclic dependency between both.

The ASAM project tries to break the cyclic dependency of this 'chicken and egg' problem by tight coherent coupling of various design stages and phases, as well as, through statically computing early performance estimates of single tasks, and combining those into likely application mappings using a probabilistic application partitioning and parallelization phase.

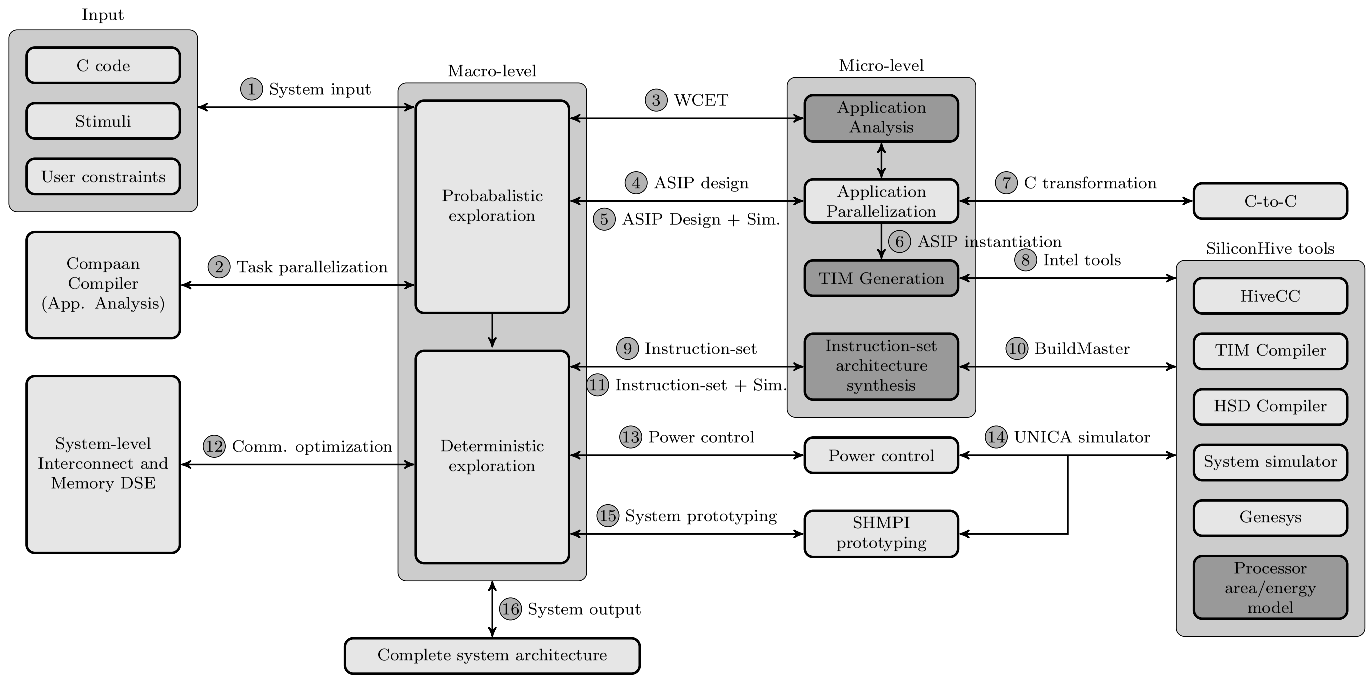


Figure 3.1: *ASAM flow overview with the contributions covered in this dissertation marked in a darker shade*[*]

[*]Source: The ASAM project

3.1 Overview

The ASAM approach divides the MPSoC design space exploration into two main stages: the macro-architecture exploration, and the micro-architecture exploration. These two stages are tightly coupled to form a coherent MPSoC architecture exploration framework. The *macro-level* architecture exploration focusses on the construction of the system out of a set of (initially unknown) VLIW ASIP processors, whereas the *micro-level* architecture exploration focusses on the analysis of a set of tasks assigned to a single ASIP and the construction of new VLIW ASIPs specialized for (groups of) specific tasks. Figure 3.1 illustrates the tight bi-directional cooperation between the macro- and micro-level architecture exploration. Further information on the ASAM flow can be found in [38]. The work presented in this dissertation lies within the micro-level architecture exploration and focusses mostly on the application analysis and the instruction-set architecture synthesis (highlighted in Figure 3.1). However, to adequately place this work in its context some more information is needed about the macro-micro interaction, as well as, the application (or more accurately intra-task) parallelization within the micro-level stage.

3.1.1 Macro- and micro-architecture exploration

The ASAM flow starts ① with its input composed of the C-code of the target application, as well as, user supplied constraints and design objectives. From these, it extracts the overall application structure and a set of compute intensive kernels using the Compaan Compiler [43], which translates these kernels into tasks in a Polyhedral process network ②.

The user supplied constraints and design objectives consist of both structural and parametric requirements. These requirements guide the automated tools and are used to control exploration aspects, such as the granularity of the computational tasks and the available processor architecture components (both structural requirements), but also to allow user defined limits on the energy consumption, throughput, and maximal area occupation (parametric requirements). Through these constraints and objectives, the user is able to control both the size and complexity of the overall exploration problem and can influence the trade-offs that are considered during the design process.

Finding good task combinations to be executed in single ASIPs

Each of the tasks of the Polyhedral process network is then analyzed by the micro-level application analysis ③. Chapter 4 presents an efficient method for determining the best-case and worst-case execution times of tasks for a future VLIW processor designed according to the SiliconHive template. These best-case and worst-case execution time estimates, computed by the micro-level for tasks assigned by the macro-level to be executed on a single ASIP, are then taken by the macro-level and used during the probabilistic system exploration [38,54,55]. Using an evolutionary algorithm combined with Monte Carlo simulation and models of the inter-processor communication the probabilistic system exploration finds promising task clusterings. For each of the task-clusterings the micro-level is then consulted to produce an initial customized VLIW processor architecture $\textcircled{4}$.

Deciding the ASIP memory hierarchy and initial VLIW ASIP architecture

In the micro-level application-parallelization and coarse architecture synthesis stage, multiple tasks within a single cluster may be transformed so that data-locality and re-use are optimized. The micro-level application parallelization tool [17,38] transforms the Polyhedral representation of the clustered loop kernels to reorder and fuse the kernel executions in such a way as to find possible trade-offs between the data-throughput and the processor area and power consumption. Although at this stage, the power consumption is assumed to be proportional to the area. Currently, loop fusion, loop tiling, and kernel vectorization are considered during this exploration, but the repertoire of transformations can be extended.

Loop fusion combines two kernels into a single new kernel which localizes any intermediate results which were communicated between the original kernel pair. This reduces the total memory requirements for the task cluster.

Loop tiling enlarges the granularity of data on which the kernel is running. This allows for a trade-off between the size of the local cache or scratch-pad memories in the processor versus the bandwidth required between the considered processor and its external memory.

Loop vectorization changes the data granularity for the computations performed within the kernel and reduces the number of instructions required for the execution of a given application (part).

The application parallelization uses estimates of the instruction-level parallelism (ILP) available in each kernel. These estimates are obtained as one of the results of the application analysis, as explained in Chapter 4. The application parallelization creates an optimized parallel structure of the application part mapped to a single VLIW ASIP and constructs a corresponding initial VLIW architecture with sufficient resources to achieve the predicted throughput. A Pareto-set of such coarse VLIW architecture designs is then returned to the macro-level exploration $\textcircled{5}$ which then uses the performance metrics of these more accurate designs in combination with a set of communication models (obtained from the communication and global memory exploration) to determine the final multi-processor system architecture. Each of these returned candidate VLIW processor architectures can be synthesized $\textcircled{6}$ (Appendix A) and a transformed C code can be generated to match the selected high-level loop transformations

7. After the ASIP synthesis and corresponding C code generation, the ASIP based hardware/software subsystem is ready for simulation using the SiliconHive tools 8.

Finalizing the ASIP design

The final optimization step is applied when the area and/or energy consumption of the thus far synthesized ASIP based platform is not yet satisfying the design constraints or for further optimization of the design objectives. This optimization is performed by the micro-level instruction-set architecture synthesis, separately for each single VLIW processor in the system when requested by the macro-level architecture exploration 9. This processor architecture optimization step tries to improve the processor architecture both by the addition of application-specific instruction-set extensions as custom operations [59–61], and by the removal of unused or scarcely used processor components which were included as part of the processor building blocks used during the construction of the initial processor prototype. The remainder of this dissertation focusses on the final processor optimization process. Several improvements to the processor area and energy models are presented in Chapter 5 and both the exploration algorithm (Chapter 7), and techniques reducing the exploration time 10 (Chapter 6) are also discussed. The resulting refined ASIP design, together with more detailed area, energy, and execution time metrics, are then returned to the macro-level architecture exploration 11.

Finalizing the MPSoC platform

Finally, the macro-level architecture exploration continues with an exploration of the MPSoC interconnect and global memory structure 12 based on the available design alternatives for the various VLIW processor cores in the system. The macro-level also selects the appropriate VLIW instances from the set of refined architectures produced through the micro-level architecture exploration. Combining the selected VLIW instances and the synthesized interconnect and global memory structure allows the introduction of system level power control 13 (e.g. voltage scaling, power gating) after which the full system can be simulated through the SiliconHive tools 14 or emulated on FPGA 15. After this final validation of the system design, the tools can finalize the design and (semi-)automatically produce the required RTL and software descriptions for performing further (ASIC) hardware synthesis 16 and the production of the final system prototype.

3.2 ASIP architecture exploration: An example

This section presents an example walk-through of the automatic ASAM micro-level architecture exploration and synthesis process, to further illustrate the ASAM approach for designing a fully customized VLIW ASIP processor. The application

```c
int input_image[N][M];
int temp_image[N/2][M];
int output_image[N/2][M/2];

void downsample2d(void)
{
  int h, w;

  // kernel 1: vertical down sampling
  for(h=0; h < N/2; h++) {
    for(w=0; w < M; w++) {
      temp_image[h][w] =
        (input_image[2*h][w] + input_image[2*h+1][w]) >> 1;
    }
  }

  // kernel 2: horizontal down sampling
  for(h=0; h < N/2; h++) {
    for(w=0; w < M/2; w++) {
      output_image[h][w] =
        (temp_image[h][2*w] + temp_image[h][2*w+1]) >> 1;
    }
  }
}
```

Listing 3.1: *2D down sampling a $N \times M$ image, original code*

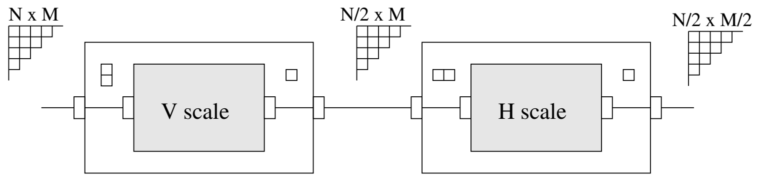


Figure 3.2: *Array-OL representation of the horizontal and vertical down-sampling kernels from Listing 3.1*

shown in Listing 3.1, 2D down sampling, was selected for this demonstration. Down sampling is a common function in image processing which benefits from most of the considered transformations without being overly complex, and therefore it is appropriate to be used for the explanation. However, the methods presented here are equally applicable to much more complex applications and algorithms.

As can be seen from the code in Listing 3.1, 2D down sampling consists of two main kernels, performing vertical and horizontal down sampling respectively. As the kernels are fairly small (with only a few operations each) and at the same time have a quite high communication requirement (half of the original image is communicated from the first kernel to the second kernel), it is fairly likely that the macro-level architecture exploration will decide to map both kernels onto a single processor.

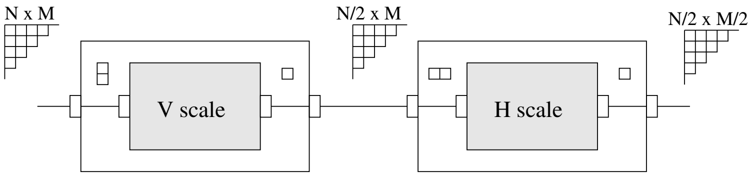
Array-OL [10], a graphically enriched representation of the polyhedral model, is used within the ASAM project in the second phase of the micro-level architecture exploration for performing the application restructuring and coarse processor architecture synthesis. Figure 3.2 shows a graphical representation of the Array-OL model corresponding to the input code of the 2D down sampling application. Both vertical and horizontal down sampling kernels (`V scale` and `H scale` respectively) are shown as grey rectangles. The input and output sizes for each kernel iteration are illustrated next to the input and output ports, both kernels consume two data elements and produce a single data element, the main difference being the orientation of the two consumed elements in the 2D data domain. The repetition domain surrounds each kernel and represents the loop-nest that wraps around each kernel in Listing 3.1.

From both the Array-OL model and the original source code, it can be seen that this implementation of the algorithm requires half of the original input image in temporary storage locations, as well as, both the complete input and output image directly accessible by the processor. This data needs to be stored either in the ASIP local memories or in a, usually slower, external memory.

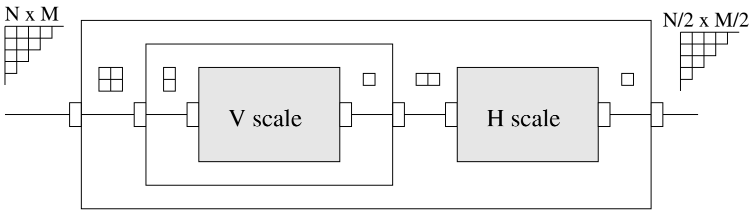
3.2.1 Application code restructuring and initial architecture construction

The application parallelization phase of the micro-level architecture exploration [17, 18, 38] starts to optimize the data locality and memory architecture of the customized processor with the Array-OL model of Figure 3.2 as input. Figure 3.3 shows the effect that some of the considered transformations have on the Array-OL model.

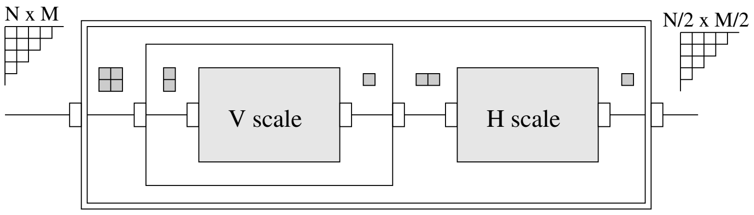
The first transformation that will be considered is loop (or kernel) fusion. The main advantages of this transformation are a reduction of the temporary data storage and an increased kernel size. Figure 3.3b illustrates the effect of kernel fusion on the example code. In order to perform loop fusion, a single repetition domain needs to be put around both kernels. Before this is possible, the `V scale`



(a) Original code structure



(b) After kernel fusion



(c) After both kernel fusion and vectorization with vectorized data elements colored gray and a double box for denoting the vectorized iteration domain

Figure 3.3: *Array-OL representation of the horizontal and vertical down-sampling kernels after different optimizations*

kernel will need to be executed twice so that it produces the appropriate data elements for the `H scale` kernel. This results in a second repetition domain wrapping only the `V scale` kernel. This second repetition domain only has two iterations and will be unrolled in the generated C code. Unrolling repetition domains with a small repetition count increases the kernel size and usually has a positive effect on the instruction-level parallelism available in the application. As can be seen in Figure 3.3b, loop fusion significantly reduces the temporary storage requirements of the application. In this case, loop fusion successfully removed the entire temporary storage except for a few single data elements.

After the application of fusion, both vectorization and tiling are explored using a genetic algorithm [17, 18]. Vectorization is mainly used to increase the number of data elements that are processed per cycle, as it changes the data granularity at which the computations are performed. Two repetition domains exist in the fused version of the application, the inner and the outer repetition domain. Both have no dependencies on previous iterations and can be vectorized at will. However, the inner repetition domain wrapping the `V scale` kernel only has two iterations and will provide only very limited performance impact when vectorized. The more logical choice is therefor to unroll this inner loop and to vectorize the outer repetition domain. The main limitation of the vectorization here is that vectorization puts a constraint on the possible input image sizes (unless strip-mining is used or explicit padding is added to the data). In this case we assume that the input image width is a multiple of 32, which results in a maximum vector width of 16 since the application needs to read two (vector) elements next to each other to feed the inner repetition domain. Figure 3.3c illustrates the vectorized and fused kernels. A double box on the outer repetition domain and the colored data elements illustrate the changed data granularity. Care should be taken when vectorizing the `H scale` kernel as it consumes two consecutive data elements to produce its result. When vectorizing such a kernel, vector shuffling operations are required to reorganize the data in such a way that the consecutive elements are put into separate vector elements so that the original kernel code can be kept. Listing 3.2 illustrates how this is achieved using the `OP_vec_odd` and `OP_vec_even` intrinsics, which select the even and odd elements of the input vectors respectively.

Finally, tiling is applied to the restructured loop nest to enable more freedom in the global mapping of the input and output data arrays. Tiling is used in the ASAM project to improve the data locality of the processor cores and to distribute data between the global memory and the local memories of each processor. This allows a significant reduction in the required size of local memories for the final processor designs. It also allows us to use direct-memory-access (DMA) controllers to perform data transfers in parallel to the actual computations. A tiled implementation therefore enables streaming operation of the kernel. This has as major benefit that the restructured kernels can start processing data while it is still being produced by the source of the data (an image sensor or other processing step in a larger application). The only requirement for a streaming application

```
#include <hive/asam_support_cell.h>
#define N_BUFFERS 2 // use double-buffering

volatile tvector MEM(ASAM_ISP_MEM_vmem) image_hive[N_BUFFERS][4];
tvector MEM(ASAM_ISP_MEM_vmem2) image_hive2[N_BUFFERS];

int height_hive, width_hive;

void down(void) ENTRY
{
  int n, w1, w2, i;

  const int final_h = height_hive>>1;
  const int final_w = (width_hive/ASAM_ISP_VEC_N_WAYS)>>1;
  const int n_tiles = final_h*final_w;

  // initialize empty space on input fifo
  for(i = 0; i < N_BUFFERS; i++) {
    OP_std_snd(FIFO_SOURCE, 0);
  }

  // run kernel
  for(w1=w2=n=0; n < n_tiles; n++, w1++, w2++) {
    // local storage for kernel
    tvector v1, v2, v3, va, vb, vc, vd;

    // acquire input token
a:  OP_std_rcv(FIFO_SOURCE) NO_ALIAS;

    va = image_hive[w1&(N_BUFFERS-1)][0] DIST(a, 1);
    vb = image_hive[w1&(N_BUFFERS-1)][1] DIST(a, 2);
    vc = image_hive[w1&(N_BUFFERS-1)][2] DIST(a, 3);
b:  vd = image_hive[w1&(N_BUFFERS-1)][3] DIST(a, 4);

    // release input
    OP_std_snd(FIFO_SOURCE, 0) AFTER(b,0) NO_ALIAS;

    // compute kernel
    v1 = (va + vc)>>1;
    v2 = (vb + vd)>>1;
    v3 = (OP_vec_odd(v1,v2) + OP_vec_even(v1,v2))>>1;

    // acquire output space, write result, and release output
c:  OP_std_rcv(FIFO_SINK) NO_ALIAS;
d:  image_hive2[w2&(N_BUFFERS-1)] = v3 DIST(c, 1);
    OP_std_snd(FIFO_SINK, 0) AFTER(d, 0) NO_ALIAS;

#pragma hivecc exhaustive, pipelining=0, stuck=3000
  }
}
```

Listing 3.2: *2D down sampling a $N \times M$ image, fused, vectorized, and tiled for streaming operation using double buffering*

is that there is sufficient data to perform at least one iteration and that there is space available to store the (partial) results.

Listing 3.2 shows the transformed C code of the 2D down sampling application. It also shows the SiliconHive annotations that control the memory mapping (`MEM`), synchronization of handshake signals and the actual data reads (`DIST` and `AFTER`), and intrinsics (`OP_`...) for communicating control signals with the DMA controller over FIFO connections to control the data transfers, as well as, for performing vector shuffling operations. This implementation uses double buffering to enable data transfers in parallel to the computation. Two external DMA channels are used to arrange the data transfers to and from the input (`image_hive`) and output (`image_hive2`) buffers respectively. These buffers are mapped into different memories in order to enable parallel access. The DMA channels are managed through FIFO channels, sending a FIFO token on an incoming channel releases space whereas receiving a FIFO token corresponds to acquiring data in the next buffer slot. The program therefore starts by computing the total number of tiles to be processed for the dataset and reporting the empty state of the input buffer to the DMA controller on lines 13–20. The kernel then enters its main loop and waits for an input tile from the DMA controller on line 23. The kernel data is then loaded into the register file when the input tile is obtained. The load operations on lines 28–33 also demonstrate the added sequencing annotations using the `DIST` directive, this directive ensures that the compiler does not reorder the load operations with respect to the FIFO handshake. Once the data is loaded into the processor, the input tile can be released (line 36) and the main computations of the fused kernels can be performed (lines 39–41). An output tile then needs to be acquired from the DMA controller on line 44 (although the compiler is free to reorder this acquisition with the kernel computation). Finally the output is written to the output memory (line 45) and the output tile is marked as finished to the output DMA channel (line 46). The entire kernel loop is then marked for exhaustive scheduling and software pipelining using a compiler directive on line 48.

During the loop transformation exploration process, the parallelism available in the core loop nest(s) of the code is estimated, using the methods presented in Chapter 4. This enables the code restructuring and initial architecture construction process to construct an ASIP architecture that has an issue-width that is appropriate for obtaining the predicted throughput. Figure 3.4 shows the processor that was constructed for the transformed C-code of the down-sampling application. The constructed processor is a wide VLIW ASIP with 4 16-way vector issue-slots, 2 vector memories for storing the input and output buffers, 3 scalar issue-slots (including fifo and control operations), and a small scalar memory for storing the width and height parameters of the kernel.

Running the restructured code, presented in Listing 3.2, on the so constructed initial processor architecture demonstrates that this ASIP architecture and application code combination is indeed capable of processing pixels at the predicted throughput of a full vector width of input pixels (16 pixels) per cycle. The

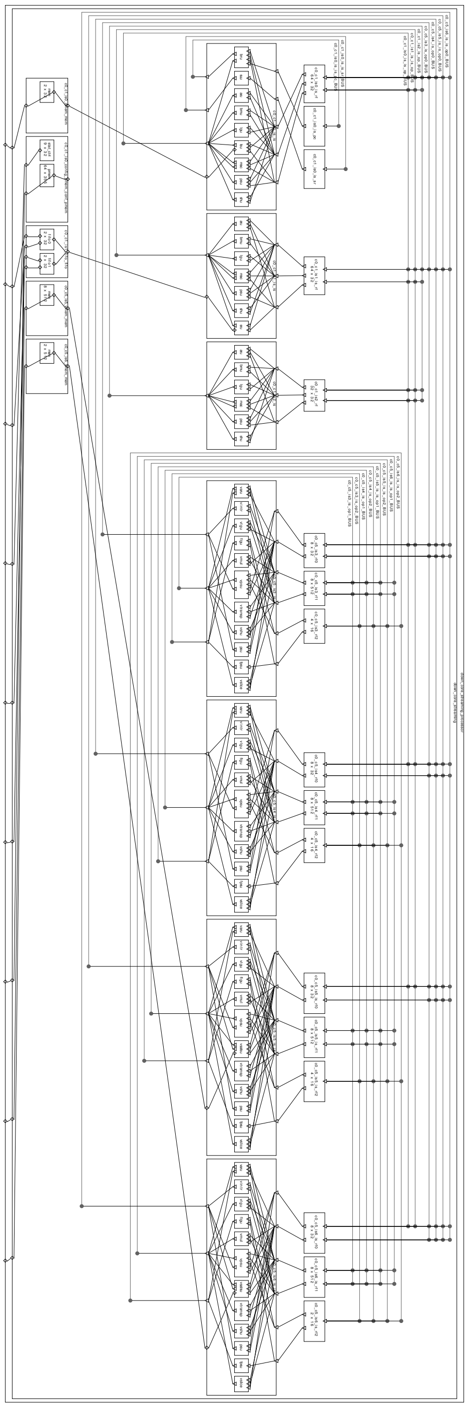


Figure 3.4: *Initial processor architecture for the down-scaling application based on code restructuring and initial architecture construction exploration decisions*

transformed code also utilizes the proposed processor architecture quite well, as it has an 85% utilization of the issue-slots in the core loop. However, large parts of the instruction-set remain unused which results in an inefficient use of the provided function units and requires an unnecessarily wide program memory. As such, this architecture is still quite overdimensioned and can substantially be reduced during the architecture refinement phase to decrease both the area and power consumption, while still realizing the required throughput.

At this point, the tasks of the code restructuring and initial architecture construction are completed and the second phase of the micro-level architecture exploration can be concluded. The proposed initial processor architecture is returned to the macro-level architecture exploration which can now explore the system memory architecture based on each processor's minimal memory size requirements. Extra buffer space can be added based on the overall mapping of the target application tasks and their respective communication buffer size requirements during the global interconnect and memory hierarchy exploration.

