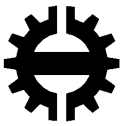

TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Information Technology

TOMMI RANTANEN

COST ESTIMATION FOR TRANSPORT TRIGGERED ARCHITECTURES

Master of Science Thesis

Subject approved by Department Council
12th May, 2004

Examiners: Prof. Jarmo Takala
Prof. Hannu-Matti Järvinen

PREFACE

This MSc thesis was completed in Institute of Digital and Computer Systems of Tampere University of Technology (TUT) in 2002-2004 as a part of the Flexible Design Methods for DSP Systems (FlexDSP) project funded by the National Technology Agency.

First and foremost, my sincerest gratitude is due to my thesis supervisor, Professor Jarmo Takala, for guiding me to a fascinating area of research and for valuable tips for the thesis. I am forever indebted to Andrea Cilio, PhD, for his unfailing patience in elaborating my work and for much invaluable tips and advice. I also record my deep appreciation to Jari Heikkinen, MSc, for numerous instructive comments. My warmest thanks are tendered to my collegues for the inspiring work atmosphere they provided.

Finally, I wish to thank my family for their support throughout my studies. Most of all, I want to thank my lovely Miia for her love and support.

Tampere, May 27, 2004

Tommi Rantanen

TABLE OF CONTENTS

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

Institute of Digital and Computer Systems

Rantanen, Tommi Olavi: Cost Estimation for Transport Triggered Architectures

Master of Science Thesis: 55 pages, 4 appendix pages

Examiners: Prof. Jarmo Takala and Prof. Hannu-Matti Järvinen

Funding: The National Technology Agency

June 2004

Keywords: transport triggered architecture, cost estimation, design space explorer

Nowadays, application-specific processors are of great interest since they offer the best possible trade-off between cost and performance. However, the design process of the application-specific processors has proven to be a difficult and time-consuming challenge. Thus, automated design space exploration is the most interesting tool for designers in the area of customizable processors. By trawling through the design space and notifying the most interesting target processor configurations, exploration tools assist the designers to find the most suitable resources for a given application. Cornerstone in the exploration tool, in addition to effective exploration algorithm, is the hardware cost estimator, which has to be fast, accurate enough, and technology independent.

MOVE framework is a set of non-commercial software tools for designing application-specific processors. The framework utilizes transport triggered architecture (TTA) programming model, where the program specifies only the data transports to be performed by the interconnection network. Operations occur as a side effect of these explicitly defined data transports.

In this thesis, a hardware cost estimator was developed into the MOVE framework. The estimator is based on the database of the costs of hardware resources. The purpose of the estimator is to evaluate the target processor for a given application in terms of chip area, power consumption and timing. The developed estimator improved the flexibility and accuracy of the old estimator of the MOVE framework. The flexibility was improved a lot in form of changing the technology, which requires only to replace the cost database with a new one that is generated for the new technology. In addition, the estimator is utilized with prime accuracy and efficiency. Furthermore, the implementation of the estimator is flexible and expandable utilizing object-oriented programming in designing the data structures and search algorithm of the cost database.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO

Tietotekniikan koulutusohjelma

Digitaali- ja tietokonetekniikan laitos

Rantanen, Tommi Olavi: Cost Estimation for Transport Triggered Architectures

Diplomityö: 55 sivua, 4 liitesivua

Tarkastajat: Prof. Jarmo Takala ja Prof. Hannu-Matti Järvinen

Rahoitus: Teknologian kehittämiskeskus (TEKES)

Kesäkuu 2004

Avainsanat: transport triggered architecture, prosessorin kustannusten arviointi

Sovelluskohtaiset käskykantaprosessorit ovat erittäin kiinnostavia, koska ne tarjoavat parhaan kompromissin kustannusten ja suorituskyvyn välille. Niiden suunnitteleminen on kuitenkin erittäin vaikea ja aikaavievä prosessi. Niinpä automaattiset suunnittelutyökalut ovat saavuttaneet suuren kiinnostuksen suunnittelijoiden parissa. Käymällä läpi mahdollisia suunnitteluratkaisuja ja ilmoittamalla kiinnostavimmista prosessorikonfiguraatioista automaattiset suunnittelutyökalut auttavat suunnittelijoita hakemaan parhaita laitteistokomponentteja annetulle sovellukselle. Kulmakivi automaattisissa suunnittelutyökaluissa on tehokkaan hakualgoritmin lisäksi prosessorin kustannusestimaattori, jonka pitää olla nopea, tarpeeksi tarkka ja riippumaton toteutusteknologiasta.

MOVE suunnitteluympäristö on joukko ei-kaupallisia suunnittelutyökaluja sovelluskohtaisten käskykantaprosessoreiden suunnitteluun. Se hyödyntää transport triggered -suoritinarkkitehtuurin (TTA) mukaista ohjelmointimallia, jossa sovellus määrittelee datan siirrot laskentayksiköiden ja rekisterien välillä. Operaatiot tapahtuvat näiden datan siirtojen sivuvaikutuksena.

Tässä diplomityössä kehitettiin prosessorin kustannusestimaattori MOVE suunnitteluympäristöön. Estimaattori perustuu tietokantaan laitteistokomponenttien kustannuksista. Estimaattorin tarkoituksena on arvioida prosessorikonfiguraatio annetulle sovellukselle pinta-alan, tehonkulutuksen ja ajoituksen suhteen. Kehitetty estimaattori paransi joustavuutta ja tarkkuutta verrattuna MOVE suunnitteluympäristön vanhaan estimaattoriin. Prosessorin toteutusteknologian vaihtaminen sujuu helposti korvaamalla kustannustietokanta uudella tietokannalla, joka on tehty uudelle teknologialle. Lisäksi estimaattori on erittäin tarkka ja nopea. Estimaattorin ohjelmistototeutus on myös erittäin joustava ja laajennettava. Kustannustietokannan tietorakenteet ja hakualgoritmi on suunniteltu käyttäen hyväksi olio-ohjelmoinnin periaatteita.

LIST OF ABBREVIATIONS AND SYMBOLS

α	Constant reflecting the importance of area
β	Constant reflecting the importance of execution time
2-D	Two-Dimensional
3-D	Three-Dimensional
ASIC	Application-Specific Integrated Circuit
ASP	Application-Specific Processor
DCT	Discrete Cosine Transform
DSP	Digital Signal Processor or Digital Signal Processing
FU	Function Unit
GCC	GNU Compiler Collection
GNU	Gnu's Not Unix
HDL	Hardware Description Language
HLL	High-Level Language
ILP	Instruction-Level Parallelism
OTA	Operation Triggered Architecture
RF	Register File
TTA	Transport Triggered Architecture
VHDL	Very high speed integrated circuit Hardware Description Language
VLIW	Very Long Instruction Word

1. INTRODUCTION

The current trend in programmable architectures, especially in digital signal processing (DSP) application domain, is to move towards high-level language (HLL) programming and customizable architectures [1]. The reason behind this is the increasing gap between the productivity of designers and increased complexity of DSP applications. By using customizable architectures, the hardware resources of the processor can be tailored according to the requirements of the application. However, it is difficult to find a satisfactory solution from the large design space; even hundreds of different architecture alternatives must be designed and evaluated. Efficient evaluation requires a set of software tools, such as HLL compiler and instruction set simulator as well as synthesis tools. Utilizing these tools manually for each architecture configuration to gather the design space extensively takes unacceptably long span of time. Thus, a design space explorer is required to automate the process.

The evaluation of a processor configuration has to be performed fast to keep the exploration time under control. Nevertheless, estimating rapidly and accurately the costs of a target processor for an application has proven to be a difficult challenge. Estimating the costs by utilizing the most accurate alternative, i.e., logic synthesis, takes far too long. Quick, accurate enough, and technology independent procedure for the estimation is optimal from the design space explorer's point of view. Speed of the estimation is appreciated almost as much as the accuracy, since the designer should be quickly conducted into the interesting design area by the explorer.

Design space explorer modifies the target processor while searching for the most suitable configuration for a given application. Thus, the processor architecture must be flexible, scalable, and customizable. Transport triggered architecture (TTA) is such an architecture [2]. The MOVE framework is a design environment utilizing the TTA concept for designing application-specific processors [3]. It contains a set of software tools such as HLL compiler, instruction set simulator, hardware estimator, and design space explorer.

In this thesis, a new cost estimation procedure based on a cost database is proposed.

The ideas of the estimation are targeted to processors, which are composed of different type of architectural resources. An example implementation of the hardware cost estimator evaluating the area and power consumption of the target processor is described in detail. The software implementation of the cost estimator is also represented, emphasis being in the flexible cost database. In addition, the experimental results verifying the accuracy and the speed of the estimator are described.

The structure of this thesis is as follows. Chapter 2 introduces the TTA concept together with its programming paradigm. In addition, the MOVE framework is described with its subcomponents. First, the hardware and software subsystems are discussed. Secondly, the design space explorer searching for the best configuration for a given application is represented. Finally, the hardware cost estimator being responsible for estimating the costs of a processor configuration is described. Chapter 3 represents the principles of the cost estimation procedure based on a cost database. Moreover, the functionality of the proposed hardware cost estimator is described in detail from the user's point of view. The software implementation of the estimator is presented in Chapter 4. The actual estimator is described shortly while the emphasis is on the cost database implementation. The data structures storing the database information are represented as well as the search algorithms supporting the database queries. Chapter 5 introduces the experimental results verifying the accuracy of the proposed estimator. Both the area and the power accuracies are analyzed. Moreover, the experiments carried out to verify the speed of the estimator are represented. Chapter 6 summarizes this thesis by representing the conclusions.

2. MOVE FRAMEWORK

Customizable processor architectures are nowadays more and more under designers' interest, since the design process of application-specific integrated circuits (ASIC) takes unacceptably long span of time. Due to the fact that customizable processors can be modified much easier than ASICs, the design process can be automated. The MOVE framework provides such a design process. The TTA concept utilized by the MOVE framework offers the flexibility, scalability, and modularity required by the automated design tools.

Section 2.1 describes the TTA concept and its architectural principles as well as the software aspects. In Section 2.2, a high-level description of the MOVE framework is given together with its hardware and software subsystems. Section 2.3 introduces the principles of the design space explorer of the MOVE framework. Moreover, the exploration algorithms are described. Section 2.4 represents the original hardware cost estimator of the MOVE framework.

2.1 Transport Triggered Architectures

Very long instruction word (VLIW) architecture has been an engrossing alternative for the DSP applications due to their modularity. The organization of an example VLIW processor with two function units (FU) is illustrated in Fig. 1. The number of FUs can be increased and even application-specific FUs can exist. However, the complexity of the bypass network and the register file (RF) increases rapidly since each input and output of an FU requires a connection to the RF. Nevertheless, the problem can be reduced, but not avoided, by partitioning the RFs and dividing the RFs into two levels. In addition, the utilization of the interconnection network is low. [2]

TTAs were developed to avoid the problems that exist in VLIW architectures. The complexity of the bypass network is reduced by moving the RFs into the same architectural level as the FUs. TTA is a flexible and modular architecture, hence an attractive alternative in embedded systems. The high-level organization of a TTA processor is

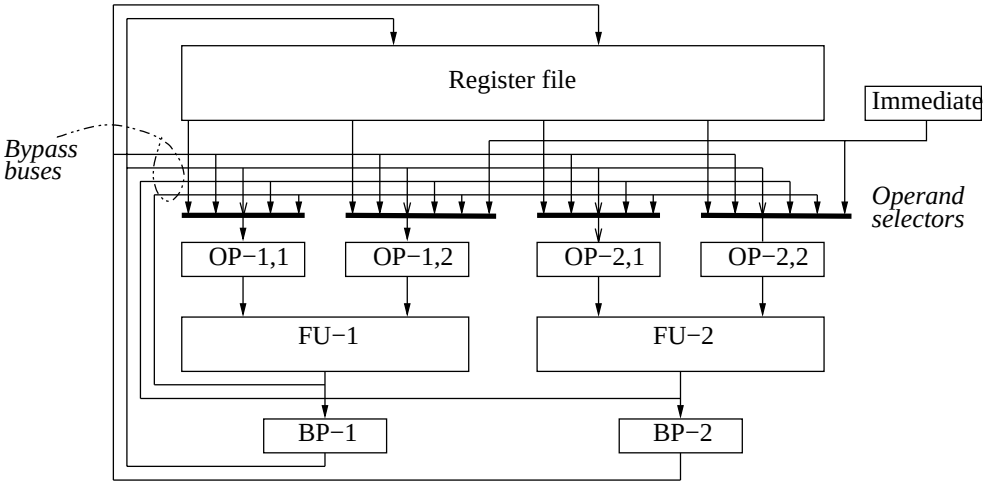

Figure 1. *The organization of a VLIW processor with two FUs.*

represented in Fig. 2. The FUs and RFs are connected to each other by an interconnection network which is controlled by the program code. In the TTA programming model, contrary to traditional operation triggered architectures (OTA) where operations are programmed, only the data transports to be performed by the interconnection network are programmed. Operations occur as a *side effect*. Since multiple data transports may occur simultaneously, instruction-level parallelism (ILP) offered by VLIW architectures is supported by the TTAs as well. [2]

2.1.1 Hardware Aspects

A TTA processor is composed of FUs, RFs, interconnection network, and control logic as well as data and instruction memory. The interconnection network includes buses and input and output sockets. Figure 3 depicts the structure of the TTA. [2]

FUs are responsible for performing the actual computation. They contain three types

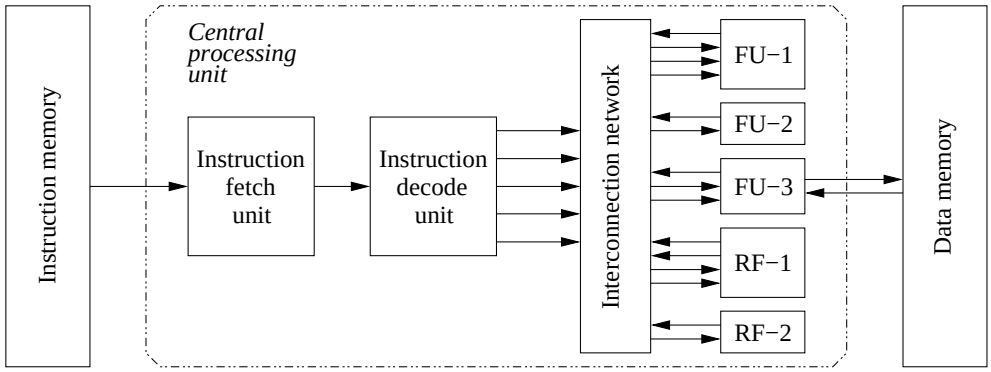

Figure 2. *Organization of the TTA.*

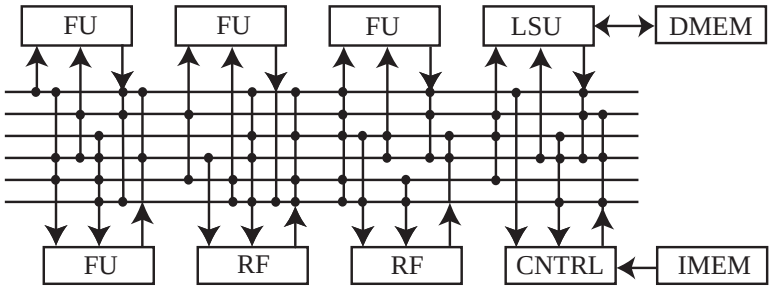

Figure 3. *TTA processor structure. FU: function unit. RF: register file. LSU: load-store unit. CNTRL: control unit. DMEM: data memory. IMEM: instruction memory. Dots represent connections between buses and sockets.*

of registers: operand, trigger and result. Operand and trigger registers function as the inputs of an FU whereas the result register is an output. The operands of an FU are transported into the input registers by the interconnection network. Transferring a value into the trigger register also initiates the functionality of the FU. Together with the trigger move, the opcode, which specifies the operation to be performed in the FU, is transferred. Some FUs have their specialized tasks, such as loading/storing values into the data memory. Heterogeneous, even application-specific, multi-operand FUs can be used without restrictions. Adding multi-operand FUs into a TTA processor is trivial from the architecture point of view. The data can be flexibly written into many input registers of an FU as well as read from the output registers since all the FUs resemble each other from the interconnection network's standpoint. Due to the same reason, an FU can be easily pipelined.

