TAMPERE UNIVERSITY OF TECHNOLOGY
Degree Programme in Information Technology

ARI METSÄHALME

INSTRUCTION SCHEDULER FRAMEWORK FOR TRANSPORT TRIGGERED ARCHITECTURES

Master of Science Thesis

Examiners: Prof. Tommi Mikkonen and Prof. Jarmo Takala
Examiners and subject approved by Department Council
April 12 2006

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Degree Programme in Information Technology
Metsähalme, Ari Juhani: Instruction Scheduler Framework for Transport Triggered Architectures
Master of Science Thesis: 53 pages
April 2008
Major subject: Software Engineering
Examiners: Prof. Tommi Mikkonen and Prof. Jarmo Takala
Keywords: transport triggered architecture, instruction scheduler

A custom-tailored application-specific processor (ASIP) can be used when no general-purpose processor (GPP) in the market can fulfill the requirements set for an embedded system. ASIPs are co-designed with the software used in the system, according to any possible restrictions in performance, energy consumption and used silicon area.

Designing application-specific processors is usually demanding, time-consuming and costly. Therefore, the design process should be automated as much as possible. TTA-Based Codesign Environment (TCE) is a toolset that provides a semi-automated design flow of application-specific processors helping embedded system developers in finding the most optimal processor architecture to run the application at hand. TCE is based on the transport triggered architecture (TTA) processor paradigm. TTA is a highly modular and flexible templated processor architecture well suited for customization.

The most important and complicated tool in the TCE toolset is the compiler. This thesis presents a software framework written for the TCE compiler back-end that performs an important part of code generation for TTA processors: instruction scheduling. For this thesis, the base interfaces of the framework were designed and implemented. Two "proof-of-concept" instruction scheduling algorithms were also written to verify the design and functionality of the framework.

In addition to the basic concepts concerning retargetable compilers and instruction scheduling for instruction-level parallel (ILP) processors, the requirements of the framework and the ideas behind the most important design decisions are described in this thesis. Finally, the verification and benchmarking results are presented.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
Metsähalme, Ari Juhani: Instruction Scheduler Framework for Transport Triggered Architectures
Diplomityö: 53 sivua
Huhtikuu 2008
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Tommi Mikkonen ja prof. Jarmo Takala
Avainsanat: transport triggered architecture, instruction scheduler

Sovelluskohtaisesti räätälöity suoritin voidaan ottaa käyttöön, kun yksikään markkinoilla oleva yleiskäyttöinen suoritin ei pysty täyttämään sulautetun järjestelmän laitteistolle asetettuja vaatimuksia. Tällaiset suorittimet suunnitellaan yhdessä niillä ajettavan ohjelmiston kanssa siten, että mahdolliset suorituskyky-, virrankulutus- ja pinta-alavaatimukset täyttyvät.

Sovelluskohtaisten suorittimien suunnittelu on usein hyvin vaativaa, aikaa vievää ja kallista. Niinpä suurin osa suunnitteluprosessista olisi hyvä automatisoida. TTA-Based Codesign Environment (TCE) on kokoelma ohjelmistotyökaluja, jotka yhdessä tarjoavat sulautettujen järjestelmien suunnittelijoille puoliautomatisoidun sovelluskohtaisten suorittimien suunnitteluvuon. Sen tarkoitus on auttaa suunnittelijoita löytämään juuri kyseiselle sovellukselle optimoitu suoritinarkkitehtuuri. TCE perustuu suoritinarkkitehtuuriin nimeltä "transport triggered architecture" (TTA). TTA on modulaarinen ja joustava arkkitehtuurimalli, joka ominaisuuksiensa puolesta soveltuu hyvin sovelluskohtaiseen räätälöintiin.

TCE:n tärkein ja monimutkaisin työkalu on kääntäjä. Tämä diplomityö esittelee sovelluskehyksen, joka on osa TCE:n TTA-kääntäjää ja suorittaa yhden käännöstyön tärkeimmistä osista: käskyjen skeduloinnin. Tätä diplomityötä varten tehty työ koostui sovelluskehyksen tärkeimpien rajapintojen suunnittelusta ja toteutuksesta. Lisäksi työtä varten kirjoitettiin kaksi skedulointialgoritmia, joiden avulla sovelluskehyksen suunnittelu ja toiminnallisuus verifioitiin.

Tässä diplomityössä esitellään käskytason rinnakkaisuutta hyödyntäville suorittimille tarkoitettujen uudelleenkohdennettavien kääntäjien yleiset periaatteet. Lisäksi työssä kuvataan sovelluskehykselle asetetut vaatimukset ja suunnittelutyössä tehdyt tärkeimmät ohjelmistotekniset ratkaisut. Lopuksi käydään läpi ne menetelmät, joilla sovelluskehyksen toimivuus varmennettiin, ja esitellään suorituskykymittausten tulokset.

PREFACE

The work for this M.Sc. thesis was carried out at the Department of Computer Systems at Tampere University of Technology as part of the Flexible Design Methods for DSP Systems (FlexDSP) project. The project involved developing a processor co-design software based on transport triggered architectures and was funded by the National Technology Agency and Nokia.

I would like to thank Professor Jarmo Takala for taking me as part of this interesting project and for his improvement ideas for the thesis. Thanks to PhD Andrea Cilio for sharing his expertise in the field of TTAs and co-operating in the design of the instruction scheduler framework. Pekka Jääskeläinen and Vladimír Guzma also deserve a big word of thanks for their effort in guiding me through the project. I am also grateful for the whole TCE team for their contribution and teamwork. Finally, I would like to thank my family and especially my dear Freija for supporting me all the way.

Tampere, April 22 2008

Ari Metsähalme

CONTENTS

LIST OF ABBREVIATIONS

ADF	Architecture Definition File
ADPCM	Adaptive Differential Pulse Code Modulation
ASIP	Application-Specific Instruction Set Processor
CLI	Command Line Interface
DAG	Directed Acyclic Graph
DIT	Decimation-In-Time
FFT	Fast Fourier Transform
FU	Function Unit
GCU	Global Control Unit
GPP	General-Purpose Processor
HDL	Hardware Description Language
HLL	High-Level Language
ILP	Instruction-Level Parallelism
IR	Intermediate Representation
IU	Immediate Unit
JPEG	Joint Photographic Experts Group
LLVM	Low Level Virtual Machine
MOM	Machine Object Model
NOP	No Operation
POM	Program Object Model
RF	Register File
TCE	TTA-Based Codesign Environment
TTA	Transport Triggered Architecture
UI	User Interface
XVID	Digital Video Compression Format based on DivX (MPEG-4)

1. INTRODUCTION

The software and the hardware of an embedded system always have their own special restrictions and demands. Even though the use of a general-purpose "off-the-shelf" processor (GPP) in such a system would be economical, it is not always possible. Requirements concerning performance, energy consumption and used silicon area may limit the processor choice so strictly that, although the possibilities are plenty, no processor currently in the market is able to fulfill them.

One solution to the problem would be to design a custom-tailored processor from scratch. These processors are called application-specific instruction set processors (ASIP). They are co-designed with the software they are supposed to run, that is, processor resources and for example the instruction set are optimized for the application. This allows superior performance when compared to using general-purpose processors and gives the developer of the embedded system the possibility to ensure that the processor meets all the set requirements. However, the design process of such a processor is demanding and may also be very costly – unless a toolset that automates most of the process could be used.

TTA Codesign Environment (TCE) currently developed at Tampere University of Technology (TUT) is one such toolset. It assists in designing application-specific processors by offering semi-automatic tools for finding the optimal processor architecture for the application at hand. The source code is compiled for each candidate architecture and simulated to verify code and processor correctness and to provide the developer with performance statistics. Other important figures such as energy consumption and processor area are also estimated. This process of finding the optimal processor architecture for the given application is called design space exploration.

The TCE toolset is based on the transport triggered architecture (TTA) processor paradigm. It is a flexible and modular templated processor architecture that consists of easily customizable set of processor resources and thus is well suited for this purpose. The architecture is kept simple by moving most complexity from hardware to software. This makes an efficient compiler a necessity and places many challenges on designing one.

Instruction scheduling is an important part of compiling efficient code for a TTA processor. This thesis presents an instruction scheduler framework designed and implemented for the TCE compiler back-end. It helps in designing and testing instruction scheduling and other code transformation and optimization algorithms for TTA processors. It can also be used for compiling for a fixed TTA target and as part of automated design space exploration. The purpose of this thesis is to describe the main design and implementation decisions concerning the instruction scheduler framework and to provide the users with enough details to enable them to use, extend and maintain the framework.

The structure of this thesis is as follows: Chapter 2 introduces the TTA architecture and describes how a TTA processor is programmed. The TCE toolset is also briefly introduced. Chapter 3 gives an overview of compilers and presents some general concepts and issues concerning compiling for a TTA processor. Chapter 4 summarizes the main requirements set for the instruction scheduler framework and Chapter 5 presents the ideas behind the design decisions and how the framework was implemented. Chapter 6 describes how the framework was tested and lists the results of the performance benchmarks run using two implemented algorithms. Chapter 7 concludes the thesis.

2. TRANSPORT TRIGGERED ARCHITECTURE

The transport triggered architecture is a flexibly customizable templated processor architecture. It is especially suitable for running specialized applications, where high performance but low production costs, processor area usage and power consumption are required.

A TTA processor is usually tailored exclusively for the application in question, giving the processor designers the possibility to optimize processor resources depending on the performance and cost requirements they have set for their system. The TTA has been developed with easy customization in mind. It is a modular architecture template that consists of a set of a few basic building blocks, such as function units, register files and transport buses. By varying these building blocks, the designer can come up with a configuration that best fulfills the set requirements.

However, despite having a flexible processor architecture template at hand, designing application-specific processors manually is still a rather demanding and possibly very expensive task. Embedded system designers need to consider whether it would be more feasible to pick an off-the-shelf processor instead. The TTA-Based Codesign Environment developed in Tampere University of Technology is a toolset that automates the most time-consuming and error-prone parts of the TTA processor design process, and thus helps in reducing design costs.

This chapter presents the organization of a TTA processor and describes how it is programmed. Finally, the TCE toolset is introduced.

2.1 Instruction-Level Parallelism

Instruction-level parallelism (ILP) defines how many operations in a program can be executed in parallel. An ILP architecture on the other hand refers to the family of processor architectures that are able to execute operations in parallel. By duplicating processor resources or for example pipelining function units these processors provide the programmer or the compiler the possibility enhance application performance.