3.2.2 ASIP instruction-set synthesis through architecture refinement

As already mentioned above, the initial coarse processor architecture proposed by the second phase of the micro-level architecture exploration is usually overdimensioned. It is composed of issue-slots taken from a standard library and thereby supports a large variation of operations in each issue-slot. However, usually not all of these operations need to be replicated into each issue-slot and many of them can be removed without any impact on the execution time of the target application. In some specific cases, even the number of the VLIW issue-slots can be reduced. Furthermore, register files also have been introduced with quite large sizes and may be reduced as well. Removing these redundant resources from the initial architecture will greatly simplify the structure of the interconnect between the issue-slot outputs and the register file inputs. This in turn can result in a large reduction of the number of program word bits required for each instruction in the program memory which further reduces the processor's area and energy consumption.

The third phase of the micro-level architecture exploration, the instruction-set architecture synthesis, performs the architecture refinement using the following three techniques; instruction-set extension, (passive) architecture shrinking, and (active) architecture reduction.

Instruction-set extension

Instruction-set extension can (optionally) be applied as the first step of the instruction-set architecture synthesis when a very high performance and/or an extremely low power solution is required. During this step, common operation patterns are identified in the target application, function units implementing each of them in hardware are constructed, and then the application specific instructions corresponding to them are added to the instruction-set of the initial prototype which was obtained from the previous exploration phase. For example, the down sampling application has a frequently occurring pattern where two values are added and the result is shifted by one place, which effectively computes the average of both values. This operation pattern is the key computation in both the vertical and horizontal down sampling kernels. Implementing the complete pattern as a single complex operation provides two benefits: it results in a smaller kernel body, as fewer operations are required to encode the entire algorithm, and it improves the latency of the kernel execution by executing both the addition and the shift in the same clock cycle. As an additional advantage, the use of complex operations also reduces the number of register file accesses as intermediate values between the operations in a pattern are no longer stored in the register file. This results in a further reduction in the energy consumption and can result in a lowered register file pressure which allows us to further reduce the register file size.

The detection and selection of candidate operation patterns for the creation of custom operations, as well as, the insertion of these custom operations into the initial prototype is performed by the designer of the processor with the help of an operation pattern enumeration tool [59–61]. This tool supports the designer by enumerating candidate operation patterns based on the frequency of their occurrence, as well as, the possibilities for hardware sharing between custom operations.

Architecture shrinking

Architecture shrinking passively strips unused components from an oversized ASIP architecture and estimates the effects of the removal of these components on the processor architecture design. During the shrinking process, individual issue-slots, function-units, register-files, memories, and/or (custom) operations can get removed from the architecture. Register files and memories can also be resized to provide exactly the required amount of space. As a result, the connectivity of interconnect, as well as, the size of the instruction word and program memory, can be drastically reduced, without any impact on the temporal performance (latency, throughput) of the overall ASIP design.

The passive shrinking approach is fully implemented in the area and energy modeling as presented in Chapter 5. Doing so allows for an efficient estimation of the benefits of a specific architectural shrinking. In our implementation, we allow for user control of the kinds of elements which are removed during the shrinking process. For example, enabling or disabling the removal of single operations from a candidate architecture can have a large impact on the re-programmability of the resulting processor. Removing all operations except those that are required for the functioning of the target application will result in an architecture that may only (effectively) support small variations on the original algorithm but will also provide a higher efficiency that is closer to a non-programmable hardware implementation. However, keeping some of the less costly (though currently unused) instructions in the final architecture design can improve the support for variations of the original algorithm at the cost of a somewhat lower energy efficiency. Deciding the granularity at which to perform the exploration depends strongly on the intended purpose of the design and, as such, is left to the designer of the processor architecture.

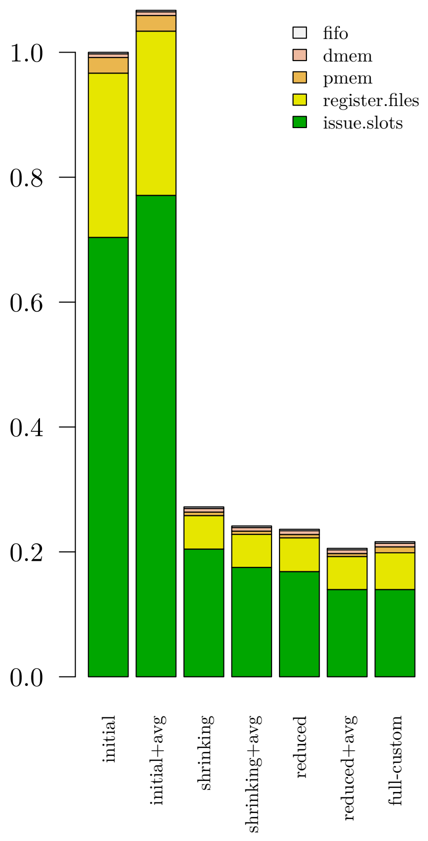
Architecture reduction

Often, performing only the architecture shrinking will not result in the most efficient architecture design. For instance, instruction scheduling heuristics may have decided to map several operations of the same kind onto different issue-slots, while these could have been mapped into the same issue-slot. This may result in the same operation being supported by multiple issue-slots when this is not strictly required for achieving the required performance of the algorithm. The ASIP architecture reduction technique implements the active component of our instruction-set synthesis framework. It actively tries to suppress the usage of specific architecture components (issue-slots or function-units) by disabling them as selection alternatives for an operation in the compiler. Doing so will render them unused in the resulting application mapping which allows for the successive architecture shrinking to remove them from the final design.

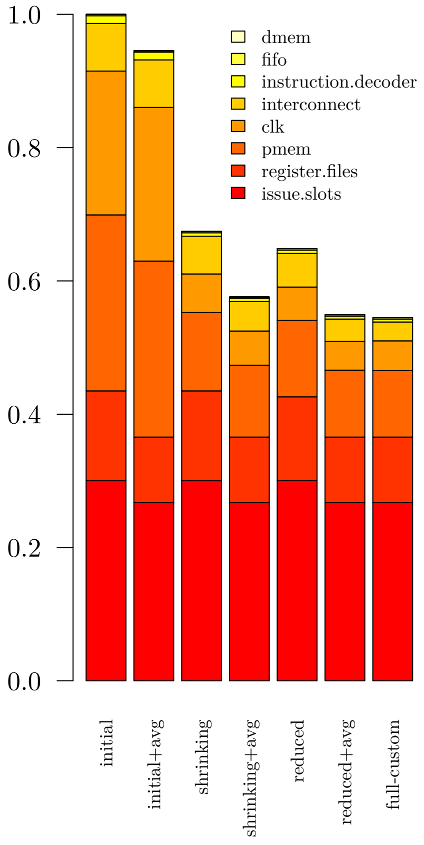
Chapter 7 compares several implementations of the presented architecture refinement methods, demonstrates the benefits of combining shrinking and reduction techniques, and compares their effectiveness and exploration time. Several different reduced architectures are usually considered and for each of those candidate architectures the target application code will need to be compiled and simulated to determine the performance of the proposed candidate. Chapter 6 introduces two intermediate result caching techniques which greatly speed-up this iterative process by remembering and reusing information on the previously considered architectures. The caching framework recognizes when previous compilation and/or simulation results can be re-used and this way provides a big improvement on the required exploration time.

Experimental results

Applying our architecture refinement techniques on the example down-sampling application demonstrates their high effectiveness. Figure 3.5 shows the architecture optimization effects during various stages of the optimization on both the area and energy consumption of different architecture variations. Each bar in the graphs is subdivided to show the area and energy distribution across the various components of the processor architectures and shows the most costly components at the bottom. It should be noted however, that these figures only show the energy and area cost of the core processor architecture with its local memories. The other components of the system such as the (usually larger) external memories are not accounted for in the graphs. The first (leftmost) bar shows the area and energy requirement of the initial prototype that was proposed by the previous micro-level architecture exploration phase. All bars are normalized with relation to the initial prototype.



(a) area



(b) dynamic energy

Figure 3.5: *Estimated area and energy during different stages of optimization, normalized to initial prototype*

The second bar illustrates the effect of adding the custom operation pattern for the add-shift (`avg`) custom operation which is applied on two vectors of data elements. It shows the increase in the processor area (due to the extra hardware) and a decrease in the active energy consumption due to a decreased number of reads from the register file. The total execution time of the algorithm did not change due to the inclusion of this operation. This can be explained since the cycle-count of the down-sampling application is limited by the initiation interval of the main loop kernel, which in turn is dominated by the amount of load operations from the input memory. Instead, the introduction of the custom operation results in a decrease of the required number of issue-slots for the processor. This is achieved by joining operations that were already scheduled in parallel into a single custom operation. The main area and energy savings from this optimization are the removal of an issue-slot and its register files, as well as, the corresponding reduction in the width of the program memory as fewer operations need to be encoded per instruction.

The result of passively shrinking both the initial architecture and the version which includes the custom operation is shown in bars three and four respectively. Shrinking has a large impact on the size of the issue-slots as many standard operations can be completely removed and others are only needed in a few issue-slots. The register files provided by the initial architecture are also significantly over-dimensioned. The results shown in this example demonstrate those obtained using a full-customization of the processor architecture and include the removal of all unused elements from the initial (extended) architecture.

Continuing the process by actively exploring further architecture reductions results in another decrease in both the area and energy requirements for the processor architecture design, as shown by the fifth and sixth bars. In both cases, the design was optimized to improve the energy-delay product of the final processor in an attempt to find a smaller, more efficient, version of the architecture without giving up too much on the temporal performance. As a result, both proposed architectures (with and without custom operation) consume about 5% less energy when compared to their shrunk versions. For the processor architecture which includes the custom operation, the architecture reduction phase was able to remove a complete vector issue-slot. This resulted in a final proposed architecture which has a 91% utilization of its issue-slots.

The exploration time required for both the original and extended initial architectures also clearly shows the effectiveness of our intermediate result caching techniques. The exploration of both architectures took 54 minutes in total on a 2.8 GHz Intel Core i7 with 6 GB of RAM memory. The caching framework was able to reuse a significant number of compilation and simulation results, over 72% of the compilation time and over 94% of the simulation time were avoided by the use of our caching techniques. Overall, the automatic exploration framework considered 407 different processor architecture variations in less than 1 hour and produced two highly optimized processor designs. Both final designs proposed by the automated architecture exploration reduce the area of the initial design by more than a factor of 4x, and the energy consumption by almost 2x.

After the exploration, we verified the predicted results by comparing them to the results obtained by actually constructing this proposed architecture using the SiliconHive tools. The result of this verification is demonstrated in the final bar of the bar-graph and the resulting architecture is shown in Figure 3.6. From this experiment we can learn that the area required for actually constructing

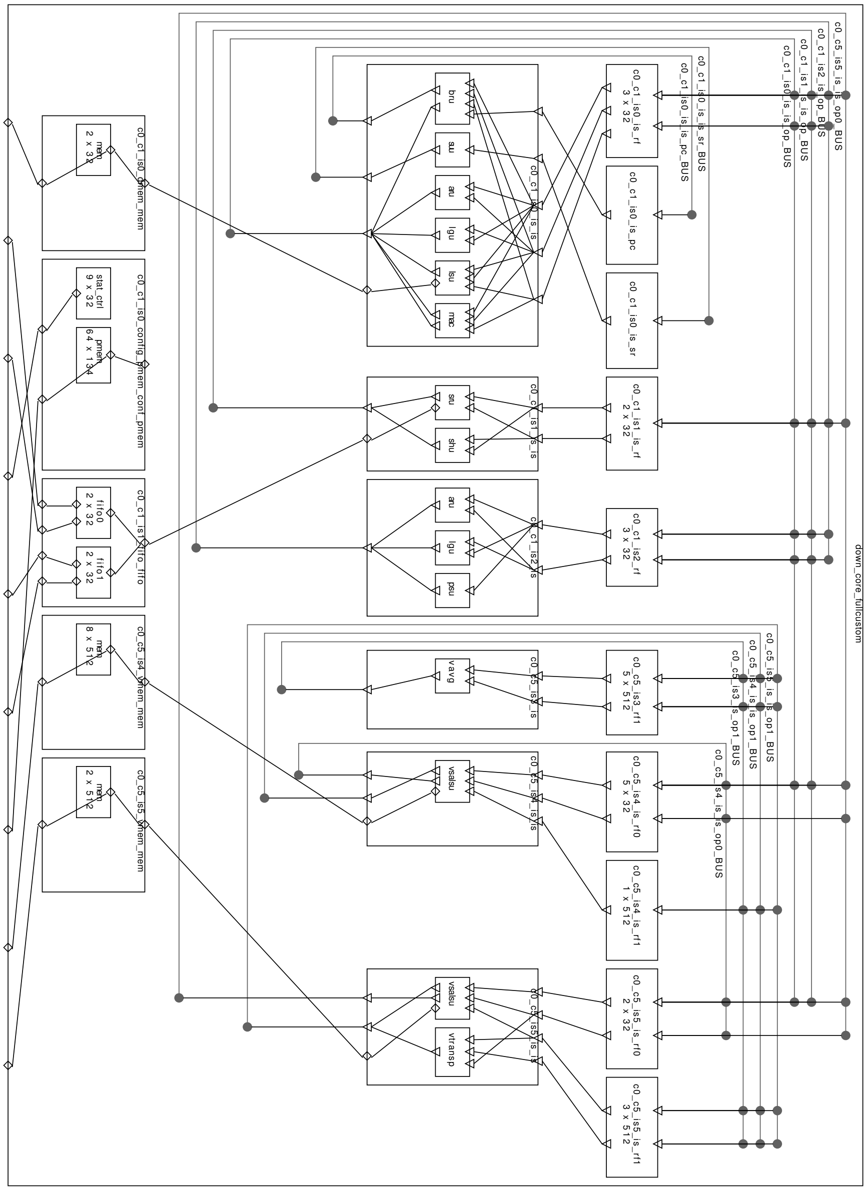


Figure 3.6: *The final full-custom architecture based on the exploration result*

the proposed processor architecture is slightly higher, while the energy required for running our algorithm on this architecture is slightly lower than predicted. These effects demonstrate some of the limitations of our current area and energy model in relation to the architecture template. In the case of our down sampling application, three discrepancies between the predicted and obtained area and energy numbers can be observed.

Firstly, the load-store unit, connected to the input memory, is only used for load operations. However, being a load-store unit, it is derived from a standard template library element which has a fixed set of input and output ports. Therefore, an extra register file must be added to be able to actually construct the processor without diverging from the current template, and in order to keep all input and output ports correctly connected. This is the 3th register file from the right in Figure 3.6, which is not used but does provide space for one 512 bit wide vector element and represents approximately 10% of the register file area in the final architecture. Improving the architecture template library such that it allows for a load-store unit with only load operations (and thus fewer input ports) would enable the removal of this extra register file and bring the total register file area back down to the predicted space. The automated tools presented in this dissertation are currently not able to perform this optimization but the SiliconHive template does allows the construction of such a load-only unit.

Secondly, the architecture model is completely agnostic of the operations implemented within function units (except for their names) and only counts accesses to register files. This is a limitation of the current implementation of the exploration tools and should be resolved as part of the future work. As a result, the architecture exploration is currently unable to recognize when a specific register file read or write port is unused in the proposed architecture. Removing such unused register file ports during the final architecture construction further simplifies the interconnect, reduces the cost of the register files in general, and reduces the number of instruction word bits required for programming the final processor architecture.

Finally, the automated exploration framework resizes the program memory based on the number of instructions required for encoding the target application. However, in its current implementation, it does not take into account the small processor initialization routine that also needs to be loaded into the program memory. In the case of the down sampling application, the addition of this initialization code results in a different rounding of the program memory size (from 32 lines to 64 lines). This results in a program memory area which is larger than predicted, though not twice as large as fewer bits are required to encode the instruction word due to the reduction of the interconnect complexity as explained previously.

3.3 Conclusion

This chapter has presented the ASAM approach to automatically designing a fully customized MPSoC based on the SiliconHive technology, and demonstrated how a single customized VLIW ASIP is automatically created for a given set of tasks. The remaining chapters of this dissertation present a more in-depth discussion of the techniques used during this automatic instantiation and customization process. In particular, we present our method for estimating the required issue-width of the new VLIW ASIP in Chapter 4, the processor enery and area model in Chapter 5, the BuildMaster instruction-set architecture exploration framework in Chapter 6, and the exploration algorithm in Chapter 7.

I've found from past experiences that the tighter you plan, the more likely you are to run into something unpredictable.

MacGyver, 1985

4 Early performance estimation

Traditionally, the issue-width decision for a VLIW processor has been based on an analysis of the instruction-level parallelism of the target application. Previous research [3, 71, 81, 85] 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. Larus [47] and Wall [85] focussed on finding the upper-bound of instruction-level parallelism over traces of a complete application. More recently, Cabezas and Stanley-Marbell [11] published a method for estimating the parallelism *distribution* 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 temporal performance at a minimal energy consumption. It is therefore important that enough parallelism is exploited 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* [80] defined their *smoothability* metric. This metric provides a score in the range of 0–100%. A program which

This chapter is based on:
Jordans, R.; Corvino, R. and Jóźwiak, L.: *Algorithm Parallelism Estimation for Constraining Instruction-Set Synthesis for VLIW Processors.* In DSD 2012.
Jordans, R.; Corvino, R.; Jóźwiak, L. and Corporaal, H.: *Exploring Processor Parallelism: Estimation Methods and Optimization Strategies.* In DDECS 2013.
Jordans, R.; Corvino, R.; Jóźwiak, L. and Corporaal, H.: *Exploring processor parallelism: Estimation methods and optimization strategies.* In International Journal of Microelectronics and Computer Science, 4 (2), 2013.

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 chapter, we will compare several methods to estimate instruction-level parallelism regarding their suitability for issue-width estimation and their computational complexity. The following methods are considered:

1. *Average parallelism (AP)* [3, 33, 34, 71, 81, 85], 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 chapter introduced in Section 4.1.1.

3. *Maximum parallelism (MP)* [11, 33, 34], providing an upper bound on the degree of parallelism that can be utilized by an application part.

4. *Required parallelism (RP)* [33, 34, 85], computing the minimal degree of parallelism as required for scheduling of an application part within a given latency bound.

We also consider the effect of software pipelining [12, 46, 70], 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 to 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).

4.1 Parallelism estimation of straight-line code

This section introduces three different methods for rapid application parallelism estimation, including our novel force based parallelism estimation method, a reference method for computing the required parallelism using constraint programming, and presents the results of our experimental research comparing the three estimation methods to the required parallelism reference.

The presented methods are used to predict the parallelism of straight line (sequential) code parts, usually called *basic blocks*. As such, the operations within a basic block form, through their dependencies, a directed acyclic graph (DAG) $G(V, E)$ with operations $V = \{v_i : i = 0, \ldots, N\}$ and operation dependencies $E = \{(v_i, v_j) : i, j = 1, \ldots, N\}$ (e.g. Figure 4.1a). We also use the algorithms for As Soon As Possible (ASAP) scheduling and As Late As Possible (ALAP) scheduling described in [19]. 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 time of the first node to start and the finish time of the last node to finish. The vector of ALAP scheduled operation start-times is similarly defined as $\mathbf{t}^L$. The application model used in the examples below 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.

4.1.1 Methods

The four parallelism estimation methods are defined as follows.

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 ($|V|$) by the required latency (λ), and provides a lower bound on the required issue-width.

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

Force based parallelism

Another estimate of the required issue-width can be obtained using a concept found in Force Directed Scheduling [65] in a novel way.

During force directed scheduling, a *distribution graph* is computed which provides information on the possible distribution of operations across the cycles of the schedule. The distribution-graph is 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 4.1. In this example, we assume that the maximum latency for the schedule is minimized, i.e. the ALAP schedule is computed for a latency of 5 cycles. Both operations v_1 and v_3 can be scheduled at 3 different moments, as shown by their ASAP-ALAP schedule interval in Figure 4.1b. Their scheduling probability is therefore $^1/_3$ for each cycle. The distribution graph of the DFG example is shown in Figure 4.1c. 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 $1^1/_3$.

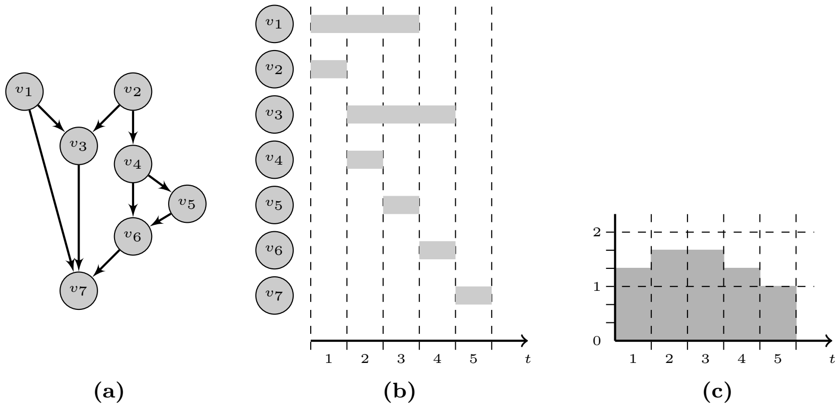


Figure 4.1: *Example DFG (a) with ASAP-ALAP schedule intervals (b), and the corresponding distribution graph used in estimating the force based parallelism (c).*

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 4.1c 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 1-dimensional IDCT benchmark code results in a value of 7.85, closely corresponding to the required parallelism of 8.

It should be noted that the force based parallelism estimate does not provide an upper nor lower bound on the parallelism, but a value close to the actually required value. Figure 4.1 shows an under-estimation while Figure 4.2 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.

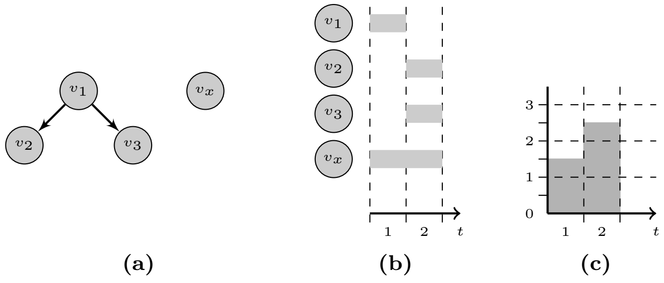


Figure 4.2: *An example DFG resulting in an over-estimation of the required parallelism by the FBP method.*

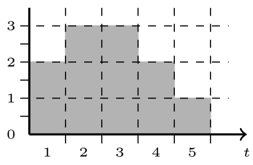


Figure 4.3: *Potential parallelism graph used in estimating the maximum parallelism for the example DFG given in Figure 4.1a.*

Maximum parallelism

The maximum parallelism [33, 34] can be estimated in a way that is similar to the estimation of the force based parallelism. The only difference is that we now compute a worst-case resource usage by counting all nodes using a weight of 1 for each cycle as shown in Figure 4.3. Estimating the maximum parallelism for the example DFG shown in Figure 4.1 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.

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.

Required parallelism

The required parallelism is used as reference in the experiments. We use constraint programming to compute the minimal degree of parallelism required for scheduling an application part within a given latency bound [33]. This is achieved by actually scheduling each basic block to its minimal latency using a common formulation [51], but then minimizing the available parallelism until a lower bound is found. Algorithm 4.1 shows the constraint formulation that was used.

Algorithm 4.1 Constraint-set for solving maximum required parallelism

```
Require: DAG $G(V,E)$ and latency bound $\bar{\lambda}$
Ensure: Calculate the minimal required parallelism $\Phi_{RP}$ of $G$ such that the
    scheduled latency $\lambda$ is inferior to $\bar{\lambda}$
1: $t_i \in (0, \ldots, \bar{\lambda} - 1)$
2: for all $(v_i, v_j) \in E$ do
3:     impose $t_i + 1 \leq t_j$
4: end for
5: impose $\Phi_{RP} = \max_{t:(0,\ldots,\bar{\lambda})} |v_i \in V : t_i = t|$
6: minimize $\Phi_{RP}$
```

The constraint programming approach is capable of finding a proven minimal degree of parallelism for each application part when given sufficient time. Time-outs of up to a minutes were sufficient for most cases except for the most complex blocks in our experiments for which 30 minutes was still insufficient, as a result a proven optimal value could be found in most cases.

4.1.2 Experimental results

All three above presented methods and our reference method for VLIW issue-width estimation using constraint programming have been implemented using the intermediate representation (IR) of the LLVM compiler framework [49]. This provides a method to compare their respective quality in the approximation of the required issue-width.