RFs contains general-purpose registers storing temporary values required by the program for a short time interval. RFs can be considered as a special FU containing input and output ports. However, the number of RF ports can be reduced considerably compared to VLIWs due to the fact that even one input port is enough for transferring the data from any FU into that RF. Since no fundamental differences exist between the FUs and RFs from the interconnection's point of view, the RFs can be easily partitioned without significant decrease in performance.

Interconnection network handles the data transfers between the FUs and RFs in the TTA. As mentioned before, the interconnection network consists of the buses and input and output sockets. The buses transfer data between the FUs and RFs. An input socket contains multiplexers feeding operands from the buses into the FUs and RFs, and an output socket contains de-multiplexers placing the result of an FU or RF into the correct bus.

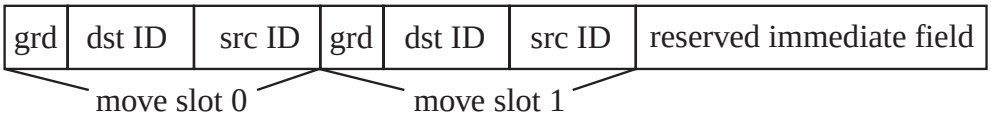

Figure 4. *The instruction format of the TTA.*

Each move bus can perform one data transport. Thus, the instruction of the TTA depicted in Fig. 4 must include a field for each move bus, hence the field is usually denoted as move slot. A move slot consists of three fields: guard ID, destination ID and source ID. Destination ID field indicates into which input socket the data from the bus is written whereas the source ID indicates the output socket from which the data is read. Guard ID field indicates whether the transport is performed or not in the bus. It can be used to implement conditional statements. In addition to the move slots, the instruction format includes reserved fields for long immediates. It is used for moving long immediate values in the bus, e.g., program counter values for jumps.

The control logic is responsible for fetching the instructions from the instruction memory. Moreover, it decodes the instruction to activate the correct sockets to perform the requested data transports utilizing the buses.

2.1.2 Software Aspects

Since only the data transports are specified by the program, only one type of operation is supported: move operation, which performs a data transport from a source to a destination. The operations occur as a side effect of the move operations. A normal OTA subtraction:

```
sub r3, r1, r2
```

can be converted to the following move operations in TTA:

```
r1 -> sub.o
r2 -> sub.t
sub.r -> r3
```

The destination register sub.o indicates the operand register of the subtractor unit whereas sub.t indicates the trigger register. The result register is indicated by the source sub.r.

The programming paradigm of the TTAs is mirrored compared to traditional OTAs. It has several advantages. It allows new scheduling and allocation techniques to be used in HLL compilers increasing the scheduling freedom. In TTAs, there are more items to schedule than in OTAs, since OTA operations of a program are divided into multiple TTA move operations. The moves can be scheduled freely as long as the operand moves occur simultaneously or before than the trigger move and the trigger move takes place before the result moves. Thus, multi-operand FUs are easy to support. HLL compilers can support optimizations such as software bypassing, dead result elimination and operand sharing. However, implementing such a compiler is a complex task due to all the advantages of scheduling possibilities. The compiler strategies of the TTA are discussed in detail in [4], [5] and [6].

2.2 MOVE Tools

The MOVE framework is a toolset for designing TTA processors semi-automatically [3]. Figure 5 depicts the general organization of the MOVE framework which is composed of three main components: design space explorer, software subsystem, and hardware subsystem. The software subsystem is responsible for generating parallel code for the target processor from the HLL code. The hardware subsystem is aimed at producing hardware description language (HDL) code of the target processor. The purpose of the design space explorer is to search for a processor configuration, which yields the best cost/performance ratio for a given application. Both the software and hardware subsystems provide statistics for the design space explorer.

The architecture description is in a central role in the MOVE framework. It fully characterizes the target processor, i.e., the type and number of FUs, RFs, RF ports and buses. In addition, the connections between the buses and sockets are described. The description does not restrict the number of different hardware resources. Even application-specific FUs can be included in the architecture description. However, the software subsystem has to be aware of the user-defined operations.

The software subsystem is described in Section 2.2.1 whereas the hardware subsystem is discussed briefly in Section 2.2.2. Section 2.3 represents the functionality of the design space explorer in detail.

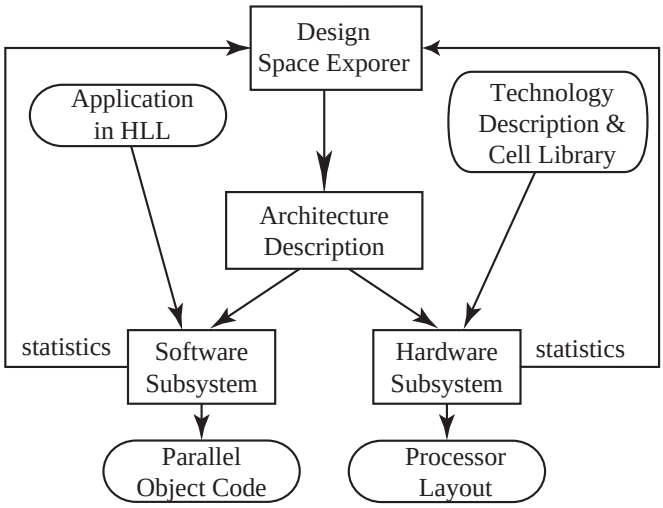

Figure 5. *The organization of the MOVE framework.*

2.2.1 Software Subsystem

The purpose of the software subsystem is to generate parallel code for the target processor for a given application. In addition, it provides execution time statistics for the design space explorer. Figure 6 depicts the tools included in the software subsystem together with their inputs and outputs. [6]

The starting point for the code generation is a HLL code specifying the application. It can be written in either C or C++. The compiler front-end, based on the GCC, converts the HLL code into sequential MOVE code where each data transport is performed after another one. The sequential code can be passed to the sequential simulator which provides profiling data of the application as well as statistics such as number of cycles and moves, most used operations and data transports, and utilization of immediate values of each bit width.

The compiler back-end is the most complex tool in the MOVE framework generating the parallel MOVE code from the sequential one. It maps the functionality of the application onto the resources described in the architecture description. The complexity is caused by the optimizations performed by the back-end to obtain an efficient and small parallel code. An optional input, i.e., the execution profile generated by the sequential simulator, assists the back-end to optimize the code even better.

After the compiler back-end has produced the parallel MOVE code, the parallel simulator may be invoked to verify the functionality of the generated code. In addition, the simulator is useful for the designer to obtain different statistics about the parallel code such as execution time, number of instructions and moves in the code, and utilization

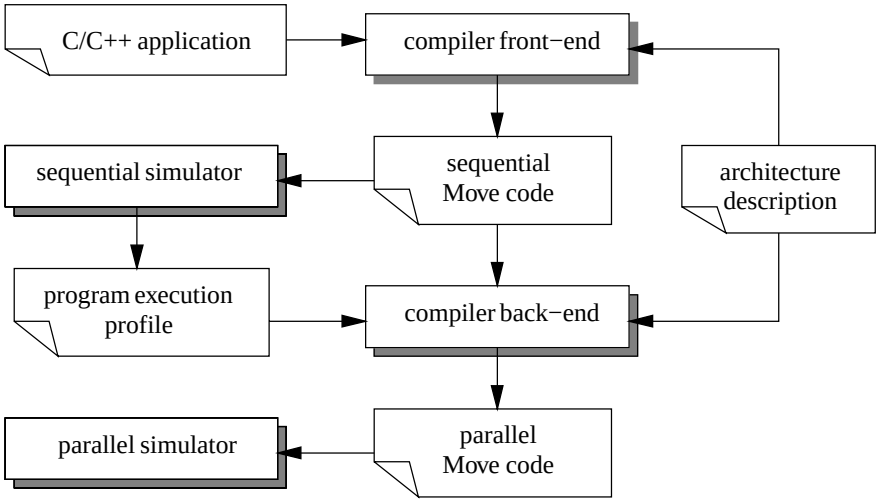

Figure 6. *The software subsystem of the MOVE framework.*

of different hardware resources. The statistics are used by the design space explorer as well.

2.2.2 Hardware Subsystem

The hardware subsystem is composed of the hardware cost estimator and processor generator. The purpose of the cost estimator, described in more detailed in Section 2.4, is to evaluate the processor in terms of area, power, and timing.

The processor generator produces a synthesizable very high speed integrated circuit hardware description language (VHDL) [7] code from the architecture description. The processor generator requires VHDL descriptions of the user-defined FUs used in the target processor as well as properties of the transport buses. The generated VHDL code can be further processed using any available commercial tool performing logic synthesis to obtain the chip layout of the target processor. Detailed discussion about the MOVE processor generator exists in [8] and [9].

2.3 Design Space Explorer

The design process of application-specific processors (ASP) is iterative requiring several modifications to the target processor before it meets the requirements of the system. In addition to the performed modifications, an iteration step includes evaluation of the processor configuration. Performing the iterations manually takes far too long if the possible design space has to be gathered extensively. In the MOVE framework,

the design space explorer searches automatically for the best possible processor configuration in terms of execution time, chip area, and energy consumption. Thus, the explorer is a huge help for the designers to quickly conduct them into the interesting area of the large design space.

The design space exploration process is composed of two independent phases: resource optimization and connectivity optimization. First, the resources of the target processor configuration are optimized by varying the number of different resources. Thereafter, the interconnection network, i.e., the connections between buses and sockets are optimized by adding and removing them. The only motivation for having separated exploration phases is practical: combined exploration of resources and connectivity would entail a prohibitively high number of evaluation steps. The resource optimization is described in Section 2.3.1 whereas the connectivity optimization is represented in Section 2.3.2. [4]

The design space explorer evaluates the processor configuration after each modification to obtain feedback about the appropriateness of the modification, i.e., to know if the modification should be revoked or not. The evaluation is done by utilizing the software and hardware subsystems of the MOVE framework illustrated in Fig. 5. The software subsystem is used to compile and simulate the given application for the target processor. Hence, the execution time of the application is obtained. The hardware subsystem, more specifically the hardware cost estimator, evaluates the given processor configuration in terms of chip area, power consumption, and timing. The original estimator of the MOVE framework is described in Section 2.4.

2.3.1 Resource Optimization

The purpose of the resource optimization phase is to find the best possible combination of different hardware resources for a given application. Since the design space is infinite and multi-dimensional, the explorer cannot consider all the aspects affecting on the goodness of the target processor configuration [4]. Only a subspace of the three-dimensional (3-D) design space is interesting for the designers. This subspace consists of the essential design points, i.e., *Pareto points*. A design point is a 3-D Pareto point if no other design point is better in all three properties (execution time, area, energy) accommodated by the explorer [4]. Pareto point concept is illustrated in two dimensions in Fig. 7 meanwhile Fig. 8 depicts the resulting Pareto points achieved from the resource optimization. However, they do not seem to be Pareto points in terms of area and execution time. If the third dimension, i.e., energy consumption, is

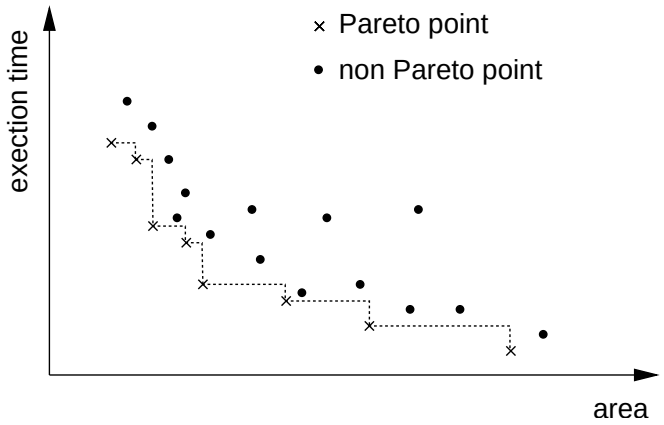

Figure 7. *Pareto point concept.*

accommodated, all the design points shown in Fig. 8 are Pareto points in 3-D design space. The most interesting design alternatives are usually those lying the nearest from the origin. Thus, some of them are selected for further processing in the connectivity optimization.

The explorer modifies four different properties of a target processor during the resource optimization: FUs, buses, sizes of the RFs, and RF ports. A modification focusing on the FUs always adds or removes an entire FU together with the sockets connected to it. Instead, buses are removed decrementally by reducing its width in steps from 64 bits to 32 and further to 1 and 0 bits. A null bit width means, of course, that the bus is removed from the configuration. Addition of resources proceeds in other direction. Registers are removed/added in groups such that the number of registers per register file is always 0, 2^n or $2^n + 2^{n+1}$, where $n \geq 0$. RF ports are easy to manage since they are just added or removed independently from other resources. [4]

Local Search Algorithm

The Pareto points are found using a local search algorithm [4]. However, the resource optimization starts from an oversized processor configuration by removing useless resources. A resource is considered useless if a processor configuration with unchanged cycle count is obtained after removing that specific resource. Since these resources do not give any contribution to the performance, they will not be considered anymore during the rest of the exploration. Thereafter, the explorer starts to search for the most cost-efficient configurations of the design space by utilizing the local search algorithm.