How much ILP is exploitable depends not only on the number of simultaneously accessable processor resources, but also on target application semantics. Control and data flow in the program create dependencies that need to be respected to make the

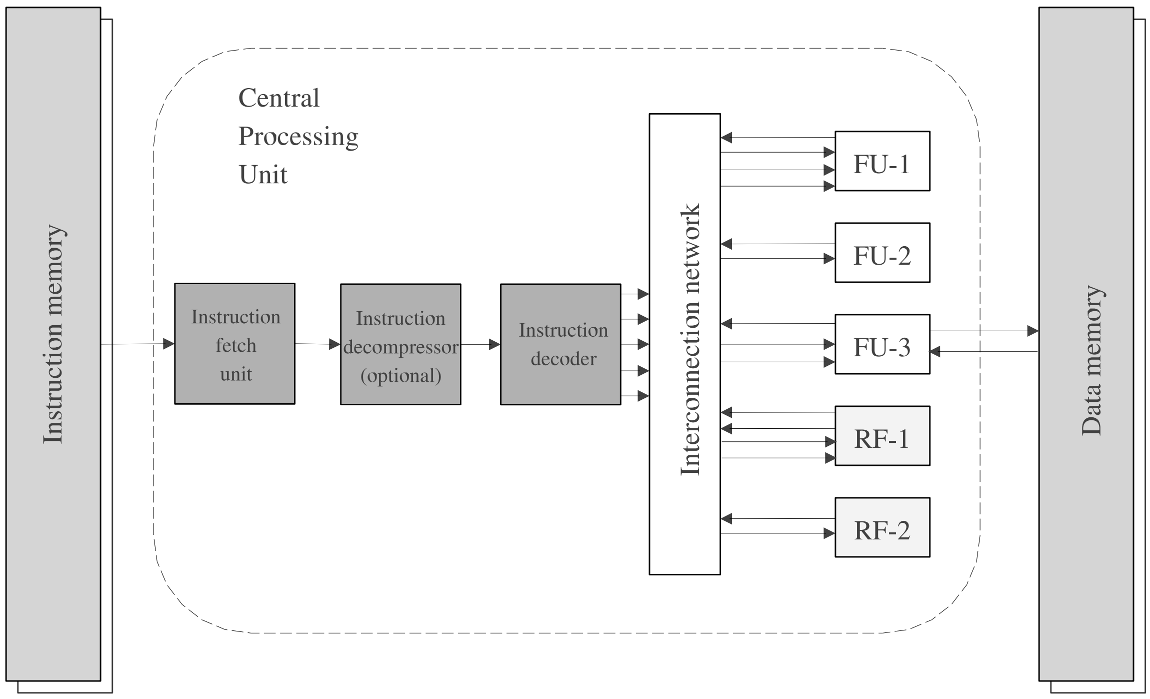

Figure 2.1: TTA processor high-level organization.

program execute correctly. These dependencies limit the number of operations that are executable in parallel. Various compiler techniques are used to increase possible ILP in programs. [1]

The TTA architecture presented in the following is an example of an ILP architecture. By duplicating its basic building blocks the designer can provide as much ILP for the target application as required.

2.2 TTA Processor Organization

The basic building blocks of a TTA processor are function units (FU), register files (RF), immediates units (IU) and the interconnection network (IC). Also, a special function unit, called the global control unit (GCU) is used to fetch and decode instructions from instruction memory and generate signals that control the processor. A load/store unit (LSU) is a function unit specialized in memory operations. The high-level TTA architecture is illustrated in Figure 2.1. Figure 2.2 represents a simple TTA processor with two function units, one register file, two transport buses and a GCU.

Each function unit can support one or more operations, ranging from simple addition and subtraction operations to complex specialized operations. In theory, a function unit could even wrap inside another TTA co-processor.

Operands for operations executed by function units are transferred through ports. The number of input and output ports in an FU depends on the operations supported

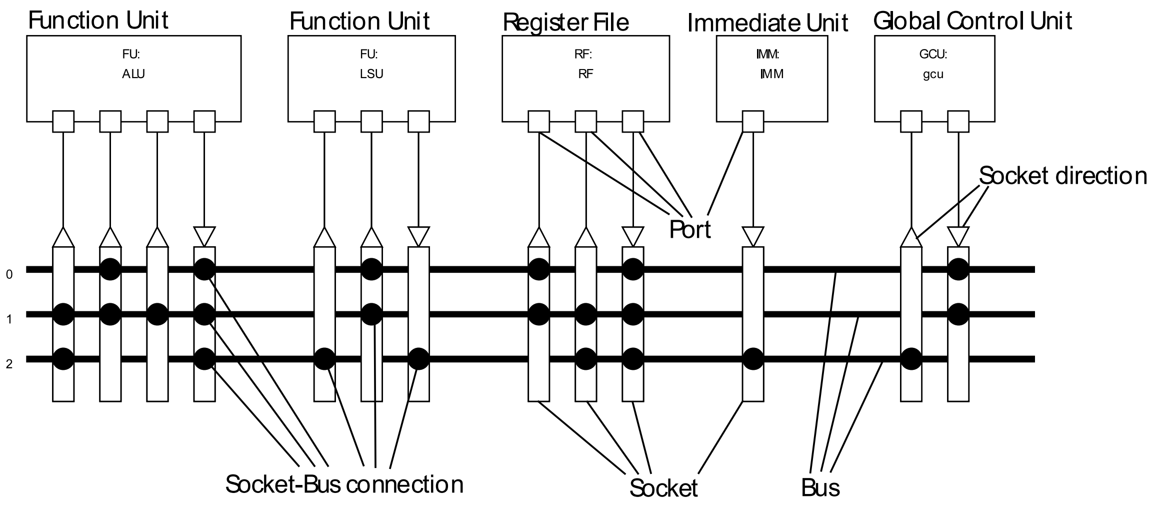

Figure 2.2: A simple TTA processor template.

by the unit. These ports are bound to the operands of each supported operation. Memory operations are also carried out by function units. Each TTA processor contains at least one so called load-store unit, which is able to read and write data memory.

Transport buses, sockets, ports and connections between them form the interconnection network. According to the control signals from the GCU, data is transported between function units and register files through those connections. The number of transport buses in the processor ultimately limits the number of concurrent data transports per clock cycle. Other available resources and data dependencies further limit operation parallelization possibilities. Temporary run-time data is stored in the registers of register files. Immediate units contain special registers for storing long immediates.

For more details on the TTA architecture, see [2].

2.3 Programming a TTA Processor

Instructions in a TTA program define data transports (later referred to as moves) instead of operations. Operations may or may not be executed as a side-effect of these data transports. The data transports are explicitly defined by the programmer, unlike in a traditional architecture, where the hardware activates the required data transports depending on the executed operations.

In the assembly language of a traditional general-purpose processor, the execution of an addition operation, which sums registers r1 and r2 and writes the result to register r3, could be programmed like this:

```
add r3, r1, r2;
```

To execute the same operation on a TTA, the programmer would need to define three data transports:

```
r1 -> add.o1;
r2 -> add.trigger;
add.o3 -> r3;
```

The two input operands of the addition operation are transported through the IC from the registers to the input ports of the function unit that implements the operation, `add.o1` and `add.trigger`. The input port that is bound to the second operand of the operation is called a trigger port. A data transport to this operand sets the opcode and triggers the execution of the operation. This behavior explains the name “transport triggered architecture”.

After the operation latency, the result is available at the output port `add.o3`, ready to be transported to register r3. In the example, the result is ready immediately at the next cycle from triggering the operation. This might not be the case especially in case of complex operations or memory accesses. Since a TTA does not automatically recognize stuctural or control hazards and stall the processor until the conflict resolves, the programmer must take operation latencies into account.

The code in the previous example is unscheduled TTA code. At this point, it is not yet mapped to any particular architecture. In this type of TTA code, data moves are defined one after another, one move per instruction. When this kind of TTA code is compiled for a concrete TTA architecture, the instructions are filled with as many moves as possible, that is, the code is scheduled. In addition, the resources used to carry out the data transports are assigned. Unscheduled TTA code has a few restrictions which ensure it is as architecture-independent as possible. These restrictions are described in detail in [3].

Since the two input operand moves in the example are independent of each other, if at least two buses and required connections are available, these moves could be carried out in parallel. In that case, the assembly code would look like the following:

```
rf.1 -> alu.o1; rf.2 -> alu.trigger.add;
alu.o3 -> rf.3;
```

Here the resources are also mapped to the architecture at hand. Register file “rf” is used to store the values. In this notation, `rf.1` refers to its register at index *1*. The function unit “alu” is used to execute the operation. `alu.trigger.add` means that the opcode for an addition operation is set and the operation is triggered by this move.

Conditional execution in TTAs is implemented using *guards*. Each move in a TTA instruction may have a guard that refers to a register, which usually is a 1-bit boolean register, or to a result port of a function unit. In case the value in the register or at the FU port evaluates to zero (false), the move is not be carried out.

2.4 TTA-Specific Optimizations

The TTA architecture allows a set of optimizations not exploitable on other traditional architectures. An example of these is *software bypassing*, which is an optimization normally done at run-time by the hardware. In TTA's though, this opimization is done by the programmer (or usually by the compiler).

In software bypassing, the results of computations are transferred directly to the operands of data dependent operations instead of transferring them through registers. This reduces the number of reads and writes to register files and reuse of registers, thus increasing ILP and reducing the cycle count of the program. [4]

As an example, let's look at the following piece of code that contains two subsequent data dependent operations:

```
rf.1 -> alu1.o1; rf.2 -> alu1.trigger.add;
alu1.3 -> rf.3;
rf.3 -> alu2.o1; rf.4 -> alu2.trigger.sub;
alu2.3 -> rf.5;
```

In this software bypassing example, the result of the first addition operation is directly transferred to the operand of the subtraction operation as follows:

```
rf.1 -> alu1.o1; rf.2 -> alu1.trigger.add;
alu1.3 -> alu2.o1; rf.4 -> alu2.trigger.sub;
alu2.3 -> rf.5;
```

Since the defining move `alu1.3 -> rf.3` is not required anymore, it is eliminated by *dead-result elimination* optimization. By applying bypassing, the cycle count is reduced as well as an unnecessary data transport and an RF access is eliminated.

Another example of a TTA-specific optimization is *operand sharing*, which means that if two operations have a common operand, the programmer can assign both operations to be executed on the same FU. The second operand move to the FU can be then eliminated, thereby saving one data transport and an RF read. [5]

In the following piece of code, the subsequent addition and subtraction operations have a common operand `rf.1`:

```
rf.1 -> alu1.o1; rf.2 -> alu1.trigger.add;
alu1.3 -> rf.3;
rf.1 -> alu2.o1; rf.4 -> alu2.trigger.sub;
alu2.3 -> rf.5;
```

If both the operations are executed on the same FU, the second operand move `rf.1 -> alu2.o1` can be eliminated, because it is already present in the operand register of the FU `alu1`. The optimized code after applying operand sharing saves a move and an RF access:

```
rf.1 -> alu1.o1; rf.2 -> alu1.trigger.add;
alu1.3 -> rf.3; rf.4 -> alu1.trigger.sub;
alu1.3 -> rf.5;
```

2.5 TTA-Based Codesign Environment

The first toolset developed for designing TTA processors was MOVE framework. The MOVE project started at Delft University in the Netherlands in the early 1990's. [5] Since 2002, development and maintenance continued at Tampere University of Technology in Finland.