The experiments reported in this chapter have been performed on a set of 3667 basic blocks taken from an MPEG4-SP encoder application. This application contains a large set of basic blocks showing different kinds of processing which are representative for general video and image processing algorithms. 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 (`-constrain-lsu=`N) was selected as a common example of an explicit resource constraint which allows at most N load/store operations to be executed in parallel. 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 4.4 shows a box-plot of the results obtained from our experiments normalized to the required parallelism (RP) for achieving the ASAP

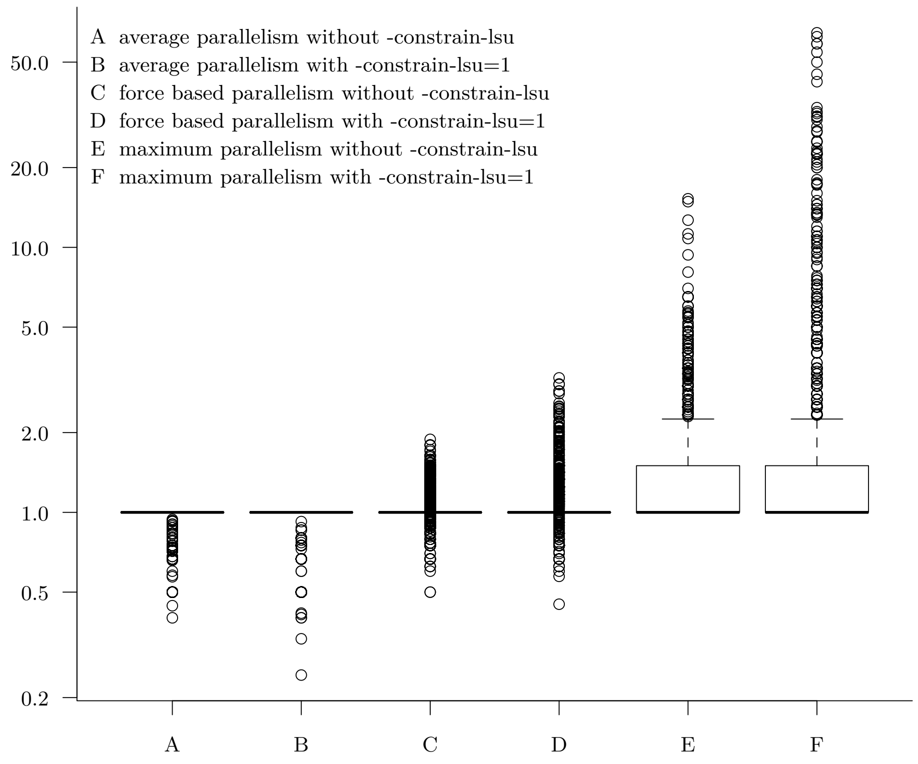


Figure 4.4: *The deviation of various parallelism estimation methods from the required parallelism. Normalized to the required parallelism when scheduled for the ASAP latency of each respective block.*

latency of each basic block found through an exhaustive search. The central box (flattened into a line for all experiments except E and F) contains the 50% of all samples surrounding the median. The whiskers extending from the box (also only visible for E and F) illustrate the tail of the sample distribution and extends up to 1.5 times the length of the center box to illustrate the bulk of the lower and upper quartile of the samples. The remaining points that fall outside of these whiskers are considered outliers and are drawn individually as small circles.

From the experimental results shown in Figure 4.4 we conclude, as expected, that the average parallelism provides a *lower-bound* on the VLIW issue-width required for executing the application, while the maximum parallelism provides an *upper-bound*. The average parallelism underestimates the required parallelism, 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 maximum parallelism provides an overestimation of up to two orders of magnitude and, on average, 31% for the unconstrained and 72% for the constrained experiments. The force based parallelism 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 force based parallelism is an underestimation of the required parallelism by 5.2x.

4.1.3 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.

4.2 VLIW issue-width optimization

The VLIW issue-width of application parts is explored at two points in the ASAM micro-architecture exploration flow. Firstly we need fast estimates of the latency and parallelism of key application parts to provide them for the coarse application parallelization and loop restructuring phase, where parallelism estimates are used to predict the required VLIW issue-width. The estimation of these input parameters is performed only once but the quality of the estimate is directly reflected in the quality of the proposed VLIW processor architecture candidates. Secondly, an exploration of the issue-width is performed during the final instruction-set synthesis and architecture refinement phase of the micro-level exploration. A highly detailed, and thus time consuming, scheduling algorithm is used at this stage. Selecting a good VLIW issue-width optimization strategy therefore can positively impact the overall instruction-set architecture synthesis time by reducing the number of required scheduling iterations.

The actually required issue-width can be found by using both a *growing* or *shrinking* strategy, where growing is the most common [16, 26, 69, 86] strategy for exploring parallelism. However, applying a linear search starting at 1 and incrementing with 1 for finding the optimal issue-width that guarantees a given latency may not be the best choice since this requires up to $\Phi_{RP} + 1$ iterations of the scheduling algorithm, with Φ_{RP} equal to the required parallelism. Finding the ASAP latency is trivial when no resource constraints are provided. As a result, only Φ_{RP} iterations are required when not considering additional resource constraints because it is possible to prove that there is no improvement possible for a solution with $\Phi_{RP} + 1$. However, an extra scheduling iteration is required when considering additional resource constraints in order to prove the optimality of the obtained result.

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.

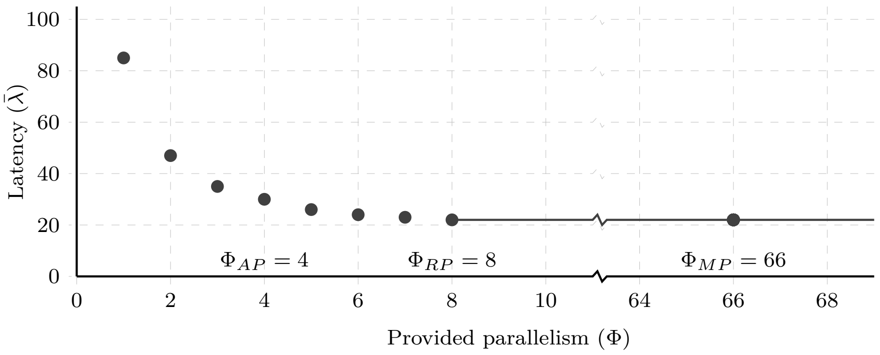


Figure 4.5: *Latency versus parallelism plot for an 8-point IDCT function with average parallelism (Φ_{AP}), required parallelism for its ASAP latency (Φ_{RP}), and maximum parallelism (Φ_{MP}) marked*

Both the previous work (e.g. [85]) 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 4.5), 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 independently of the division strategy of the remaining ranges. For instance, using the value of the force-based parallelism estimate Φ_{FBP} as first pivot, and continuing 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.

4.2.1 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.

Linear search

Both the 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 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 result satisfying our latency bound, it is required to verify that this is the optimal result. Assuming that the schedule latency decreases monotonically with respect to an increase in the issue-width of a VLIW processor allows us to only check for narrower architectures, 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.

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 4.5. 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 4.2 shows how such an unbalanced binary search can be implemented using a balancing parameter α, while using Φ_{FBP} as the first pivot.

Algorithm 4.2 Computing required parallelism using an unbalanced binary search where the balance is controlled by parameter α

```
Require: Basic block $BB$, latency bound $\bar{\lambda}$, and balance parameter $\alpha$ with $0 < \alpha < 1$
Ensure: Calculate the issue-width $\Phi$ of $BB$ such that the scheduled latency $\lambda \leq \bar{\lambda}$
 1: $\Phi_{max} \leftarrow \Phi_{MP}$
 2: $\Phi_{min} \leftarrow \Phi_{AP}$
 3: $\Phi_{pivot} \leftarrow \lfloor \Phi_{FBP} \rfloor$
 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:     $\Phi_{pivot} \leftarrow \Phi_{min} + \lfloor \alpha \cdot (\Phi_{max} - \Phi_{min}) \rfloor$
12: end while
13: return $\Phi_{RP} \leftarrow \Phi_{min}$
```

4.2.2 Experimental results

For the experiments with the search strategies 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 needed 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 (presented in Table 4.1) were obtained using basic blocks of at least 10 operations from the MPEG4-SP encode application to prevent bias from very small blocks that will not provide much parallelism.

It should be noted that the quality of our result strongly depends 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 monotonically decreasing latency in relation to an increasing issue-width to prevent the issue-width exploration from getting stuck at an early local minimum. This means that using a more effective (but slower) scheduling algorithm we will be able to achieve an even higher result quality. From the many available list-scheduler heuristics [76], we selected the *dependency height* (i.e. the difference between the goal latency and the ALAP schedule time of the node $\bar{\lambda} - t_i^L$) 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

[1]On average within 3% of the actually required parallelism as computed using an optimal

Table 4.1: *Total number of scheduler iterations required for finding the required parallelism for all (534) basic blocks with 10 or more operations of an MPEG4-SP encoder for several linear and binary search strategies*

method	**start**	α	**unconstrained**	**constrained**
linear	1		3283	1782
	AP		1133	914
	ceil(FBP)		1538	1401
	floor(FBP)		1388	1233
binary search	AP–MP	0.5	1942	2111
		0.2	1467	1519
		0.1	1428	1399
		0.04	1417	1347
		0.02	1458	1336
binary search	AP–FBP–MP	0.5	792	853
		0.2	783	849
		0.1	802	869
		0.04	833	887
		0.02	869	896

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.

4.2.3 Conclusion on the issue-width optimization

Usage of a binary search strategy with the force based parallelism 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 31% reduction from the currently used method of linear search from the average parallelism for the unconstrained experiments, and in a 7% reduction for the constrained experiments. We therefore recommend to use a combination of our force based parallelism metric with binary search when looking for a single design point using a time-consuming scheduling algorithm. However, we recognize that the 7% reduction in exploration time is much more likely as most architecture explorations will involve some form of resource constraints. As such, it might not be worth the added complexity of the exploration algorithm considering that the full Pareto-front of solutions can also be very interesting in many cases. Computing the Pareto-front requires the exhaustive linear search starting from 1.

scheduler based on constraint programming [51].

4.3 Parallelism estimation of pipelined loops

The parallelism estimation methods presented above focus on estimating the parallelism of single basic blocks. As such, they do not take inter block dependencies into account. Taking inter block dependencies into account often allows for a further performance improvement and results in an increase in instruction-level parallelism. One of key types of inter block dependencies are inter iteration dependencies of a loop kernel. These inter iteration dependencies are usually exploited using a technique called software pipelining [46]. With software pipelining, increased utilization of parallel resources is achieved by overlapping the execution of multiple iterations of a loop core. Figure 4.6, 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 4.3.1 provides a more in-depth explanation of this example.

Two main techniques based on different approaches are used for creating software pipelined schedules: *modulo scheduling* [46, 70], and *unroll-and-jam* [12]. Our estimation methods build upon their common concepts and are independent of the used software pipelining method.

4.3.1 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 is constrained by two factors: *1)* the available resources, and *2)* the inter-iteration dependencies of the loop core. Considering that both factors provide a bound on the II, II_{res} for the resource constraint, and II_{rec} for the inter-iteration dependencies (or recurrences), we find that the overall II can be defined as their maximum:

$$II = \max(II_{res}, II_{rec})$$

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. Within the VLIW ASIP architecture template used for the ASAM project, for each memory only a single load/store operation is performed per cycle[2]. As we do allow the existence of

[2]Memory solutions supporting multiple parallel LSUs are possible within the SiliconHive template but their usage is not explored within the ASAM project.

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 4.1. Figure 4.6a 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 4.1, 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_{res} is 3 (cf. Figure 4.6b)
2. When both arrays are mapped onto different memories the minimal II_{res} is 2 (cf. Figure 4.6c)

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

Listing 4.1: *Example loop nest showing an initiation interval constrained by the number of available load/store unit(s).*

In our architecture, accessing (reading or writing) 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.

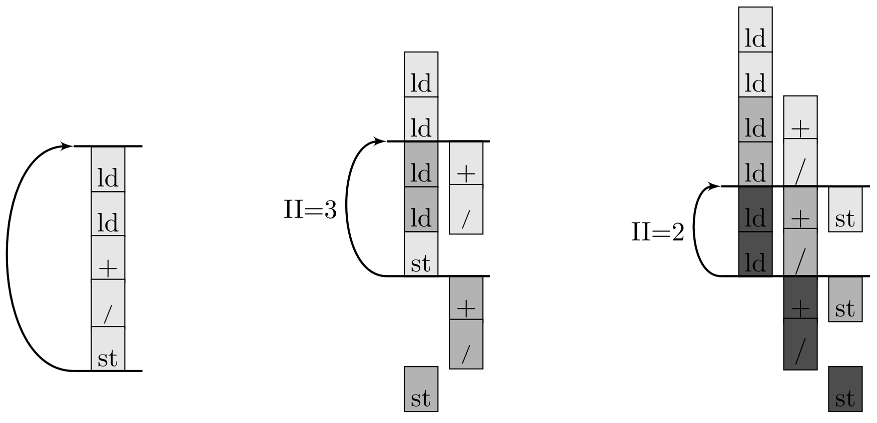
Inter-iteration dependencies

Inter-iteration dependencies appear when a loop iteration requires a result that was produced by an earlier loop iteration. So called *reduction loops* are frequently occurring examples of this kind of behaviour. Listing 4.2 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 4.2: *Example loop nest having an initiation interval constrained by a loop carried 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 memory carried



(a) Original (b) Single memory (c) Two memories

Figure 4.6: *Simplified schedules of the loop in Listing 4.1 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.*

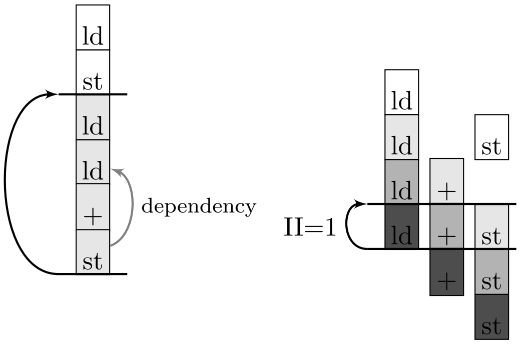
inter-iteration dependency can be translated into a register carried memory inter-iteration dependency by introducing a local variable whose values are stored in a register. Listing 4.3 shows how this can be achieved for the code shown in Listing 4.2.

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

Listing 4.3: *Restructured version of the code shown in Listing 4.2, changing a memory carried dependency into a register carried dependency.*

Figure 4.7 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.

In general, all memory carried inter-iteration dependencies can be translated to register carried ones by inserting one or more temporary variables into the code.



(a) Original **(b)** Transformed

Figure 4.7: *Simplified schedules for the original (Listing 4.2) and transformed (Listing 4.3) 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.*

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 lower, more manageable number.

In our work on VLIW ASIP design the size of the register file can be adapted to fit the required number of temporary variables. As such, register carried dependencies limit the initiation interval solely through their length. However, we are still able to insert custom operations that can significantly reduce the critical path length that constrains the initiation interval during the instruction-set architecture synthesis. 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 chapter are solely based on the resource constraints of the architecture (i.e. we consider $II_{rec} = 1$), and in particular on the memory access constraints described above.

4.3.2 Methods

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

Utilization-based parallelism 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 the loop body ($|V|$) 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 4.7b 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 4.7b.

Duplication-based parallelism estimation

Our second method computes the number of parallel copies of the loop body in the software pipelined schedule. We compute this number by dividing the latency of a single execution of the loop body (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 4.7b, 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 4.2. Other parallelism estimates, such as found using the average parallelism or force-based parappelism methods, are also usable. However, using the parallelism estimates obtained with the average parallelism 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 force-based parallelism method will produce very similar results compared to our choice of using the computed parallelism from Algorithm 4.2.

4.3.3 Experimental results

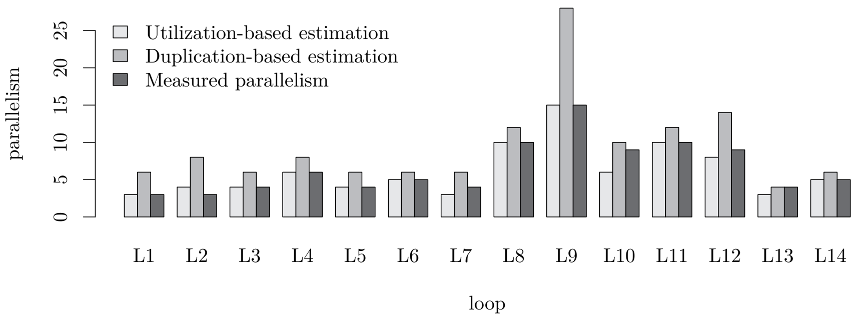


Figure 4.8: *Comparison of estimated parallelism versus observed parallelism after software pipelining in custom built architectures.*

For our experiments related to parallelism estimation of pipelined loops we mapped several kernels (partially) from the Polybench benchmark [68] 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.

Figure 4.8 shows the estimated parallelism using our two methods together with the actually obtained parallelism achieved with the SiliconHive tools when running the loop code on an architecture which was manually customized for that particular loop. As such, our measured reference is the II that was achieved using the manually optimized code on an overly wide simulated SiliconHive processor architecture. 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 results in a quite large over-estimation (54% on average). However, there are several cases where the utilization-based estimation underestimates the required parallelism.

Further investigation into the cases where this underestimation occurs shows us that their source is outside of our method, but in the abstraction of LLVM's intermediate representation (IR) over the actual instruction-set of our target architecture. The LLVM IR includes an operation `getelementpointer` which is used to perform address computation. In the target-specific back-end of the compiler, this operation gets lowered into zero or more target specific operations depending on the complexity of the addressing and the available architecture features. The cost of this operation is determined during the parallelism estimation through the machine model which estimates the required number of operations for the target architecture. In our experiments miss-prediction of the required number of operations for the address computation was responsible for each of the observed underestimations. 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) where the number of operations directly translates into the parallelism of the loop. Overestimations of the parallelism are similarly related to the IR abstractions, loops L2 and L4 for example both include IR operations that get combined into larger operations such as multiply-add. Improving the machine model for our estimator or implementing this estimation at a lower, more accurate, level (after expansion of the address computation and detection of operation patterns) of the compiler can therefore result in better quality 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 parallelism estimates of software pipelined loops. The average error margin of the utilization-based method was shown to be less then 1%. Furthermore, the observed errors were not caused by our method, but by the abstractions of the LLVM-IR on which our analysis was performed. Implementing the method in a later stage of compilation providing more precise information will result in even more accurate estimations.

4.4 Conclusion

In this chapter, we presented and compared three methods for estimating the required VLIW ASIP issue-width for a given target application. We experimentally demonstrated that our force based parallelism estimation proposed in this chapter delivers results with only 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 considered 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 up to 31%. 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 very accurately, with less than 1% error on average. Finally, we found that the remaining estimation errors were caused by the application code abstractions of the LLVM IR on which we based our estimations. Such an estimation inaccuracy is a direct consequence of the high abstraction level at which these estimates were obtained. Lowering the abstraction level through re-implementing the method in a later compilation stage is expected to further improve its accuracy.

The whole is other than the sum of its parts

Kurt Koffka, 1886–1941

5 Area and energy modeling

A key to an effective and efficient automated instruction-set architecture exploration is a rapid, but accurate enough, quality estimation of a proposed processor design. The three quality aspects of a proposed processor and application combination that are usually considered are the processing speed, processor area, and energy consumed when running the application. An actual complete construction of a finalized hardware and software design for each processor variation considered during the processor's instruction-set architecture exploration requires a significant amount of effort, which is at least impractical, but usually prohibitive. Therefore, the actual customization of the processor and related software alternatives, as well as, their analysis through simulation or emulation should be replaced by analytic models as much as possible. These models should be accurate enough to provide reasonable estimates of the processing speed, area, and energy, and fast enough to enable a thorough design space exploration.

Modeling the execution time of a given application software alternative on a proposed processor architecture, as well as, modeling the processor area, are relatively simple when compared to the modeling of the energy consumption. The execution time (as expressed in clock cycles) can usually be obtained from a cycle-accurate simulation of the application and does not require a full construction of the actual processor architecture. Similarly, the estimation of the processor area only requires knowledge about the internal structure and organization of the

This chapter is based on:
Jordans, R.; Corvino, R.; Jóźwiak, L. and Corporaal, H.: *An Efficient Method for Energy Estimation of Application Specific Instruction-set Processors.* In DSD 2013.

processor. Such information can be derived from a gate-level implementation of the processor architecture, either directly (more accurate) or through the summation of the area of its components as remembered from previous synthesis results (less accurate).

The modeling of energy consumption is much more difficult, as it requires both information about the physical structure of the finalized processor design from hardware synthesis, as well as, information on the activity of the various components (preferably as toggle rates at the wire level) for a given application software version. Combining these together as a weighted sum of the component activities and the energy consumption of these components for a single transition yields the overall dynamic energy consumption estimate. The most accurate results can be achieved when the activities of single wires in the design are considered (using gate-level simulation) but this kind of simulation is usually very time consuming. This problem gets exaggerated further when we consider that a full simulation, using a sufficiently large set of input data of the application on the proposed architecture is often required in order to get a representative estimation of the consumed energy. Therefore, several techniques have been proposed in the past to model the energy consumption of a processor at the instruction level (e.g. [8, 44, 45, 72, 88]). These techniques have been reported to have average error margins of approximately 1–5%. While not exact, such error margins can be tolerated during architecture evaluation during instruction-set synthesis. The downside of these techniques is that they usually require an extremely time consuming simulation to obtain the component activity for each proposed architecture candidate. This makes the iterative instruction-set architecture exploration process slow and severely limits the number of candidates that can be considered during the exploration.

In this chapter we first present a brief description of the area and energy models used within the ASAM project. We then present the existing methods for obtaining the required activity counts and introduce our novel method which, in most cases, completely eliminates the need for re-simulation of the target application for new variations of a previously considered processor design. Our new energy estimation method enables both to significantly speed-up the design space exploration process, and the exploration of a larger design space.

5.1 Estimating area and energy

A set of area and energy models was made available within the ASAM project for estimating the physical characteristics of proposed ASIP architectures. Overall, these models use methods that are very similar to those of the related work. These models predict the area and energy in three parts; total area, static energy consumption, and dynamic energy consumption.

The total area (A_{total}, equation 5.1) is computed as the sum of the major ASIP component areas. The specific contributions of these components are discussed below in more detail.

$$\begin{aligned}A_{total} = &\sum_{IS}\left(A_{is_{misc}} + \sum_{FU_{is}} A_{fu}\right)\\ &+ \sum_{RF} A_{rf}\\ &+ \sum_{MEM} A_{memory}\\ &+ A_{interconnect}\\ &+ A_{misc}\end{aligned} \tag{5.1}$$

Energy which is consumed independently of the activity (or inactivity) within the ASIP is considered static energy consumption. For a large part, this energy consumption is caused by leakage in the transistors of the actual ASIP implementation. As such, the static energy consumption depends largely on the implementation choices for each of the ASIP components. In the ASAM ASIP model the static energy consumption is computed through the accumulation of a multiplication of the area of each component by a leakage constant for components of that type, similar to the area estimation.

One important aspect to realize when modeling the static energy consumption is that it represents the constant energy consumption for the whole the runtime of the program. In the ASAM model, the static energy consumption has been modeled per cycle and needs to be multiplied by the number of cycles that the target application is executed before it can be combined with the dynamic energy consumption into a total energy consumption.

The total dynamic energy consumption (E_{dyn}, equation 5.2) is computed by combining the activity (α) of each component with a normalized energy consumption (E_{norm}) for components of that type. Again, the specific contributions of these components are discussed below in more detail.

5.1.1 Issue-slots and operations

The area of each issue-slot is composed of the accumulated areas of its function-units and a small miscellaneous part, which takes a.o. a part of the overhead from the instruction decoding and distribution of the decoded control signals to the function-units of each issue-slot into account. The area of each function-unit type is considered a fixed number for function-units operating on a given data width, whereas the miscellaneous area overhead component is considered as a function of the number of operations contained within the issue-slot.

Dynamic energy consumption within the issue-slots depends directly on the executed operations. A weighted sum, combining the function-unit activity with

$$\begin{aligned}
E_{dyn} = & \sum_{OP} \alpha_{OP} E_{norm_{OP}} \\
& + \sum_{RF} \left(\sum_{reads} \alpha_{rf_{read}} E_{norm_{rf_{read}}} + \sum_{writes} \alpha_{rf_{write}} E_{norm_{rf_{write}}} \right) \\
& + \sum_{MEM} \left(\sum_{reads} \alpha_{mem_{read}} E_{norm_{mem_{read}}} + \sum_{writes} \alpha_{mem_{write}} E_{norm_{mem_{write}}} \right) \\
& + E_{interconnect} \\
& + E_{misc}
\end{aligned} \tag{5.2}$$

a normalized dynamic energy cost is used to compute the contribution of each issue-slot to the total dynamic energy consumption.

Normalized area and dynamic energy consumption numbers are available at a function-unit granularity. Obtaining a finer granularity is difficult since multiple operations implemented within the same function unit often share hardware. For example, a function unit which provides multiplication, addition, and multiply-accumulate operations will commonly only contain a single multiplier and a single adder. This makes it very difficult to predict when, and by how much, the area and dynamic energy consumption of a function-unit will change with the removal of one or more of its operations. Two solutions for this problem exist, either a worst-case estimate (using the normalized numbers for the full function-unit) is used, or a set of characterizations for different configurations is provided for each function-unit. The tools used in this dissertation use the former method but can be extended to the latter as part of future improvements to the architecture modeling.

5.1.2 Register files and memory-like interfaces

All register files and components with memory-like interfaces are treated in a similar way within the ASAM area and energy models. Small local memories may be implemented using register-based structures while large register files may structurally resemble a small memory. Area is consumed based on the internal storage requirements and control hardware, while energy is directly correlated with the read/write access activity counts. Components with memory-like interfaces are ASIP parts such as local memories, FIFO interfaces, and master interfaces for external memories.