The explorer requires a basis to determine which of the two configurations is better. Thus, a formula combining the characteristics and quantities of a configuration in one

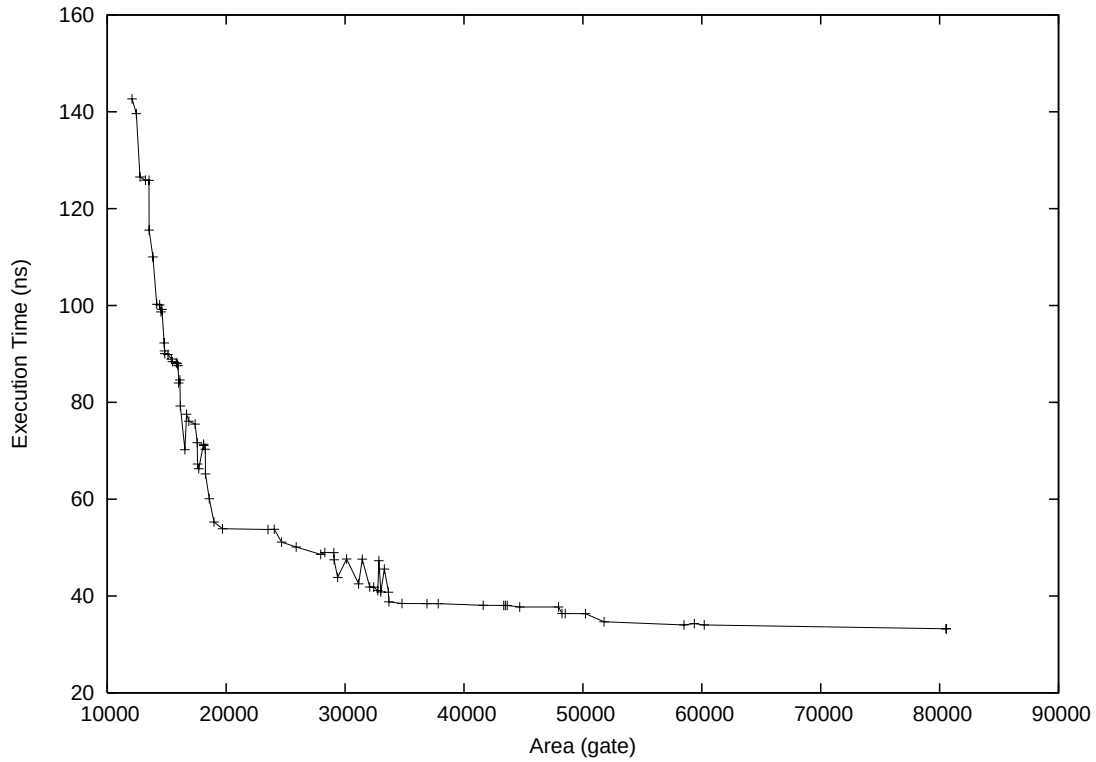

Figure 8. *The most interesting design points found by the resource optimization.*

number, which represents its "quality", is defined as

$$quality = \left(\left(\frac{t_0}{t} \right)^{\alpha} \cdot \left(\frac{A_0}{A} \right)^{\beta} \right)^{\frac{1}{\alpha+\beta}}, \quad (1)$$

where α and β are constants, usually in range from 1 to 5, that are reflecting the importance of execution time and area, respectively, t_0 and A_0 are the execution time and the area of the initial processor configuration, while t and A are the execution time and the area of the processor configuration currently being evaluated.

The local search algorithm proceeds by alternating two type of search phases, called reduce and extend phases. In a reduce phase, resources are removed one by one from the target processor configuration until a minimum configuration is found. A target processor configuration is considered "minimal" if the compiler fails to map the data transports specified by the application onto the target processors obtained by removing any resource from the "minimal" configuration. The process is reversed during the subsequent extend phase: removed resources are put back into the target processor configuration in a different order until the initial configuration is reached again.

Both extend and reduce phases are performed five times during the local search algorithm with different quality functions. The values of α and β exponents that determine

the quality function used in reduce and extend phases are varied according to the following set of (α, β) 2-tuples:

reduce: {(1,1), (1.5,1), (2,1), (2.5,1), (3,1)}
extend: {(1,1), (1,1.5), (1,2), (1,2.5), (1,3)}

Due to different quality functions, the explorer moves in the different area of the design space each time an extend or reduce phase is performed since the area and execution time has different weighting in the function.

Each time a hardware resource is removed from the target processor, the local search algorithm evaluates the design points that lie in the neighbourhood of the current design point, i.e., they can be obtained from the current design point by removing a resource. Thereafter, one of them is selected and the process is repeated with the selected point. Which resource is removed is determined by the quality function. In the *first fit* technique, the first resource which yields a target processor configuration with a higher quality than the current configuration is chosen. If no such resource can be found, the resource that gives the best possible quality is removed. In the *best fit* technique, the resource that gives the best possible quality is removed always. Similar procedure is used while adding resources.

Backtracking

The backtracking search algorithm is an optional extension of the local search algorithm. The backtracking phase starts after the local search phase is completed and it explores a more extensive neighbourhood of the Pareto configurations found by the local search algorithm and often finds several new Pareto processor configurations. However, the exploration time increases approximately two to five times when the backtracking algorithm is used.

The neighbourhood of the initial Pareto configuration is searched by a recursive algorithm sketched in Alg. 1. It evaluates all possible processor configurations obtained by removing up to $depth\_limit$ hardware resources from the initial Pareto configuration where $depth\_limit$ is a constant indicating the backtracking depth limit. Whenever a new Pareto configuration is found, its neighbourhood is also searched.

At each step, one of the hardware resources is removed from the initial Pareto configuration. If the obtained configuration is not a new Pareto configuration, the algorithm

Algorithm 1 ReduceResourcesBT(*depth*)

```
if depth = 0 then
    return
end if
for all hardware resources r in the processor configuration C do
    remove r from C
    if C is a Pareto configuration then
        ReduceResourcesBT(depth_limit)
    else
        ReduceResourcesBT(depth-1)
    end if
    put r back in C
end for
```

recursively removes another resource from the newly found configuration. This process continues until one of two conditions is met:

- the number of steps exceeds the backtracking depth limit
- the obtained configuration is a new Pareto point.

In the first case, the algorithm backtracks one recursion level, to the configuration obtained before the last reduce step, and then tries removing another resource from it. In the second case, the same procedure is repeated all over again on the obtained configuration.

2.3.2 Connectivity Optimization

The resource optimization is definitely the most important part of the design space explorer. However, the connectivity optimization is also essential part of the design process when a cost-efficient target processor is designed for a given application. By reducing connections between buses and sockets, the capacitive load of the buses is reduced, which may shorten the critical path of the overall system and, therefore, increase the maximum clock frequency at which the target processor can run. In addition, removing connections also results in smaller area due to simpler (de)multiplexers in the sockets. Furthermore, the instruction size may decrease since the number of addressable locations per bus is lower. [4]

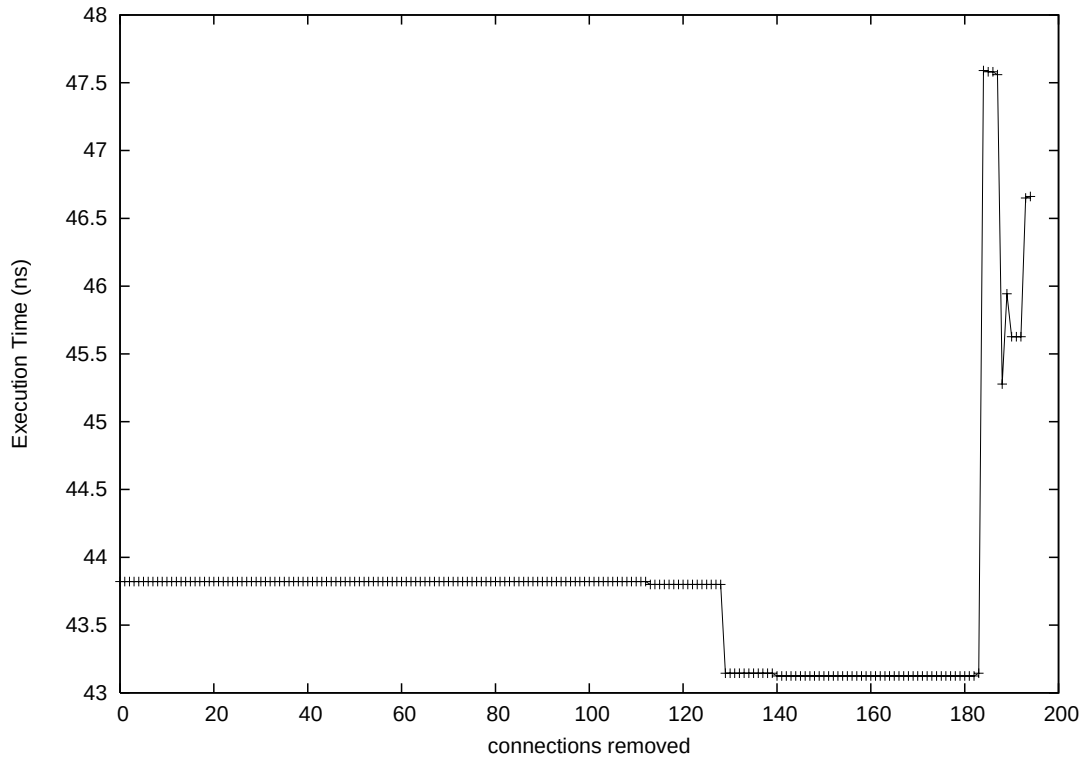

Figure 9. *The execution time in terms of removed connections in the connectivity optimization.*

The connections are reduced in a round robin fashion. The connection that will be removed from the bus is the first connection that has no influence on the cycle count. If no such connection exists, the connection with the lowest influence on the cycle count will be taken. This process is repeated until the execution time starts to increase. Figure 9 depicts the behavior of the execution time in function of number of removed connections. Selecting the fastest design alternative, where the most of the connections are removed, gives the best cost-efficiency. However, the programmability of such a specific processor configuration is poor indicating that even a slight change or bug fix to the application might turn the processor useless. Thus, selection of the processor from the alternatives given by the connectivity optimization is a trade-off between programmability and cost-efficiency. [4]

2.4 Original Hardware Cost Estimator

The hardware cost estimator is responsible for evaluating the target processor in terms of chip area, power consumption, and timing for a given application. The results are mostly used in the design space explorer to compare these statistics with other configurations. Thus, the explorer is able to guide the process of finding the most appropriate

configuration for the given application. In addition, the cost estimator may be used to obtain quickly the suggestive costs of a processor configuration.

The estimation procedure of the original MOVE estimator included FUs, RFs, buses, input and output sockets as well as instruction memories. However, the control logic was not taken into account separately but it was accommodated in the costs of other resources. Moreover, the area and timing were evaluated but the power consumption analysis was not supported. [10]

The original MOVE estimator was based on the modeling of each hardware resource. Mathematical functions representing area and timing characterics of the architectural resources are derived through experimental data about the behavior of the TTA resources. The equations depend on the architecture parameters as well as on the technology characteristics. The equations can be found in [10] with detailed reasoning.

Nevertheless, the properties of the original MOVE estimator described in Section 2.4 were not completely satisfactory. It had a lack of flexibility in form of changing the technology, which was hard-coded into the source code. The original model consisted of several technology characteristics as well as coefficients in the equations that were hard-coded in the source code. In addition, some of the technology characteristics were too detailed, for example, area of a tri-state buffer, output capacitance of a tri-state buffer, and time constant of a flip-flop output. Thus, the buses must be formed out of tri-state buffers and the implementation of the interconnection network could not have changed. Technology characteristics and the coefficients of the model could have been moved into a file to avoid recompilation when the technology changes. Thus, the technology could have been changed by modifying these variables. However, modifications of the source code would have been needed if a new technology characteristic had been adopted into the estimation or if the model had changed even slightly.

3. COST ESTIMATION

The purpose of the hardware cost estimator is to evaluate the target TTA processor in terms of area, power consumption, and timing. The area indicates the chip area taken by the processor configuration from the silicon when implemented. The power consumption means the power consumed by the processor for the execution of a given application. The maximum clock frequency supported by the processor configuration is characterized by the timing evaluation. The estimation of the mentioned properties has to be performed by utilizing a procedure which is automated, accurate enough, quick, and technology independent. The best possible accuracy could be obtained by performing logic synthesis together with gate-level simulation. However, the estimator is an important tool in the design space exploration process as depicted in Fig. 10. The explorer evaluates hundreds, or even thousands of processor configurations using the hardware cost estimator. Thus, as logic synthesis takes unacceptably long span of time, it is not an alternative to be used as the estimation procedure. Due to the high speed requirements, the accuracy has to be compromised.

Technology independency is required from the estimation process in order to be flexible for the user to try different technologies. Thus, changing the technology should be made as easy as possible. In addition, the estimation procedure must be automated so that the design space explorer can take advantage of the estimator to obtain the costs of the target processors.

Alternative estimation methods are described in Section 3.1. High-level principles of the proposed estimation procedure is presented in Section 3.2. Section 3.3 discusses the information on which the estimation procedure is based. The evaluation of the target processor area is described in Section 3.4 whereas Section 3.5 introduces the power consumption evaluation. The timing estimation is represented in Section 3.6.

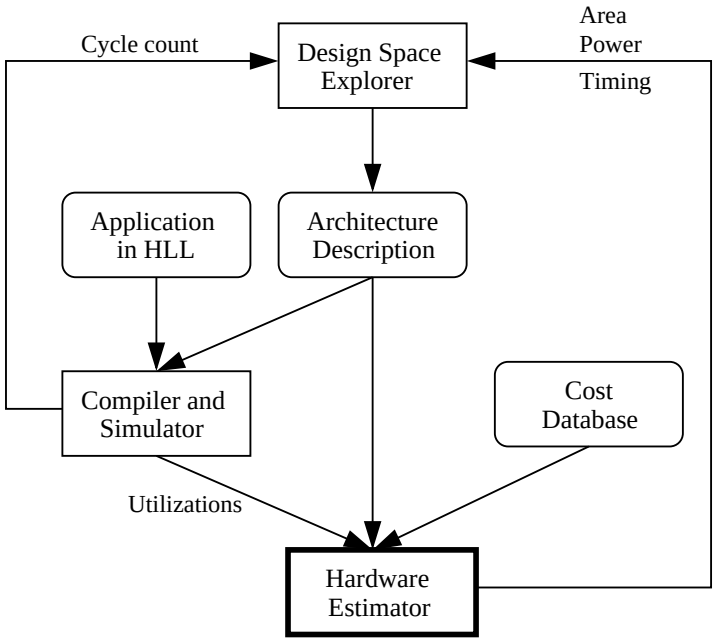

Figure 10. *The role of the hardware cost estimator in the design space explorer.*

3.1 Alternative Estimation Methods

Extensive research efforts have been used on estimation methods but a fast, accurate enough, automated method does not exist. A common approach for the estimation is physical modeling where specific formulas evaluate the costs of a processor. The formulas are usually depending on the technology characteristics from different libraries as well as some metrics about the architecture. The problem of these methods in larger architecture domains seems to be the lack of accuracy. However, the methods are usually extremely rapid. In [11], several methods of physical modeling, such as Rent's rule and Donath's wiring model, have been applied for evaluating the area and delay of the protocol processor architectures. The method was elaborated in [12] by probing experimentally the Rent's exponents for each hardware resource in the processor and applying linear approximation. The area accuracy is in the range of 10% to 40%. In [13], the costs of the units are taken from a library while the interconnection is evaluated premised on the average area of each entity in the interconnection network. An interesting and completely different approach based on *the order of superiority* of the processor configurations is proposed in [14]. The absolute values cannot be obtained from the cost estimation procedure but different architectures can be arranged into the order of superiority, which is derived from the formula of the *fidelity* describing the goodness of the architecture. In principle, the explorer does not need to know the absolute costs but it is only interested in the relative costs of the design alternatives.