The main purpose of the toolset was to allow researchers to experiment new ideas and extend the framework with new algorithms easily. In the long run, due to the chosen software architecture in the MOVE framework, it became very difficult if not impossible. The aim of the project started at TUT was to completely redesign the MOVE framework, focusing on extendability and flexibility. The new toolset was named TTA-Based Codesign Environment (TCE).

The initial input to the TCE toolset is the source application written in a high-level programming language (HLL). The front-end compiler transforms the original source to bytecode that is then used as an input to the following parts in the design flow. Originally, the TCE front-end was based on GCC version 2.7.0 [6]. Currently, an LLVM-based (Low Level Virtual Machine) implementation is used. A detailed description on the LLVM framework can be found from [7].

The rest of the TCE design flow can be divided in the following main phases: processor design space exploration, code generation and analysis, and program image and processor generation [8]. The design flow is illustrated in Figure 2.3.

Processor design space exploration. The design space exploration is a process of finding an optimized processor for running the application at hand, considering the set requirements and restrictions. This process can be automated using the Design Space Explorer tool in the TCE toolset. It starts by modifying the resources of an initial processor configuration and evaluating the effects of the modifications. The evaluation process consists of estimating processor area usage, energy consumption and performance for each configuration. Finally, the exploration produces a set of candidate architectures that fulfill the given performance and cost requirements.

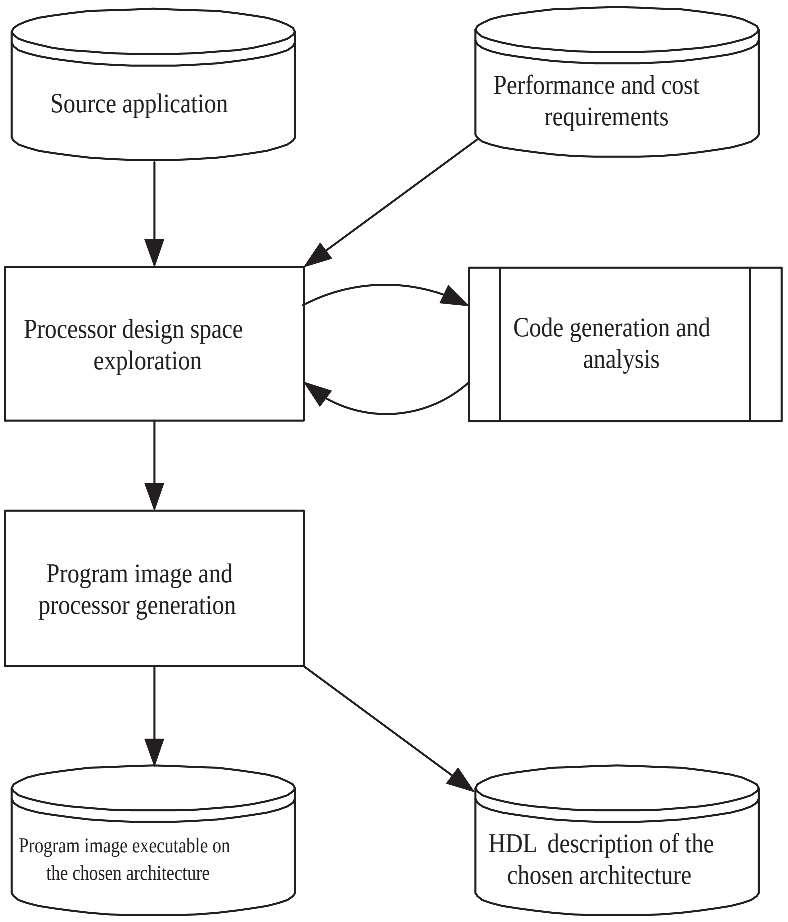
Source application
Performance and cost requirements
Processor design space exploration
Code generation and analysis
Program image and processor generation
Program image executable on the chosen architecture
HDL description of the chosen architecture

Figure 2.3: TCE design flow.

In manual design space exploration, the user is allowed to fully customize the processor architecture. Each custom configuration is evaluated similarly to the automated exploration. According to the evaluation data, the user can modify the architecture as necessary and repeat the process until the optimal architecture is found.

Code generation and analysis. In the code generation and analysis phase, the source code is scheduled and optimized for the given architecture and then analyzed by simulation. The simulator provides data such as processor utilization and cycle counts, which are necessary for energy consumption and performance estimates. The compiler maps the input program to parallel code that utilizes the resources of the given processor architecture as efficiently as possible. This is definitely the most demanding task in the TCE design flow, and therefore TCE provides researchers a framework for developing and experimenting with different scheduling algorithms and optimizations.

The detailed requirements of the instruction scheduler framework are presented in Chapter 4, and the ideas behind the design and implementation decisions are discussed in Chapter 5.

Program image and processor generation. In the final phase of the TCE design flow, a hardware description language (HDL) description of the chosen architecture is generated using the Processor Generator tool in the TCE toolset. Additionally, the executable bit image of the scheduled program is generated with the Program Image Generator tool. A detailed presentation of both these tools can be found in [9].

3. CODE GENERATION FOR TRANSPORT TRIGGERED ARCHITECTURES

This chapter discusses the most important concepts concerning instruction scheduling and code generation with main focus on transport triggered architectures.

3.1 General Concepts

Generally, a compiler is a program that translates a program written in some language, the *source* language, to another language, the *target* language, while keeping program semantics equivalent. The source and target languages may be basically anything. Usually though, an HLL representation of the program is translated to the machine language of a computer. [10]

In TCE, the compiler is used to translate a program written in a high-level language into executable code for the TTA at hand. Especially for TTA-based processor architectures, an efficient compiler is an essential tool. That is because of the basic concept of the TTA: moving complexity from hardware to software.

Compiling for a TTA involves assigning processor resources to every data transport while avoiding any conflicts in resource usage. At the same time, all possible ILP should be exploited to make the code execute as efficiently as possible. Particularly with larger applications, this would be a very time consuming and demanding task to do manually. That is why the programmer is better off leaving this task to an automated tool. The effort of writing a good TTA compiler pays off also because of its retargetability: changes in the architecture or the instruction set of the processor does not require compiler rewriting.

3.2 Structure of a Retargetable ILP Compiler

Compilers are typically divided in three parts: a front-end that is HLL-specific, a middle-end that performs machine-independent optimizations on the output of the front-end, and a target architecture dependent back-end. Each part of the compiler lowers the level of abstraction in the source code and finally comes up with a program representation that corresponds directly to the machine code of the target processor. [11]

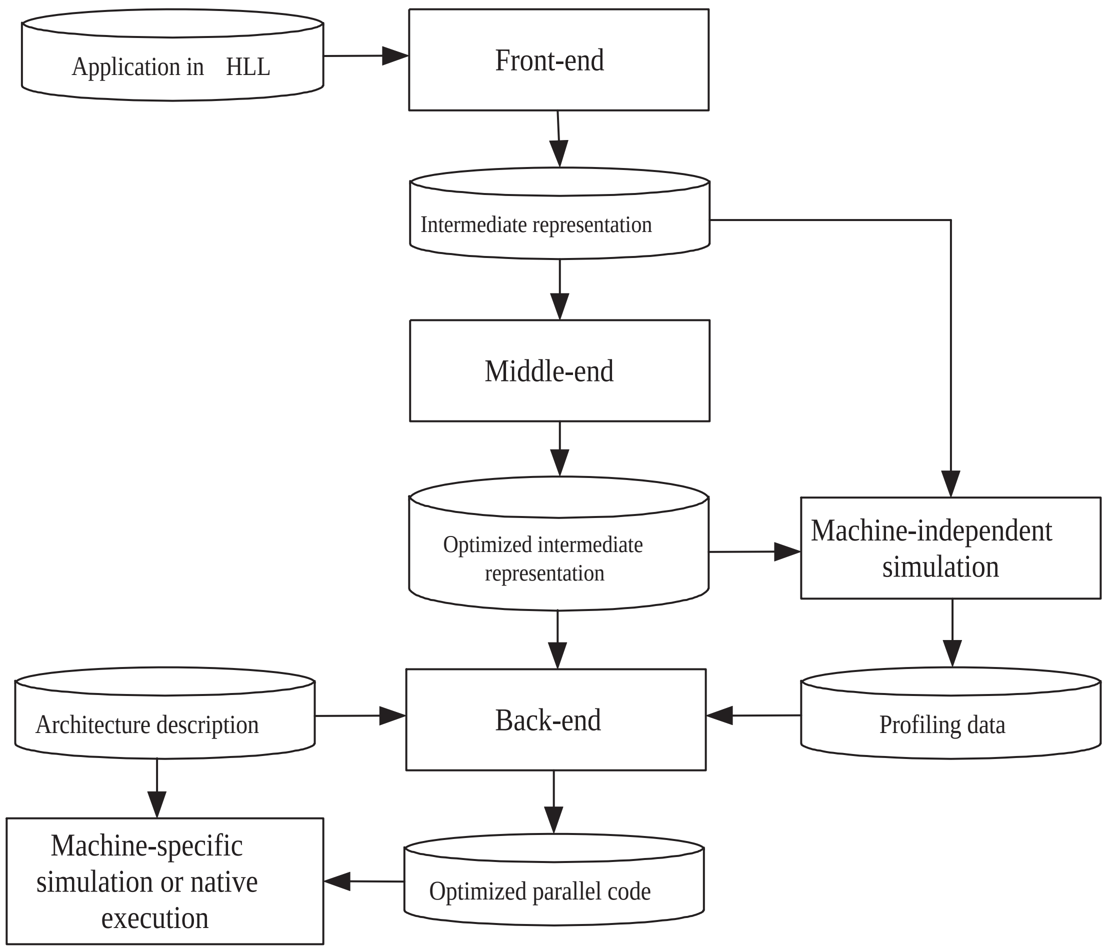
Application in HLL
Front-end
Intermediate representation
Middle-end
Optimized intermediate representation
Machine-independent simulation
Architecture description
Back-end
Profiling data
Machine-specific simulation or native execution
Optimized parallel code

Figure 3.1: Data flow in a retargetable ILP compiler.

Front-end. The front-end translates the source application code written in HLL into an intermediate program representation (IR) not compiled for any particular architecture. Compilers usually have multiple front-ends for different programming languages.

A simulator can be used to verify the correctness of the code and generate profiling data to be used in the next phase of the compilation. This profiling data may contain for example execution counts for each basic block and each control flow edge.

The IR and all possible auxiliary data are the inputs for the compiler middle-end, or to the back-end if no optimizations are performed on the IR. The back-end then also requires the architecture description.

The data flow in a typical retargetable ILP compiler is illustrated in Figure 3.1.

Middle-end. The middle-end performs high-level language- and architecture-independent optimizations on the intermediate program representation produced by the front-end. These optimizations may include for example dead-code elimination, which removes unnecessary instructions, or function inlining and loop unrolling, which aim at increasing exploitable ILP. [4]

Back-end. The compiler back-end reads the machine-independent IR, the architectural description and possible profiling information, then translates the code into parallel code for the target architecture.