Register file area is modeled for each register file separately. Register files are characterized through their size (both data-width and number of entries), and type (the number of read and write ports). Both read and write accesses are counted separately for each register file. This information, combined with the read and write cost for a register file of the given size and type allows us to compute the total energy spent on register file accesses.

The cost for components with memory-like interfaces are counted similarly to register files, but usually use specialized models for obtaining their area and access energy cost. Furthermore, many of these components will have a relatively stable configuration during the ASIP instruction-set architecture exploration. For example, the local memory sizes, the external memory interfaces, and the number and configuration of FIFO interfaces are already determined during the construction of the initial architecture in the second phase of the micro-architecture exploration. As such, their area and energy cost per access are relatively constant for the ASIP architecture variations considered during the instruction-set architecture exploration.

Each VLIW ASIP instance also contains a program memory. This memory is modeled similarly to the local memories but differs in that its accesses are implicit instruction word reads, and that only read accesses are performed. The program memory area and access cost are also much more susceptible to variation compared to the local memories. Different compositions of the ASIP instruction-set will result in different instruction widths, and different levels of instruction-level parallelism will result in different program lengths. During the instruction-set architecture exploration, a new instruction-word width is computed based on the remaining resources and the program memory is resized accordingly. The length of the program memory can also be optimized during the instruction-set architecture exploration. The number of operations listed in the final compiled version of the target application is optionally rounded to the next power-of-two bound to provide room for program updates, and used to determine a suitable minimal program memory length. Both the access cost and program memory area are then based on the new width and length of the program memory.

5.1.3 Interconnect

The interconnect component mainly represents the result select network. This network transports the results of computations from the outputs of the issue-slots to the inputs of the register files. The area and energy consumption of the network are determined by both its complexity and the activity on the network. The activity can be counted similarly to that of the other components in the ASIP architecture but its complexity is much more difficult to model.

Two important factors determine the interconnect complexity. Firstly, the logic part of the network contains multiplexers that route the data from the source issue-slots to one or more register-file write-port destinations through a multi-bus network where each issue-slot usually has a private result bus. Secondly, the wiring of the network also contributes significantly to its area and switching energy costs. The current version of the model uses the number and data-widths of the switches in the routing network to estimate its area and energy per access.

Accurately estimating the wiring cost is quite difficult as it also depends on the layout of the finalized ASIP design. However, we are estimating the ASIP area and energy costs before the layout is done. As such, the interconnect model currently ignores the layout effects and estimates the area and energy costs solely based on the complexity of the network.

5.1.4 Miscellaneous

Besides the above listed ASIP architecture components, there are several other components which are more difficult to categorize into a single of the above categories. These components usually provide the underlying infrastructure of the ASIP and are grouped into a miscellaneous category. Examples of such components are the distribution of clock signals, parts of the instruction fetching and decoding processes that are not yet counted as part of the issue-slots or program memory, and the overall status and control logic for the ASIP. These components either have a fixed cost (e.g. status and control logic) or have a cost that is proportional to the overall ASIP area such as the clock distribution network.

5.1.5 Model calibration

The ASIP architecture model relies strongly on the provided area and energy cost numbers for each of the used ASIP components. Two methods exist for obtaining such metrics. The most general approach is to characterize the ASIP architecture building-block library into a set of generic area and energy cost numbers for a specific implementation technology and clock-frequency requirement. Using this approach, it is possible to construct a generic library of components with their associated costs and to provide a fast instruction-set architecture exploration. However, this approach fails when non-standard blocks are included in an ASIP design (e.g. a new function unit with custom operation patterns). Adding non-standard blocks to the library requires a full hardware synthesis and characterization of the new block. This characterization is performed manually using external hardware synthesis tools and is currently not integrated into the ASAM architecture exploration framework.

The second approach is more robust; it involves retraining the model based on a fully constructed version of the initial ASIP architecture prototype as produced after the application parallelization step (cf. Figure 3.1) of the micro-level architecture exploration. The initial architecture is synthesized using Cadence's RTL-Compiler and the area cost numbers are extracted directly from the synthesis reports. Synopsys Primetime-PX is then used to simulate (a representative part of) the target application on the synthesized platform to obtain the energy cost numbers for each of the ASIP architecture components. Such a simulation is extremely time consuming though and significantly increases the overall ASIP instruction-set architecture synthesis time. The advantage of this method is that it can also be applied after instruction-set extensions have been added to the initial processor architecture, thus automating the incorporation of these custom components into the energy and area models. A full RTL description of the blocks functionality will still need to be provided, but it will now be synthesized and analyzed as part of the overall model calibration which now automatically provides the integration of the custom operation model.

The current ASAM ASIP architecture cost model also provides options for estimating area and energy cost of architectures after changing the data-width of computations. These extrapolations are implemented as linear approximations and have a relatively large error margin. Without changes in the data-width, the ASAM ASIP architecture model has an error margin of approximately $\pm10\%$. This error margin can easily double when changes to the data-width are considered.

Several improvements of the models are still possible which can further reduce the error margin of the model to those reported in the related work. One key opportunity is the incorporation of better models for the local memories, e.g. by using specialized memory modeling tools such as CACTI [58], or by directly using models provided by a memory IP block vendor. The research presented in this dissertation does not explore the data-width of the computations as these are decided during the second phase of the micro-level architecture exploration before the initial prototype ASIP architecture is constructed. This allows us to train the area and energy cost models for an ASIP with a fixed data-width which reduces the error margin. As such, the ASAM ASIP area and energy cost model provides a sufficient accuracy for demonstrating the advantage of our automated instruction-set exploration methods. However, it is important to realize that the trend of the model versus the synthesised reality (answering if architecture variant A is preferable over B) is more important for the exploration of ASIP architectures than the absolute accuracy of the model (answering the specific energy and area requirements of each architecture variant). The absolute accuracy only becomes important when hard area or energy constraints are considered. Further improvements to the ASIP area and energy cost models, while useful, are therefore beyond the scope of this work.

5.2 Activity estimation

As stated above, the estimation of dynamic energy consumption relies heavily on activity counts for each of the ASIP building blocks. In that, our energy model is quite similar to the approaches used in the previous works [8, 44, 45, 72, 88]. Our energy model distinguishes itself by the way that these activity counts are collected. So far, we have observed two methods for obtaining activity counts in previous research works; trace-based and profile-based. In this section, we extend these methods by introducing our improved profile-based energy estimation.

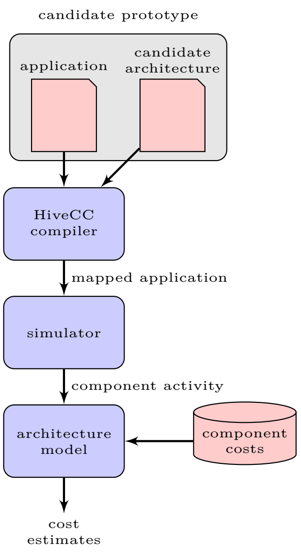


Figure 5.1: *Trace-based energy estimation*

5.2.1 Trace-based energy estimation

Traditionally, a simulation of the target application software on the proposed processor hardware is used to obtain the activity counts for the key components of the processor. When using a gate-level simulation these key components can be single gates and/or wires. However, when using a cycle-accurate instruction-set simulator the granularity of these key components increases to operations, function units, and register files.

Figure 5.1 illustrates the energy estimation process using the traditional simulation-based method. Using a re-targetable compiler (in our case the SiliconHive HiveCC compiler), the application gets compiled for the proposed processor architecture which results in the mapped application. This mapped application is then fed into a cycle accurate simulation of the proposed processor architecture, which in turn results in an execution trace. This execution trace is a list of all activities for each component of the processor architecture. This allows us to then compute the total energy consumption as the sum of components activities weighted by their individual cost. During the model calibration process, a gate-level simulation of a test program which triggers each component with several input patterns is used to determine the component costs.

5.2.2 Profile-based energy estimation

One of the major limitations of the trace-based energy estimation is a large trace size when a large enough representative set of inputs is used for the target application. In most cases, this results both in a slowdown of the simulator due to the large amount of data (hundreds of megabytes when simulating a larger application) that needs to be written into the trace file, and a significant amount of time (tens of minutes) in extracting the aggregate counts from the acquired data. Some of these inefficiencies can be resolved by storing the application trace in an efficient database format which allows for an easy extraction of the aggregate numbers, such as is done in the TCE-project [52, 67]. While this does alleviate the problem somewhat, it still does require the gathering of the entire execution trace.

A more efficient method seems to be to collect the aggregates directly. For example, both the MOVE project [16] and the PICO project [42] use the *application profile* which contains only the basic block execution counts, and combines this with the scheduled assembler code to estimate the execution time of the target application while reducing the need for a full cycle-accurate simulation into a, much simpler, structural simulation of the code. Collecting the application profile directly from the simulation significantly lowers the storage requirement and completely removes the time required for processing the application trace.

A similar method can be used to obtain aggregate usage counts other than the total instruction count. Combining the application profile with the information in the scheduled assembly code allows us to compute the exact activity counts for all key resources in the processor (i.e. operations, register files, and memories). While applying such a technique may run into inaccuracies when considering processor architectures supporting out-of-order execution, this does not happen in our case. Exact activity counts can be obtained for our target architecture as it performs such optimizations at compile-time.

Figure 5.2 illustrates the process of the profile-based energy estimation method. As can be seen from the figure, the activity estimation step now computes the key component activities based on the scheduled assembly code (obtained from the HiveCC compiler) and the execution profile of the application. The advantage of this technique is that it completely removes the application trace from the equation and only uses the aggregate basic block access counts from the application profile. This has both the advantage of requiring much less data to be communicated between the simulation and energy estimation frameworks, but also allows for a much less detailed simulation. However, this advantage in evaluation speed comes at a cost of a lower accuracy. By directly collecting the aggregate activity counts we no longer have access to the individual input patterns applied to each component. Profile-based energy estimation can therefore only be used in combination with an operation-level energy estimation framework which does not take the change in signal values between consecutive inputs into account. As a result, the accuracy of the usable models [8, 52, 72, 88] can be a few percent lower than otherwise feasible [44, 45]. So far, this lower energy estimation accuracy has been tolerated [8, 52, 72, 88], as it enables the exploration of much larger design spaces without significantly increasing the exploration time.

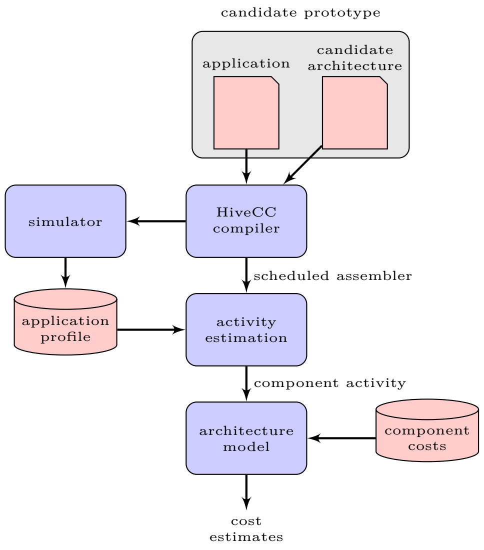


Figure 5.2: *Profile-based energy estimation*

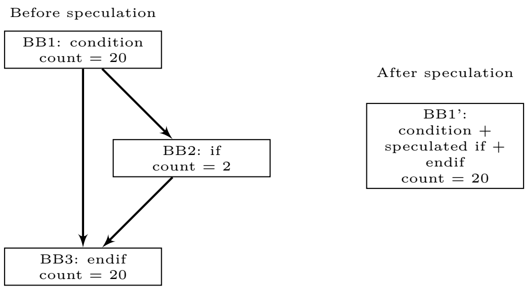
5.2.3 Improved profile-based energy estimation

Both methods presented above focus on energy estimation for single architecture evaluations and will perform a simulation of the program running on the proposed architecture before producing an estimation result. A further improvement of the evaluation time can be realized when we account for the fact that there is only a limited variation of the application profile during an iterative instruction-set architecture exploration. As such, we may be able to re-use the profile information from previous candidate architecture evaluations and remove the need for re-running the simulation entirely. In some cases, however, a change in the proposed processor architecture can trigger a different combination of software optimizations and scheduling decisions which results in a profile change. Such code transformations need to be detected properly in order to be able to determine when new profile information needs to be obtained.

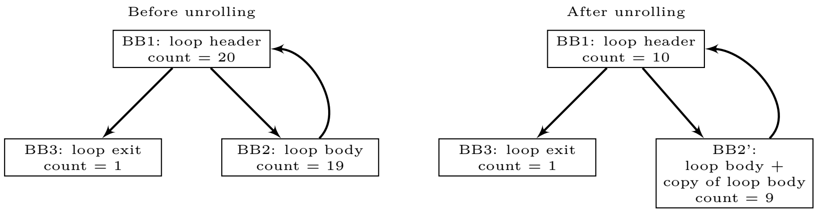
Handling of the code transformations

The estimation of the component activity is trivial when the target compiler does not apply control-flow transformations which influence the application's profile.

In general, we can only reuse information from the application profile when the activity estimator can recognize the transformation from the changed control-flow graph of the application. For example, transformations like *if conversion* and *speculation* (cf. figure 5.3a) can move an operation from a basic block in the program to its parent. In some cases, such a move causes the original basic block to become empty, which results in the basic block being removed from the control-flow graph. Such a transformation can easily be recognized from the changed control-flow graph and the profile can be adjusted accordingly. However, other transformations are more difficult to recognize. For example, *loop unrolling* (cf. figure 5.3b) duplicates a loop body and adjusts the iteration count of the corresponding loop to compensate for the duplication. Both the original and the unrolled loop have the same control-flow graph shape. This makes it impossible to recognize unrolling without a corresponding annotation from the compiler or an in-depth analysis of the resulting assembly code.


Before speculation
BB1: condition count = 20
BB2: if count = 2
BB3: endif count = 20
After speculation
BB1': condition + speculated if + endif count = 20

(a) if conversion (speculation)


Before unrolling
BB1: loop header count = 20
BB3: loop exit count = 1
BB2: loop body count = 19
After unrolling
BB1: loop header count = 10
BB3: loop exit count = 1
BB2': loop body + copy of loop body count = 9

(b) loop unrolling

Figure 5.3: *Code transformations and their effects on the application profile*

The above described problem of taking code transformations into account after they have been applied is related to our current implementation of the activity prediction module outside the compiler. Integrating the activity prediction into the compiler will make it possible to directly handle the code transformations that can currently not directly be recognized from the control-flow graph by an external tool. The overhead of merging the activity estimation with the standard compiler activities should be relatively small in most cases, as most of the optimizing compilers are already profile-driven and already have all the necessary data for the activity estimation.

Current implementation of the method

In principle, our method should produce exactly the same prediction results as the simulation based method. However, some operations (e.g. conditionally executed or guarded operations) may have a different cost depending on the application state. In these cases, the static profile-based estimation needs to be able to guess which is the correct cost. Our current implementation takes the worst-case cost for these operations as it has no way of recognizing the origin of the operation. Better approximations are possible when the activity prediction is combined with the compilation process. If the compiler remembers the basic block from where a guarded operation originated, it can use the profile count of that block to determine how often the guard is true and how often it is false. This would make it possible to accurately model the application state.

5.2.4 Further improvements

Keeping track of the code transformations that have been applied by the compiler seems a bit artificial when one considers that the compiler can often also use the same profile information to make more informed optimization decisions. It can be expected that a compiler will keep track of the changes it makes to the code structure and tracks their expected effect on the application profile for usage by subsequent optimization decisions. If this is already the case, it should also be feasible to output the updated profile information directly, thus completely removing the need for any re-simulation of the target application.

Figure 5.4 illustrates this further improved energy estimation flow. This flow presents two extra opportunities besides the ability of avoiding the need for any re-simulation. The first advantage is that this method can use any profile which accurately reflects the application structure as input. As such, it is possible to compile the target application for a different, much faster, processor architecture and obtain the application profile through native execution on that processor architecture. This method is much faster than any simulation can offer but will still produce exactly the same results as in the case when the target architecture was simulated, as long as the code structure remains unchanged. The second advantage is that the execution count of specific application parts may be computed statically or may be estimated by the compiler. A profile composed of such estimated or analytically derived information can also be used. As such, the information obtained through this further improved method for energy estimation may even open up opportunities for energy aware optimizations by the compiler itself.

5.3 Initial experiments

To demonstrate the value and benefits of the improved profile-based energy estimation method on a real-life application, we selected the Pan-Tompkins QRS detection algorithm [63] for an electro-cardiogram (ECG) monitor, performed the architecture exploration for this application, and mapped this application onto

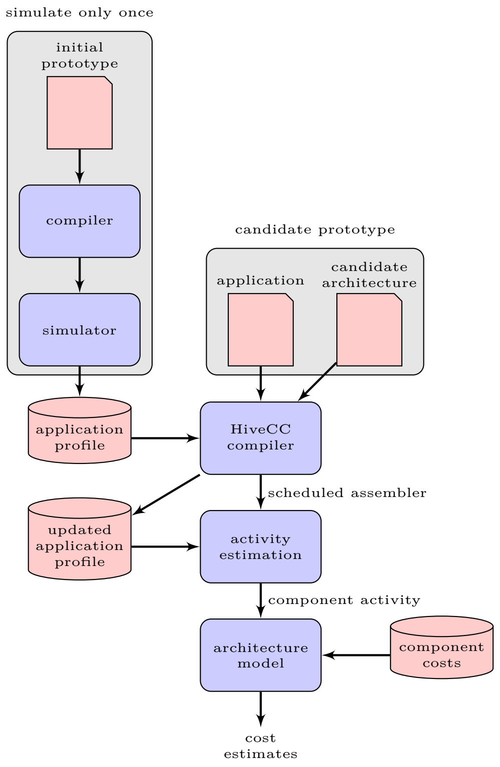


Figure 5.4: *Improved profile-based energy estimation*

the selected architecture. For this application, both the power consumption and real-time behaviour of the created architecture are critical.

In our experiments, we have compared the quality and speed of our ASIP architecture exploration when using each of the three energy estimation techniques: trace-based, profile-based, and our improved profile-based. For this purpose, we have implemented all three cost estimation techniques into our automated instruction-set architecture exploration framework and compared the resulting exploration times. Figure 5.5 shows how the exploration time increases as dependent on the number of considered design points for each cost evaluation method. Two experiments, both using the trace-based cost estimation method for evaluation, took more than one day. The exploration time for these experiments has been extrapolated based on a linear regression of the exploration time as a function of the number of considered design points and was predicted to be approximately 35 hours (for 347 design points) and 85 hours (for 835 design points) respectively. Figure 5.5 clearly shows the advantage of our fast cost-estimation technique when it comes to the total exploration time of larger design spaces.

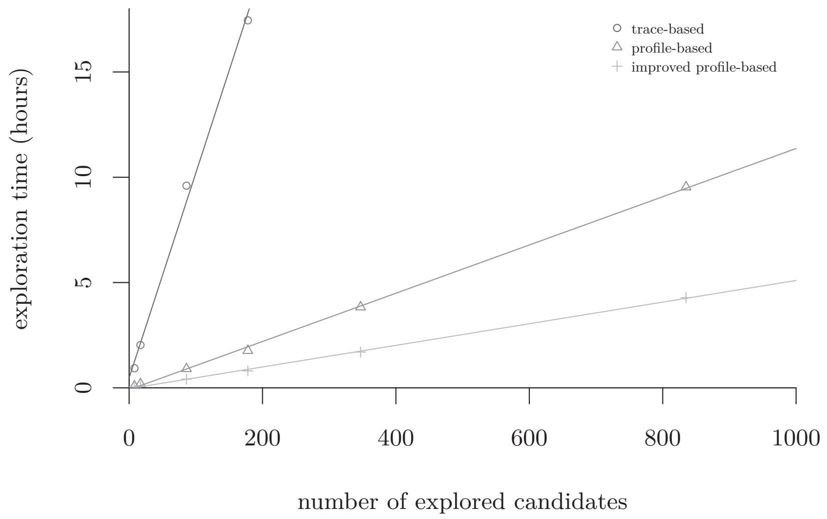


Figure 5.5: *The total exploration time plotted for all three different cost evaluation techniques with added regression lines*

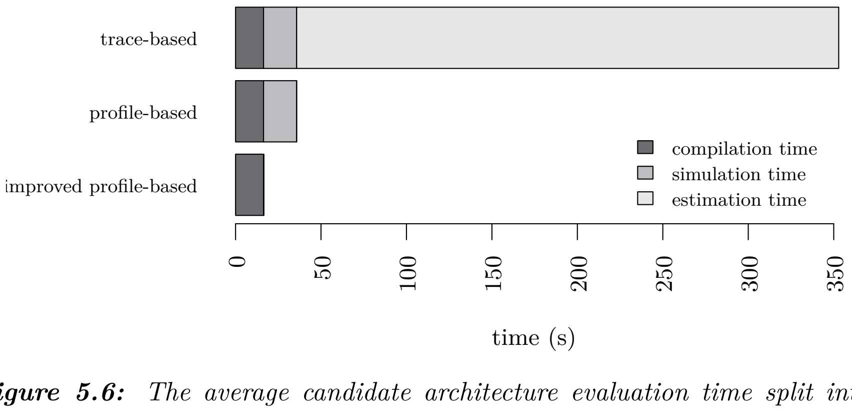


Figure 5.6: *The average candidate architecture evaluation time split into its major components for the three estimation techniques during an exploration considering 178 candidates*

Figure 5.6 presents the distribution of the candidate architecture evaluation time in its main components; compilation, simulation, and evaluation. From this figure we find that the trace-based estimation method indeed offers a unnecessarily slow estimation. Even for the used example trace, processing 10000 samples with the Pan-Tompkins QRS detection algorithm (taking 50 seconds in real-time), the parsing of the application trace of 68 megabyte and computation of the aggregate energy estimates takes over 5 minutes. Switching to the profile-based estimation method reduces the estimation time to a negligible 5 milliseconds and reduces the average evaluation time to 35.6 seconds. Including then the re-use of the simulation results by our improved profile-based energy estimation further reduces the evaluation time. The selected ECG application implementation provides no opportunities for performing code transformations that result in a changed application profile. As a result, only a single simulation was required for the entire exploration, thus reducing the average simulation time by a factor of 178 (the number of explored candidate architectures). This resulted in a reduction of the average simulation time from 19.3 seconds to 109 milliseconds. The remaining candidate evaluation time, 16.3 seconds, represents time required for the compilation of the target application on the proposed candidate.

The experiments reported here are for a single example application of low to medium complexity for which the application profile does not change during the architecture exploration. Especially the trace based estimation is highly dependant on the application run-time and test data size. We have observed several cases where larger tests would take hours to estimate energy using the trace-based approach while our profile-based approach would still take well below a single second.

5.4 Conclusion

In this chapter, we discussed our efficient ASIP area and energy estimation models and methods. In particular, we proposed and discussed a novel method for the estimation of the energy consumption of VLIW ASIP architectures, and we have experimentally demonstrated its effect on the instruction-set architecture exploration time. Using our method, we can explore many more design points within the same time compared to other approaches, without any loss in accuracy.

Further experiments using the improved profile-based energy estimation are presented in the next chapter, where this technique is used to implement the simulation caching in the BuildMaster framework.

There are three great virtues of a programmer; Lazyness, Impatiance, and Hubris

Larry Wall, “Programming Perl”, 1996

6 Intermediate result caching

During the process of ASIP instruction-set architecture refinement, many similar candidate architectures are considered. The evaluation of each of the considered architecture variations consists of two major components; compilation and simulation (see the previous chapter). In the previous chapter we demonstrated how the reuse of the application profile, consisting of basic block execution counts, helps to avoid unnecessary simulations which results in a significant reduction of the total exploration time.

In this chapter, we provide methods to automatically detect when such reuse is possible and implement a simulation cache based upon this detection. Similarly, we also detect when the proposed changes to the architecture are not expected to result in changes of the compilation result with respect to a previously considered candidate architecture and use this detection ability to implement a compilation cache. Both these caches storing intermediate evaluation results and a supporting VLIW ASIP instruction-set architecture refinement framework are bundled into the BuildMaster framework presented in this chapter. This framework allows us to implement our intermediate result caching techniques independently of the exploration strategies, which greatly simplifies the addition of new strategies to our architecture exploration tools.

This chapter is based on
Jordans, R.; Diken, E.; Jóźwiak, L. and Corporaal, H.: BuildMaster: *Efficient ASIP Architecture Exploration Through Compilation and Simulation Result Caching.* In DDECS 2014.

The techniques presented in this chapter will be used in the experimental research discussed in the next chapter. This chapter mainly investigates their specific benefits, demonstrates the effectiveness of both compilation result caching and simulation result caching, and shows the exploration time improvements due to using the exploration strategies presented in the next chapter.

6.1 The simulation cache

The simulation cache builds upon our efficient cost estimation method and is responsible for automatically recognizing when a previously obtained application profile can be reused in our improved profile-based cost estimation method. It keeps track of basic block execution count changes resulting from key loop transformations (we currently track changes in software pipelining) and uses 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 does 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 currently 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 research work.