3.2 Principles of Proposed Hardware Cost Estimator

The proposed estimation procedure is based on a priori information on area, power, and timing statistics of the hardware resources. First, each hardware resource is characterized on a given technology. Secondly, the area and power of a target processor are evaluated based on this information.

The estimator requires information about the used technology with which the target processor is going to be implemented. Circuit manufacturers provide information on the physical characteristics of their processes and devices in various formats and different abstraction levels. It would be too difficult to support all these formats in the estimator. Instead, a description at a higher abstraction level is used to provide the physical information of the target technology. This description is structured as a database, henceforth to be called the *cost database*. Since the cost database includes area, power and timing statistics of the hardware resources, estimation procedure does not need to take the used technology into account. In this approach, technology dependency lies in the cost database, to be exact, in the creation of the database. Thus, it can be easily changed.

The role of the hardware cost estimator in the design space explorer as well as its inputs and outputs are depicted in Fig. 10. The estimator takes as an input the cost database and the architectural description of the processor to be evaluated. In addition, the utilization statistics is obtained from the simulator to achieve more accurate power evaluation. As an output, the estimator provides an evaluation of the processor configuration for a given application in terms of chip area, power consumption and timing. The output of the estimator is mostly used by the design space explorer to guide a semi-automatic design process but the estimator may be invoked independently from the explorer to obtain suggestive statistics.

The estimator accommodates FUs, RFs, buses, and input and output sockets in the evaluation as well as the control logic. Control logic includes, in addition to the control signals, registers at least for the program counter, return address, short and long immediates, boolean values and instruction word. Memories, however, are excluded from the processor evaluation.

Table 1. *Properties characterizing the hardware resources.*

Resource	**Characterized by**
Function unit	supported operations, bit width, cycle time, latency
Register file	size, read ports, write ports, bit width, cycle time
Bus	fanin, bit width, cycle time
Input socket	fanin, bit width, cycle time
Output socket	fanout, cycle time
Control logic	density of the interconnection network

3.3 Cost Database

The cost database includes area, power, and timing statistics of the hardware resources existing in the TTA target processors. Each hardware resource is characterized by some of its properties as illustrated in Table 1. The cost database includes statistics of the resources for different characteristics. Since power consumption depends on the utilization of a hardware resource piecewise linearly, power values need to be given for different utilizations. One expression of the statistics and characteristics forms a database entry. For each resource of the TTA, i.e., FUs, RFs, buses, and input and output sockets, several entries exist in the database. In addition, the database contains one entry for the control logic. During the creation of the database, area, power, and timing of the resource have to be identified for different combinations of the characteristics mentioned in Table 1.

The creation of the cost database can be done by utilizing the preferred way of the designer, e.g., the designer may know some of the statistics of his hardware resources. Thus, the statistics can be written by hand into the database. However, utilizing any commercially available synthesis tool is recommended. In addition, the creation should be automated in a way or another to easily recreate the database or to add comfortably new entries.

While the estimator requests entries from the database to evaluate the costs of a target processor, flexibility should be supported. The database should manage queries where the perfect match for a certain hardware resource does not exist in the database. Perfect match means that each characteristic of a hardware resource is equal with the database entry. Statistics of other entries can be utilized in the evaluation if the behavior of the resource supports that. The following query types, usually called as match types, are supported for each characteristic:

- exact match: matches if equal characteristic is found from the database
- superset: matches if greater characteristic or a superset is found from the database
- subset: matches if smaller characteristic or a subset is found from the database
- interpolation: matches if smaller and greater characteristic is found from the database; then linear approximation is used for calculating new statistics for the new database entry.

Nevertheless, some match types cannot be used for certain characteristics due to data types which represent their values. For example, interpolation cannot be used for sets.

Basically, the format of the database may be anything, e.g., a convinient, human readable text file or very specific, small binary file. The proposed estimator uses currently a textual database, which is created utilizing Synopsys Design Compiler [15]. The database contains multiple entries for each hardware resource type of the TTA. An example database is composed of a couple of entries for each resource type as represented in Appendix A. Each entry has the following fields representing the statistics:

area Area of an entry. The area unit (mm^2, equivalent gates) can be freely chosen, on condition that the same unit is used throughout the database.

delay Critical path of an entry, in nanoseconds. The critical path is defined as the maximum delay inside an entry. In pipelined FUs it means the maximum delay between any two adjacent pipeline stages.

power Total power consumption of an entry given as multiple (utilization, power) pairs. The utilization here is the fraction of the total cycle count where the entry is used. A model for power consumption as a function of utilization can be constructed by means of piecewise linear interpolation.

In addition to the statistics, an entry contains the properties characterizing the resource. A common characteristic for all entries, i.e., cycle time, is described as follows:

clk Length of clock period for which the block is optimized, in nanoseconds. Using this characteristic, it is possible to distinguish implementations of the same entry designed for high speed (large area and power) or for small area (low speed). The queries support subset match type for this characteristic.

FU entries include, in addition to the common property, the following characteristics:

oper Set of operations the function unit can perform. The queries support superset match type for this characteristic.

data Bit width of the FU, i.e., the width of the widest operand in the FU. The queries support interpolation match type for this characteristic.

latency Latency of the operations supported by the FU. The queries support only exact match for this characteristic.

RF entries accept the following characteristics:

size Number of registers in the RF. The queries support interpolation match type for this characteristic.

rd Number of read ports. The queries support interpolation match type for this characteristic.

wr Number of write ports. The queries support interpolation match type for this characteristic.

data Bit width of the RF, i.e., the width of one register in the RF. The queries support interpolation match type for this characteristic.

Bus entries have the following characteristics:

fanin Number of FU or RF output sockets connected to the bus. In addition, one connection is added if the bus supports short immediates, since it can be considered as an output socket. The queries support interpolation match type for this characteristic.

data Bit width of the bus. The queries support interpolation match type for this characteristic.

An input socket entry is composed of the characteristics as follows:

fanin Number of buses the input socket is connected to. The queries support interpolation match type for this characteristic.

data Bit width of the input socket. The queries support interpolation match type for this characteristic.

The characteristic included in the database for the output socket entries is as follows:

fanout Number of buses the output socket is connected to. The queries support interpolation match type for this characteristic.

The information about the control logic in the database is different from other entry types. It consists of the two declarations having statistics as follows:

connectivity Relative connectivity of the transport network. It is computed as the ratio between the number of connections in the interconnection network divided by the number of connections in the fully connected version of the same interconnection network.

area Area of one register element in the control logic for the given connectivity.

power Power consumption of one register element in the control for the given connectivity.

3.4 Area Estimation

The total area of the processor configuration is obtained by the sum of the area of each hardware resource as follows

$$A = \sum A_{FU} + \sum A_{RF} + \sum A_{bus} + \sum A_{insock} + \sum A_{outsock} + A_{cntrl} \quad (2)$$

Area of a specific resource is obtained by querying the corresponding entry from the database.

However, characterization is not straightforward in case of interconnection elements, i.e., buses and input and output sockets. A data transport from one register to another takes one cycle. During that time, the data passes an output socket, a bus and an input socket. Thus, each interconnection resource is not utilized throughout the entire clock cycle. The following assumptions are made of the time, which a certain resource is active from the whole clock cycle:

- bus is utilized 25% of the clock cycle

- input socket is utilized 30% of the clock cycle
- output socket is utilized 30% of the clock cycle
- buffers between the interconnection elements are utilized 15% of the clock cycle.

Analysis of example TTA processors indicated that the area and power of the buffers are negligible. Thus, they are ignored in the evaluation process.

Slightly different approach is used for the evaluation of the output sockets, for which each bit line is characterized in terms of its fanout and cycle time since the bit width cannot be assigned to the output sockets. The total area of an output socket is achieved by adding up the area of each bit line as follows

$$A_{outsock} = \sum_{i=n}^{1} b_i A_i \tag{3}$$

where n is the number of buses in the target processor, A_i is the area of a bit line driving i buses, and b_i is the number of bit lines driving i buses. An output socket which is driving four buses of various bit widths is depicted in Fig. 11. Its total area is

$$A = 8A_4 + 8A_3 + 0A_2 + 16A_1 \tag{4}$$

The first eight bit lines of the output socket are driving all of the four buses since each of the buses is composed of eight or more bits. Due to the fact that the bit width of three buses, i.e., namely Bus2, Bus3 and Bus4 is more than eight and at least 16, next eight bit lines drive these three buses. The upper 16 bit lines of the output socket are driving only one bus, i.e., Bus4, since no other bus has more than 16 bits. Bit lines that are driving exactly two buses do not exist.

A completely different approach is used for the estimation of the control logic. Our approach is based on the number of registers in the control. The control logic is actually not characterized in the cost database but it is rather physically modeled to obtain the area of one register in the control. The model is based on the density of the interconnection network, which means the amount of connections from the maximum number of connections, i.e., all possible connections exist between buses and sockets. The total chip area taken by the control is

$$A_{control} = n_{regs} A_{reg} \tag{5}$$

where n_{regs} is the number of registers in the control and A_{reg} is the area of one register. The area of one register in the control is

$$A_{reg} = A_0 + dA_s \tag{6}$$

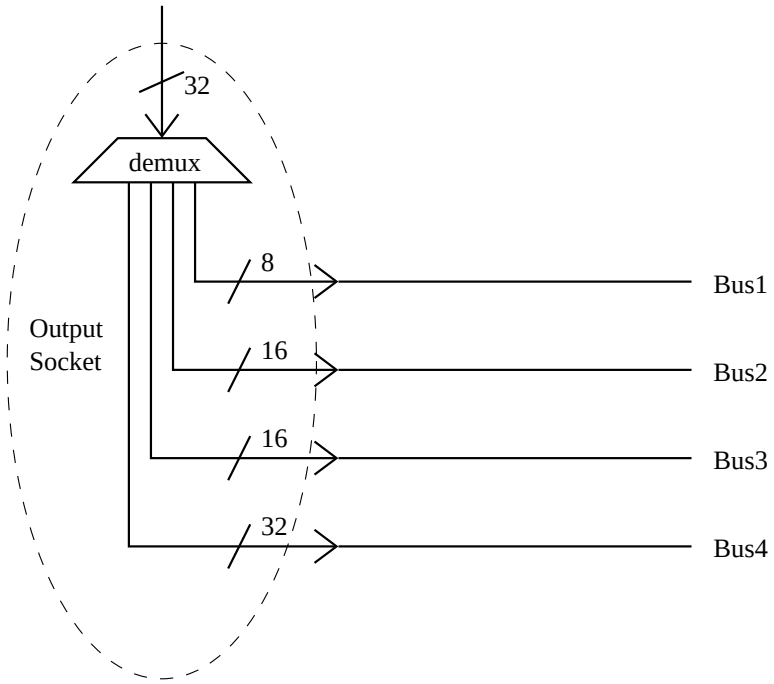

Figure 11. *An output socket driving four buses of various bit widths.*

where A_0 is the base area of one register in the control, d is the density of the interconnection network, and A_s is the slope of the area of one register in function of the density of the interconnection network. A_0 and A_s can be obtained from the cost database whereas d is a property of the target processor. The number of the registers in the control can be achieved by summing up all the registers caused by numerous properties of a target processor. Table 2 illustrates the number of registers caused by different processor elements to the control logic.

3.5 Power Estimation

The estimate for the total power consumption is

$$P = \sum P_{FU} + \sum P_{RF} + \sum P_{bus} + \sum P_{insock} + \sum P_{outsock} + P_{cntrl} \tag{7}$$

Thus, it is obtained utilizing the same principles as for the area estimation. In the beginning of the power evaluation, the application is compiled and simulated to obtain utilization statistics of the hardware resources. Contrary to the area, the final power of a resource cannot be obtained directly from the database but it must be approximated piecewise linearly from the utilization, and weighted according to the used cycle time as follows

$$P = \left(\frac{t_{db}}{t}\right)(P_0 + UP_s) \tag{8}$$

Table 2. *Number of registers in the control logic caused by the processor elements.*

Element	**Number of registers**
Program counter	$\log_2(number\ of\ instructions\ in\ application)$
Return address	$\log_2(number\ of\ instructions\ in\ application)$
Long immediate	$size\ of\ long\ immediate\ unit$
Short immediate	$length\ of\ short\ immediate + 1$
Input sockets	$\log_2(number\ of\ connections\ to\ buses) + 1$
Output sockets	$number\ of\ driven\ buses$
Boolean register	$number\ of\ boolean\ registers$
Register file	$\log_2(number\ of\ registers) \cdot numbers\ of\ writeports$
Function unit	$\log_2(number\ of\ supported\ operations) + 1$
Instruction register	$size\ of\ the\ instruction\ word$

where t is the cycle time utilized by the processor, t_{db} is the cycle time characterizing the resource, P_0 is the base power, i.e., the power with the closest possible utilization which is smaller than the requested utilization, U is the utilization, and P_s is the slope of the power in function of the utilization in the requested utilization range.

The power evaluation contains similar differences from the main estimation procedure as the area estimation. The interconnection elements are characterized differently from other resources. In addition, the power consumption of the output sockets is calculated differently. The power obtained using Eq. 8 gives the power of one bit line. Thereafter each bit line has to be added up to obtain the total power consumption of the output socket. Moreover, the power consumed by the control logic is obtained using the same model as for the area estimation.

3.6 Timing Estimation

Currently the cost estimator does not calculate the minimum cycle time. However, the designer can provide it as a parameter for the estimator. In the future, the timing evaluation should be included but it was not the first priority since the designer might actually have better knowledge of the desired cycle time. In addition, a really accurate timing evaluation is even more difficult than the estimation of the power consumption. Therefore, the inaccuracies would exist even more if the cycle time, which is used for characterizing the hardware resources, is not accurate in the database queries.

4. IMPLEMENTATION

The previously described estimation process was implemented in the MOVE framework. In other words, the hardware cost estimator of the MOVE framework was rewritten to achieve more accurate and flexible estimator.

The implementation of the estimator consists of the two main parts: the cost database and the application, i.e., client using the services provided by the database to calculate the costs of the processor configuration. The main focus in the rewrite process was to implement a flexible and expandable cost database without ignoring the efficiency. Expandability means that the following additions are easy and straightforward to implement to the database:

- new entry type to the database
- new field type to a database entry
- new data type to an entry field
- new statistic element.