The back-end performs various analyses on the program, such as control flow and data flow analysis and memory reference disambiguation analysis. Using this information, it performs several optimizations that require specific knowledge of the target implementation. These optimizations include register allocation and instruction scheduling, which are important parts of generating efficient code executable on the target processor. [4]

3.3 TCE Compiler Overview

The TCE compiler follows the basic structure of a retargetable ILP compiler presented in the previous section. It is illustrated in Figure 3.2. The front-end of the TCE compiler is the LLVM C front-end. It transforms an application written in C to LLVM bytecode, which is an architecture-independent intermediate program representation used in the LLVM framework. [7]

The IR is then optimized in the middle-end and may be simulated with the LLI for verification. LLI is a tool in the LLVM framework that executes the bytecode using an interpreter or a just-in-time compiler. [7]

The back-end of the TCE compiler requires the architectural description of the target processor (Architecture Definition File or ADF, described in [12]). Machine-dependent code transformations, such as instruction selection and register allocation are performed on the input bytecode in the LLVM back-end.

Then the optimized code, that now contains both machine-independent and dependent information, is passed to the TCE back-end, that performs instruction scheduling, applies TTA-specific optimizations and finalizes the code generation process. The instruction scheduler framework, which is the main topic in this thesis is part of the TCE-backend. Instruction scheduling, together with resource assignment is its responsibility.

3.4 Code Transformations

There are a number of code transformations that can or need to be run on the code before, during or after instruction scheduling. Two mandatory transformations, *instruction selection* and *register allocation*, are presented in the following sections.

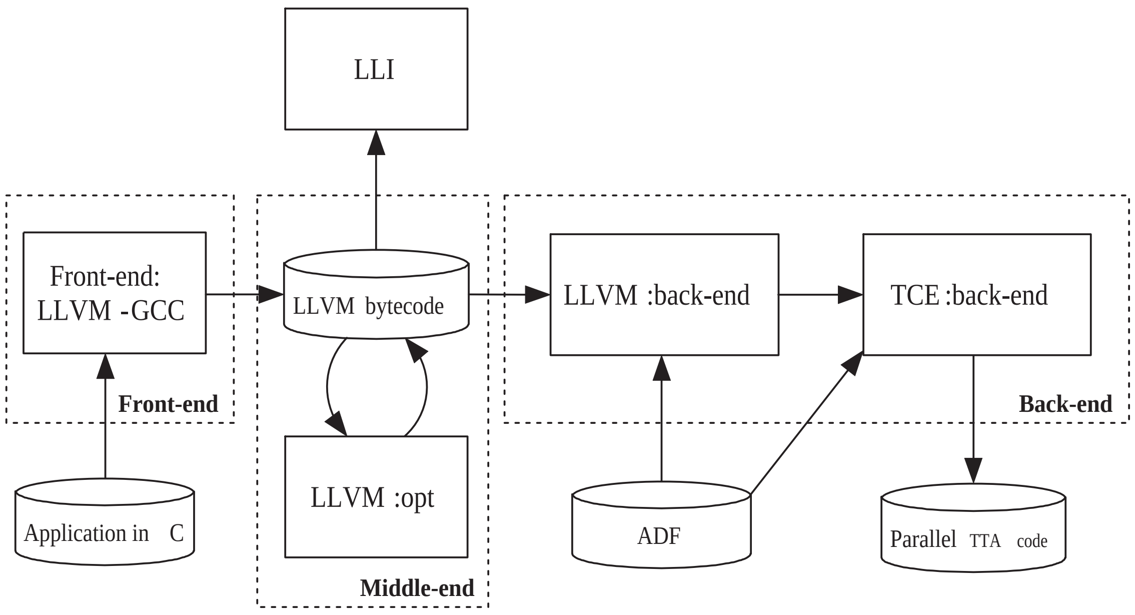

Figure 3.2: Structure and data flow in the TCE compiler.

3.4.1 Instruction Selection

Instruction selection is the process of replacing the instructions defined in the intermediate program representation by instructions from the instruction set of the target architecture. [11]

Instruction selection is an important code transformation that may have a considerable effect on program execution time, especially in a retargetable compiler such as a TTA compiler, where the instruction sets of the possible target architectures may vary significantly. Instruction selection is also crucial in the sense that it must be done in order to make the code executable on the target.

In the TCE compiler, the instruction selection is done by the LLVM back-end before instruction scheduling.

3.4.2 Register Allocation

Register allocation means assigning variables in the program to registers in the target processor. A traditional method of register allocation involves creating and coloring an *interference graph* that shows which variables in the program are live at the same time.

A fundamental problem concerning register allocation is the phase-ordering problem: whether to perform register allocation before or after instruction scheduling, or to do them at the same time. A register allocator tries to minimize the need to spill variables to memory and tends to reuse registers, thus creating scheduling constraints (data dependencies) that the instruction scheduler must cope with. An instruction scheduler on the other hand tries to exploit all available ILP and would benefit from the scheduling freedom caused by a large number of used registers. In consequence, these two important optimizations have rather conflicting goals. [13]

The LLVM backend in the TCE compiler implements a *linear-scan* register allocator, that attempts to allocate registers in linear time proportional to the number of instructions in the code. The register allocation is done before instruction scheduling, that is, before the code is handed over to the TCE back-end. This approach prioritizes efficient register use over exploiting ILP, which works well for machines with few available registers. [11]

3.5 Instruction Scheduling

Instruction scheduling is a fundamental phase in code generation. In ILP compilers for statically scheduled architectures like TTA, the instruction scheduler identifies operations that can be executed in parallel and reorders and packs them into as small number of instructions as possible. This is done keeping in mind that the semantics of the program must remain intact and that the operations executed in parallel should not use the same hardware resources.

The compiler also needs to take care of operation latencies, because a statically scheduled processor is not automatically stalled in case of a conflict caused for example by reading a result before it is ready. In addition to avoiding conflicts, the compiler should take the latency slots in use by filling them with other independent operations.

In many dynamically scheduled architectures, the processor provides hardware support for stalling the processor when a structural or control hazard occurs. For these architectures, instruction scheduling is only an optimization that aims at minimizing stall cycles. [14]

A TTA instruction scheduler schedules data transports, or moves, instead of operations. Moves that belong to the same operation introduce another scheduling constraint that needs to be respected. That's why moves of the same operation are usually scheduled in one shot. [5]

In the TCE compiler, the instruction scheduling is performed by the TCE back-end.

3.5.1 Scheduling Scopes

Schedulers are usually categorized by the scope of the program or region they handle. A *basic block* is a sequence of consecutive instructions that has no join points in the middle and contains no branching except at the end of the sequence [10]. A *local scheduler* works on the scope of these basic blocks. Any scheduling decisions made

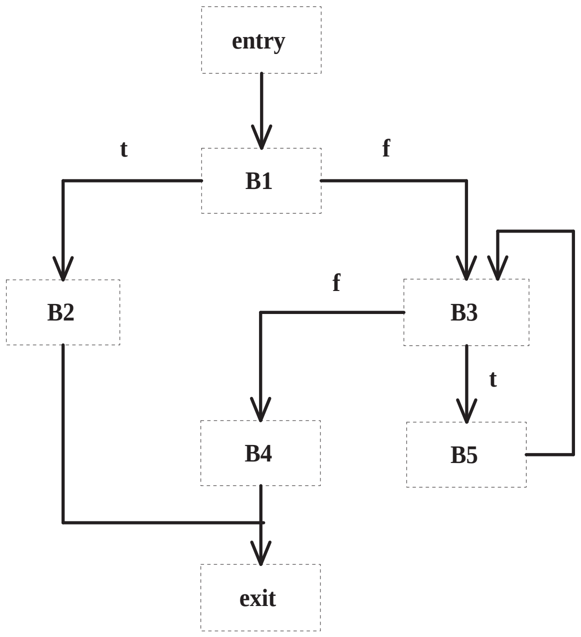

Figure 3.3: An example control flow graph.

in a basic block has no effect on the scheduling of other basic blocks.

Basic blocks are usually not more than a few operations in size, so the exploitable ILP is often limited. Therefore, more advanced schedulers work on scopes that cross basic block boundaries. These are called *global schedulers.* Global schedulers are able to move operations between basic blocks that belong to the scheduling scope. [5]

3.5.2 Control Flow Analysis

To get a global perspective on the program semantics, we need to analyze how the control flows between the basic blocks. This often involves constructing a control flow graph (CFG). Each node in the CFG is a basic block and edges represent changes in the flow of control. These changes are due to conditional execution or jumps.

Figure 3.3 depicts a simple CFG with 5 basic blocks and additional entry and exit nodes. The edges *t* and *f* represent *true* and *false* conditions, respectively. The branch selected depends on how the condition at the end of the basic block evaluates. Blank edges are unconditional jumps or fall-throughs from previous basic blocks. Refer to [15] for more details and algorithms for constructing CFG's.

3.5.3 Data Flow and Dependency Analysis

Data flow analysis provides information about how a part of a program (such as a basic block or a procedure) manipulates data [15]. It consists of computation of live-ranges of variables and finding variable definition-use chains. This information can be exploited when allocating registers. [4]

Data dependency analysis is a tool of vital importance to the instruction scheduler. It determines the ordering constraints between operations which need to be respected during scheduling for the code to execute correctly. [15] These relations can be represented by a data dependency graph (DDG). The instruction scheduler uses the DDG when deciding which operations are executable in parallel without breaking the dependency constraints [5].

For example, let's look at the following piece of pseudocode:

```
S1. x := y + z
S2. x := x + 1
S3. y := a
S4. a := x
S5. y := 0
```

It introduces data dependencies that are represented in the DDG in Figure 3.4. The solid line represents a *flow dependency*, which means that the value defined by a statement is used by the dependent statement. The dashed line represents an *anti-dependency*, which means that the value defined by a statement is redefined by the dependent statement. Another type of dependency is *output dependency*, which is similar to an anti-dependency. It occurs when reordering statements would affect the final value of a variable. This is illustrated in the example DDG by the coarsely dashed line between statements *S3* and *S5*.

Flow dependencies are also known as *true* dependencies, because they cannot be eliminated. On the contrary, output and anti-dependencies are called *false* dependencies, because they are caused by variable naming. [16]

All these dependencies reduce available ILP. If register assignment is done before instruction scheduling, it can be used to reduce these dependencies and thus have a notable effect on the parallelization of the program.

Data flow and dependency analysis algorithms can be found in [15].