The 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 set of loop transformations is found, we use the stored application profile with our improved 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.

6.2 The compilation cache

The BuildMaster framework automatically recognizes when two similar architecture prototypes should result in the same optimized ASIP architecture 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 in the same final design point. Figure 6.1 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 (red color). Let's notice that the compiler did not require FU_2 from issue-slot 1 and FU_1 from issue-slot 3 (cf. figure 6.1a, yellow color). When, later in the exploration, we consider a similar architecture but now also disable FU_2 from issue-slot 1 we expect that the result will be as shown in figure 6.1b and has the same performance metrics as found for the first situation.

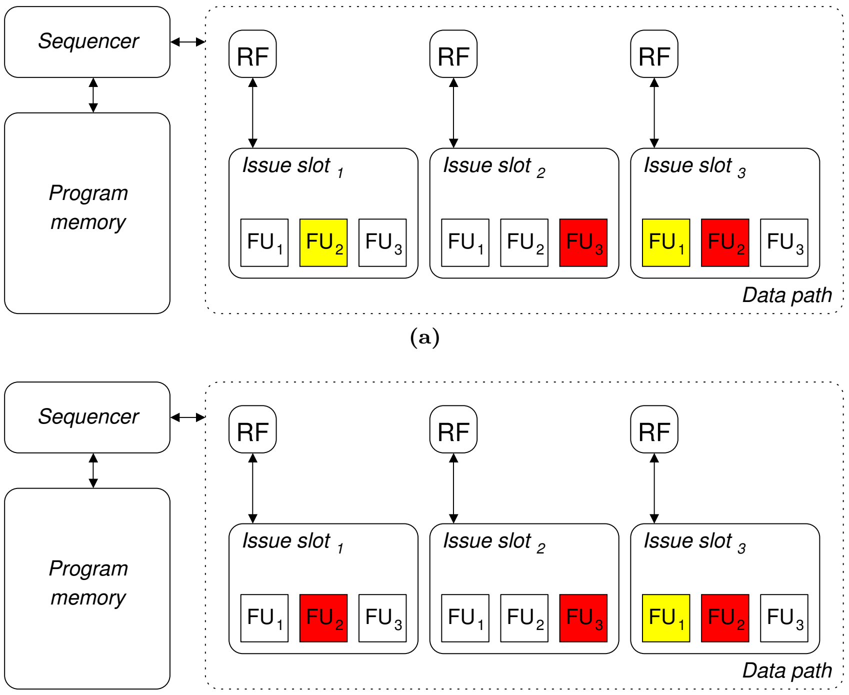


Figure 6.1: *Two equivalent architectures showing resources removed during the selection of candidates (marked red) and unused resources after compilation (marked yellow)*

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 explicitly disabled in the cached prototype, and additionally, explicitly disables a subset of the unused resources of the reduced prototype, is considered a hit of 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.

6.3 Experiments

We have implemented the BuildMaster framework and integrated it into our design space exploration framework [35]. This allowed us to test various cache configurations under different exploration runs. In this section we compare the differences between enabling and disabling of either one or both the compilation and simulation caches. Our experiments show the speedup of the total design space exploration (cf. figure 6.2 and figure 6.3) as well as the cache hit-rates when both caches are enabled (cf. figure 6.4).

The experiments have been performed using six applications from different application domains and having different characteristics, which have been preprocessed using the first two stages of the ASAM micro-level architecture exploration. This pre-processing has prepared them for usage with our exploration framework and provides a fair starting point for the comparison of our exploration strategies and intermediate result caching. Two ECG hart-beat detection applications *ecg-1* and *ecg-2*, two AES encryption and decryption applications, one using a small test sequence (*aes-1*) and one using a large test sequence (*aes-2*), 2D down sampling (*down*), and a low-pass spatial filter (*lpsf*) were selected for the experiments. Each application was explored using both our heuristic exploration strategies *best-match* and *first-match* which will be presented in the next chapter. The experiments used the energy-delay product as a criterion to guide the candidate architecture selection process but other cost-functions should yield similar results with regard to the cache performance.

6.3.1 Exploration time speedup

Figure 6.2 shows the distribution of the obtained speedup due to caching for different architecture exploration runs as a boxplot. The center line in the box illustrates the median value and is surrounded with a box extending to encompass both of the quartiles surrounding the median. The whiskers extending from the box illustrate the first and last quartile of the samples, samples that are over 1.5 times the total length of the box into the first or last quartile are considered outliers and are drawn as separate points (e.g. the topmost sample for the compilation cache).

We observed no cases where the addition of either cache resulted in a slowdown 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

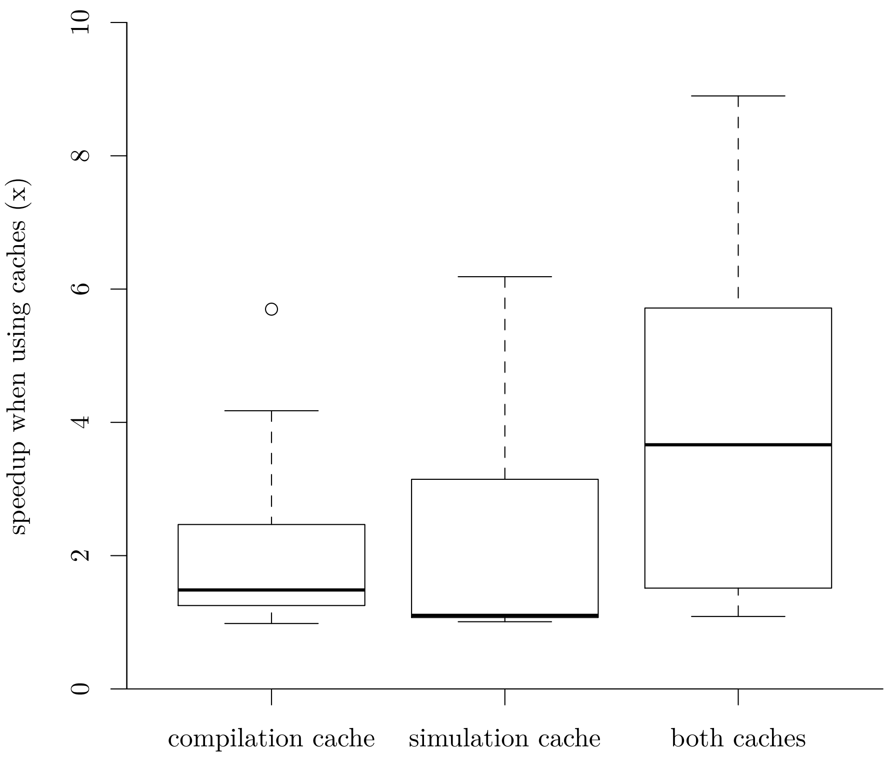
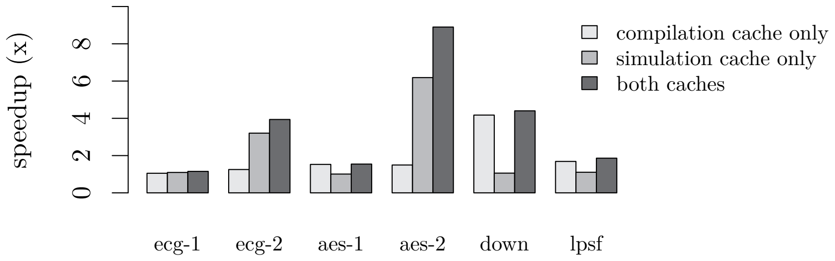
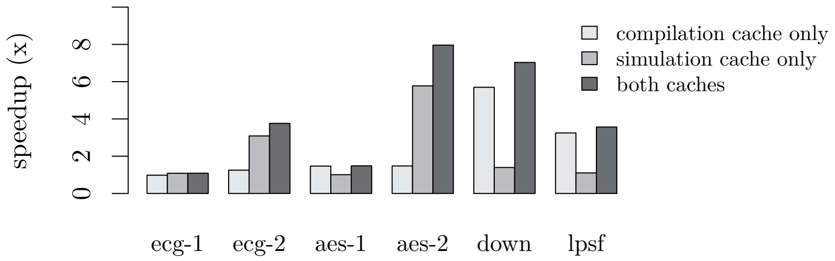


Figure 6.2: *A boxplot showing the exploration-time speedup ranges using different caching strategies*

from the simulation cache. Figure 6.3 provides a more detailed view of the architecture exploration speedup for two different strategies *best-match search* and *first-match search* that will be explained in the next chapter. The AES encryption applications (*aes-1* and *aes-2*) demonstrate the impact of the size of the input data. Traditionally it takes 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. Application *aes-2* shows us (under both exploration strategies) that a larger architecture is found to be cost efficient for the AES encryption when more realistic input data is used. Applications with a relatively small input size, and especially those with a large final architecture (such as *down* and *lpsf*) tend to have more benefit from the compilation cache. This can be explained through the larger size of the final architectures (5 and 7 issue-width VLIW processors respectively) with



(a) best-match search



(b) first-match search

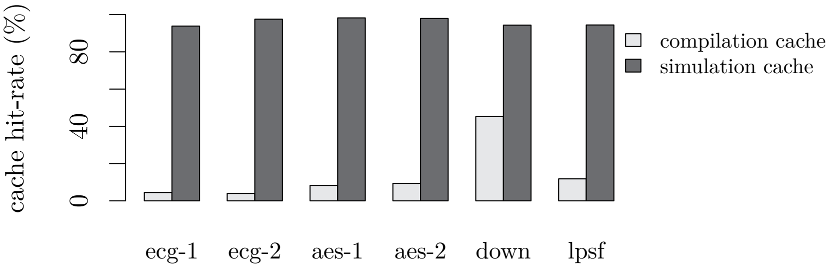
Figure 6.3: *Exploration-time speedup ranges using different caching strategies*

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 likely to more often trigger hits in the compilation cache when a large architecture is considered. Furthermore, applications *down* and *lpsf* 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.

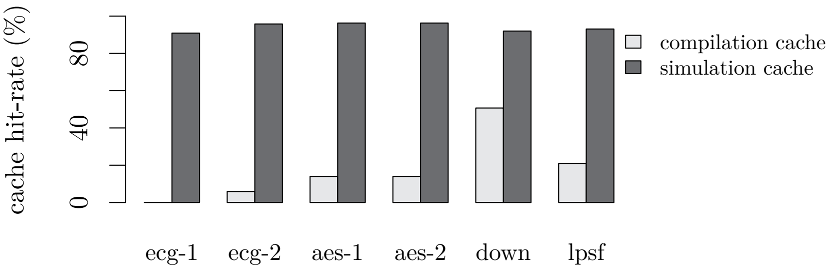
The influence of the input data size makes it difficult to precisely compare the proposed caching methods to the traditional exploration without caching. It is clear that a lot can be gained and that this method allows for an efficient usage of substantially larger input datasets. This last feature helps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

6.3.2 Cache hit-rates

The hit-rates of both caches may give us a better insight into the actual benefits of the caching. Figure 6.4 shows the hit-rates observed in our experiments for



(a) best-match search



(b) first-match search

Figure 6.4: *Observed cache hit-rates for two different architecture exploration strategies*

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 directly translated 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 6.4 we can see, for example, that application *down* 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 additional cause of this speedup. It is discussed next.

6.3.3 Caching induced exploration path divergence

Both the simulation cache and the compilation cache try to detect when a previously obtained evaluation result can be reused instead of (re-)computed. However, small variations can occasionally occur between the cached result and the actual result that would be obtained through compilation and/or simulation. For exam-

Table 6.1: *Detailed experimental results showing caching induced exploration path divergence on the* down *and* lpsf *applications 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

ple, the simulation cache may not properly recognize a loop transformation that was applied, or the compiler may produce a slightly different code when presented with a reduced set of resources (even though all resources that were previously in use are still available). These slight variations between the cached results and results from the compiler can result in variations in the exploration choices, we call this effect caching induced exploration-path divergence.

For our experiments, 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 6.1 gives two example applications to illustrate these effects. 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 whether 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 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 *down*) 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 *down*) the optimization to blame was loop peeling, which was performed as part of 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 *ecg-1*) 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 *lpsf*), 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 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.

6.4 Conclusion

In this chapter 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 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.

The most exciting phrase to hear in science, the one that heralds new discoveries, is not "Eureka" but "That's funny..."

Isaac Asimov, 1920–1992

7 Automated design space exploration

After completing both the construction of an initial ASIP architecture and the calibration of the area and energy models, it is time to optimize the initial design which satisfies the temporal performance constraints, in order to produce a more compact and energy efficient final design. In the ASAM project, this final optimization of the design is performed during the third phase of the micro-level architecture exploration, i.e. instruction-set architecture synthesis.

Design space exploration is an iterative evolutionary process by its nature. One or more candidate designs are proposed and evaluated, and based on the evaluation results new (hopefully better) candidates are proposed. The evaluation of candidates can be completely analytical (based on mathematical quality models as used in the model-based design approaches), but may also require a partial or complete construction and simulation of the proposed design. Different exploration strategies may be selected depending on factors such as the complexity of the design problem, the time required for the evaluation of individual candidates, and the time available to perform the exploration. Design space exploration has been used by many previous works and there exist many different exploration strategies [37,39,40] ranging from simple heuristics to highly complex algorithms. Some of these methods are quite generic (such as genetic algorithms) whereas

This chapter is based on:
Jordans, R.; Corvino, R.; Jóźwiak, L. and Corporaal, H.: *Instruction-set Architecture Exploration Strategies for Deeply Clustered VLIW ASIPs.* In ECyPS 2013.
Jordans, R.; Jóźwiak, L. and Corporaal, H.: *Instruction-set Architecture Exploration of VLIW ASIPs Using a Genetic Algorithm.* In MECO 2014.

other methods may be highly specialized (such as custom tuned heuristics) and combinations utilizing concepts from multiple techniques can be encountered as well.

This chapter introduces the design space exploration algorithms that are used in processor and instruction-set architecture synthesis, and compares them for both the required exploration time and the quality of the produced designs. We first provide an overview of the exploration goals and strategy and introduce the possible exploration algorithms in Section 7.1. Section 7.2 then presents our heuristic exploration methods, followed by Section 7.3 which presents the implementation of a genetic algorithm that was used to verify the quality of our heuristics. Section 7.4 then presents the results of our experiments using the selected exploration methods.

7.1 Exploration method

One of the first aspects to decide when considering to automate a design exploration problem is the formulation of the design goal. Is there a single goal and do we want to focus on optimizing this single aspect of the design (e.g. energy consumption) with constraints on the others? Or are there trade-offs which have to be considered among the different aspects, do we want to limit the impact of the design aspects that are less of a problem? For example, reducing the energy consumption can easily result in an execution slow-down of the target application. Deciding how much of a slow-down can be tolerated can sometimes be modeled as a hard constraint on the execution time, but often a less restrictive soft constraint or minimization objective can help in getting an even better energy consumption. In general, such a multi-objective exploration can be performed in two variations. Either a set of Pareto-optimal (non-dominated)[1] solutions is produced as the result of the exploration, and the multi-objective score aggregation and trade-off decision making is left to the human designer, or and automated algorithmic aggregation and trade-off decision making is performed using one of several possible aggregation methods. For instance, a cost function can be defined and optimized which combines the key design attributes (energy, area, delay) into a single aggregate value.

Previous works [1, 15, 31, 32, 42] mostly focused on obtaining a set of Pareto-optimal designs, and are considering either the area-delay or energy-delay trade-offs. In the ASAM project however, such a Pareto-optimal set of design candidates is already determined as part of the application restructuring and coarse

[1]A solution dominates another solution when it scores the same or better in all considered quality metrics and scores better in at least one of the metrics. A non-dominated solution has the characteristic that no one of the other possible solutions dominates it. In other words, a non-dominated solution is either equivalent to another non-dominated solution or better on one or more of the considered quality metrics than any other of the possible solutions (thus also non-dominated solutions). The full set of non-dominating solutions therefore provides insight in the trade-offs among various quality aspects.

architecture exploration phase [38]. From the results of this second phase of the micro-architecture exploration the initial prototype is selected for further refinement using the algorithms presented in this chapter. During the architecture refinement phase considered here we focus on refining a single Pareto-optimal solution constructed in the second phase into a better solution. We have to perform here a local search around one selected Pareto-optimal coarse architecture to refine this architecture in a good way. In order to avoid a further increase in the number of ASIP design alternatives presented to the ASAM macro-level architecture exploration we are not focusing on finding Pareto-optimal solution sets, but on further refinement of already Pareto-optimal high-level solutions towards a single aggregate value or *cost function*. For this purpose, our ASIP instruction-set architecture exploration framework offers several different cost functions 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. Any other custom cost functions to accommodate specific design concerns can easily be added into the existing exploration framework. Please refer to appendix A for more details on, and a brief introduction into, the customization of the developed design space exploration tools.

7.1.1 Growing versus shrinking strategies

In essence, there are two approaches when proposing architecture variations as new candidate architectures. Either resources are added to the original architecture, or they are removed from it. Moving a resource from one location (e.g. issue-slot) to another counts as the combination of an addition and a removal. Constructing each of the candidate processor architectures consumes a significant part of the exploration time in a traditional architecture exploration process. Even for relatively basic (though ANSI C compliant) 32-bit RISC processor architectures, the construction of a processor RTL description, simulator, and its support libraries from a high level description can take several minutes. In case any significant number of designs should be explored, avoiding the repeated construction of processors is critical for achieving a fast and effective design space exploration.

In this work we have decided for a shrinking approach, as this enables us to completely overcome the problem of repeated architecture construction. The retargetable HiveCC compiler provides us with the option to (partially) disable specific resources during the compilation process. This makes it possible to construct an oversized initial architecture, and then only virtually ‘remove’ components by disabling them. The resulting impacts on both the area and energy requirements of the thus reduced architecture are taken into consideration by forcing the area and energy models to ignore the removed resources. Currently, our model is capable of modeling the effects of removing function-units, issue-slots, (partial) register files, and (partial) data memories. It also takes the effect of such removals into account when judging the required size of the program memory and the complexity of the interconnect. Using this combination of architecture shrinking and our adaptive modeling allows us to limit the construction of candidate architectures to two architectures. Only the initial architecture and the final, reduced, architecture need to be constructed. This way, we are able to significantly reduce the required exploration time.

Guaranteeing that the initial architecture provides a sufficient temporal performance that satisfies possible related constraints is key to enabling the shrinking approach. Our methods for parallelism estimation, presented in Chapter 4, have therefore been selected on their ability to provide an upper bound on the application's parallelism. This enables us to directly construct an initial processor architecture that has a sufficiently large number of issue-slots and memory interfaces to provide the required performance. The addition of custom operation patterns to the processor should also be performed before the reduction process is started. Currently, we manually add such custom operation patterns and related hardware to the initial prototype and let the shrinking process decide if these operations should be kept or not.

7.1.2 Active versus passive exploration

The most direct implementation of a shrinking strategy is to simply remove all resources that have not been used by the target application. As could already be seen from the example architecture exploration discussed in Chapter 3, this passive approach to instruction-set specialization already results in a much improved design. However, only using the passive shrinking still does leave significant room for further improvement. Some costly operations may still be duplicated across issue-slots if the scheduler is not specifically instructed to try to schedule the executions of these operations onto less function units. A more active approach to the instruction-set architecture exploration, which explicitly attempts to remove some unnecessary function units which are still in use, can solve this problem.

One of the key advantages of the passive shrinking approach is that it can be incorporated completely in the architecture and energy model of the processor architecture. As such, a simple switch in the model can be used to observe the effects of removing the unused function units in detail. The active exploration does not allow this and requires a rescheduling of the application onto the reduced architecture, before the effects of a function unit removal can be observed. During this rescheduling of the application, different optimization and scheduling decisions may result in a change of the program structure. In consequence, a re-simulation of the application may be required if the application structure changes substantially, in order to be able to accurately estimate the energy consumption for the proposed design. Deciding when to perform re-simulations is handled by the simulation cache in our BuildMaster exploration framework which was presented in Chapter 6.

Taking the significant improvement in the evaluation time due to our intermediate result caching techniques into account also opens up the possibility of performing the exploration directly using only an active exploration. However, only using active exploration introduces extra complexity in the exploration algorithm. Where the effect of removing a resource was directly noticeable previously through our passive reduction, this effect may be less clear when only using active exploration as more function units will need to be removed before an architecture is reduced into a representative final candidate. As a result, a significant amount of back-tracking needs to be performed when an exploration needs to be able to reconsider earlier decisions. Separating the exploration into the combination of an active and passive element avoids the added complexity of intensive backtracking in the exploration algorithm.

Performing an active architecture exploration is limited by the components that can be removed from the architecture. Due to the availability of flags for enabling/disabling instructions at the function unit level, we have decided to limit the active component of our explorations to the issue-slot and function unit levels. The exploration of the other resources such as register file sizes and interconnect is left to the processor modeling as a part of the passive architecture reduction. However, flags for controlling the available number of entries in each register file are also present in the SiliconHive tools. These are currently not used during the exploration, the register file size is part of the passive architecture reduction, but may be added as an extra exploration consideration in the future work. The current reasoning for not exploring the register file sizes actively is that we try to prevent any register file spilling from happening in the final architecture. Register file spilling saves (temporarily unused) values from the register file into one of the memories and comes with cycle-count and energy penalties for the resulting application, especially if the spilled value needs to be moved into a memory by a load/store unit that is not directly connected to the register file from which it is spilled.

7.1.3 Exploration algorithms

For the validation of our parallelism estimation methods in Chapter 4, we used a constraint programming approach to find the optimal issue-width while scheduling operations at the LLVM IR abstraction level. While attempting to extend this method to provide a similar reference point for the final instruction-set architecture refinement we ran into several problems. First of all, the LLVM IR is an abstraction of the actual processor instruction-set. We found that it was very difficult to obtain representative exploration results at this abstraction level during the final architecture refinement. However, increasing the detail at which the architecture was modeled also proved difficult as this greatly increased the complexity of the constraint-programming formulation. Evaluating the original abstract formulation already was very time consuming and the time required for solving the more complex formulation increased rapidly. Adding the extra freedom of a variable instruction-set to the constraint programming problem further complicated the problem and resulted in another huge increase of the exploration time, which was already measured in days. For example, finding the minimal set of function-units required for computing the 1D 8-point IDCT already took several hours, without the added complexity of then distributing these function units over a set of issue-slots. Secondly, implementing our own scheduler in the constraint programming formulation led us to obtain different scheduling results than obtained from the HiveCC compiler. This made it difficult to reproduce the obtained instruction-set architecture exploration results on the actually constructed processor architectures. To avoid these problems, we decided to keep the existing SiliconHive scheduler in the exploration loop and base our exploration decisions on the results obtained using this scheduler.

When manually exploring the instruction-set architecture of a processor, an engineer is limited in the analysis extent, quality, and number of design decisions that can be made. Making smart decisions is therefore a key aspect of the engineer's work. A part of the intelligence behind these smart decisions can be codified as a set of heuristics and used to drive an automatic exploration algorithm. Such an approach often results in highly efficient exploration algorithms which can produce reasonable results. However, in many cases, it can be difficult to accurately demonstrate the effectiveness of such 'rule of thumb' heuristics. This makes it difficult to obtain a sufficient degree of trust in the exploration algorithm such that it will be used for the actual processor architecture exploration in an industrial setting. As an alternative, genetic algorithms are commonly used to automatically solve design problems, where the full complex problem description cannot be directly modeled into the solver. Previous research [37, 39, 40] has shown that exploration using genetic algorithms (and derivatives thereof) can consistently deliver good results when the algorithm is given a long enough run-time. The exploration time for problems with huge design spaces can be very long, often making exploration using a genetic algorithm practically infeasible. However, incorporation of our efficient energy exploration method (Chapter 5) and by using our intermediate result caching methods (Chapter 6) allows us to significantly speed-up the process and obtain exploration results within a more reasonable time. This allows us to use a genetic search tool to establish a baseline of known feasible ASIP architecture designs, which makes it possible to judge the quality of the results obtained when using our heuristic search methods.

7.2 Heuristic search

The proposed heuristic based design space exploration closely follows the traditional design process as performed by an expert ASIP designer. Candidate architectures with specific resources removed are proposed and the effects of the removal of one or more resources is considered. If found beneficial, the resources are actually removed from the system and the process is repeated until either the design criteria are met or no further improvement of the current candidate is found to be possible. Again, we build upon the fact that the initially proposed coarse architecture already meets the temporal requirements of the target application and that the focus of the design space exploration lies on refining this architecture towards lower energy consumption and lower area.

Like with the genetic algorithm, we can select to perform this exploration at different granularities within the processor architecture template. We can choose to either explore the removal of complete issue-slots or the removal of separate function-units. These are the architectural parameters that are explored as part of our active refinement. The other architectural components (e.g. register files and interconnect) are reduced passively using our smart area and energy models.