In addition, the queries should support different match types, i.e., exact match, superset, subset and interpolation, as well as the following modifications:

- adding a new match type
- implementing a completely new search algorithm.

The flexibility and expandability requirements mentioned above cannot conduct the database implementation into an inefficient solution. In as much as the design space explorer represented in Section 2.3 evaluates hundreds of processor configurations in one completion, the cost estimation must be followed through as quickly as possible.

High-level architecture of the hardware cost estimator is composed of the two modules: *CostDatabase* and *Application*. The *CostDatabase* module is described more specifically in Section 4.1 whereas Section 4.2 represents the *Application* module being responsible for the cost result calculations.

4.1 Cost Database

The cost database implements the services for storing the data and querying the entries from the database. It highly utilizes object-oriented programming paradigm [16] to implement required services. In addition, two design patterns are used, namely **Façade** and **Strategy** [18]. The cost database is composed of two modules and a few classes as depicted in Fig. 12. The *CostDatabase* class is in a central role in the module. It provides the services for the clients of the database system. An entry is stored utilizing the *Entry* module sketched in detail in Section 4.1.1. However, the most important service of the database is the *Search()* interface implementing the queries to the database exploiting the *Query* module. Section 4.1.2 describes the implementation of the queries, i.e., the class hierarchy to realize a flexible and expandable search algorithm. *CostDBReader* is a class for implementing the reader of the *CostDatabase* from a specific file utilizing lex and yacc [17].

4.1.1 Data Storage

The cost database needs to store the entries in a flexible and expandable way. The class hierarchy used for the data storage of the database that supports the requirements is depicted in Fig. 13. The *CostDatabase* class contains a map for storing different entry types. *InsertEntry()* function can be used to add an entry to the database.

CostDBEntry is the main class for representing a cost database entry. It consists of the

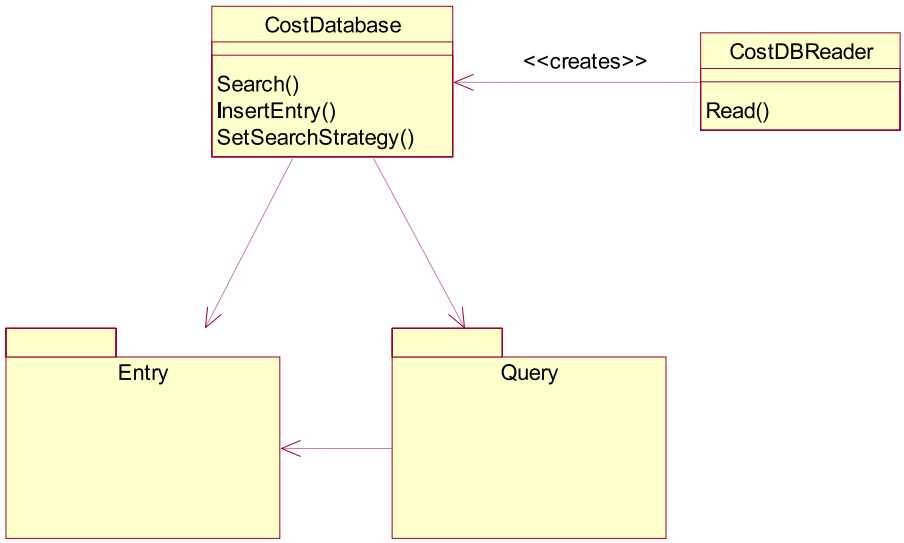

Figure 12. *Class diagram of the cost database.*

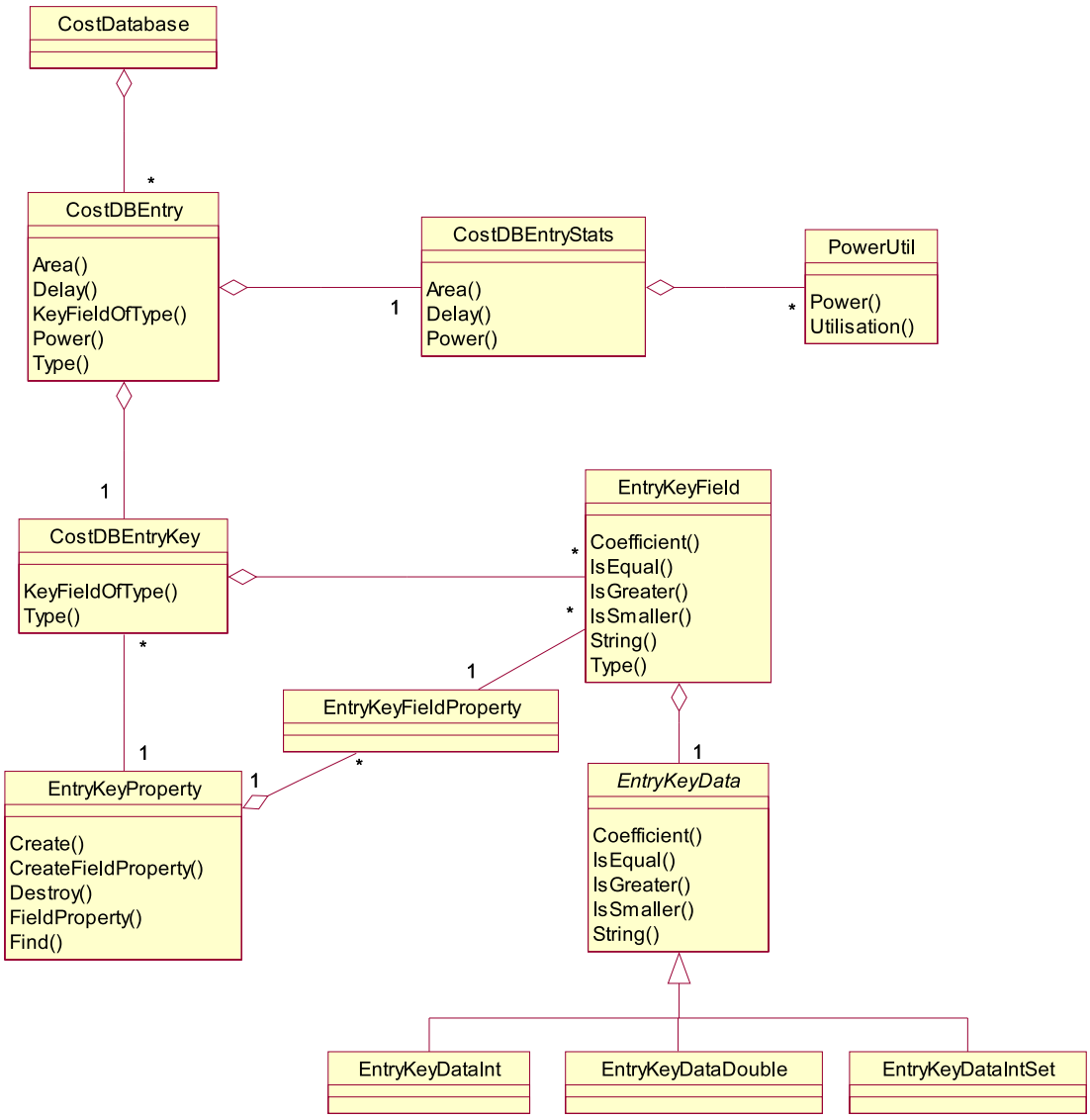

Figure 13. *Class diagram of the module storing the entries.*

search key and the statistics represented by the *CostDBEntryKey* and *CostDBEntryStats* classes, respectively. *CostDBEntryStats* class comprises the data for area, delay, and power consumption, which is represented as pairs of utilization and power. *CostDBEntryKey* incorporates the fields of an entry that characterizes it, i.e., search key. Nevertheless, the *CostDBEntry* is a *façade* for forwarding the interface calls to the complex subparts, which can be also used independently. This is one of the purposes of the **Façade** design pattern. The advantage is to group together the statistics and the search key of the database entry, and to still allow to design them separately. Being the façade, the clients of the database entries need only to manage *CostDBEntry* ob-

jects instead of its subparts, hence promoting weak coupling between an entry and its clients. The clients of the database entries do not need to be aware of the complex internal structure if they do not care about it. However, the **Façade** design pattern does not prevent the clients from using the subpart classes directly if they need to. In fact, this is required since the use of the search key is needed independently from the statistics in the client. An alternative for the **Façade** pattern would have been to compose an entry of each search key field and statistic. Thus, if an entry would have been used as a search key, the statistics would have always existed as unwarranted information. [18]

CostDBEntryKey, then, describes the properties of an entry that are used as a search key. Thus, it is composed of the type of an entry, and fields, i.e., *EntryKeyField* objects representing a search key field of an entry. From a *CostDBEntryKey* object, type and specific fields can be requested.

Each field of an entry key contains some data, which is represented by the *EntryKeyData* class. *EntryKeyField* class is composed of the data and the type of the field, which are both represented by own classes, namely *EntryKeyData* and *EntryKeyFieldProperty*, respectively. However, *EntryKeyData* is an abstract base class for different data types that the field can represent. It defines the interface that the data types should implement providing the search algorithms a way to compare different fields. Currently, the following data types are implemented:

- integer represented by *EntryKeyDataInt* class
- double represented by *EntryKeyDataDouble* class
- operation set represented by *EntryKeyDataIntSet* class.

The interface of the *EntryKeyData* provides the clients a flexible and generic way to handle the data. However, the actual data value cannot be obtained, which is a drawback. Fortunately, it is not a huge problem since the clients usually require little information about the actual data values, if at all. Nevertheless, if the client needs the data value itself, it can be obtained as a string and converted to the correct format.

Each database entry, as well as the fields of the entries have a type. They are represented by the classes *EntryKeyProperty* and *EntryKeyFieldProperty*, respectively. Class *EntryKeyProperty* contains static methods for creating and obtaining certain entry types. The class ensures that only one instance of a specific entry type exists. Hence, the comparison between the entry types can be done using pointers, which gives an important efficiency advantage. Each entry type contains the type element,

i.e., *EntryKeyFieldProperty* instance for each different field. The policy and the advantage with the field types are the same as with the entry types, i.e., the *EntryKeyProperty* class takes care of the fact that each field type has only one instance.

The design of the data storage part of the cost database represented above meets the flexibility and expandability requirements of the database. Adding a new type of entry or entry field does not require any changes to the database implementation, i.e., the client only has to create and use the new types. Adding a new type of data included in the entry field is extremely easy, i.e., the new class should be derived from the base class *EntryKeyData* and the whole interface of it should be overloaded.

4.1.2 Filtering Search

Generic search algorithm is one of the main requirements of the cost database. Since the structure of the database is not stable, the algorithm cannot depend on a specific field or entry types but it should be flexible and generic supporting any entry type the client is able to create. The *Query* module, for which a class diagram is depicted in Fig. 14, realizes such a search algorithm.

The genericness is accommodated already in the highest level where **Strategy** design pattern is applied. *CostDatabase* class contains a reference to the search strategy that is used for querying the entries from the database according to a specific search key. The reference points to an instance of the class derived from *SearchStrategy* which is an abstract base class defining the interface for all different search algorithms. Using the *SetSearchStrategy()* function of the *CostDatabase* object, the client can freely bring desired algorithm on line, i.e., dynamic changes of the algorithms are possible, which is one of the advantages of the **Strategy** pattern. Due to the **Strategy** pattern, unnecessary conditional statements are completely avoided. In addition, the pattern completely hides complex, algorithm-specific data structures from its clients. [18]

The responsibility of the *CostDatabase* in the queries is only to forward the search request, i.e., the search key and search type, to the active strategy. Moreover, the *CostDatabase* object adds the database entries for the *SearchStrategy* as the data from which to search certain entries. The type of the match applied to a specific entry field is denoted as *match type*. It is represented by class *MatchType*. The type of the whole search of an entry is called *search type*. It is composed of the list of match types, which, when correctly made, contains a match type for each field of an entry.

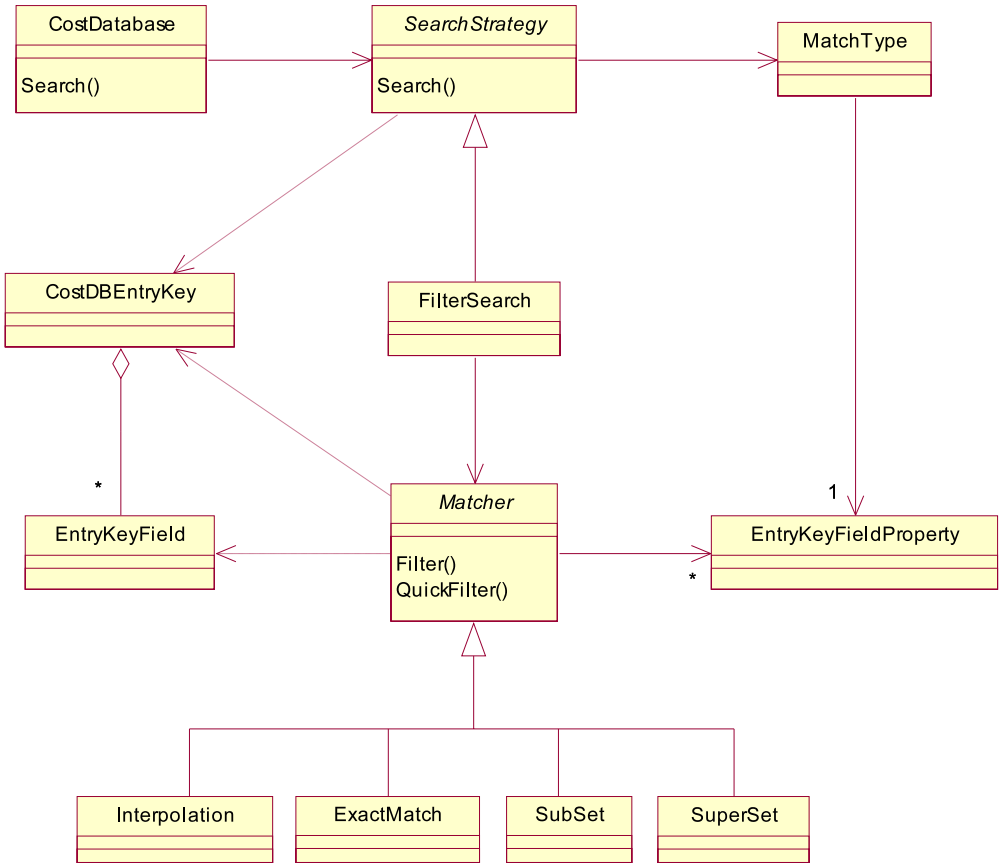

Figure 14. *Class diagram of the module implementing the queries to the database.*

Principles

One search strategy, class *FilterSearch* in Fig. 14, is implemented based on the filtering of unwanted entries out from the results. Entries that do not match with the search key in an entry field according to the used match type cannot be a match for the query, and hence can be removed from the results. Continuing this procedure for all fields results into an entry collection which matches the request. If more than one entry exist in the results, the client is responsible for enlarging on the collection of the appropriate entries further.