3.5.4 List Scheduling

The most popular technique used for instruction scheduling is *list scheduling*. Its popularity is based on its rather good effectiveness combined with reasonable compile time. A list scheduler repeatedly assings operations to cycles or cycles to operations without any backtracking or lookahead heuristics. An instruction-based list scheduler fills instructions cycle by cycle, trying to place as many operations in the current instruction as possible. An operation-based list scheduler repeatedly selects

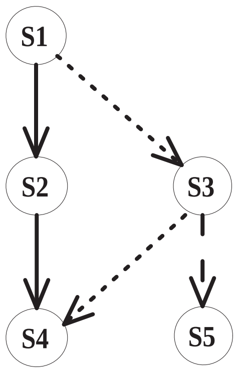

Figure 3.4: An example data dependency graph.

operations ready for scheduling and finds a suitable instruction for them. TCE uses an operation-based list scheduler, in which we concentrate in the following.

An operation-based list scheduler works on a DDG, which is a directed acyclic graph (DAG) which prevents possible lock-ups. In top-down approach, the DDG is walked through in topological order so that no operation is scheduled before all its predecessors have been scheduled. In bottom-up approach, an operation is scheduled after all its successors have been scheduled. An operation ready for scheduling is said to be a member of the *ready set.* Any operation from this set can be selected for scheduling next. The selection is prioritized by a priority function, that may for example favor candidates that are on the *longest path* on the DDG. [17]

3.5.5 Resource Assignment

Duplication of resources and pipelining in function units provides parallelism at the level of processor resources. On the other hand, if resources are scarce, producing an effective schedule for a program becomes challenging for the instruction scheduler. In addition to respecting program semantics, the produced code should not contain any resource usage conflicts. Because a TTA compiler is retargetable, and the amount and quality of resources can vary, the resource assignment process becomes even more complex. [5]

If the registers in register files have already been assigned before instruction scheduling, the scheduler must still assign operations to function units, moves to transport buses and sockets and decide whether an immediate is encoded in the immediate field of the instruction or in the source field of a move. [4]

It is an important decision when and in which order each processor resource is assigned. Resource assignments usually have side-effects and each assignment decision affects following assignments. Each resource type also requires different heuristics for selecting a resource from a group of candidates. Whether for example a *first-fit* or a *more specialized resource first* principle should be used depends on the case at hand.

The Chapters 4 and 5 describe how the instruction scheduler framework in the TCE compiler back-end helps in solving problems concerning instruction scheduling as well as resource assignment.

4. REQUIREMENTS AND HIGH-LEVEL ARCHITECTURE

This chapter presents the main requirements set for the instruction scheduler framework completed with the most important use cases, and gives a brief architectural overview.

4.1 Product Perspective

The instruction scheduler framework is a fundamental part of the TCE compiler back-end. This is because instruction scheduling plays a crucial role in compiling an application for a TTA, especially in making it run as efficiently as possible. The capabilities of a powerful TTA processor cannot be fully utilized without effective instruction scheduling. The framework is used to implement the instruction scheduler and to perform any supporting pre- or post-scheduling code analyses or transformations. It can be used as a stand-alone tool or as part of another application.

The main inputs of the framework are the unscheduled TTA code usually generated by the TCE front-end compiler, the description of the target architecture and optionally some supporting auxiliary data such as program profiling information. The framework applies user-defined optimization and code transformation passes on the input program. These passes and their parameters are given in a scheduler configuration file. From the given inputs the framework produces as output the processed target program mapped to the given target architecture and optionally some auxiliary data. The inputs and outputs of the framework are illustrated in Figure 4.1.

The framework provides means for writing and configuring the scheduler passes mentioned above and easily plugging them in without time-consuming recompilation of the whole framework. This is one of the most important requirements for the framework. It should help users in experimenting with different optimizations and parameters to find the most suitable combination for their needs and evaluate their effects on target application performance. It should also help the users in trying out different scheduling algorithms to produce the most effective schedule for a given target architecture.

The most important use cases are presented in the following sections. Refer to [18] for more details on the scheduler framework and TCE instruction scheduler

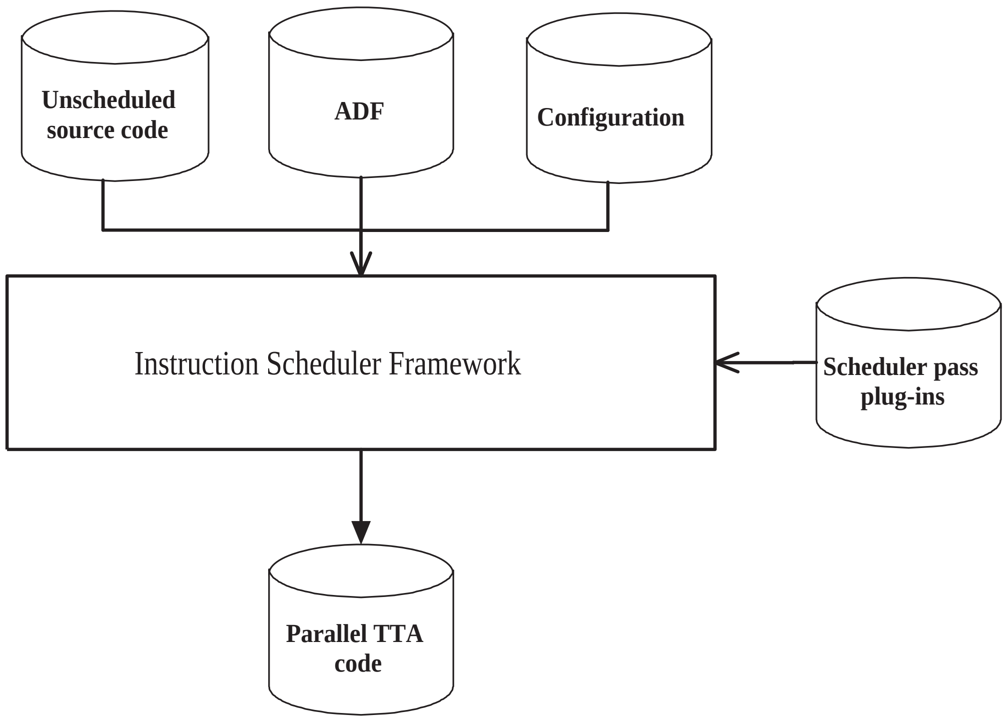

Figure 4.1: Instruction scheduler framework inputs and outputs.

requirements.

4.2 Key User Needs

The key needs of the user of the instruction scheduler framework can be briefly listed as follows:

1. Map unscheduled TTA code to parallel TTA code for a given target architecture, exploiting available instruction-level parallelism as well as possible.

2. Map partially processed or scheduled TTA code to parallel TTA code.

3. Evaluate effects of different parameters that control the code optimization and transformation passes on target application performance.

4. Evaluate effects of different code transformations and scheduling algorithms on target application performance and select the combination that best meets the set requirements.

4.3 Use Cases

Scheduling as part of design space exploration. A typical user of the scheduler framework uses the scheduler as part of design space exploration. The user experiments with different code optimizations and parameters, trying to come up with the most suitable combination for his needs, and evaluates what effects they have on exploration results. In this case, the scheduler is not used as a stand-alone application.

Scheduling for a fixed target. Another important use case is mapping a target application for a known fixed target architecture. In this case the scheduling process is started from command line or another scheduler user interface (UI) client. The scheduler framework is given the unscheduled TTA code, the target architecture description and a configuration file, in addition to possible command line parameters. The scheduling process is then launched according to given configuration and user-defined scheduler passes are performed on the target application.

Afterwards, the user evaluates the results for example by using the Simulator or the Estimator, and if required, repeats the process using different parameters for the scheduler passes, or even using a completely different scheduling algorithm, until a satisfying result is found.

Developing new algorithms. Finally, an advanced type of user is a researcher developing completely new scheduling algorithms or code optimization and transformation techniques. The user exploits the flexibility and configurability of the framework to quickly experiment with new ideas, then evaluating the performance of the result and verifying its correctness for example with the Simulator.

The developer may choose to re-implement a set of interfaces for some optimization subtask defined by the framework, or alternatively implement a completely new technique independent of the interfaces provided by the framework.

The main flexibility points of the scheduler framework software and a description of its modular architecture are presented in detail in Chapter 5.

4.4 Architectural Overview

The scheduler framework was designed with the important requirement of easy configurability and flexibility in mind. It provides a base on which co-operating modules can be plugged in at run-time. A scheduler pass may be implemented as a single module or as a group of modules working together. One of these modules is the main module and the others are called helper modules. Only the main module is visible outside the pass and is independently startable.

Each module implements the base plug-in interface described in Section 5.1.1. This interface allows each module to be replaced at run-time by another module performing the same task. It is also possible to bypass most of these interfaces if the user wishes, as long as the minimum base plug-in interfaces are implemented.

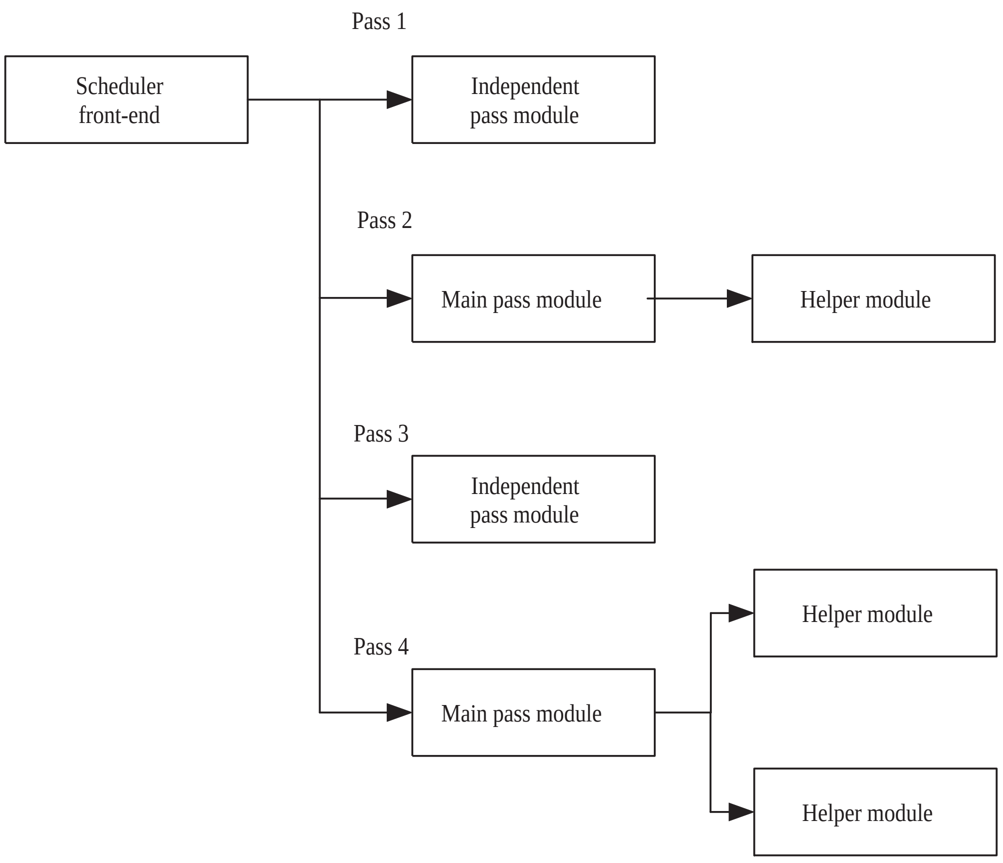

Figure 4.2: An example code transformation and scheduling chain.