In our early research [35], we found that exploring either the issue-slots or the function-units in a single run has its limitations. Limiting the active exploration to the issue-slot level removes the ability of the exploration to try to combine costly operations executed in multiple issue-slots onto fewer issue-slots. For example, the initial architecture construction uses issue-slots from a library which contain a standardized set of function-units. Thus, architectures constructed with this template library will often have the same function-unit available within different issue-slots. When only actively exploring the issue-slot level it will not be able to notice when two parallel multiplication operations, executed in different issue-slots, can be serialized and scheduled onto a single issue-slot if each of these issue-slots is also used for other operations. As such, active exploration at only the issue-slot level provides only a limited capacity for customization and leads to low specialization of the resulting ASIP architecture design. On the other hand, actively exploring the function-unit composition of a processor architecture results in a much larger exploration space. While exploring at the function-unit level does provide the possibility to overcome the above limitation of exploration at the issue-slot level, we found that it was very difficult to make good quality design decisions during the early stages of the exploration process. One of the key problems of the exploration at the function-unit level is that there is often a high degree of symmetry caused by the replication of function-units into several issue-slots, in the early design choices. During the early exploration stages, it can be very difficult to decide in which of the available issue-slots to keep a given function-unit, especially when only a single function-unit of a specific type is needed. As a result, we found that an exploration at the function-unit level has a high probability of getting stuck on a sub-optimal solution. Backtracking of design choices [32] helps to improve the results for the function-unit level and provides an option for getting the design space exploration out of a locally optimal solution, though at the cost of a further increase in the exploration time.

As an alternative, we propose a two-stage approach for our heuristic design space exploration method. In the first stage we perform a coarse exploration of the issue-slots to find the proper issue-width of a refined architecture. While in the second stage, for the found issue width, we refine the composition of each of the remaining issue-slots through an exploration of their function-units. Our initial experiments [35] showed that this offers the benefits of both having a high degree of possible specialization in the resulting design and having a fairly limited number of design alternatives to reach the conclusion.

Similarly to [32], we provide two exploration strategies, *best match* and *first match*, both based on the same cost metrics. The *best match* exploration considers the removal of each separate component and selects the component that shows the best improvement of the cost metric. For example, when removing an issue-slot from an n issue-slot initial prototype during the first exploration phase, selecting the best match exploration strategy will construct a set of candidate prototypes with $n-1$ issue-slots containing each possible subset of $n-1$ issue-slots from the initial prototype. The *first match* exploration considers different alternative components sequentially and selects the first component that shows an improvement of the cost metric. The first match strategy results in a faster search due to a much smaller number of considered design points. This is not a problem when the choices are symmetric, i.e. there is no quality difference between two choices (e.g. removing one of two equal issue-slots), but may result in suboptimal solutions when the design choices are asymmetric (e.g. when using a mix of different types of issue-slots).

7.3 Genetic algorithm

Inspired by biological evolution, genetic algorithms are a generalized search heuristic that mimics the process of natural selection. Possible solutions to the problem to be explored are encoded in a genome. These solutions are then explored by deriving new generations of solutions through techniques such as inheritance, selection, mutation, and cross-over. In our case, the genome is used to identify different candidate processor architectures. For our exploration we use a bit-vector as a genome, where each bit in the genome represents the presence (or absence) of a specific function unit in the processor, as these are the decisions that we explore in our active exploration.

We have implemented the genetic search algorithm in our architecture exploration framework using the AI::Genetic::Pro library [50]. Usage of such a library simplifies the implementation of some of the complex optimizations possible with genetic algorithm. Only two support functions (*fitness* and *terminate*) and some basic configuration settings need to be provided to customize the genetic algorithm for our purpose. These customizations are presented in the section below.

7.3.1 Genetic algorithm configuration

During the lifetime of a genetic algorithm, a population of candidate solutions is evolved, via several intermediate population generations, into a (set of) final design(s). The size of the population, the number of generations, and the ways one generation leads into others all contribute to the quality of the final result and to the time required to arrive at that result. For our ASIP instruction-set architecture exploration, the values of these parameters have been determined experimentally. However, finding a good combination is a very time consuming process. The configuration presented in this section resulted in a reasonably high success rate for the genetic algorithm during our experiments, but better configurations may be possible.

The size of the population is selected based on the size of the genome through the reasoning that a more complex problem benefits from a larger variation in the (initial) population. 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 without providing substantial further benefits. We initially found that using a population being three times the number of bits in the genome (the number of actively explored resources) 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, when the population is already trending towards a final solution. We therefore shrink the size of the population after each round of evolution by 10% which results in a significantly reduced exploration time.

A new set of candidate architectures is generated after each round of evolution. New genomes inherit possibly beneficial features through combining features of the best genomes from previous generations. 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 locally optimal solutions. We selected a relatively high mutation rate (15%) for our experiments because the exploration space can be very irregular at times, and this high mutation rate greatly reduced the frequency at which the genetic algorithm got stuck in a sub-optimal solution. To make sure that the best solutions do not get lost during the evolution, we also copy the 3 best solutions of the current generation into the next generation.

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

7.3.2 Fitness function

The fitness function evaluates the fitness of the proposed candidate solutions. It uses the BuildMaster framework (Chapter 6) to compile and simulate the target application on the proposed architecture, so that its execution time, as well as, the area and energy costs can be estimated. The BuildMaster framework decides if compilation and/or simulation are required for this specific instance and performs 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.

The cost functions, as used in our exploration framework, result in a lower value for a better design. However, the genetic algorithm implementation requires a fitness function (i.e. higher is better) and does not allow negative fitness values. We therefore translated the cost metrics into corresponding fitness functions by using the reciprocal value of the cost function as a fitness measure.

7.3.3 Terminate function and number of generations

After several generations, the population of candidate architectures converges into a final design. After each generation, a terminate function is called which is used to detect if the solution has stabilized. We detect this by comparing the current best solution with several of the previously found ones. We have reached a stability region when all these solutions are the same. We can conclude that the exploration got stuck (hopefully through finding the best solution) when such a region of stability spans many generations. Our architecture exploration tool allows the user to set this stability region length and provides a default value of 10 generations.

7.3.4 Further optimizations to the genetic algorithm

In order to get at least one design point that is guaranteed to work we insert the initial prototype into the initial population. This ensures that there is at least one working architecture that satisfies the temporal performance constraints in the initial population. However, even with this addition, the exploration times were still very high in our initial experiments. We improved upon this by inserting more known-to-work architectures into the initial population. These architectures were obtained from a quick heuristic exploration of the VLIW issue-width using the first-match search strategy presented in the next section. 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 this also increases the risk of running into a locally optimal solution.

7.4 Experiments

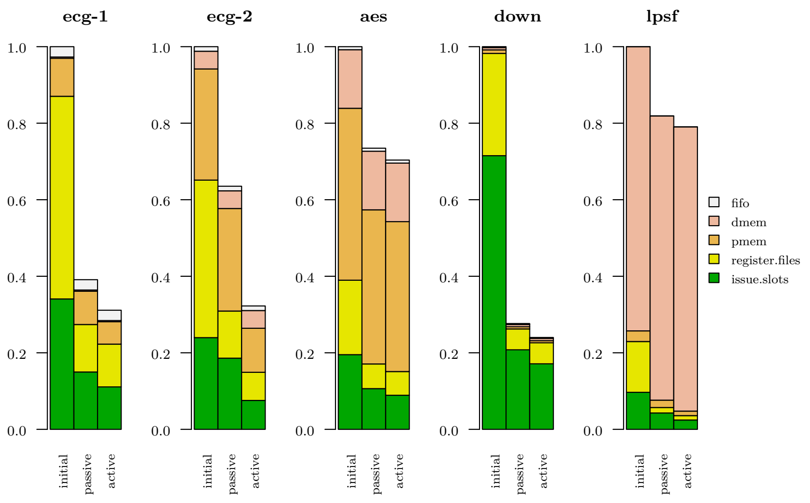
This section presents the experimental results that have been used to verify the proposed instruction-set architecture exploration algorithms. First, we investigate the effectiveness of the architecture exploration separation into a passive and active component. We use our modified area and energy models to implement both our passive shrinking approach which 'removes' all unused elements from the proposed coarse processor architecture design, and our active exploration using one our above discussed design space exploration algorithms. Secondly, we compare the quality of the optimized processor architectures obtained from our various design space exploration methods to investigate their effectiveness. Finally, we conclude this section with a comparison of the required exploration time.

Five test cases were used for this evaluation. These test-cases are two heart-rate detection (ECG) applications (*ecg-1* [20] and *ecg-2* [35, 36, 63]), one AES encryption/decription application (*aes* [36]), and two image processing kernels from the Polybench[2] benchmark suite (*down* performing 2D 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 that 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 also doesn't provide much opportunities for loop optimizations. The image processing kernels *down* and *lpsf* do provide substantial opportunities for loop transformations, vectorization, and software-pipelining. This makes the exploration space much more irregular which results in a much more difficult exploration problem. However, the *lpsf* benchmark requires the complete image data for processing which makes it difficult to tile, as such it has a much larger local data memory than the *down* application which can easily be tiled (as was demonstrated in Chapter 3). This all can be observed from the experimental results. A hand-crafted initial prototype was constructed for each of these test cases to accurately represent the input of the final micro-architecture exploration. These initial prototypes include both the (overdimensioned) processor architecture, as well as, a hand optimized version of the C code that reflects the selected loop transformations for vectorization and which allows for an effective software-pipelining (if applicable).

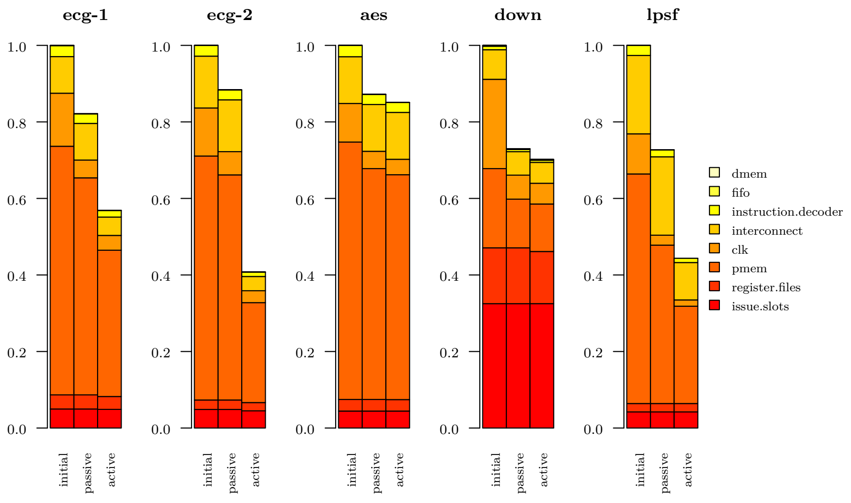
7.4.1 Separation into passive and active exploration

A set of initial processor architectures was constructed using the issue-width estimations presented earlier. These initial processor architectures were constructed from a library of issue-slots based on the result of the application restructuring and

[2]Online: `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`



(a) area



(b) active energy

Figure 7.1: *Area and energy distribution and improvements after passive and active exploration stages*

coarse ASIP synthesis phase of the micro-level architecture exploration. In this first set of experiments we compare the area and energy estimates for each experiment during different stages of the architecture refinement. Figure 7.1 shows the distribution of both the area and active energy consumption as distributed over the proposed processor architecture of each experiment. Each plot shows three bars and both the area and energy scales have been normalized to the estimates of the initial prototype for each experiment (the first bar). The second and third bar of each plot show the estimates after using passive shrinking exploiting the power model and after the complete processor architecture exploration using the *first-match* exploration strategy, respectively.

Figure 7.1 shows the large architecture variation that results from our method of constructing the initial processor architectures. Each application has been optimized for data locality and comes with a processor architecture that provides a set of local memories, register files, and issue-slots that matches the throughput requirements of the application. Streaming interfaces are used for the communication of handshake signals during streaming operation of the application and control the management of the local input and output buffers mapped onto the data memories. The global memory hierarchy has not been taken into account in these processor designs, and is decided in the ASAM project by the macro-level exploration in cooperation with a separate global memory system and communication exploration. In general, this has resulted in relatively small local memories for the explored architectures, with the low-pass spatial filter *lpsf* application being a notable exception, as it requires a relatively large segment of the data while processing. The *aes* application also requires a significant memory area as it uses several lookup tables for its computations, which are stored in their own local memories to allow parallel access. One other notable variation in the initial processor architectures that have been considered is the 2D down sampling application (*down*) which was already partially discussed in Chapter 3. Vectorization was found to be very effective for this application. As a result, the processor area is dominated by its datapath with very large register files and large issue-slots containing the function-units implementing the vector operations.

It can be concluded that both the passive and active exploration strategies can have a significant impact on the area and energy cost of the final processor architecture design. The initial processor architecture uses a set of standardized issue-slots taken from the processor architecture template library. These issue-slots provide a quite rich set of operations, which results in a very rich instruction-set for the initial processor. Combining this rich instruction-set with relatively large register files and a fully connected interconnect results in an architecture that is guaranteed to meet the required temporal performance of the target application. However, it usually also guarantees a significantly oversized processor architecture which needs a very wide program memory to be able to store the encoded instructions. The passive architecture shrinking step can already take care of reducing the number of supported operations and the register file sizes which results in a significant reduction of the processor area and energy cost

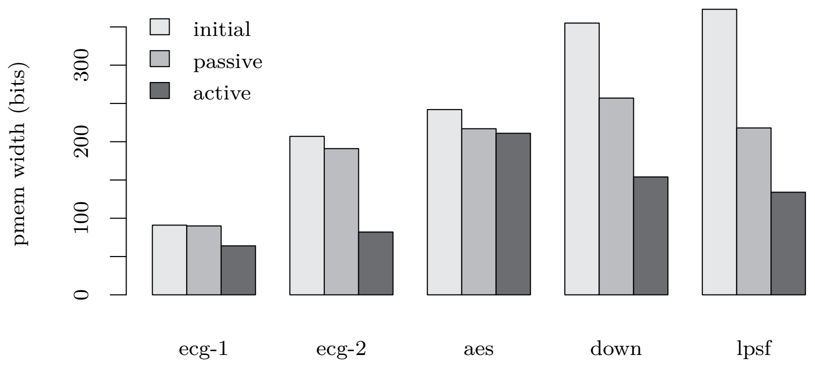


Figure 7.2: *Program memory width after passive and active exploration stages*

without any impact on its temporal performance.

Significant further improvements can be achieved when allowing for slight variations in the temporal performance of the target application. This allows the active exploration to explore the trade-off between the area, energy, and execution time. By recombining the execution of costly operations onto fewer execution resources it becomes possible to further optimize the instruction-set architecture. Among others, the active exploration allows us to significantly decrease the instruction-word length, which often results in a significant decrease of both area and energy cost of the resulting processor architecture as can be seen in Figure 7.2. From this figure, which shows the program memory width reduction achieved using the *first-match* heuristic, we can see that the wider architectures (*down* and *lpsf*) already benefit significantly from passive reduction. This is caused by the high degree in resource duplication across the standardized issue-slots of the initial architecture. The smaller architectures (*ecg-1* and *ecg-2*) provide less obvious redundancy and do not reduce much when only using passive reduction. Also applying the active exploration results in significant further improvements, reducing the required program memory width by up to 47% with respect to the passively reduced case (geomean 36%).

7.4.2 Quality of the active exploration results

In our active instruction-set architecture exploration experiments we have used the energy-delay product as the cost metric. Similar results and observations are expected when using a different cost metric. Final scores are presented as the improvement factor in comparison to the initial architecture after passive shrinking. Higher final scores therefore represent better results achieved by using the active exploration stage.

Figure 7.3 shows the final scores obtained using the two heuristic approaches

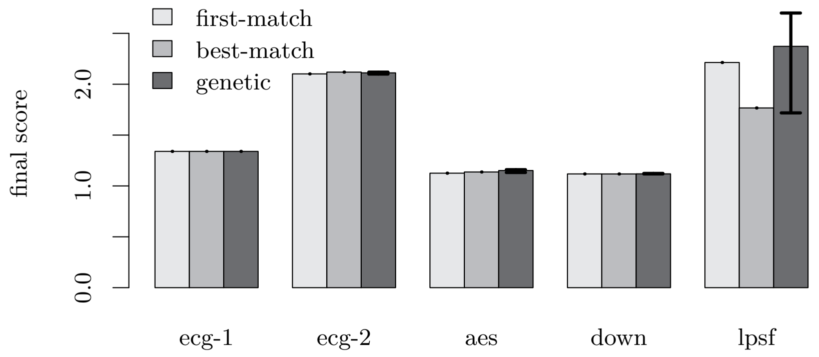


Figure 7.3: *Final optimization score for each benchmark (higher is better)*

(*first-match* and *best-match*), and compares them against the results obtained with the genetic algorithm. The heuristic exploration algorithm is a deterministic process which will produce exactly the same configuration given the same inputs. However, the genetic algorithm is non-deterministic through its probabilistic nature and may not always stabilize on the best solution. Figure 7.3 therefore presents the average final score (higher is better) through height of the bar itself, while the error-bars illustrate the minimum and maximum scores observed over 6 runs of the algorithm.

As can be seen from Figure 7.3, 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 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.

7.4.3 Exploration time

Figure 7.4 compares the time required for exploring the processor instruction-set 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.

From Figure 7.4, 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. This

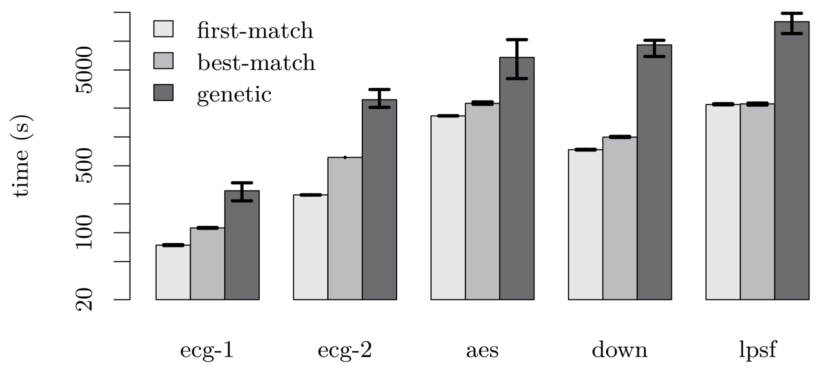


Figure 7.4: *Exploration times using different strategies (on logarithmic time scale)*

optimization allowed us to significantly shorten the exploration time, in several cases by reducing the required number of generations by half, without an impact on the final design score.

Key to making the genetic exploration practically feasible was our usage of caching (presented in the next chapter) of 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%.

From the experiments it follows that using both the 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 several times 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.

7.5 Conclusion

In this chapter we have proposed and discussed a practically feasible instruction-set architecture exploration method for application-specific VLIW instruction-set processors using a genetic algorithm, and compared its result quality and exploration time to two heuristic exploration approaches developed by us. Moreover, we investigated the effect of caching of the intermediate compilation and simulation results on the exploration time. Our experiments show that both the genetic algorithm and the heuristic algorithms produce similar final solutions, but the genetic algorithm is more tolerant to highly irregular design spaces such as for the *lpsf* benchmark which includes several software-pipelined kernels. We also find that usage of our automated intermediate result caching methodology significantly reduced the exploration time of the genetic algorithm.

If one could always choose the right question, then every answer should be as obvious

Steven Erikson, "Malazan Book of the Fallen; Midnight Tides", 2004

8 Conclusions and future work

This chapter concludes the dissertation. It is subdivided in two main sections. The first section revisits the issues identified in the problem statement of Chapter 1 and argue how they have been addressed in this dissertation. The second section of this chapter presents some of the problems that still remain open, as well as, some ideas that remain untested as suggestions for future work.

8.1 Conclusions

In Chapter 1 we presented our problem statement and identified five major problems which needed to be addressed for an effective and efficient automatic processor architecture exploration. The research reported in this dissertation proposed a solution to each of these problems and resulted in a new effective and efficient method and prototype tool for the automatic instruction-set architecture synthesis of VLIW ASIP based hardware/software systems. Specifically, the presented research contributes the following improvements to the ASIP architecture design process:

1. Both the distribution of tasks on a, yet to be constructed, MPSoC platform, as well as, the application restructuring step, require early estimates on the kinds of parallelism available in a particular application part and the expected performance of the application part on the (yet to be constructed) parallel VLIW ASIP. High quality parallelism and execution time estimates help by both improving the selection of the proper task distribution among the processors in a MPSoC, but also aid the construction of initial ASIP

and MPSoC architecture proposals that have a high chance to be closer to the final design.

This dissertation proposed **a new parallelism estimation method** for estimating the application instruction-level parallelism. This method provides information needed for the exploration of the application restructuring, but is also used for determining the appropriate number of issue-slots in the initial architecture. As a result, we are able to construct an initial processor architecture that both meets the performance requirements for the target application, and at the same time, is reasonably close to the final optimized processor design. In Chapter 4 we have shown that our parallelism estimation techniques offer high quality estimations for both the expected instruction-level parallelism and latency of particular application parts. Especially our utilization based parallelism estimation comes on average within 1% of predicting the actually required issue-width of the initial processor. As such, it allows us to produce a high-quality initial processor architecture design before even starting the instruction-set architecture exploration algorithms. From this fact we can conclude that the techniques presented in Chapter 4 efficiently solve the first problem by providing parallelism and performance estimates of sufficient accuracy.

The automated design flow within the ASAM project [38, 56] directly benefits from these techniques, but the presented methods can easily be used as part of other early design space exploration approaches [24] and can support the traditional manual MPSoC system architecture exploration process. As such, the presented early parallelism (issue-width) estimation method provides a significant improvement to the existing (both manual and automated) design flows, as well as, a required initial step for the remainder of the work presented in this dissertation.

2. The current state-of-the-art implementations for the evaluation of proposed candidate architectures commonly depend on an activity trace of (part of) the target application. Both obtaining and processing such a trace can be very time consuming, which limits the effectiveness of the architecture exploration by forcing the use of (less representative) shorter execution traces.

   Chapter 5 presents **a rapid energy consumption estimation methodology** which combines the block execution profile of the target application simulation with the application's scheduled assembly listing. This makes our energy estimation method independent of the number of simulated processor clock cycles and enables the use of larger, more representative, input data sets, allowing for both faster and more realistic evaluation of the candidate designs. Thus, the techniques presented in Chapter 5 effectively address the second problem by drastically changing the evaluation time of candidate architectures from minutes (or even hours) into milliseconds.

Furthermore, this profile-based energy estimation technique also represents an important step towards the further high gains in architecture analysis and evaluation delivered by our intermediate result caching techniques presented in Chapter 6.

The presented energy consumption estimation technique can, in most cases, replace the existing trace-based estimation methods as used by most of the design flows presented in the related work section. Doing so will significantly reduce the evaluation time of candidate designs, but also makes the evaluation time less dependant on the size of the test data used for the evaluation. This allows a designer to use much larger, often more representative, sets of input data which will better demonstrate the actual trade-offs decided during the design process.

3. Efficient implementation of different exploration strategies requires tracking of previously explored design points. When getting closer to the final architecture, many design points will differ only slightly. Recognizing when previously obtained results will be so similar to the current results that they are available for re-use offers an opportunity for a substantial exploration efficiency improvement. This intermediate result tracking is, to a large degree, independent of the exploration strategy.

   Chapter 6 presents **our extensible architecture exploration framework** called BuildMaster, which simplifies the implementation of our architecture refinement exploration strategies. This framework automatically detects when compilation and simulation results obtained for previously considered candidate architectures can be re-used for the evaluation of newly proposed candidate architectures. This intermediate result caching system allows us to avoid on average over 90% of the originally required simulation time by re-using the previously obtained profile information for the energy estimation.

   The BuildMaster framework addresses this third problem by implementing this functionality as part of an exploration framework which greatly simplifies the creation of new processor architecture exploration methods, as many of the bookkeeping tasks are automatically managed by the framework. Similar automated caching approaches may be used as part of automated design space exploration tools targeting different architectures or that are built upon related design flows.

4. Current state-of-the-art methods need to construct, analyze, and evaluate each proposed candidate processor architecture. This is a very time consuming process which significantly impacts the exploration efficiency.

   Chapter 7 introduces **a processor architecture refinement method** which allows us to avoid the (time consuming) construction and simulation of numerous intermediate candidate processor architectures. Through configuring the compiler to disable some selected processor resources during scheduling, combined with our static shrinking technique, we are able to estimate the execution time, as well as, the energy and area costs for the reduced candidate architectures without actually constructing the candidate architectures. In consequence, our approach only needs to construct the initial and final designs, while all intermediate candidate architectures need not be constructed. This approach addresses the fourth problem in an effective way and saves a large amount of the exploration time (often several minutes for each considered architecture).

This style of exploration can also be applied to other processor architecture templates as long as *a)* the compiler provides the opportunity to disable a user defined set of resources during the compilation process, and *b)* the area and energy models for the target architecture are adapted to enable cost estimation for the thus reduced architecture candidate.

5. Refining the instruction-set architecture of an initially proposed prototype ASIP architecture is a process that involves proposing and comparing many different candidate architectures. A smart candidate construction and selection strategy is key to an efficient exploration.