The division of the search to the smaller subparts is based on the **Strategy** pattern. *Matcher* is the abstract base class, i.e., the *strategy* defining the interface for the filters of single entry fields. Since the type of the filter must be flexibly changed, the **Strategy** design pattern is extremely appropriate for this context offering the possibility to change the concrete strategies dynamically [18]. For each *MatchType*, an own *Matcher* will be derived to implement specific type of filtering.

Algorithm 2 FilterSearch::Search(*search_key*, *entries*, *search_type*)

1: create *ML* (list of *Matcher*s) from the *search_type*
2: **for all** matchers *M* in the *ML* **do**
3: *M*.Filter(*search_key*, *entries*)
4: **end for**
5: **return** entries

The high-level algorithm of the filtering search is represented in Alg. 2. The search algorithm takes three parameters as input: the search key, i.e., the characteristics of an entry to be searched for, the list of entries from which to find the matches, and the search type. In the beginning of the algorithm, instances of the subclasses of *Matcher* are created according to the requested search type, which is indicated by the list of *MatchType* instances. Thereafter, the filtering of each *Matcher* object is applied to the list of database entries using the search key. The *Matcher* object itself knows the field into which apply the filtering. In the end of the represented algorithm, the entries contain only appropriate database items accepting the search criteria, i.e., search key with certain search type.

Subalgorithms

The cost database supports four different match types for a field: exact match, superset, subset and interpolation. Thus, own class derived from the base class *Matcher* exists for each of them to implement subparts of the filtering search. They implement a specific behaviour to filter out entries being inappropriate in a specific field according to the requested match type.

Queries requiring equal results for a specific field are using a match type called exact match. The algorithm for filtering according to exact match is represented in Alg. 3. It checks whether an entry has an equal value with the search key in a specific field or not.

Superset is a match type for querying greater field value or a superset of the search key for a specific field. The filtering algorithm is illustrated in Alg. 4. In the following, the algorithm is explained in detail:

- 1: Scroll through the whole input entry list.
- 4-6: If the entry is not equal or greater than the search key further processing is not required and the entry can be ignored.

Algorithm 3 ExactMatch::Filter(*search_key*, *entries*)

```
1: for all entries E in entries do
2:     E_field = E.KeyFieldOfType(Matcher::field_type)
3:     if E_field.IsEqual(search_key) then
4:         results.Add(E)
5:     end if
6: end for
7: entries = results
```

- 7: The results already found by the algorithm are browsed through.
- 8-10: The result entry is not handled furthermore if it does not belong to the *same group of entries*, i.e., has equal field values in each field except in the key field for which the superset algorithm is applied to.
- 12-13: If the current entry is smaller than the current result entry, the resulting entry can be deleted from the results list.
- 14-15: If the current entry is greater than the current result entry, the result collection already contains a better alternative than the current entry which will not be added to the results later on.
- 18-20: If the results do not contain an entry of this group, it will be added. The entry will be added also if it is the best entry in this group that is found so far.
- 22: Finally, the resulting entries are assigned into the output.

Subset search type is used for finding smaller field value or a subset of the search key for a specific entry field. The principles and the algorithm for subset filtering are the same as for the superset. However, *IsSmaller()* function calls are replaced with *IsGreater()* calls and vice versa.

The flexibility requirements for the database queries demand that linear approximation of the statistics should be possible to use if the behavior of a hardware resource for a certain characteristic is accordant with that. Therefore, interpolation match type, for which the filtering algorithm is depicted in Alg. 5, exists. The algorithm is much more complicated than for other search types due to the genericness requirements of the search algorithm and complexity of the interpolation.

The filtering algorithm in the interpolation is composed of two phases. In the first phase (lines 1-26), *entry pairs* composed of a smaller and a greater entry are constructed

Algorithm 4 SuperSet::Filter(*search_key*, *entries*)

```
 1: for all entries E1 in entries do
 2:     add_entry = true
 3:     E1_field = E1.KeyFieldOfType(Matcher::field_type)
 4:     if !(E1_field.IsEqual(search_key) or E1_field.IsGreater(search_key)) then
 5:         continue
 6:     end if
 7:     for all entries E2 in results do
 8:         if !OnlyThisFieldDiffers(Matcher::field_type, E2, E1) then
 9:             continue
10:         end if
11:         E2_field = E2.KeyFieldOfType( Matcher::field_type)
12:         if E1_field.IsSmaller(E2_field) then
13:             results.Delete(E2)
14:         else if E1_field.IsGreater(E2_field) then
15:             add_entry = false
16:         end if
17:     end for
18:     if add_entry then
19:         results.Add(E1)
20:     end if
21: end for
22: entries = results
```

from the entry list given as an input for the algorithm. Both the smaller and the greater belong to the same group of entries. The smaller entry in the pair embodies an entry which has a smaller value than the search key in the field for which the search is applied to. The greater entry has similar meaning, i.e., representing greater value. The smaller entry of a pair *P* is represented as *P*.smaller and the greater entry as *P*.greater in Alg. 5. In the second phase (lines 27-34), each pair is combined together to form one entry having linearly approximated area, power and timing statistics.

In the following, the whole interpolation algorithm represented in Alg. 5 is explained in detail line by line:

- 1-4: The creation of the entry pairs starts by scrolling through the whole input entry list. Inside that, the pairs already found are browsed through.

Algorithm 5 Interpolation::Filter(*search_key*, *entries*)

```
1:  for all entries E in entries do
2:      new_pair = true
3:      E_field = E.KeyFieldOfType(Matcher::field_type)
4:      for all pairs P in pairs do
5:          if !OnlyThisFieldDiffers(Matcher::field_type, P.smaller, E) then
6:              continue
7:          end if
8:          if E_field.IsEqual(search_key) then
9:              P.smaller = E
10:         else if E_field.IsSmaller(search_key) and E_field.IsGreater(P.smaller) then
11:             P.smaller = E
12:         else if E_field.IsGreater(search_key) and E_field.IsSmaller(P.greater) then
13:             P.greater = E
14:         end if
15:         new_pair = false
16:         break
17:     end for
18:     if new_pair then
19:         if E_field.IsEqual(search_key) or E_field.IsSmaller(search_key) then
20:             pair.smaller = E
21:         else
22:             pair.greater = E
23:         end if
24:         pairs.Add(pair)
25:     end if
26: end for
27: for all pairs P in pairs do
28:     if P.smaller.KeyFieldOfType(Matcher::field_type).IsEqual( search_key) then
29:         results.Add(P.smaller)
30:     else if P.HasBothElements() then
31:         results.Add(Combine(P.smaller, P.greater, search_key)
32:     end if
33: end for
34: entries = results
```

- 5-7: The pair is not handled furthermore if it does not belong to the same group of entries.
- 8-14: The current entry pair is updated if the current entry is better than the smaller or the greater entry in the pair.
- 15: If the algorithm got through the test on line 5, new entry pair is not required after the loop since a pair exists already for this group.
- 16: The loop can be finished, since only one pair exists for a certain group.
- 18-26: If an entry pair does not exist for the current entry, it has to be created. Current entry is assigned for the smaller or greater entry of the new pair, depending on whether it is smaller or greater than the search key.
- 27: Each pair that were found are scrolled through.
- 28-29: If the pair is composed of an entry that is equal to the search key, it is added to the results without any treatment.
- 30-31: If both smaller and greater entry exist in the pair, they will be combined into one entry. Otherwise, the pair is inappropriate and it will be ignored.
- 34: Finally, the resulting entries are assigned into the output.

The combination of the smaller and greater entry is made by linearly approximating area, power, and timing according to the difference of the entry fields from the search key. For example, if the bit width requested is 20 and the smaller entry has 16 as the bit width and the greater 32, the requested bit width is 25% of the total gap between the bit widths. If the area of the smaller entry is 200 gates and 350 for the greater, the requested are will be 237,5 gates. Going into the details, the function *EntryKeyData*::*Coefficient()* in the *Entry* module returns the coefficient required by the linear approximation of the area, power, and timing. In the example above, it would have returned 0.25.

Since the power values exist for different utilizations of a hardware component, the combination of the power is slightly more complicated. The requested entry contains power value for each utilization in the both smaller and the greater entry. For example, if one entry contains power values for utilizations 0,1; 0,6 and 0,9, and another one for 0,2 and 1,0, the entry obtained by combining these two entries would contain power data for utilizations 0,1; 0,2; 0,6; 0,9 and 1,0. Power for each utilization is obtained similarly to the area and timing.

Optimizations

The algorithm represented in Alg. 2 can be optimized to make the queries much more efficient. Two optimization methods have been used in the optimized version represented in Alg. 6: *cache* and *quick filtering*.

The TTA processors have several similarities. All the buses of the processor configuration are usually identical as well as RFs, and input and output sockets. FUs have more differences than the other resources since they support different operation sets. The purpose of the database is to provide statistics for the estimator, which evaluates the costs of a processor configuration. Since the processor is composed of several similar resources, the database encounters identical query requests. Thus, filtering search supports *cache*, i.e., results of the previous queries can be quickly used if identical search is requested [19]. In the algorithm depicted in Alg. 6, the cache appears in the beginning (lines 1-3), where the cache is checked whether it already contains the results for this query. In the end of the algorithm, i.e., on line 11, the results of the new query are added to the cache.

However, the size of the RF as well as the number of read and write ports of the RF may vary especially in the design space explorer. In addition, the fanin of the input sockets and fanout of the output sockets contains also some variations in the connectivity optimization of the explorer. Thus, the advantage of the cache cannot always be fully utilized. Nevertheless, it has a significant decreasing impact on the query times of the application.

The cost database usually consists of several hundreds, or even thousands of entries. Only a few of them satisfies the requirements of the search request, and most of them are completely inappropriate when quickly checking their properties. In addition, performing filtering of the most complex match types for the huge entry collections takes a quite long time. Due to these facts, an optimization called *quick filtering* is used for the database queries. It includes a filtering of unwanted entries out of the resulting entry collection before calling the actual filtering. In the Alg. 6, lines 5-7 depicts the usage of the quick filtering, which is performed for each *Matcher* instance before any of the actual filterings take place in line 9. The speed of a quick filter call must be in order of growth $O(n)$, where n is the number of entries passed to the quick filter algorithm. The filtering itself can be of any order of growth, since it must result in an entry collection satisfying the requirements of the search request.

The filtering algorithms represented earlier are divided into quick filtering and actual

Algorithm 6 FilterSearch::Search(*search_key*, *entries*, *search_type*)

```
1: if cache.Check(search_key, search_type) then
2:     return cache.Entries(search_key, search_type)
3: end if
4: create ML (list of Matchers) from the search_type
5: for all matchers M in the ML do
6:     M.QuickFilter(search_key, entries)
7: end for
8: for all matchers M in the ML do
9:     M.Filter(search_key, entries)
10: end for
11: cache.Add(search_key, search_type, entries)
12: return entries
```

filtering algorithms for the optimized version of the filtering search. The filtering of exact match is $O(n)$ in the order of growth. Thus, it can be moved to the quick filtering function as it is, and the actual filtering remains empty. In the interpolation, an entry cannot be removed from the results without having information about other entries. Hence, nothing can be done in the quick filtering phase and the filtering algorithm is the same as it is without quick filtering. Some filtering can be done for superset search in the quick filtering phase. Each entry containing equal or greater value in the field under interest must be left in the results and other entries can be removed as illustrated in Alg. 7. The algorithm does not work if it removes entries with smaller value from the results, since for some data types, such as sets, a value may be neither smaller nor greater than another value.

The filtering algorithm of the superset search can be slightly modified since the algorithm can trust that the quick filtering represented in the Alg. 7 is already run before the filtering itself is executed. Thus, each entry in the input is equal or greater than the search key and lines 4-6 of the superset algorithm represented in Alg. 4 can be removed. Furthermore, it can be noticed that comparison with the search key is not needed anymore, since each entry is equal or greater than the search key after the quick filter call. Similar changes can be done for subset search when quick filtering is used.

In general, the quick filtering speeds up the queries a lot, since the size of the entry collection passed to the filtering itself reduces significantly. Of course, in a very specific case the query time may increase because of quick filtering. For example, if an entry

Algorithm 7 SuperSet::QuickFilter(*search_key*, *entries*)

```
1: for all entries E in entries do
2:     E_field = E.KeyFieldOfType(Matcher::field_type)
3:     if E_field.IsEqual(search_key) or E_field.IsGreater(search_key) then
4:         results.Add(E)
5:     end if
6: end for
7: entries = results
```

having the greatest value of the whole database on a field for which subset match type is applied, is searched, the quick filtering algorithm of the subset is run and no entry is removed from the results.

Analysis

The filtering algorithm and its subalgorithms are extremely generic and flexible. They are completely independent of the entry types, field types of the entries and data types of the entry fields. Thus, these properties of the database can be varied by the client as much as is required and the search algorithm still works. As an example, after the implementation of the cost database was finished, a new entry type, i.e., output sockets had to be added to the database. The addition was fast, easy, and completely trivial due to the genericness of the database design. Output sockets consists of two fields, namely fanout and cycle time. Adding this data into the database did not had any effect on the *Query* module of the cost estimator. Changing the estimation process to support new entry type took only about one work day.

Due to the general interface of the data, the search algorithms can handle any kind of data. Thus, the database entries can contain any kind of data if it implements *EntryKeyData* interface. However, a filtering algorithm may require some new, search-specific interfaces. For example, interpolation requires *EntryKeyData*::*Coefficient()* function, which is not required by other filtering algorithms. Nevertheless, operation set type cannot implement this function, which implies that interpolation cannot be applied to the fields containing operation set type of data. If the client tries to use interpolation match type for operation set type of data, a run-time error will occur preventing the search algorithm from giving incorrect and meaningless results. On the other hand, the whole comparison interface of the *EntryKeyData* class is specific for the queries. *IsEqual()* function is required by the exact match filtering as well as the other match

types. In addition, *IsSmaller()* and *IsGreater()* functions are needed to make superset and subset filterings possible. However, these three functions can be considered as the basic set for making any kind of requests to the database.

The search type is specified by a list of *MatchType* objects. The supported match types are exact match, superset, subset and interpolation. Any combination of them can be used to form the search type. Particularly, multi-dimensional interpolation is flexibly possible. However, some client may be interested in any field values, i.e, to not filter any entries out of the results according to a specific field. This functionality can be obtained by using a special match type called *match all*.

The genericness and flexibility of the queries gives a huge advantage since the format of the database is not static. Nevertheless, it has always the drawback of being a bit inefficient. If the efficiency of the database queries becomes an issue, faster algorithms need to be considered. More speed can be achieved, for example, by making assumptions of the entry types or field types of the entries. The filtering search algorithm does not perform any checks for the values of other fields but the field under filtering. Adding more dependencies between the fields can give some efficiency advantage to the exclusion of genericness.