However, the users are encouraged to take advantage of the interfaces provided by the framework.

There are no restrictions on the activities performed by a scheduler pass set by the framework. It is the responsibility of the user to configure the code transformation and scheduling chain so that the passes are run in the correct order and that running a pass does not bring the target program in an inconsistent state.

Figure 4.2 presents an example scheduling chain consisting of four passes and depicts the connections between the modules that form it. Pass 1 and pass 3 are carried out by single independent modules, whereas the main modules in pass 2 and pass 4 use helpers to perform certain subtasks.

The main responsibility of the scheduler front-end also shown in the picture is to load and run the plug-in modules that implement the passes of the scheduling chain. It also acts as an interface between the clients and the scheduler application. See Section 5.1 for additional details.

4.5 Core Interfaces

The scheduler framework offers a set of interfaces that help the user in implementing the most crucial part of the code transformation and scheduling chain, that is, mapping the unscheduled code to the target architecture.

Program Representation. Each scheduler pass works on a representation of the target program or a part of it. The framework provides the user with a few different object models for the purpose.

The most basic program representation is the Program Object Model (POM), which is a simple flat structure of sequentially ordered instructions. The program representations more suitable for instruction scheduling and effective code transformations are graph-based and represent data and control dependencies.

The program representations provided with TCE are described in more detail in Section 5.2

Scheduler Pass Hierarchy. Scheduler passes are classified according to the scope of the program they are able to handle, and to the type of data they accept as input. A pass may analyze or transform a whole program in one shot. On the other hand, it may be able to work in the scope of a single procedure only, or even just a basic block. Each provided program representation also has a corresponding pass interface.

Each pass should implement an appropriate part of the hierarchy depending on its required inputs. More details on the scheduler pass interfaces can be found in Section 5.3.

Resource Manager. The resource manager keeps book of used processor resources and other scheduling constraints on each cycle of execution on the part of the target program it handles. It is responsible for assigning these resources for the program to be used. Also, if the actual assignment is done outside the resource manager, it verifies the correctness of the assignment and then updates its resource bookkeeping accordingly.

Internally, the resource manager uses a hierarchy of specialized managers that are each responsible for a single type of resource. These are called resource brokers. Sections 5.4 and 5.5 give a detailed view of the resource management services provided by the framework.

The main client of the resource manager is the pass that performs the instruction scheduling, although other passes that analyze or optimize the code may also require resource management services.

5. DESIGN AND IMPLEMENTATION

This chapter describes the core components of the framework and ideas behind their design and implementation decisions in detail.

5.1 Scheduler Front-End

Scheduler front-end, implemented by the *SchedulerFrontend* class, is the top-level component of the framework. It provides a simplified interface to be used by the clients of the scheduler to configure and start the scheduling process. A client of the scheduler may be any kind of graphical or command line user interface (CLI) or a separate controller module. In other words, the scheduler may be deployed as a stand-alone terminal-based application that is launched and configured directly from command line, or as part of another application such as the Explorer.

The front-end loads the domain object models such as the target processor and the source program, loads required plugin-modules and sets up the scheduling chain according to the scheduler configuration (see Sections 5.1.2 and 5.1.3 for more details). The scheduler is configured either from command line using a configuration file or by providing the front-end with pre-constructed object models. The scheduler configuration file format is defined in [18].

After the required object models are set up, the scheduling process can be started. Upon completion of the process, the front-end returns the scheduled program or writes it to a file, depending on how it was called.

5.1.1 Plug-In Module Interfaces

The scheduling process is an ordered sequence of scheduler passes that analyze the target program and apply optimizations and code transformations to it. All pass modules – including instruction scheduling – conform to a generic module interface.

Each module is an instance of the class *BaseSchedulerModule*. It is a base class that defines the minimum interface each module is required to implement. These services include setting up the module with required object models and starting it. If a scheduling pass consists of multiple modules, the main module shall implement the *StartableSchedulerModule* interface, which is a specialization of the base class. This also is the base class for modules that alone form a complete pass. Helper modules that cannot be run independently implement the *HelperSchedulerModule*

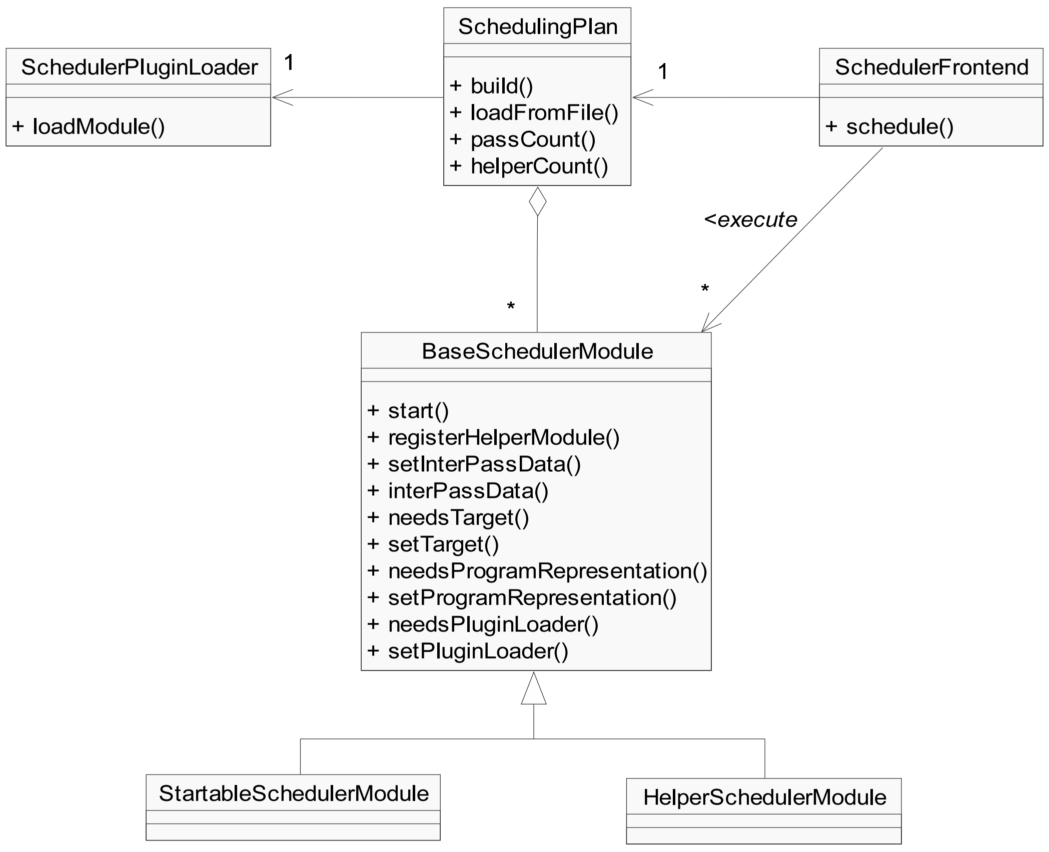

Figure 5.1: Plug-in interfaces.

interface.

5.1.2 Loading Plug-Ins

The scheduler front-end uses a general interface for loading plug-ins. This interface is implemented by the class *SchedulerPluginLoader*. It loads scheduler modules from dynamically linkable object files at run-time and manages them through their lifetime. Although the scheduler front-end is the main client for the plug-in loader, the interface is useful for any pass module that may require helper plug-in modules to be loaded during the scheduling process.

The main advantages of a separate abstraction layer for loading and linking scheduler plug-ins are that direct communication from plug-in modules to the scheduler front-end is avoided, and that the details of pass module handling, such as construction and destruction as well as name and file matching, are completely hidden from the client.

5.1.3 Constructing the Scheduling Chain

The front-end delegates the setup of the scheduling chain to the *SchedulingPlan* class. It constructs the scheduling chain from an object tree representation of the scheduler configuration file. This tree is generated by the configuration file parser, that is implemented by the class *SchedulerConfigurationSerializer*. The plug-in loader presented earlier is used to load any defined plug-in modules.

The scheduler front-end follows through the scheduling process by launching the passes stored in the scheduling plan in the defined order.

The class diagram of the scheduler plug-in interfaces and the related classes is depicted in Figure 5.1.

5.2 Program Representations

The TCE toolset provides a set of program object models that enable TCE applications to access, analyze and modify TTA programs. Each provided program representation is intended for a specific purpose, but a common property for all the representations is to help TCE applications generate efficient code for the given target TTA processor.

The program representations are not just interfaces, so they are not part of the actual instruction scheduler software framework, but more like domain utilities. Though, most of them are mainly intended to be used by the scheduler passes. These program representations are presented in the following sections.

5.2.1 Program Object Model

The Program Object Model (POM) is the most basic program representation in the TCE toolset. It only provides the most generic services. It does not model control or data flow or any other auxiliary data structures, such as basic blocks. All client-specific services are left for the special purpose program representations such as different types of graphs.

POM has a simple treelike structure. The highest level of abstraction in POM is implemented by the *Program* class. Each program is seen as an ordered sequence of procedures, implemented by the *Procedure* class. Similarly, each procedure is an ordered sequence of instructions, whereas instructions – implemented by the *Instruction* class – contain a non-ordered set of moves and long immediates. These are implemented by the *Move* and *Immediate* classes, respectively. The class relations are depicted in Figure 5.2.

A *Program* instance represents a complete TTA program. Instructions in the program have indices instead of real addresses. Instructions such as jumps that point to other parts of the program refer directly to instructions instead of addresses or

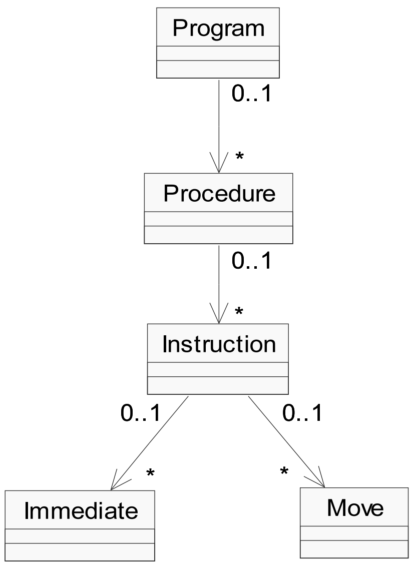

Figure 5.2: Program Object Model.

indices. The instruction indices are automatically adjusted as a side-effect of adding or removing instructions or procedures and the *InstructionReferenceManager* class takes care that any affected references are updated. It is the responsibility of the parent objects to keep book of the relative positions of the model parts in their parent structures. For example, an instruction does not know its own position in the program. Instead, it has a handle to its parent procedure.

The *Move* class represents a data transport through the interconnection network of the target TTA. It has a source and a destination and references to required resources of the target processor. The *Immediate* class represents a special data transport of a constant long immediate value encoded in the parent instruction to a dedicated immediate unit register.