   Chapter 7 also presents **a set of exploration strategies** which effectively refine the processor architecture and a comparison between these strategies with respect to both the quality of the obtained result, as well as, the required exploration time. We show that the proposed exploration heuristics find results of quality comparable to the results found using a genetic algorithm, while requiring an order of magnitude shorter exploration time. These heuristics present a fast method of performing the architecture refinement process and, as such, efficiently address the last problem.

   The instruction-set exploration techniques presented in this final chapter are highly specific for the VLIW architecture template and mainly serve to demonstrate the power of the previously presented estimation/construction/exploration methods. However, similar exploration strategies may be implemented for an alternate processor architecture template when the prerequisite exploration framework, intermediate result caching, rapid area and energy estimation models, and initial architecture construction are provided.

Chapters 3 and 7 demonstrate that combining the presented techniques results in a highly efficient and easily extensible instruction-set architecture exploration methodology. Our experiments showed that our framework is able to explore hundreds of processor architecture variations per hour, while consistently producing compact results that meet the expected performance. As such, the methods presented in this dissertation provide a definite advantage over those used in the current state-of-the-art.

8.2 Future work

More opportunities for improvement of the various ASIP design process steps have been identified during the progress of both the research for, and writing of, this dissertation. Some of these opportunities deserve to be mentioned as possible extensions of the presented work.

- **Automatic detection and insertion of custom operation patterns**: The presented instruction-set architecture exploration is capable of exploring architectures supporting custom operation patterns that have been manually inserted as custom function units. In the ASAM project candidate operation patterns are already automatically detected using [60, 61]. However, the creation of a proper hardware description of the corresponding function unit, and the insertion of this custom function unit into the initial architecture are left to the designer. Furthermore, the HiveCC compiler only has a limited capability for recognizing operation patterns in the C code of the application. Manual insertion of intrinsics into the C code is currently required for complex operation patterns to make sure that they are properly detected during compilation. Using intrinsics forces the compiler to use the selected custom operations which prevents the exploration framework from removing the thus selected operation patterns from the final instruction-set.

- **Compile time data locality optimizations**: In the ASAM project data locality is optimized during the early stages of the micro-level architecture exploration by exploring loop fusion, loop tiling, and loop vectorization. These transformations are then applied as source-to-source transformations of the input C code. Properly predicting the effect of such transformations on the final compiled application requires a strong relation between the application of transformations at the code level and the effects of later transformations by the compiler. Such a relation is difficult to guarantee and often results in either miss-predictions or in a low level of optimization by the compiler (caused by skipping potentially disrupting optimizations). A better approach would be to extend the compiler with the capability of performing such transformations automatically. Adding this capability to the compiler will also aid the manual application development process as it should significantly improve the optimization capabilities of the compiler.

- **Improved energy model accuracy**: Although the energy models used in this dissertation are exploiting the same activity counting based methods as are used by the related work, their actually obtained accuracy still needs to be further analyzed and can possibly be further improved. Currently we rely less on the absolute accuracy of the area and energy models but mostly on their direction (i.e. removing a resource results in a reduction of the predicted values). This is sufficient for basic exploration but more detailed trade-off decisions and hard exploration constraints are difficult to properly implement using such a less accurate model. However, the presented methods for profile based energy estimation apply to any energy estimation method based on component activities. Therefore, similar gains in exploration efficiency can be expected from improved, more accurate, versions of these energy models. Furthermore, depending on the accuracy of a generic model a calibration phase may be added at the begin of the architecture refinement phase which allows for an improved accuracy with regard to the explored architectures. More intermediate calibration steps may be required if the reduced architectures are found to be difficult to model. However, the architecture template for the SiliconHive processors provides a quite well structured view on the final implementation and the expectation is that it can be modeled without requiring any intermediate calibration.

- **Avoid the simulation cache altogether**: Chapter 5 introduced our profile based energy estimation. After recognizing that the application profile is relatively constant across different architecture variations, we presented a framework which automatically recognizes when a previously obtained version of the application profile can be re-used (the simulation cache from Chapter 6). However, as mentioned in Chapter 5, this caching may completely be avoided if we can alter the compiler to output an updated profile of the application. Profile guided optimizations are frequently used when attempting to get optimal performance from an application. Since the compiler is aware of the code transformations it performs, it should also be able to track their impact on the application profile. Enabling the compiler to output an updated application profile completely removes the need for a re-simulation of the application and opens up opportunities for a tighter coupling between the energy model and compilation process which may aid in the development of related energy-aware compilation techniques.

- **Extending the active exploration**: The current architecture exploration tools focus on finding the right combination of function-units and their distribution over issue-slots, and shrink the register file sizes and interconnect through for passive reduction. However, the register file size can also be explored actively using existing compiler flags in the SiliconHive compiler. In some cases, it may be beneficial to allow for small amounts of register file spilling in some non-critical application parts to allow for an even more compact solution through actively exploring the register file sizes. Moreover, adding the interconnect to the active exploration can also be considered although this will currently require added controls in the SiliconHive compiler. Without these added controls the exploration will need to fall-back to the complete construction of candidate architectures which will significantly impact the exploration time.

- **Exploring hardware support for multiple vector lengths**: This work currently only explores architectures providing SIMD operations support for vector operations of a single total bitwidth, but different application parts may require different vectorization levels. This is solved by selecting a common vector length for the hardware implementation. However, extending the processor architecture exploration to enable heterogeneous vectorization can also provide benefits [21–23] when this better matches the parallelism available in the application.

- **Reducing the instruction word length by generating compact ISA**: Although the architecture refinement techniques presented in this dissertation are currently aimed at physically removing functionality from a processor architecture design, they may also be used to create limited processor 'views' for the purpose of code compaction. By only exposing a sub-set of the full instruction-set in a reduced processor mode, the instruction length may be significantly reduced for large parts of the program. Automatically finding good instruction-set sub-sets for parts of the application could aid a processor designer when deciding the available processor views.

- **Energy aware scheduling**: Similarly to the generation of processor views for code compaction, the presented architecture reduction techniques may also aid a related energy aware compilation through fine grain clock-gating of processor resources. By scheduling the code such that some parts of the processor are idle for longer, more consecutive, periods of time may enable more aggressive clock-gating of these idle processor resources, enabling further energy saving opportunities.

Bibliography

[1] S. Aditya and V. Kathail. *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter Algorithmic Synthesis using PICO, pages 53–74. Springer Science, 2008.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *ISCA 1992 – Proceedings of the 19th Annual International Symposium on Computer architecture*, pages 342–351, 1992.

[4] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros. The archc architecture description language and tools. *International Journal of Parallel Programming*, 33(5):453–484, 2005.

[5] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*, pages 244–263. Springer, 2010.

[6] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16. IEEE Computer Society, 2004.

[7] M. Bekooij. *Constraint Driven Operation Assignemnt for Retargetable VLIW Compilers*. PhD thesis, Eindhoven University of Technology, 2004.

[8] L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria, and R. Zafalon. A power modeling and estimation framework for VLIW-based embedded systems. In *Proceedings of the International Workshop on Power And Timing Modeling, Optimization and Simulation PATMOS*, volume 1, pages 2–3, 2001.

[9] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Notices*, 43(6):101–113, 2008.

[10] P. Boulet. Array-ol revisited, multidimensional intensive signal processing specification. *Rapport de Recherche Institut National de Recherche en Informatique et en Automatique*, 6113:1–27, Februari 2007.

[11] V. C. Cabezas and P. Stanley-Marbell. Parallelism and data movement characterization of contemporary application classes. In *SPAA 2011 – Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 95–104, New York, NY, USA, 2011. ACM.

[12] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on,*, volume 1, pages 183–192. IEEE, 1996.

[13] L. Charvat, A. Smrcka, and T. Vojnar. Automatic formal correspondence checking of ISA and RTL microprocessor description. In *Microprocessor Test and Verification (MTV), 2012 13th International Workshop on*, pages 6–12. IEEE, 2012.

[14] A. Chattopadhyay, I. G. Ascheid, and P. Ienne. *Language-driven Exploration and Implementation of Partially Re-configurable ASIPs (rASIPs).* PhD thesis, Lehrstuhl für Integrierte Systeme der Signalverarbeitung, 2008.

[15] H. Corporaal. *Transport Triggered Architectures; Design and Evaluation.* PhD thesis, Technische Universiteit Delft, 1995.

[16] H. Corporaal and J. Hoogerbrugge. Cosynthesis with the MOVE framework. In *Symposium on Modelling, Analysis, and Simulation*, pages 184–189, 1996.

[17] R. Corvino, E. Diken, A. Gamatie, and L. Jóźwiak. Transformation based exploration of data parallel architecture for customizable hardware: A JPEG encoder case study. In *DSD 2012 - 15th Euromicro Conference on Digital System Design*, Cesme, Izmir, Turkey, September 2012.

[18] 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 - 12th International Conference on Embedded Computer Systems*, Samos, Greece, July 2012.

[19] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1st edition, 1994.

[20] E. Diken, R. Jordans, R. Corvino, and L. Jozwiak. Application analysis driven ASIP-based system synthesis for ECG. In *Embedded World Conference*, Germany, February 2012.

[21] E. Diken, R. Jordans, R. Corvino, L. Jóźwiak, H. Corporaal, and F. A. Chies. Construction and exploitation of VLIW ASIPs with heterogeneous vector-widths. *Microprocessors and Microsystems*, 38(8-B):947–959, 2014.

[22] E. Diken, R. Jordans, L. Jóźwiak, and H. Corporaal. Construction and exploitation of VLIW ASIPs with multiple vector-widths. In *MECO 2014 - 3rd Mediterranean Conference on Embedded Computing*, pages 244–247, Budva, Montenegro, June 2014.

[23] E. Diken, M. O'Riordan, R. Jordans, L. Józwiak, H. Corporaal, and D. Moloney. Mixed-length SIMD code generation for VLIW architectures with multiple native vector-widths. In *ASAP 2015 - 26th IEEE International Conference on Application-specific Systems, Architectures and Processors*, Totonto, Canada, July 2015.

[24] J. F. Eusse, L. G. Murillo, C. McGirr, R. Leupers, and G. Ascheid. Application-specific architecture exploration based on processor-agnostic performance estimation. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, pages 84–87. ACM, 2015.

[25] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *European Design and Test Conference, 1995. ED&TC 1995, Proceedings.*, pages 503–507. IEEE, 1995.

[26] FlexASP project. TTA-based co-design environment. Online: `http://tce.cs.tut.fi`.

[27] M. Gillespie. Amdahl's law, Gustafson's trend, and the performance limits of parallel applications. *Online: http://software.intel.com/sites/default/files/m/d/4/1/d/8/Gillespie-0053-AAD_Gustafson-Amdahl_v1__2_.rh.final.pdf*, 2008.

[28] G. Goossens, D. Lanneer, W. Geurts, and J. Van Praet. Design of ASIPs in multi-processor socs using the chess/checkers retargetable tool suite. In *System-on-Chip, 2006. International Symposium on*, pages 1–4. IEEE, 2006.

[29] T. Grosser, A. Groesslinger, and C. Lengauer. Polly: Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04), 2012.

[30] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[31] J. Hoogerbrugge. *Code Generation for Transport Triggered Architectures.* PhD thesis, Technische Universiteit Delft, 1996.

[32] J. Hoogerbrugge and H. Corporaal. Automatic synthesis of transport triggered processors. In *Proceedings of ASCI*, pages 1–10, 1995.

[33] 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*, pages 1–4, Cesme, Izmir, Turkey, September 2012.

[34] R. Jordans, R. Corvino, L. Jóźwiak, and H. Corporaal. Exploring processor parallelism: Estimation methods and optimization strategies. In *DDECS 2013 - 16th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 18–23, Karlovy Vary, Czech Republic, April 2013.

[35] 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*, pages 38–41, Budva, Montenegro, June 2013.

[36] R. Jordans, L. Jóźwiak, and H. Corporaal. Instruction-set architecture exploration of VLIW ASIPs using a genetic algorithm. In *MECO 2014 - 3rd Mediterranean Conference on Embedded Computing*, pages 32–35, June 2014.

[37] L. Jóźwiak. Advanced ai search techniques in modern digital circuit synthesis. *Artificial Intelligence Review*, 20(3-4):269–318, 2003.

[38] L. Jóźwiak, 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*, 37(8):1002–1019, October 2013.

[39] L. Jóźwiak and N. Nedjah. Modern architectures for embedded reconfigurable systems - a survey. *Journal of Circuits, Systems, and Computers*, 18(2):209–254, 2009.

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

[41] K. Karuri, A. Chattopadhyay, X. Chen, D. Kammler, L. Hao, R. Leupers, H. Meyr, and G. Ascheid. A design flow for architecture exploration and implementation of partially reconfigurable processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(10):1281–1294, 2008.

[42] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D. Cronquist, and M. Sivaraman. PICO: automatically designing custom computers. *Computer*, 35(9):39–47, 2002.

[43] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving process networks from Matlab for embedded signal processing architectures. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign*, pages 13–17. ACM, 2000.

[44] F. Klein, G. Araujo, R. Azevedo, R. Leao, and L. C. dos Santos. A multi-model power estimation engine for accuracy optimization. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, pages 280–285. ACM, 2007.

[45] F. Klein, G. Araujo, R. Azevedo, R. Leao, and L. C. Dos Santos. On the limitations of power macromodeling techniques. In *IEEE Computer Society Annual Symposium on VLSI*, pages 395–400, 2007.

[46] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *ACM SIGPLAN Notices*, 23(7):318–328, 1988.

[47] J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *Parallel and Distributed Systems, IEEE Transactions on*, 4(7):812–826, 1993.

[48] J. Leijten, G. Burns, J. Huisken, E. Waterlander, and A. van Wel. Avispa: A massively parallel reconfigurable accelerator. In *System-on-Chip, 2003. Proceedings. International Symposium on*, pages 165–168. IEEE, 2003.

[49] LLVM. Project website. Online: `http://www.llvm.org/`.

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

[51] 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*, 17(1):37–54, February 2008.

[52] J. Mäntyneva. Automated design space exploration of transport-triggered architectures. Master's thesis, Tampere University of Technology, Tampere, Finland, July 2009.

[53] P. Mattson, W. J. Dally, S. Rixner, U. J. Kapasi, and J. D. Owens. Communication scheduling. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 82–92, New York, NY, USA, 2000. ACM.

[54] L. Micconi. *A Probabilistic Approach for the System-Level Design of Multi-ASIP Platforms*. PhD thesis, Technical University of Denmark, 2014.

[55] L. Micconi, D. Gangadharan, P. Pop, and J. Madsen. Multi-ASIP platform synthesis for real-time applications. In *SIES 2013 - 8th IEEE International Symposium on Industrial Embedded Systems*, Porto, Portugal, June 2013.

[56] L. Micconi, J. Madsen, and P. Pop. System-level synthesis of multi-ASIP platforms using an uncertainty model. *Integration, the VLSI Journal*, 2015.

[57] P. Mishra and N. Dutt. *Processor Description Languages*, volume 1. Morgan Kaufmann, 2011.

[58] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, 2009.

[59] A. S. Nery, L. Jóźwiak, M. Lindwer, M. Cocco, N. Nedjah, and F. M. França. Hardware reuse in modern application-specific processors and accelerators. *Microprocessors and Microsystems*, 37(6):684–692, 2013.

[60] A. S. Nery, N. Nedjah, F. M. Franca, L. Jozwiak, and H. Corporaal. Automatic complex instruction identification for efficient application mapping onto ASIPs. In *Circuits and Systems (LASCAS), 2014 IEEE 5th Latin American Symposium on*, pages 1–4. IEEE, 2014.

[61] A. S. Nery, N. Nedjah, F. M. G. Franca, L. Jóźwiak, and H. Corporaal. A framework for automatic custom instruction identification on multi-issue ASIPs. In *Proceedings of the 12th IEEE International Conference on Industrial Informatics*, pages 428–433. IEEE computer society, IEEE, 2014.

[62] Y. Ökmen. SIMD floating point processor and efficient implementation of ray tracing algorithm. Master's thesis, TU Delft, Delft, The Netherlands, October 2011.

[63] J. Pan and W. J. Tompkins. A real-time qrs detection algorithm. *Biomedical Engineering, IEEE Transactions on*, BME-32(3):230–236, 1985.

[64] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Paek, and E. Earlie. Register file power reduction using bypass sensitive compiler. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 27(6):1155, 2008.

[65] P. Paulin and J. Knight. Force-directed scheduling for the behavioral synthesis of asics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(6):661–679, 1989.

[66] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, pages 933–938. ACM, 1999.

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

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

[69] 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*, 25(7):1209–1229, 2006.

[70] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74. ACM, 1994.

[71] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, 21(12):1405–1411, December 1972.

[72] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria. An instruction-level energy model for embedded VLIW architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(9):998–1010, 2002.

[73] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. Architecture implementation using the machine description language LISA. In *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design*, pages 239–244. IEEE, 2002.

[74] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. Pico-npa: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 31(2):127–142, 2002.

[75] D. She, Y. He, and H. Corporaal. Energy efficient special instruction support in an embedded processor with compact ISA. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 131–140. ACM, 2012.

[76] M. Smotherman, S. Krishnamurthy, P. S. Aravind, and D. Hunnicutt. Efficient dag construction and heuristic calculation for instruction scheduling. In *MICRO 1991 – Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 93–102, 1991.

[77] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66, 2010.

[78] Tampere University of Technology, Department of Computer Systems. *TTA-based Co-design Environment v1.9 User Manual*, Januari 2014.

[79] A. Terechko. *Clustered VLIW Architectures: a Quantative Approach.* PhD thesis, Eindhoven University of Technology, 2007.

[80] K. B. Theobald, G. R. Gao, and L. J. Hendren. On the limits of program parallelism and its smoothability. In *MICRO 1992 – Proceedings of the 25th Annual Symposium on Microarchitecture*, pages 10–19, 1992.

[81] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers*, 19(10):889–895, October 1970.

[82] A. Tridgell, J. Rosdahl, et al. ccache–a fast c/c++ compiler cache. Online: `http://ccache.samba.org`.

[83] E. van Dalen, S. G. Pestana, and A. van Wel. An integrated, low-power processor for image signal processing. In *Multimedia, 2006. ISM'06. Eighth IEEE International Symposium on*, pages 501–508. IEEE, 2006.

[84] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, 2012.

[85] D. W. Wall. Limits of instruction-level parallelism. In *ASPLOS 1991 – Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, 1991.

[86] C. Wolinski and K. Kuchcinski. Automatic selection of application-specific reconfigurable processor extensions. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2008.* IEEE, 2008.

[87] H. Yang. Computer aided design of cluster-based ASIPs. Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, August 2013.

[88] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proceedings of the 37th Annual Design Automation Conference*, pages 340–345. ACM, 2000.

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

Douglas Adams, “Mostly Harmless”, 1992

A
ASIP construction and exploration tools

Several tools were created as part of the design-space exploration research presented within this dissertation. This appendix describes two important tool-sets which can be used to both reproduce and extend the presented experimental results.

The first part of this appendix presents the TimGen tool. This program is used to translate a high-level XML description into its TIM equivalent. This tool connects the high-level design obtained from the second micro-level architecture exploration phase with the final architecture refinement phase.

The second part of this appendix presents the design-exploration tools built upon our BuildMaster framework. It describes the currently available exploration strategies and discusses the implementation of a new cost function. This second part also demonstrates the instruction-set architecture exploration on one of the applications from our benchmark set.

A.1 Processor architecture construction

The purpose of the TimGen tool is to simplify the construction of the initial processor architecture for the ASIP architecture refinement phase. It transforms a high-level XML description of a processor architecture into its TIM equivalent. Version 3.0 (the current version) is based upon the original prototype XML-to-TIM tool developed by Hubiao Yang [87]. Version 3.0 introduces a new XML description format and is not backwards compatible with the earlier tool.

A.1.1 Features

- New XML format allows custom issue-slots
- Supports the addition of new issue-slots to the library without requiring any changes to the tool
- Support for flexible connectivity
- Support for multiple register files per cluster

A.1.2 Installation and usage

The `timgen3` tool is available on the server. Adding the following line to your `.bashrc` will enable it on your account.

```
alias timgen3='java -jar /home/tools/tue/TimGen3.0.jar'
```

Listing A.1: *Installation of the TimGen tool*

The tool is used from command-line and takes the input XML file as argument.

```
timgen3 sample.xml
```

Listing A.2: *Running TimGen*

A.1.3 XML input specification

Basics

The input of the TimGen tool is a file in XML format, it starts with a file type definition and XML comments are allowed.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Optional comment -->
```

Listing A.3: *XLM input snippet, document preamble*

The entire architecture is wrapped in a `<timplatform:ASIP>` node. This node has an argument which provides the name of the processor. This processor name is also used when naming the output file, the generated output file will be *processorname*.tim.

Constant values are defined next. The names of constants are used directly in the generated TIM code and must be valid constant names for the TIM language.

```
<timplatform:ASIP name="sample">
```

Listing A.4: *XLM input snippet, declaring platform name*

```
<constant name="value"/>
```

Listing A.5: *XLM input snippet, declaring constants*

Issue-slots are defined based on their name and their related register files and memories. The `sequencer` is recognized by the presence of a program memory. Each processor must have exactly one sequencer. The interface of an issue-slot is described by the `port` list. Ports that are within a register file `rf` node are input ports and ports outside the `rf` nodes are the output ports.

Issue-slots can be grouped into clusters. All elements within the same cluster are fully connected and ports may be marked as `global` (by adding the attribute `global="true"` to the port) to provide connectivity at higher cluster levels (see also the example below).

Example

Listing A.6 shows an example XML description of a processor containing a single scalar issue-slot and a second vector issue-slot.

A.1.4 Limitations

The TimGen tool assumes that the input files will be automatically generated and provides only a very limited sanity check on the described architecture. For example, it does not check if structure of the building blocks used in an architecture description matches with the library

A.2 Design-space exploration tools

The final ASIP architecture exploration tool takes as its input an initial ASIP prototype as generated by the second phase of the micro-architecture exploration. Each initial ASIP prototype already represents a coarsely explored final ASIP architecture. The initial ASIP prototype fixes the application mapping, the number and sizes of local memories, and the vectorization of various application parts. It also provides an upper bound to the number of issue-slots considered for the final ASIP architecture.

The tasks of the final ASIP architecture exploration tool presented here are:

- Minimize the number of issue-slots in the final ASIP architecture.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<timplatform:ASIP name="sample">

<constant intWidth="32" />
<constant vecWidth="256" />
<constant vecWays="8" />
<constant pmemWidth="210" />
<constant immBits="5" />
<constant immBits2="10" />

<cluster>
  <is type="eva" width="vecWidth" ways="vecWays" imm="immBits">
    <rf size="4">
      <port name="ip0" width="intWidth" global="true"/>
      <port name="ip1" width="intWidth"/>
    </rf>
    <rf size="8">
      <port name="ip2" global="true"/>
      <port name="ip3"/>
    </rf>
    <rf size="2">
      <port name="ip4" width="vecWays" global="true"/>
    </rf>
    <port name="op0" width="intWidth" global="true"/>
    <port name="op1" width="vecWidth" global="true"/>
    <port name="op2" width="vecWays" global="true"/>
    <lm name="vmem" size="8192"/>
  </is>
  <is type="ana" imm="immBits2" width="intWidth">
    <rf size="64">
      <port name="ip0" global="true"/>
      <port name="ip1"/>
    </rf>
    <port name="op" global="true"/>
    <lm name="dmem" size="8192"/>
    <pmem size="4096" width="pmemWidth"/>
    <fifo capacity="2" count="4"/>
  </is>
</cluster>
</timplatform:ASIP>
```

Listing A.6: *Example XML input with one scalar issue-slot ('ana') as sequencer and one vector issue-slot ('eva')*

- Explore the available operations in each issue-slot of the final ASIP architecture.
- Explore the sizes of the register files available in the final ASIP architecture.
- Minimize the local communication structures within the ASIP architecture.
- Minimize the size of the program memory of the final ASIP architecture.

The aim of the exploration tool is to create a final, reduced, ASIP architecture which efficiently implements the target application in terms of both area and energy consumption while maximizing a given fitness model. Currently the following fitness models are supported:

- 1/energy-delay product (ed)
- 1/energy-delay-squared (edd)
- 1/energy-squared-delay (eed)
- 1/energy-delay-area product (eda)

Defining new fitness models is very easy and is discussed below.

A.2.1 Interface

Input

- Initial ASIP prototype

Output

- Final optimized ASIP prototype
- Simulated performance characteristics

A.2.2 Initial prototype preparation

The ASIP architecture exploration tool requires a small addition to the SH project of the initial prototype.

`$(EXTENDED_CFLAGS)` needs to be added to the to the `CFLAGS` line for the hive code in the Makefile as shown below.

Furthermore, the standard library and emulation libraries need to be disabled using `-nostdlib` and `-fno-emul-arith` as functions from these libraries are compiled for the complete processor and will disrupt the shrinking process.

```
SYSTEM = ecg_core_system
METHODS = sched
PROGRAMS = ecg

HOST_FILES     = host.c
HOST_CFLAGS    = -W -Wall

ecg_CELL      = ecg_core
ecg_FILES     = ecg.hive.c
ecg_CFLAGS    = -Werror -nostdlib -fno-emul-arith $(EXTENDED_CFLAGS)
ecg_LDFLAGS   = -embed
```

Listing A.7: *Makefile prepared for exploration*

A.2.3 Usage

The ASIP architecture exploration tool is currently implemented as a command-line tool to aid integration into the framework through scripting. Two versions of the exploration tool are available, `asam-asip-explorer` which is based on a heuristic exploration algorithm, and `asam-asip-explorer-genetic` which uses a genetic search.