If the algorithms implemented do not satisfy some requirements, they can be easily changed. The whole filtering algorithm can be changed by deriving a new algorithm from the interface class *SearchStrategy* and by changing the client to use it. Moreover, adding a new match type is trivial. Filtering search can support the new match type by deriving a new class from the *Matcher* base class.

4.2 Application

The hardware cost estimator is responsible for calculating the costs of a processor configuration for a given application. The class diagram of the estimator is illustrated in Fig. 15. *Application* is a module which implements the actual evaluation utilizing the services provided by *CostDatabase* class. Since the estimator is integrated into the MOVE framework described in Chapter 2, the *Application* is implemented as an old C-style module taking advantage of the old code as much as possible. Thus, the structure of the old C-style estimator is not changed but it is only converted to use the new, flexible and extendable cost database. Implementing a flexible client for the database was not the purpose. Instead, the effort has been put on the estimation principles as well as on the design of the cost database.

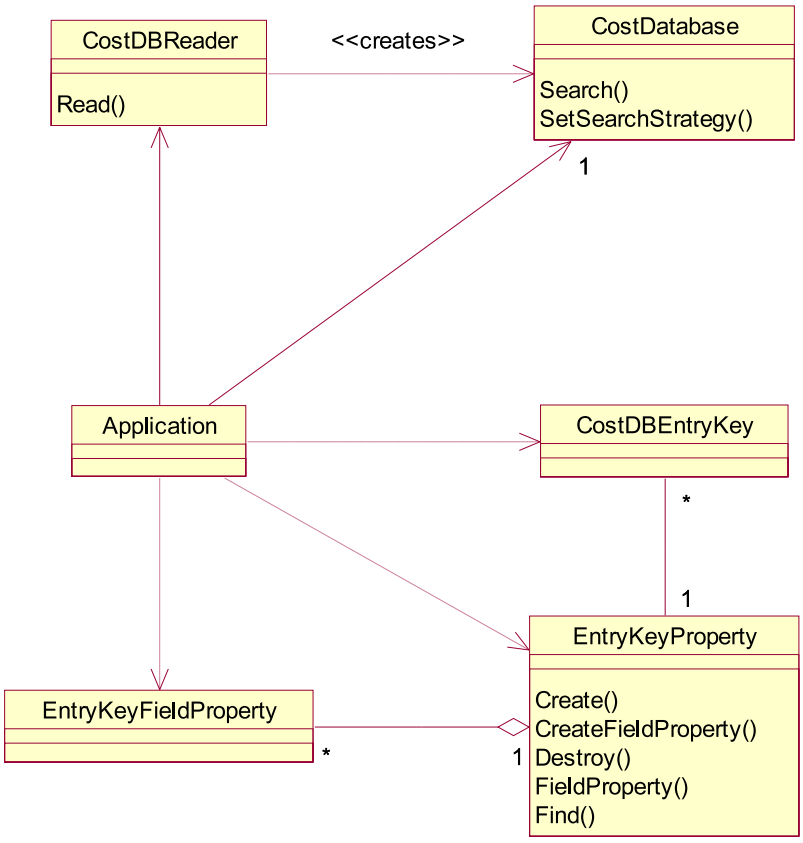

Figure 15. *Class diagram of the hardware cost estimator.*

The functionality implemented in the *Application* consists the following steps:

1. Read the processor configuration to be evaluated.
2. Create entry types, i.e., *EntryKeyProperty* instances, and entry field types, i.e., *EntryKeyFieldProperty* instances, for the cost database.
3. Read the cost database.
4. Create and assign a search strategy, i.e., *FilterSearch* instance, to the cost database.
5. Create the search type of the queries, i.e., *MatchType* list, for each entry type.
6. Compile and simulate the application to obtain execution time and utilization statistics of the hardware resources.
7. Evaluate the area and power of each hardware resource.
8. Sum up the costs of each resource.

9. Output the results.

Steps 2-5 are actions required by the cost database from its clients, before the database queries can properly take place. Steps 1-6 prepare the actual estimation which is performed in steps 7 and 8. Step 7 includes the evaluation of FUs, RFs, buses, input and output sockets, and control logic. An evaluation of a hardware resource can be divided into steps as follows:

1. Obtain the type of an entry, i.e., *EntryKeyProperty* instance, to be evaluated.
2. Create the search key, i.e., *CostDBEntryKey* instance containing *EntryKeyField* objects constructed as follows:
   (a) Obtain the type of an entry field, i.e., *EntryKeyFieldProperty* instance.
   (b) Create an instance of the appropriate class implementing the interface *EntryKeyData*.
   (c) Create *EntryKeyField* object.
3. Perform a database query.
4. Process the resulting entry collection furthermore, i.e., select the correct entry from the results if more than one exist.
5. Enlarge on the statistics further if required.
6. Store the results.

Steps 1 and 2 prepares for the database query performed in step 3. Steps 4-6 includes client-specific further processing of the results obtained from the database. In step 4, the client can freely choose an entry from the results if there are more than one entry. In the search algorithm, the entries are chosen based on the characteristics. However, the client can choose an entry from the results using any decent principle based on the characteristics or statistics. For example, an FU query might give as a result multiple entries which support different operation sets. For example, the smallest area or the smallest power consumption can be chosen depending on which properties are preferred.

5. PERFORMANCE EVALUATION

The evaluation principles of the proposed hardware cost estimator were represented in Chapter 3 together with the exact procedure to obtain evaluations on the chip area, power consumption and timing. Meanwhile, Chapter 4 described the implementation of the estimator.

The estimator is based on the database of the costs of different hardware resources. This approach has several advantages. Changing the technology is trivial since only a new database has to be created for the new technology. The actual estimator does not need to be modified at all. For example, the user might want to change the bus type from tri-state buffers to and-or gates. The statistics of the interconnection entries, i.e., buses, and input and output sockets have to be replaced with the new statistics representing the costs of these resources when the buses are composed of and-or gates. Similarly, if the user finds a more efficient implementation for an FU performing multiplication, only the statistics of the multiplier FU entries have to be replaced with the statistics of the new implementation. Furthermore, adding an application-specific FU into the target processor does not create more problems than addition of any other resource. The evaluation will handle a user-defined FU when the database includes the statistics for it.

This evaluation approach based on the cost database offers, in theory, possibilities for better accuracy than the alternatives based on mathematical modeling. The first phase in the mathematical modeling is to collect data about the behavior of the resources. Thereafter, equations are forced to fit into the data as well as possible. Inaccuracies always exist at some points of the model. In our database approach, the collected data is directly put into the database from which the estimator obtains it. Of course, the collected data cannot represent the values in all different cases, but they have potential for very accurate results overall.

However, the estimator cannot be considered as a significant alternative for the target processor evaluation without analyzing its properties. First, the accuracy of the results has to be verified which is done in Section 5.1. Secondly, the speed of the estimator

Table 3. *Hardware resources of the architectures for experiments.*

Resource	**A**	**B**	**C**
Function units	8	6	5
Register files	8	8	6
Registers	58	64	14
Bus	12	6	2

has to be proven to satisfy the efficiency requirements of the design space explorer. Section 5.2 describes efficiency analysis of the estimator.

5.1 Accuracy Analysis

The accuracy of the proposed estimator was verified by evaluating several processor configurations for one application, two-dimensional (2-D) 8×8 discrete cosine transform (DCT). First of all, each configuration was evaluated in terms of area and power using the Synopsys Design Compiler [15]. The clock frequency was 100MHz during the evaluation. For the power analysis, gate-level simulation was performed utilizing ModelSim [20] to obtain gate-level activities. Hence, the power analysis of the Design Compiler is more reliable. Thereafter, Synopsys Design Compiler was used to produce a cost database for the proposed estimator which was, finally, invoked to evaluate each configuration. The results of the Design Compiler were used as references against the results of our cost estimator.

Figures 16 and 17 illustrate the experimental results as proportions of the area/power of FUs, RFs, interconnection network (IC) containing buses, and input and output sockets, control logic (Cntrl), and other elements such as buffers between interconnection elements. The results are in couples, from which the left-hand result is the reference whereas the right-hand experiment utilizes the proposed hardware cost estimator. Three different architectures were evaluated using three different connectivities for each of them: large (L), medium (M) and small (S). Large connectivity indicates that the interconnection network is fully connected whereas small connectivity means minimum number of connections required for the compilation to succeed. Medium connectivity indicates that the number of connections between the buses and sockets is quite close in the middle of the number of connections of the large and small connectivities. The number of different hardware resources used in the evaluated architectures are represented in Table 3.

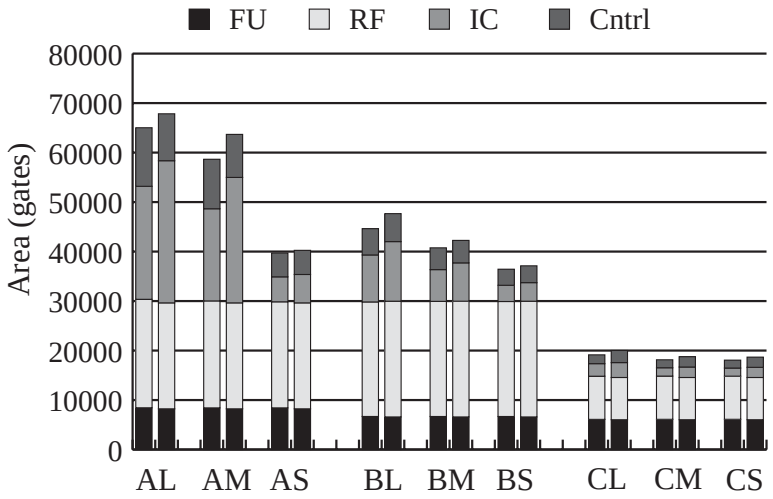

Figure 16. *Area of the references and the estimator. The reference is represented with the left-hand bar and the estimate with right-hand one.*

In the aggregate, the accuracy of the area results depicted in Fig. 16 are excellent since the average error is only 4,2% and the maximum error 8,6%. The evaluation of the FUs and RFs correspond to the references as well as the control logic. The only notable inaccuracy is caused by the interconnection network due to the problems in the database creation, i.e., the area of the buses in the database does not correspond to the synthesized bus area.

Power results are illustrated in Fig. 17. The power consumption of the FUs and RFs evidence satisfactory accuracy. Problems arise in the interconnection network as well as in the control logic. The estimator underrates the power consumption of the interconnection whereas the power of the control is overestimated a lot, due to the fact that activities of more fine-grained items than the database includes have significant effect on the results, and they are cumbrous to accommodate in the evaluation. In addition, perusing the results indicates that the power consumed by the buses includes the inaccuracy of the interconnection network due to the problems in the database creation. The impreciseness of the control logic estimates clearly indicates that our simple model does not work for the power estimation of the control. One problem are interconnection buffers that consume some extra power which is not taken into account in the estimation process. Thus, the average error in the total power is 16% whereas the maximum error is 27%.

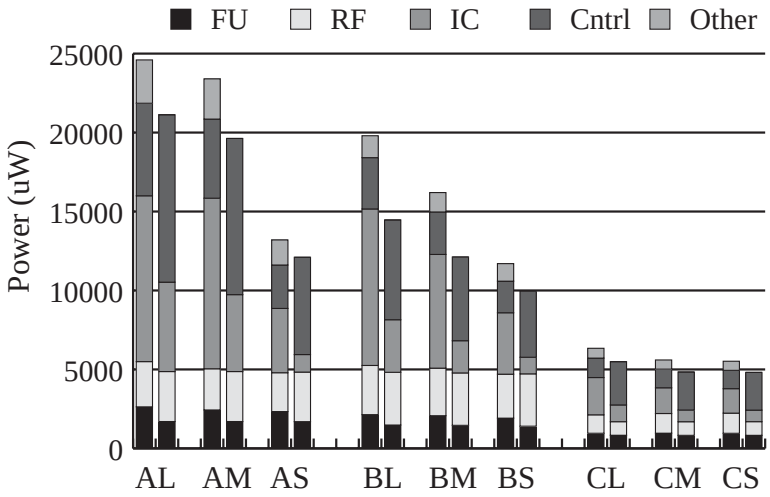

Figure 17. *Power consumption of the references and the estimator. The reference is represented with the left-hand bar and the estimate with right-hand one.*

5.2 Efficiency Analysis

The efficiency of the estimator was tested by evaluating a huge processor configuration for a given application utilizing an extensive cost database. The only significant inefficiency of the estimator can be caused by the database queries since other computation included in the estimator is trivial and fast. Thus, the efficiency analysis is focused on the database queries. In addition, the usefulness of the optimizations, i.e., cache and quick filtering, were verified.

The speed of the estimator mostly depends on the following issues related to the database queries:

- The number of entries in the cost database.
- The number of database queries, which depends directly on the number of hardware resources in the target processor to be evaluated.
- The speed of one query to the database.
- Internal structure of the search type, i.e., the order of match types in the list since the filtering of the fields will be done in that order.

The first issue cannot be optimized by the search algorithm implementation. The cache reduces the number of queries that has to be performed thoroughly, hence decreasing the time consumed for the second point. The third issue is optimized by quick filtering since it reduces the size of the entry collection from which to search using complex

filtering algorithms. In addition, quick filtering reduces the size the fourth problem as a side effect. Applying complex filtering algorithm on a huge entry collection is a time consuming step. This happens especially when the filtering is applied to the first field of an entry since the number of entries is not decreased at all. Thus, quick filtering reduces this problem by decreasing the size of the entry collection passed to the filtering of the first field. However, statistics about the cost database and the target processor configuration might be used to order the match types in an effective way inside the search type.

Section 5.2.1 describes the target processor configuration and the cost database used in the efficiency tests as well as the application. In Section 5.2.2, the results of the efficiency tests are represented.

5.2.1 Data Used in Efficiency Tests

A huge target processor configuration and an extensive cost database were used in the efficiency evaluations to obtain the worst case execution time of the estimator. Of course, the user may always have more entries in the database and the processor configuration may be more awkward from the estimator's point of view. The data used in the efficiency tests should still be as awkward as possible in this size of target processor category.

The benchmark used in the efficiency evaluations of the estimator was 8×8 DCT. However, the application used in the evaluations is an insignificant factor when verifying the efficiency of the estimator. Of course, the compilation and the simulation time of the application affects on the execution time of the estimator since they are required by the power consumption analysis of the estimator to obtain the utilizations of hardware resources. Nevertheless, in the experiments the compilation and the simulation times are excluded since the estimator cannot optimize them at all.

The processor configuration used in the efficiency evaluations was tried to make awkward from the database query's point of view. This is achieved by using hardware resources composed of different characteristics as much as possible. The configuration contains the resources represented in Table 4, which describes the number of hardware resources for each resource type together with the number of resources with different characteristics. In addition, the minimum number of different characteristics in the configuration is given. The number of resources in the minimum configuration is equal with the configuration used in the tests.