For a complete description of POM with all the details and the rest of the classes explained, refer to [19].

5.2.2 Graph-Based Program Representations

All graph implementations in TCE are based on the Boost Graph Library (BGL) [20]. It provides efficient implementations of some most frequently used basic graph operations. The boost graph library is encapsulated in a class hierarchy that represents the program at a level of abstraction suitable for instruction scheduling.

The base class for all graphs in TCE, *BoostGraph*, is a BGL-based implementation of a graph. It is inherited from an implementation-independent templated base interface, *GraphBase*. Each graph contains a set of nodes and edges, implemented by the classes *GraphNode* and *GraphEdge*, respectively.

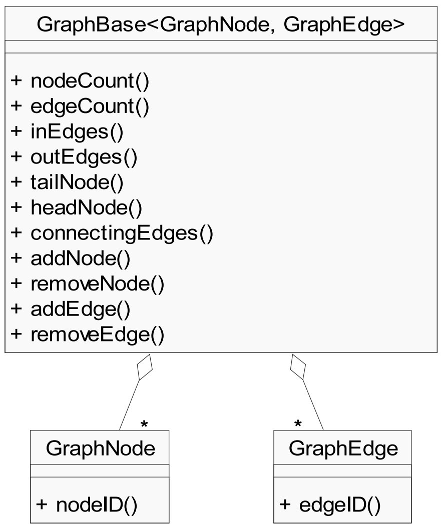

Figure 5.3: Base graph interfaces.

The purpose of these base classes is to encourage a consistency in the interfaces of all graph implementations in TCE and to suggest a minimum set of services each graph should provide. These are for example addition and deletion of nodes and edges, tests for connectivity and simple search algorithms, which depend only on the topological structure of the graph, not on the specialized data that resides in its nodes and edges.

For each specialized graph type, the required implementations of the node and edge classes are inherited from the *GraphNode* and *GraphEdge* base classes. Most importantly, the graph class itself is specialized from the appropriate base class. The base classes are illustrated in Figure 5.3.

Control Flow Graph. The control flow graph in TCE is a more specialized program representation that implements the concept of the control flow graph as described in Section 3.5.2. Its class diagram is depicted in Figure 5.4.

The payload data in the nodes of the CFG are basic blocks. The nodes are implemented by the class *BasicBlockNode*. Each node consists of a single basic block, represented by the class *BasicBlock*. The additional data in the nodes include information whether the node is an entry or exit node, or whether it just contains an ordered sequence of instructions. The basic blocks also contain statistics of how many instructions, moves or long immediates there are in the basic block. This information can be retrieved from the *BasicBlock* object.

The edges in the CFG represent changes in the flow of control from one basic block to another. This change is always caused by a jump or a call, or a fall-through from previous basic blocks. The change in control flow may be conditional (*true* or

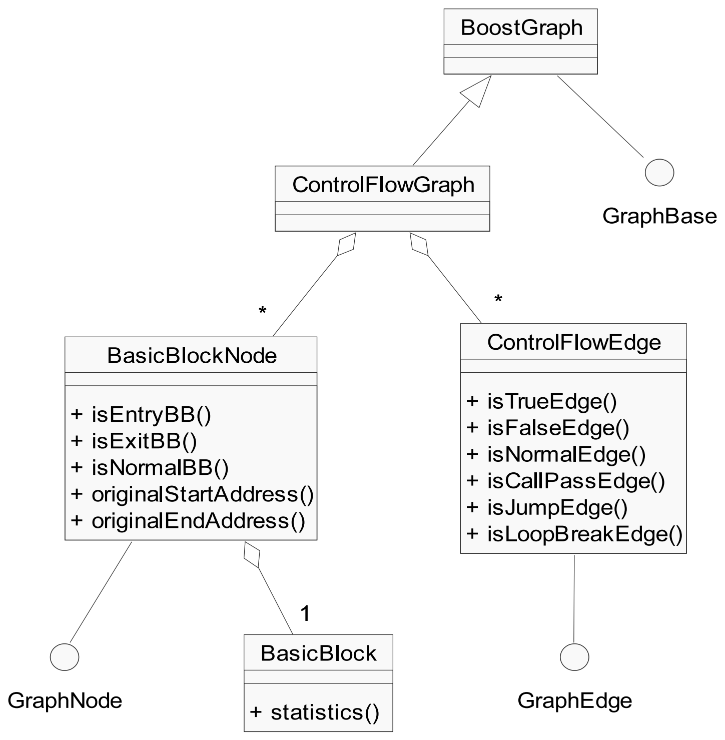

Figure 5.4: Control flow graph class diagram.

false edges) or unconditional.

The control flow graph is constructed in two main phases: First, the beginnings and ends of basic blocks are detected and basic block instances are created. Then the whole program is parsed through again, and each time a change in the flow of control (a jump or a call) is found an edge is created. Finally, artificial entry and exit nodes are created and added to the graph.

See [15] for more details on the algorithm used to build the CFG.

Data Dependency Graph. TCE also provides an implementation of the data dependency graph (see Section 3.5.3). The class diagram is presented in Figure 5.5.

The nodes in the DDG are single moves and the edges represent different types of data dependencies between them.

The possible dependency types are *read after write*, *write after write* and *write after read*. The first type is the basic case, the data should be written before it is read. The two latter are anti-dependencies: consecutive writes must be done in the right order, and a write operation is possible only after the previous value has been consumed.

The edge reason tells where the dependent data resides: in a register or in memory.

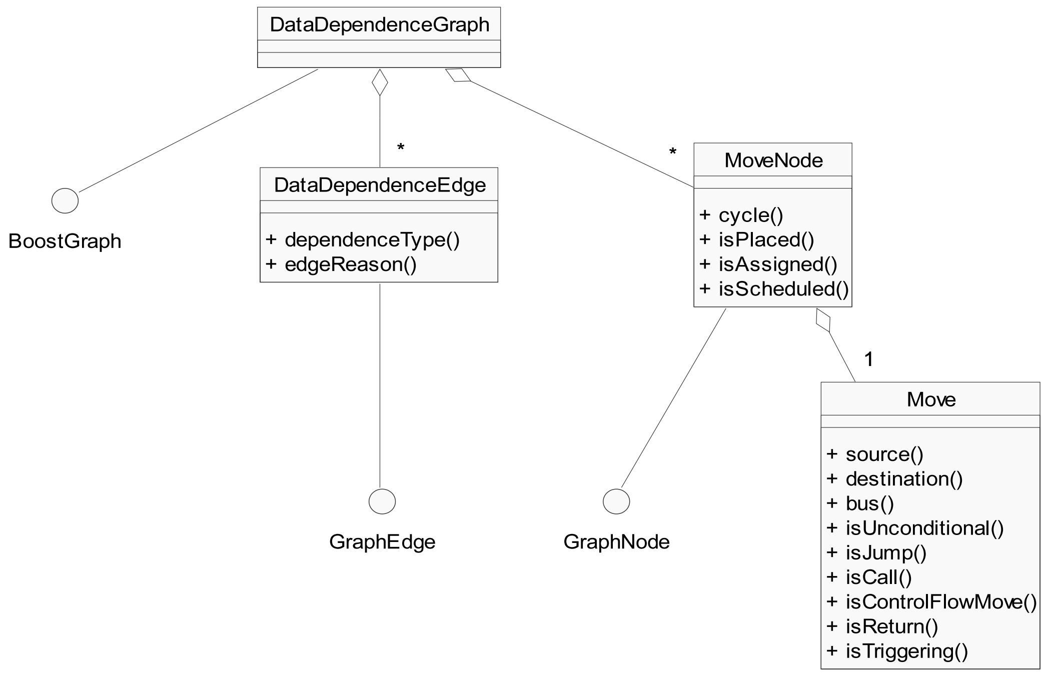

Figure 5.5: Data dependency graph class diagram.

In TTA, the data dependency may also be between the moves of a single operation, or between writes to an FU, if the operation has a state.

The data dependency graph may be constructed for a single basic block or for a whole control flow graph. Building a DDG for a CFG is a bit more complicated than for a single basic block: The CFG is walked through starting from the first basic block in topological order. In addition to the data dependencies inside the basic blocks, the dependency edges that cross basic block boundaries are also created.

The algorithm used in building the DDG can be found in [15].

5.3 Scheduler Pass Hierarchy

Each scheduler pass may require different inputs, produce different outputs and handle different scopes of the input program. Figure 5.6 depicts a simplified diagram of the interactions between a scheduler pass and its inputs and outputs.

The class *SchedulerPass* is the base interface for all scheduler passes. All passes should specialize the appropriate subclass in the hierarchy, depending on the type of data and the scope of the input program they handle.

The scheduler pass hierarchy contains special interfaces for passes that handle whole programs, procedures or just basic blocks. Passes that work with control flow or data dependency graphs also have their own base classes, as illustrated in Figure 5.7.

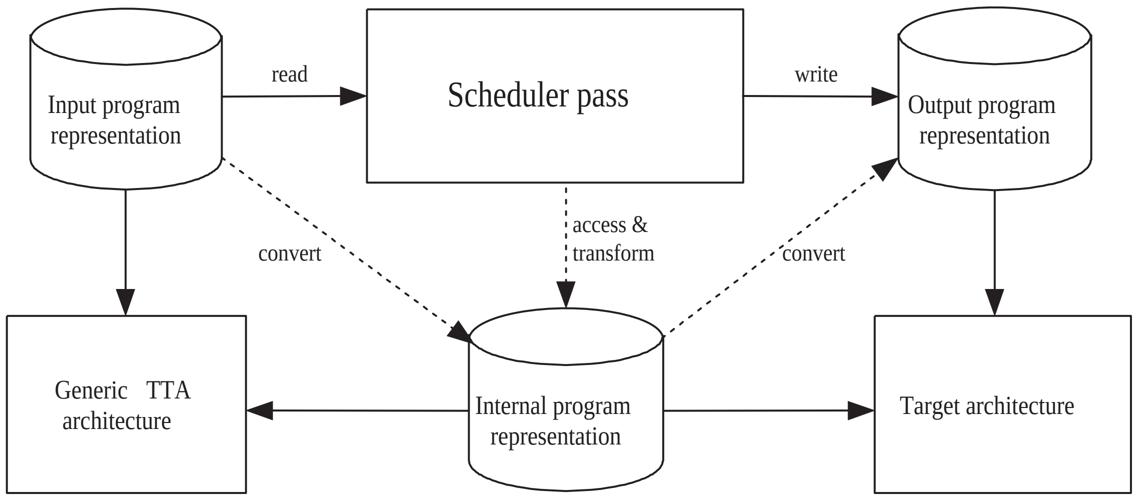

Figure 5.6: Scheduler pass inputs and outputs.

Scheduler passes communicate using an auxiliary class *InterPassData*. It provides a general storage for data of any type and a generic interface for accessing it. Additionally, it provides shortcuts to access the most frequently used data. An *InterPassData* instance is passed through every pass in the scheduling chain that requires the data.