Common options

Both tools share a set of common options which control common search parameters such as the following.

- The granularity of the exploration
  (`-style=issue-slot` or `-style=function-unit`)
- Fitness model selection and parameters
  (`-fitness-model` and `-required-latency`)
- Output detail
  (`-verbose`)
- Compilation and simulation caching
  (e.g. `-no-compilation-cache` and `-no-simulation-cache`)

The tools also provide a `-help` option which prints the complete list of options with their default values.

Heuristic search options

The heuristic version adds the following options.

- Heuristic search candidate selection strategy
  (`-strategy=first` or `-strategy=best`)

- One extra exploration style option
  (`-style=two-phase`)

Genetic search options

The genetic version adds the following options.

- Controlling the minimum and maximum number of generations the exploration will run
  (`-min-generations=10` and `-max-generations=50`)
- Controlling the mutation rate
  (`-mutation-rate=.15`)
- Early exit when the genetic algorithm reaches a value which stays constant for a given number of generations
  (`-plateau-length=10`)

A.2.4 Examples

The ASIP exploration tool is demonstrated on the ECG use-case from ST, but was also tested on several other applications. The ECG use-case consists of a single kernel and its code can not be transformed by the second phase exploration since the explorations performed in the second phase require a set of two or more communicating loops. This results in a single considered application mapping which makes the ECG explicitly suitable for demonstrating the effects of the third phase of the micro-level architecture exploration.

During the first phase, the minimum and maximum execution time of the application are estimated together with the parallelism requirement for achieving the minimum execution time. The initial ASIP prototype is then constructed according to this parallelism requirement. The parallelism requirement of the ECG use-case is 3, as is shown in table A.1, resulting in a 3 issue-slot initial ASIP prototype. Furthermore, software pipelining is not possible for the ECG application, since the algorithm mainly involves decision processes and requires a large amount of control-flow to which software pipelining does not apply [71].

Table A.1: *Predicted cycle counts for the ECG use-case*

issue-width	cycle count
1	47648
2	31992
3	31977

```
$ asam-asip-explorer -style=issue-slot ecg
Exploring initial ASIP prototype 'ecg'
Found initial prototype mapped on: core_3b
Loading information from APEX file...
Initial prototype has 3 issue-slots

Searching for best 'ed' fitness solution...
 Using 'issue-slot--first' strategy

P1: #IS: 3, Totals: 34336 cycles, 1.954E+03 nJ, 33680 um2 (logic),
    195910 um2 (memory), improvement=1.000000
P3: #IS: 2, Totals: 34373 cycles, 1.460E+03 nJ, 25563 um2 (logic),
    193597 um2 (memory), improvement=1.336916
Built 4 prototypes

Best solution FUs:
  bp_core_3b_s1_aru
  bp_core_3b_s1_bru
  bp_core_3b_s1_lgu
  bp_core_3b_s1_lsu
  bp_core_3b_s1_psu
  bp_core_3b_s1_shu
  bp_core_3b_s1_suu
  bp_core_3b_s2_aru
  bp_core_3b_s2_lgu
  bp_core_3b_s2_psu
  bp_core_3b_s2_shu

P3: #IS: 2, Totals: 34373 cycles, 1.460E+03 nJ, 25563 um2 (logic),
    193597 um2 (memory), improvement=1.336916
```

Listing A.8: *Example exploration output using issue-slot first-match strategy*

Using this information, we constructed the initial ASIP prototype being a 3 issue-slot processor with one internal memory large enough to store the test-samples and internal variables. We then used our final ASIP exploration tool to reduce the initial ASIP prototype architecture and find solution with the maximum fitness. These results were obtained using the command shown in listing A.8 (with the initial ASIP prototype stored in the directory `ecg` and using the default ED fitness model).

So far the tool only explored the number of issue-slots and found an optimal solution using 2 issue-slots which improves upon the fitness of the original design by 34%. The required function-units for the best solution are listed and allow the designer to instantiate the final processor design.

It is also possible to further reduce the contents of the issue-slots by removing function units and related operations. This effect is achieved using the command shown in listing A.9.

This will remove operations and function-units from candidate prototypes found using the issue-slot exploration in an attempt to further improve upon the cost metric. However, for this example it didn't result in a better design.

A.2.5 Implementing custom fitness models

The fitness of a design is decided by the fitness models which are embedded in the `asam-asip-explorer` script which is written using Perl. The provided fitness models are defined in `lib/fitness_models.pm`.

A fitness model takes a reference to a hash containing the metrics of the current design. These metrics can be used to compute the fitness and this fitness value is then returned. A higher fitness implies a better design, a fitness of 0 implies that the design did not function correctly.

Listing A.10 shows the energy-delay product which also checks a hard maximum latency (delay) constraint as an example.

The fitness model does not need to be a simple linear function as can be seen from this listing. The third line, for example, tests if the metrics were supplied correctly (which does not happen in case of a build or simulation failure), if the latency requirement is given `($required_latency != 0)` and, if the given requirement is met `($required_latency < $metrics->{'cycle_count'})`.

The processor metrics hash currently contains the following fields:

A.2.6 Current status and limitations

The current implementation of the ASIP architecture exploration tool implements all the above described options. However, there are still several limitations which should be addressed before this tool can actually be used in a industrial environment.

The ASIP exploration tool uses the ASAM power model to estimate the performance of candidate prototypes. It therefore shares the limitations of the

```
$ asam-asip-explorer ecg
Exploring initial ASIP prototype 'ecg'
Found initial prototype mapped on: core_3b
Loading information from APEX file...
Initial prototype has 3 issue-slots

Searching for best 'ed' fitness solution...
 Using 'two-phase--first' strategy

P1: #IS: 3, Totals: 34336 cycles, 1.954E+03 nJ, 33680 um2 (logic),
    195910 um2 (memory), improvement=1.000000
P3: #IS: 2, Totals: 34373 cycles, 1.460E+03 nJ, 25563 um2 (logic),
    193597 um2 (memory), improvement=1.336916
Built 5 prototypes

Best solution FUs:
  bp_core_3b_s1_aru
  bp_core_3b_s1_bru
  bp_core_3b_s1_lgu
  bp_core_3b_s1_lsu
  bp_core_3b_s1_psu
  bp_core_3b_s1_shu
  bp_core_3b_s1_suu
  bp_core_3b_s2_aru
  bp_core_3b_s2_lgu
  bp_core_3b_s2_psu
  bp_core_3b_s2_shu

P3: #IS: 2, Totals: 34373 cycles, 1.460E+03 nJ, 25563 um2 (logic),
    193597 um2 (memory), improvement=1.336916
```

Listing A.9: *Example exploration output using two-phase first-match strategy*

```
sub ed {
    my ($metrics) = @_;
    return 0 if (not defined $metrics or ($required_latency != 0 and
        $required_latency < $metrics->{'cycle_count'}));
    return 1/($metrics->{'total_energy'}*$metrics->{'cycle_count'});
}
```

Listing A.10: *Example fitness function implementing energy-delay product.*

```
$metrics = {
    'issue_width'         => $issue_width,
    'cycle_count'         => $total_cycles,
    'total_energy'        => $total_energy,
    'logic_area'          => $logic_area,
    'memory_area'         => $memory_area,
    'n_prototypes_built' => $n_prototypes_built,
    'used_fus'            => \%fu_list,
};
```

Listing A.11: *Metrics available for implementing fitness functions.*

ASAM power model. The most important one being that it is unable to predict the energy consumption of code that is not part of the application (e.g. emulation code or elements from the standard C library).

Secondly, the ASIP exploration tool uses the BuildMaster framework to avoid repetitive simulation of the application by estimating the energy cost based on a application profile and the assembler code of the compiled application. However, some optimizations performed by the compiler may change the application's profile and thereby result in incorrect energy cost prediction. The BuildMaster framework tries to correct for this but cannot do this in all cases. Any final solution obtained using the ASIP exploration tool will need to be simulated in order to verify the predicted cost.

The final limitation of the ASIP architecture exploration tool is that currently it does not automatically add custom operations. In order to explore custom operations, the appropriate function units need to be added, the compiler needs to be able to recognize occurrences of the custom operation pattern, and the area and power models need to be provided for each custom function unit. If these three constraints are met, the current implementation of the ASIP exploration tool is capable of exploring the architecture including the effects of the addition of the considered custom operations.

Samenvatting

Hoge eisen aan zowel de energie-efficiëntie als de prestaties voor de beeld en signaal bewerking die tegenwoordig onderdeel is van mobiele en autonome geïntegreerde systemen, maken het niet langer mogelijk om alleen gebruik te maken van ongespecialiseerde processor systemen. Hierdoor is er een sterke verschuiving opgetreden richting het gebruik van heterogene processor systemen, welke opgebouwd worden rond meerdere gespecialiseerde processoren. Alhoewel er een zekere hoeveelheid automatisering reeds beschikbaar is worden dergelijke systemen veelal met de hand ontworpen. Hierbij worden belangrijke ontwerp besluiten genomen waarbij sterk gebruik gemaakt wordt van inaccurate schattingen. De combinatie van deze hoge mate van interactiviteit binnen het ontwerpproces met de korte opvolging tussen verschillende product generaties resulteren in een sterk gereduceerd aantal alternatieve ontwerpen dat beschouwd kan worden. Als direct gevolg hiervan zal vrijwel altijd een suboptimaal ontwerp ontstaan.

De huidige technologieën bieden slechts beperkte ondersteuning voor automatisering van het ontwerp proces en richten zich vooral op de automatisering van belangrijke tussenstappen zoals de constructie van een nieuw processor ontwerp vanuit een hoog niveau omschrijving, het evalueren van een kandidaat ontwerp door middel van simulatie of emulatie, en door middel van het voorstellen van mogelijke uitbreidingen aan een bestaand ontwerp. Ondanks dat de huidige methoden reeds significante verbeteringen hebben gebracht blijft er nog altijd ruimte voor verdere versnelling van het ontwerpproces. De, in dit proefschrift voorgestelde methoden, richten zich dan ook vooral op verbeteringen in zowel de ontwerp evaluatie als de verdere automatisering van de ontwerp kandidaat selectie.

Een drie stappen aanpak voor processor architectuur ontwerp wordt voorgesteld, beginnend met een nieuwe applicatie analysemethode gericht op het vinden van het beschikbare parallellisme en het vaststellen van vroege prestatie schattingen voor rekenkundig intensieve programmadelen. Deze schattingen worden vervolgens gebruikt gedurende het herstructureren van het doel programma, waarbij het beschikbare parallellisme verder verhoogd wordt en een geschikte verdeling van de benodigde programmagegevens over de processor geheugens wordt bepaald als tweede stap. Door de getransformeerde programma code te combineren met de vastgestelde geheugen hiërarchie en de originele parallellisme schattingen is het vervolgens mogelijk een initiële processor architectuur te construeren. Als derde stap wordt vervolgens deze initiële architectuur verfijnd tot een sterk gespecialiseerde processor architectuur omschrijving.

Het onderzoek dat is gepresenteerd in dit proefschrift concentreert zich op verbeteringen in de volgende stappen van het ontwerpproces.

- Een methode voor het inschatten van parallellisme op het instructie niveau dat beschikbaar is in een programma. Deze methode biedt vroegtijdige feedback welke van pas komt gedurende de verkenning van mogelijke herstructureringen van het doel programma, maar wordt ook gebruikt voor het bepalen van het juiste aantal operaties dat parallel gestart dient te kunnen worden in de initiële processor architectuur. Als gevolg zijn we in staat om een initieel processor ontwerp te construeren dat de geëiste prestaties kan leveren, maar dat ook redelijk dichtbij het uiteindelijke verfijnde processor ontwerp ligt.
- Een processor architectuur verfijningsmethode welke de mogelijkheid biedt om de tijdrovende constructie van kandidaat processoren over te slaan. De gepresenteerde methode vereist alleen de constructie van het initiële architectuur voorstel en het uiteindelijke processor ontwerp. Alle tussenliggende kandidaat architecturen hoeven niet geconstrueerd te worden.
- Een snelle methode voor het inschatten van het energieverbruik van een kandidaat ontwerp, gebaseerd op de executie frequentie van programma delen en hun assembler instructies. Hierdoor wordt de energieschattingsmethode onafhankelijk van het gesimuleerde aantal processor klok cycli waardoor langere, meer representatieve, test sequenties gebruikt kunnen worden om zodoende tot een snellere en meer realistische evaluatie te komen.
- Een architectuur exploratie platform genaamd BuildMaster dat het makkelijker maakt om nieuwe zoekstrategieën te implementeren. Dit platform detecteert automatisch wanneer compilatie of simulatie resultaten van eerder beschouwde kandidaat architecturen geschikt zijn voor hergebruik bij de evaluatie van een nieuw kandidaat ontwerp. Dit systeem stelt ons in staat om, bijvoorbeeld, gemiddeld 90% van de traditioneel benodigde simulaties over te slaan door het hergebruiken van reeds beschikbare executie frequentie gegevens gedurende de energie schatting.
- Een aantal architectuur exploratie strategieën die op effectieve wijze de processor architectuur verfijnen en een vergelijking tussen deze strategieën met betrekking tot hun effectiviteit en benodigde exploratie tijd. Hierbij wordt aangetoond dat de voorgestelde heuristiek in staat is een vergelijkbaar resultaat te verkrijgen als wordt bereikt met behulp van een genetisch algoritme terwijl het slechts een fractie van de exploratie tijd vereist.

Het combineren van de gepresenteerde technieken resulteert in een zeer efficiënte en uitbreidbare instructie-set architectuur exploratie methode. In onze experimenten wordt aangetoond dat het exploratie platform in staat is om honderden processor architecturen tegen elkaar af te kunnen wegen per uur, maar ook consistent een compacte architectuur als resultaat kan presenteren.

Acknowledgements

The time has come to finalize writing this dissertation. Overall, it has been an interesting experience, these last five years of my life. With many great moments but also with more difficult ones. My thanks go out to those that supported me in achieving this goal.

First of all, I would like to thank my promotor and co-promotor, Henk Corporaal and Lech Jóźwiak, for providing me with this opportunity and for their support during the process over the years. I am grateful for the important lessons and contributions that resulted from your critical views and insightful questions, as well as, the many insights that I have obtained when it comes to planning projects and the execution of plans.

My thanks to the members of my PhD committee, Ton Backx, Jarmo Takala, Koen Bertels, Peter de With, Jeroen Leijten, and Bart Mesman for their participation and contributions to the final version of my dissertation. Bart, thank you for suggesting this position to me when it became available.

I would also like to thank the many people that I cooperated with through the ASAM project. Our project discussions have had a significant impact on the directions chosen for this research and provided me with lots of ideas and different views on the considered problems. In particular I would like to thank Menno, Erkan, Alexandre, Felipe, Rosilde, Giuseppe, Laura, Paolo, Lech, and Jan, for their contributions within this project. Especially Erkan, thank you for the fruitful cooperation, discussions, your contributions to the exploration framework and tools presented in Chapter 6 and Appendix A, together with the many shared publications that resulted from this; Menno, thank you for your many insights and very thorough review of this dissertation; and to Giuseppe: EBBE'ECCO!

Also thanks to the members of our bi-weekly PARsE meetings; Cedric, Gert-Jan, Maurice, Erkan, Dongrui, Luc, Mark, Yifan, Zhenyue, Rosilde, and Henk. We've had many interesting discussions and explaining my problems and ideas to you usually greatly helped my own understanding of them.

Thanks also to the people that shared their offices (in Potentiaal and Flux), lunch, and coffee table time with me. There were to many interesting (and possibly disturbing) discussions during these times to remember them all, and I guess it's good that not all of them are remembered equally well. It has been a great experience to have you as colleagues and I guess I will keep seeing you around. Either at the university, at conferences, in companies, or during some other random encounter. Thank you, Andrew, Cedric, Gert-Jan, Luuk, Manil, Davit, Sven, Luc, Mark, Raymond, Martijn, Reinier, and Joost (was ik je bijna vergeten jòh).

During the writing of this dissertation I managed to flee the country for a while to visit Movidius in Dublin and work on the LLVM project as part of a HiPEAC collaboration grant. This didn't directly result in new contents for my dissertation but an important part of the writing did happen over there in parallel to lots of interesting compiler related work and discussions. Thanks to Martin, Stephen, David, and all the others at Movidius for their welcome reception, many great discussions, and teaching me how to appreciate a good pint of Guinness. Cheers guys!

Special thanks to all the people that kept me distracted from my work during the times that I needed that. Sometimes it's needed to step outside of the tight technological view of designing application specific instruction-set processors, in order to find the actual problems that need to be solved. Thanks to the members of ESAC, in particular het 33e bestuur, de ESAC, and de AC, for preventing my back from taking too much of the shape of my office chair. Thanks to de Donderdagavond ploeg and de Bier Brigade for listening to my complaints when things weren't working out at times, and for temporarily relieving my mind from those problems. And finally, thanks to the Eindhoven Museum, WEA, and Waldfyrd groups for helping me completely forget about modern technology so that I could focus my thoughts on campfires, and working with hammer and anvil. Also thanks to my family, for supporting me in my quest for knowledge, letting me find my own path, and for not panicking too much when deviations extended the duration of my study yet another bit. I'm not going to name you here, I'm just too afraid that I will miss important people here and I have no clue about the order I should put you in. You're the guardians of my sanity, thank you for that.

That's all, I'm sure that there are many people that deserved to be mentioned, either directly or indirectly, but which I have failed to thank here anyway. So to those people that feel missed, thank you as well… So long, and thanks for all the fish.

About the author

Roel Jordans 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 within the ASAM project. His research interest include VLIW architectures and the automatic synthesis of application specific instruction-set processors, as well as, related compilation and code optimization techniques for these platforms.

Author's publications

Journal Articles and Book Chapters

[1] Diken, E.; **Jordans, R.**; Corvino, R.; Jóźwiak, L.; Corporaal, H. and Chies, F. A.: *Construction and Exploitation of VLIW ASIPs with Heterogeneous Vector-Widths.* In Microprocessors and Microsystems, 38 (8-B): 947-959, 2014.

[2] **Jordans, R.**; Corvino, R.; Jóźwiak, L. and Corporaal, H.: *Exploring processor parallelism: Estimation methods and optimization strategies.* In International Journal of Microelectronics and Computer Science, 4 (2): 55-64, 2013.

[3] Jóźwiak, L.; Lindwer, M.; Corvino, R.; Meloni, P.; Micconi, L.; Madsen, J.; Diken, E.; Gangadharan, D.; **Jordans, R.**; Pomata, S.; Pop, P.; Tuveri, G.; Raffo, L. and Notarangelo, G.: *ASAM: Automatic architecture synthesis and application mapping.* In Microprocessors and Microsystems, 37 (8): 1002-1019, 2013.

[4] Stuijk, S.; Kumar, A.; **Jordans, R.** and Corporaal, H.: *Implementing Time-Constrained Applications on a Predictable MPSoC.* In Multicore Technology: Architecture, Reconfiguration, and Modeling, chapter 2, pages 41-60, CRC Press, Boca Raton, Fl, USA, 2013.

Conference and Workshop proceedings

[1] Diken, E.; O'Riordan, M.; **Jordans, R.**; Jóźwiak, L.; Corporaal, H. and Moloney, D.: *Mixed-Length SIMD Code Generation for VLIW Architectures with Multiple Native Vector-Widths.* In ASAP 2015 - 26th IEEE International Conference on Application-specific Systems, Architectures and Processors, Totonto, Canada, 2015.

[2] **Jordans, R.** and Corporaal, H.: *High-level software-pipelining in LLVM.* In SCOPES '15 - 18th International Workshop on Software and Compilers for Embedded Systems, pages 97-100, Sankt Goar, Germany, 2015.

[3] Diken, E.; **Jordans, R.**; Jóźwiak, L. and Corporaal, H.: *Construction and Exploitation of VLIW ASIPs with Multiple Vector-Widths.* In MECO 2014 - 3rd Mediterranean Conference on Embedded Computing, pages 244-247, Budva, Montenegro, 2014.

[4] **Jordans, R.**; Jóźwiak, L. and Corporaal, H.: *Instruction-set Architecture Exploration of VLIW ASIPs Using a Genetic Algorithm.* In MECO 2014 - 3rd Mediterranean Conference on Embedded Computing, pages 32-35, Budva, Montenegro, 2014.

[5] **Jordans, R.**; Diken, E.; Jóźwiak, L. and Corporaal, H.: BuildMaster: *Efficient ASIP Architecture Exploration Through Compilation and Simulation Result Caching.* In DDECS 2014 - 17th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, Warsaw, Poland, 2014.

[6] **Jordans, R.**; Corvino, R.; Jóźwiak, L. and Corporaal, H.: *An Efficient Method for Energy Estimation of Application Specific Instruction-set Processors.* In DSD 2013 - 16th Euromicro Conference on Digital System Design, pages 471-474, Santander, Spain, 2013.

[7] **Jordans, R.**; Corvino, R.; Jóźwiak, L. and Corporaal, H.: *Instruction-set Architecture Exploration Strategies for Deeply Clustered VLIW ASIPs.* In ECyPS 2013 - EUROMICRO/IEEE Workshop on Embedded and Cyber-Physical Systems, pages 38-41, Budva, Montenegro, 2013.

[8] **Jordans, R.**; Corvino, R.; Jóźwiak, L. and Corporaal, H.: *Exploring Processor Parallelism: Estimation Methods and Optimization Strategies.* In DDECS 2013 - 16th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, pages 18-23, Karlovy Vary, Czech Republic, 2013. *Received best paper award.*

[9] **Jordans, R.**; Corvino, R. and Jóźwiak, L.: *Algorithm Parallelism Estimation for Constraining Instruction-Set Synthesis for VLIW Processors.* In DSD 2012 - 15th Euromicro Conference on Digital System Design, pages 152-155, Cesme, Izmir, Turkey, 2012.

[10] Jóźwiak, L.; Lindwer, M.; Corvino, R.; Meloni, P.; Micconi, L.; Madsen, J.; Diken, E.; Gangadharan, D.; **Jordans, R.**; Pomata, S.; Pop, P.; Tuveri, G. and Raffo, L.: *ASAM: Automatic Architecture Synthesis and Application Mapping.* In DSD 2012 - 15th Euromicro Conference on Digital System Design, pages 216-225, Cesme, Izmir, Turkey, 2012.

[11] **Jordans, R.**; Siyoum, F.; Stuijk, S.; Kumar, A. and Corporaal, H.: *An automated flow to map throughput constrained applications to a MPSoC.* In Bringing Theory to Practice: Predictability and Performance in Embedded Systems, pages 47-58, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2011.

Other publications (non peer-reviewed)

[1] **Jordans, R.** and Moloney, D.: *A high-level implementation of software-pipelining for LLVM* (presentation). EuroLLVM 2015 - European LLVM Conference. London, United Kingdom, 2015.

[2] Diken, E.; **Jordans, R.** and O'Riordan, M.: *moviCompile: An LLVM based compiler for heterogeneous SIMD code generation* (presentation). FOSDEM 2015. Brussels, Belgium, 2015

[3] Diken, E.; **Jordans, R.**; Corvino, R. and Jóźwiak, L.: *Construction and Exploitation of VLIW ASIPs with Multiple SIMD Widths* (poster). ICT.Open 2013 - The interface for Dutch ICT-Research. Eindhoven, The Netherlands, 2013.

[4] **Jordans, R.**; Diken, E.; Corvino, R.; Jóźwiak, L. and Corporaal, H.: *Buildmaster: Efficient ASIP Architecture Exploration* (poster/presentation). ICT.Open 2013 - The interface for Dutch ICT-Research. Veldhoven, The Netherlands, 2013.

[5] Diken, E.; **Jordans, R.**; Corvino, R. and Jóźwiak, L.: *Application Analysis Driven ASIP-based System Synthesis for ECG* (paper/presentation). In Embedded World Conference, pages 1-8, Germany, 2012.

[6] **Jordans, R.**; Diken, E.; Corvino, R. and Jóźwiak, L.: *Automated Architecture Synthesis and Application Mapping for ASIP Based Adaptable MPSoCs* (poster). ICT.Open 2011 - The interface for Dutch ICT-Research. Veldhoven, The Netherlands, 2011.

[7] Diken, E.; **Jordans, R.**; Corvino, R.; Jóźwiak, L. and Lindwer, M.: *Automated architecture synthesis and application mapping for ASIP based adaptable MPSoCs* (abstract/poster). In ACACES 2011 - 7th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, pages 135-138, Fiuggi, Italy, 2011.

[8] **Jordans, R.**; Siyoum, F.; Stuijk, S.; Kumar, A. and Corporaal, H.: *An automated flow to map throughput constrained applications to a MPSoC* (poster). STW.ICT Conference 2010. Veldhoven, The Netherlands, 2010.