Table 4. *Statistics about the target processor configuration used in the tests.*

Resource	**Number of resources**	**Different resources**	**Diff. res. in Original**
FU	28	13	7
RF	8	5	1
Bus	20	20	1
Input socket	72	8	1
Output socket	44	16	1
Total	**172**	**62**	**11**

Without any optimization the number of required database queries is directly proportional to the number of resources. From the cache optimization's point of view the number of different characteristics is a significant factor, since it is directly proportional to the number of database queries required during the evaluation. In this target processor, this number is quite huge compared to normal case. For example, in the original processor configuration the interconnection network is fully-connected. Thus, each bus has equal number of connections as well as both input and output sockets.

The cost database was composed of several entries for each hardware resource type. Table 5 illustrates the number of entries included in the database for different resource types as well as the number of different values for each characteristic of them. The entries gather possible resources really extensively up to the processor configuration used in the efficiency tests. For example, buses have fanin from 0 to 45, the cycle time exist for 2ns, 3ns, 5ns, 7.5ns, and 10ns, and the bit width has values 16, 32, and 64. Statistics exist in the database for the entries consisting of any combination of these characteristics, hence the number of bus entries being 690 ($= 46 \cdot 5 \cdot 3$). An exception from other characteristics is the latency of an FU. It is not varied through certain values while creating FU entries since different operations support very different latencies. In average, 1.5 different values exist for each combination of other characteristics. This indicates that every second operation has implementations only for one latency and every second for two different latencies. In addition, the write ports of a RF are not varied over the size of the RF. Thus, 3.4 different values exist in average for each combination of other characteristics.

Table 5. *Number of entries for different resource types in the cost database together with the number of different values for each characteristic of them.*

Characteristic	**FU**	**RF**	**Bus**	**Input socket**	**Output socket**
Cycle time	7	1	5	7	8
Bit width	3	3	3	3	-
Operations	28	-	-	-	-
Latency	1.5	-	-	-	-
Fanin	-	-	46	19	-
Fanout	-	-	-	-	20
Size	-	10	-	-	-
Read ports	-	4	-	-	-
Write ports	-	3.4	-	-	-
Total	924	408	690	399	160

5.2.2 Efficiency Tests

The efficiency tests were carried out on 3GHz Pentium 4 machine with 1GB of RAM as well as on 600MHz Pentium 3 machine with 256MB of RAM. The estimator was invoked utilizing four different versions of filtering search. First of all, an unoptimized version was used after which only quick filtering optimization was applied. Thereafter, filtering search used cache to optimize the queries, and finally, both quick filtering and cache were in use. For each version, the internal order of match types was set to the best possible order, i.e., the fields were filtered in the order that leads the query into the fastest possible execution time. Furthermore, the order was changed to the worst possible to obtain maximum execution time of the estimator with the specified input data.

The execution times of the estimator in the tests are shown in Table 6. The execution time taken by the compilation and simulation required by the estimation are excluded. The execution times are averages of five completions of the estimator to reduce the variation caused by the changes in the load on the test machine. The results clearly indicate that optimizations reduce the execution time really significantly in the worst case. Moreover, the best case time is also decreased about 40% on both machines which is a significant improvement. It is important to notice that the best and the worst case are equal when the optimizations are used.

If the order of match types in the search type were always the best, the optimizations

Table 6. *Execution time of the estimator.*

Optimizations	**P4 3GHz**		**P3 600MHz**	
	best	worst	best	worst
Unoptimized	1.3s	4.6s	5.0s	19.0s
Quick filtering	1.2s	1.4s	4.5s	5.4s
Cache	0.8s	2.4s	3.3s	10.0s
Optimized	**0.8s**	**0.8s**	**3.0s**	**3.0s**

would not be needed. In addition, if all the users had powerful machines, the optimizations would be unnecessary. However, the worst case execution time without optimizations on Pentium 3 machine is quite long. Thus, the optimizations are really needed. In addition, the size of the database or processor configuration may increase, whereupon the optimizations would be required to keep the estimation time under control.

Even though the estimation time is only 19 seconds in the worst case tested, the advantage achieved with optimizations, i.e., 16 seconds down to 3 seconds, is significant if the design space explorer is considered. It is not unusual that the explorer evaluates two thousand configurations. Speedup of 16 seconds in one evaluation would mean 8,9 hours faster execution of the explorer.

6. CONCLUSIONS

In this thesis, a hardware cost estimation procedure for transport triggered architectures was described together with an implementation of the hardware estimator into the MOVE framework. The estimator creates the statistics of the target processor for a given application in terms of chip area, power consumption and timing. Function units, register files, buses, input and output sockets, and the control logic of the target processor are evaluated, whereas both the instruction and data memory are excluded from the evaluation. The main purpose of the proposed estimator was to improve the accuracy and the flexibility of the old estimator of the MOVE framework. Both accuracy and flexibility was achieved by isolating the area, power, and timing information of the hardware resources into a cost database. The estimation procedure was based on this database from which the statistics of each resource were obtained for evaluating the costs of the entire target processor. The estimator uses the compiler and simulator of the MOVE framework to obtain utilization statistics of hardware resources for achieving more accurate power evaluation.

The cost database improves dramatically the flexibility of the estimator. Changing the technology is of major importance, and hence it is done extremely easy in the proposed estimator. Only the database has to be replaced with a new database that is generated for the new technology. The actual estimator is independent of the technology.

From the software systems's point of view, the estimator has to be done in a flexible and expandable way to make future modifications possible since the actual estimation procedure might slightly change. The most important part in the software implementation was to design the data structures and search algorithm for the cost database in a flexible and expandable way. This was achieved by taking advantage of the object-oriented programming paradigm. The designed search algorithm is extremely flexible. It is based on filtering unwanted database entries out of the results by one entry field at a time, i.e., entries that do not meet the search criteria in a specific entry field will be removed from the resulting entry collection. Thus, when all the fields are handled, the resulting entry collection contains only entries that are valid according to the search

request. The algorithm does not depend on entry or field types, i.e., the cost database can include any type of entries and entry fields without requiring any changes in the algorithms or the data structures of the cost database implementation. Only their usage has to be changed.

The accuracy of the estimator was verified by comparing the results of the Synopsys Design Compiler and the proposed estimator. Three different target processors were evaluated with three different interconnection networks for each of them. The application in the evaluations was 8×8 discrete cosine transform. The accuracy of the estimator is superb in area. The obtained power evaluations are quite satisfactory even though inaccuracies exist in the interconnection network as well as in the control logic. Since the power consumption is much harder to evaluate than the area, the results of the estimator can be considered fairly good. Thus, the high-level estimation procedure has proven to be an interesting alternative.

Since an estimation includes several database queries, the efficiency of the estimator has to be verified using experimental tests. The application in the tests was 8×8 DCT, and the target processor and the cost database were awkward for estimator from the efficiency's point of view. The effect of the optimizations used in the database queries, i.e., cache and quick filtering, were also tested. Each test was carried out on 1GHz Pentium 4 as well as on 600MHz Pentium 3 machine. The speed of the estimator is outstanding since the estimation takes less than a second on Pentium 4 machine and about three seconds on Pentium 3. In addition, the effect of the optimizations of the database queries is significant.

Even though the experiments evidence great accuracy, the estimator can be improved a lot. First of all, the inaccuracies of the interconnection network should be fixed by finding out the exact items included in the interconnection using commercial synthesis tools. Secondly, alternative methods have to be considered and experimented for the evaluation of the control logic. In general, current model seems to be completely inaccurate.

Accuracy verification of the proposed hardware cost estimator has to be done more extensively. Several experiments have to be carried out with varying applications, cycle times and target processors. Thereafter, a completely new issue, longest path analysis will be researched and implemented to obtain the maximum clock frequency the processor can support.

BIBLIOGRAPHY

[1] J. M. Rabaey, W. Gass, R. Brodersen, T. Nishitani, and T. Chen, "VLSI design and implementation fuels the signal-processing revolution," *IEEE Signal Processing Mag.*, vol. 15, no. 1, pp. 22–37, Jan. 1998.

[2] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*. Chichester, UK: John Wiley & Sons, 1997.

[3] H. Corporaal and M. Arnold, "Using transport triggered architectures for embedded processor design," *Integrated Computer-Aided Eng.*, vol. 5, no. 1, pp. 19–38, 1998.

[4] J. Hoogerbrugge, "Code Generation for Transport Triggered Architectures," Ph.D. dissertation, Delft Univ. Tech., Feb. 1996.

[5] J. A. A. J. Janssen, "Compilation Strategies for Transport Triggered Architectures," Ph.D. dissertation, Delft Univ. Tech., Sept. 2001.

[6] A. G. M. Cilio, "Code Generation and Optimization for Embedded Processors," Ph.D. dissertation, Delft Univ. Tech., Sept. 2002.

[7] *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076-1993, 1994.

[8] A. Smit, "The MPG manual," Master's thesis, Delft Univ. Tech., Delft, The Netherlands, June 2000.

[9] J. Sertamo, "Processor Generator for Transport Triggered Architectures," Master's thesis, Tampere Univ. Tech., Tampere, Finland, Sept. 2003.

[10] M. Arnold, "Instruction Set Extension for Embedded Processors," Ph.D. dissertation, Delft Univ. Tech., Mar. 2001.

[11] T. Nurmi, S. Virtanen, J. Isoaho, and H. Tenhunen, "Physical modeling and system level performance characterization of protocol processor architecture," in *18th IEEE Norchip Conference*, Turku, Finland, Nov. 5–6 2000.

[12] T. Ahonen, T. Nurmi, J. Nurmi, and J. Isoaho, “Block-wise extraction of rent’s exponents for an extensible processor,” in *IEEE Computer Society Annual Symposium*, Feb. 20–21 2003, pp. 193–199.

[13] J. Septien, D. Mozos, R. Hermida, and A. Sotelo, “Bounding the design space in hardware allocation,” in *Circuits and Systems, 1991*, Shenzhen, China, Jun. 16–17 1991, pp. 913–916.

[14] J. Gerlach and W. Rosenstiel, “A scalable methodology for cost estimation in a transformational high-level design space exploration environment,” in *Design, Automation and Test in Europe*, Paris, France, Feb. 23–26 1998, pp. 226–231.

[15] *Synopsys Online Documentation*, Synopsys, 2002.

[16] S. B. Lippman and J. Lajoie, *C++ Primer*, 3rd ed. Addison Wesley Longman, Inc., 1998.

[17] J. R. Levine, T. Mason, and D. Brown, *lex & yacc (2nd ed.)*. O’Reilly & Associates, Inc., 1992.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, USA: Addison-Wesley, 1995.

[19] T. Connolly and C. Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management*, 3rd ed. Addison Wesley, 2002.

[20] *ModelSim SE User’s Manual*, Model Technology, 2003.

Appendix A

EXAMPLE COST DATABASE

```
FU

oper    ld_ldh_st_sth
data    32
clk     20
latency 3
area    909.000000
delay   3.99
power   0.1  138.9168 uW
power   0.8  271.2466 uW

oper    ld_ldh_st_sth
data    32
clk     4
latency 3
area    909.750000
delay   3.64
power   0.1  695.6682 uW
power   0.5  1.0190 mW
power   0.9  1.3578 mW

oper    mul
data    32
clk     15
latency 3
area    4167.000000
delay   5.36
power   0.1  520.1469 uW
power   0.9  1.2272 mW

oper    mul
data    32
clk     12.5
latency 2
area    3115.750000
delay   9.05
power   0.1  192.4314 uW
power   0.8  1.0154 mW

oper    mul
data    32
clk     5
latency 3
area    4188.250000
delay   4.55
power   0.1  1.5584 mW
power   0.8  3.7165 mW

oper    shl_shr
data    32
clk     12.5
latency 1
area    424.500000
delay   0.999
power   0.1  72.1206 uW
power   0.8  297.2523 uW
```

```
oper    and_ior_xor
data    32
clk     10
latency 2
area    771.000000
delay   2.65
power   0.1  127.4627 uW
power   0.4  306.3020 uW
power   0.9  539.0090 uW

oper    eq_gt
data    32
clk     12.5
latency 2
area    607.500000
delay   3.12
power   0.1  88.4481 uW
power   0.8  348.3014 uW

oper    add_sub
data    32
clk     7.5
latency 1
area    1397.750000
delay   3.26
power   0.1  218.2494 uW
power   0.9  1.0837 mW


RF

size    4
rd       1
wr       2
data    32
clk     4
area    1664.000000
delay   3.82
power   0.1  976.9204 uW
power   0.9  3.1914 mW

size    4
rd       2
wr       2
data    32
clk     4
area    2090.000000
delay   3.82
power   0.1  1.1272 mW
power   0.8  3.9484 mW

size    6
rd       2
wr       2
data    32
clk     4
area    2925.000000
delay   3.83
power   0.1  1.3148 mW
power   0.9  4.6454 mW

size    8
rd       2
wr       3
data    32
clk     4
area    4641.500000
delay   3.83
power   0.1  2.4534 mW
power   0.3  3.4283 mW
power   0.8  7.0832 mW

size    16
```

```
rd       2
wr       1
data    32
clk     4
area    5942.750000
delay   2.80
power   0.1  1.4219 mW
power   0.8  7.6209 mW

size    32
rd       3
wr       3
data    32
clk     4
area    18288.250000
delay   3.84
power   0.1  7.6193 mW
power   0.8  18.4731 mW


BUS_WRITE

fanin   8
data    32
clk     2
area    331.500000
delay   1.96
power   1.0  3.3075 mW

fanin   17
data    32
clk     2
area    675.250000
delay   1.97
power   1.0  9.2625 mW

fanin   7
data    32
clk     5
area    290.250000
delay   4.56
power   1.0  1.2162 mW

fanin   45
data    32
clk     5
area    1800.000000
delay   4.98
power   0.1  2.8745 mW
power   1.0  9.6934 mW

fanin   22
data    32
clk     7.5
area    780.000000
delay   6.46
power   1.0  3.1948 mW

fanin   37
data    32
clk     10
area    1428.000000
delay   6.36
power   1.0  3.9584 mW


INPUT_SOCKET

fanin   2
data    32
clk     3.0
area    64.000000
delay   0.999
power   1.0  285.2014 uW
```

```
fanin   5
data    32
clk     4.0
area    181.250000
delay   3.98
power   1.0  567.3033 uW

fanin   10
data    32
clk     0.8
area    3016.250000
delay   0.999
power   1.0  27.5040 mW

fanin   20
data    32
clk     2.0
area    1179.250000
delay   2.00
power   1.0  5.0206 mW


OUTPUT_SOCKET

fanout  3
clk     3.75
area    3.750000
delay   0.25
power   1.0  3.8770 uW

fanout  7
clk     6.0
area    8.750000
delay   0.34
power   1.0  9.0082 uW

fanout  20
clk     6.0
area    25.000000
delay   0.64
power   1.0  26.7752 uW

fanout  20
clk     0.9
area    27.750000
delay   0.54
power   1.0  30.2229 uW


CNTRL

connectivity 0.21
area         8.31967213115
power        0.0105257611241 mW

connectivity 0.92
area         10.7053415061
power        0.0120907180385 mW
```