The data is stored in key-value-pairs, a string as the key and an instance of the class *InterPassDatum* as the value. The *InterPassDatum* class is a simple interface that the data required should implement.

As a concrete example of inter-pass data passed in the basic block scheduler (see Section 6.1.2), the registers that are left unallocated in the register allocation pass are passed to the instruction scheduling pass to be used as temporary data storages as required due to reduced connectivity.

5.4 Resource Model

The resource model is a dynamic abstraction used for keeping book of processor resource usage at each cycle of the current scheduling scope. It is a hierachy of interdependent objects that each represent processor components and other scheduling constraints. With the resource manager (see Section 5.5) it provides means to allocate and assign resource during instruction scheduling.

A resource model does not need to describe the processor architecture in every detail as the Machine Object Model (MOM) does. It only represents hardware and other constraints as seen from the point of view of instruction scheduling.

The base class of the hierarchy is *SchedulingResource*. It provides a generic interface for handling all resource types. One of the most important services is testing for availability, that is, answering to the question “can this resource be assigned to

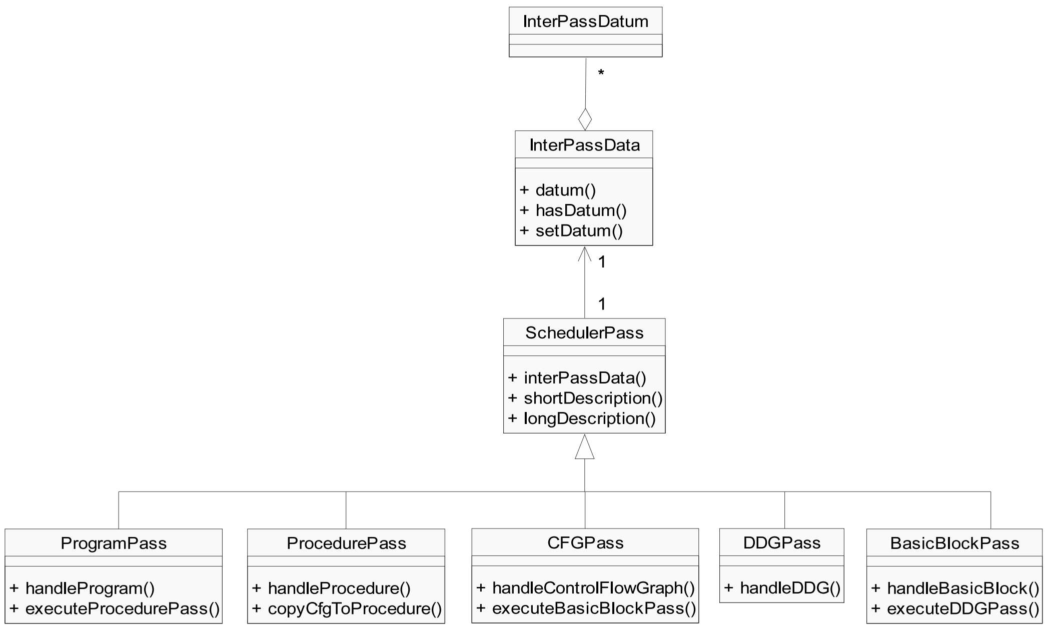

Figure 5.7: Scheduler pass class hierarchy.

the given move at the given clock cycle?” An available resource can then be assigned, or an already assigned resource may be freed. They can also be queried for dependent or related resources. The advantage of providing such a general interface is that clients do not need to have knowledge of the specifics of each resource type. Instead, all type-specific information is hidden under the common interface.

Dependent resources are resources tightly connected to each other. If a dependent resource is assigned, the resource it depends on is also assigned as a side-effect. On the other hand, if a resource is assigned, at least one resource of each type from the list of dependent resources needs to be also assigned. For example an execution pipeline is an inseparable part of a function unit, and therefore an execution pipeline resource is dependent on a function unit resource.

Resources related to each other must also be assigned in conjunction. If no related resource of a given type is available for assignment, it is not possible to assign the other resource, even if itself is available. For example, bus and socket resources are related to each other. If all buses connected to an output socket are in use in the given cycle, the socket cannot be assigned and vice versa.

Altough the main purpose of the resource model is to define a base interface and determine general design guidelines, the instruction scheduler framework includes a complete implementation of a simple resource model to be used by the resource manager. The concrete resource types derived from the base class *SchedulingResource* mostly correspond to the basic TTA building blocks represented in Chapter 2, although depending on the resource in question, the relation to concrete hardware

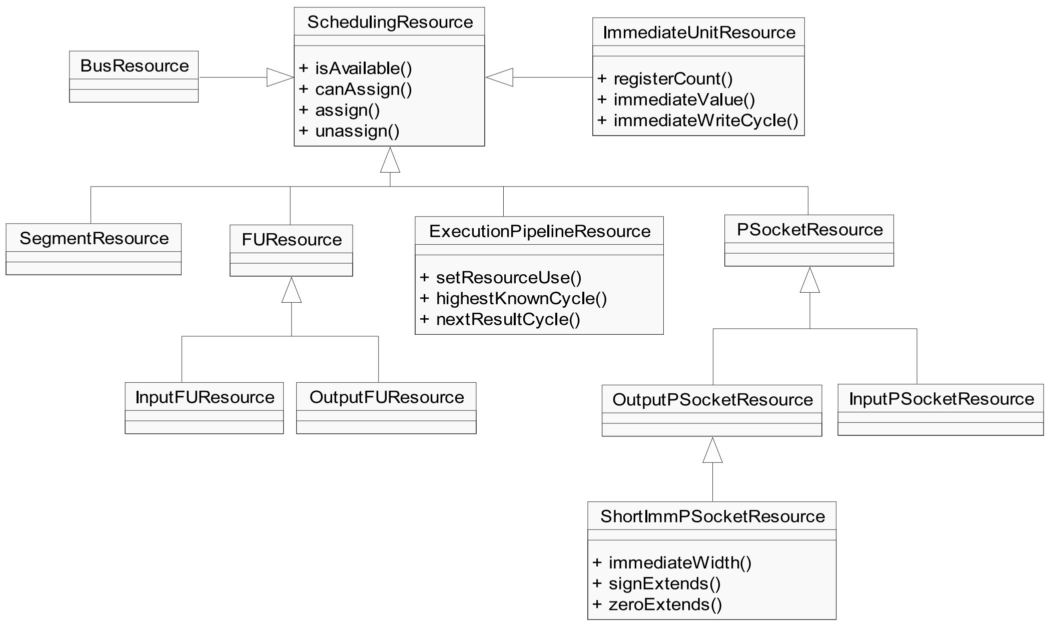

Figure 5.8: Resource model class diagram.

blocks may not always be direct. In addition, some resource types in the model do not exist in hardware at all, but represent more abstract scheduling restrictions that need to be taken into account while assigning resources to program elements.

Register files are the only significant resource completely missing from the model, because the resource manager implementation assumes registers pre-assigned.

The resource types are presented briefly in the following and a complete description is available in [21]. The class diagram of the resource model is depicted in Figure 5.8.

Input and Output P-Socket Resource. When input or output ports of a unit are assigned, one and only one socket is implicitly assigned as well. That is because in a TTA a socket can be connected to multiple ports, but a port is connected to only one socket of the corresponding direction. That is why ports and sockets are modelled by a single resource.

An “input p-socket” resource represents an input socket and the ports connected to it. Similarly, an “output p-socket” resource represents an output socket and the ports it is connected to.

The related resources of a p-socket resource are unit inputs or outputs and bus segments. If an input p-socket is a combination of an input socket and a triggering FU port, then the related input FU resource has an additional dependent resource: the execution pipeline resource.

Short Immediate P-Socket Resource. The short immediate p-socket is an artificial resource in the sense that it has no direct hardware correspondence. It is assigned when the source of a move is an immediate value.

The first segment of each bus has a related short immediate p-socket resource that represents the immediate transport capabilities of the bus. Consequentially, a bus can carry out only one short immediate transport per clock cycle, even if it is segmented.

Input and Output Function Unit Resource. The resource model includes two different FU resources, input and output FU resources, which represent the set of ports of the same direction.

When an FU resource is assigned, a p-socket and usually an execution pipeline resource must be assigned as well. Since inputs and outputs of an operation are bound to ports, the p-socket resource to be used is determined by the operation that is being executed. If the port used sets the opcode, a corresponding execution pipeline resource is also assigned. Therefore, p-sockets and execution pipelines are resources dependent on the FU resource.

Execution Pipeline Resource. Different function units have different internal resources that are used by the operations it implements. Each operation supported by the FU has a resource usage profile. Each time an operation is triggered, the resources are utilized according to the profile. An execution pipeline resource keeps book of the resource usage, that is, the state of the execution pipeline in each cycle of execution. If the resource usage profile of a triggered operation conflicts with the current state of the pipeline, the execution pipeline resource is unavailable at that cycle.

The execution pipeline resource also keeps track of operand writes and result reads. For example, if one operand of an operation is written in cycle x and the other, triggering operand in cycle y, it is not possible to write an operand of some other operation to the same FU between cycles x and y. Also, if the result of an operation is ready at cycle i and it is read at cycle j, it is not possible to start an operation that would produce a result between cycles i and j, thus overwriting the previous result.

Bus and Segment Resource. A transport bus consists of one or more segments. A bus segment can perform a data transport from source to destination. These are modelled by bus and segment resources, respectively. Segments are dependent resources of the corresponding bus resource. Input and output p-sockets connected to bus segments are related resources.

TCE version 1.0 does not support segmented buses. Therefore, in the current implementation of the resource model, there are only single-segment buses. Consequentially, on higher level, the concept of a segment can be completely ignored.

Immediate Unit Resource. Immediate units are used to store long immediate values. The immediate unit resource keeps book of each register an IU contains. Between definition of an immediate register and its last use, the register is considered as being “in use”.

The p-sockets used to transport the immediate values to or from the IU are related resources. The instruction template assigned to the definition of the immediate value is a resource dependent on the assigned IU.

Instruction Template Resource. An instruction template specifies the set of move slots used for encoding immediate bits instead of data transports, and a destination immediate unit. One of the registers of the specified IU is used for storing the immediate value. This is modelled by the instruction template resource.

5.5 Resource Manager

The purpose of the resource manager is to assign available resources on moves and other elements of the program within the limits of its scope. The resource assignment itself can also be carried out by an external entity, in which case, the resource manager is used only to verify that there are no resource conflicts due to these assignments.

Resources of the same type are managed by resource brokers, which are presented in Section 5.5.4. Co-ordination of resource brokers is delegated to the broker director. Using the given assignment plan, it decides in which order the brokers are invoked and resources tested for availability and assigned on nodes. See Sections 5.5.3 and 5.5.5 for more details.

The resource manager implementation included in the framework is based on the resource model presented in Section 5.4, but any implementation compatible with the resource manager interface may be used as a replacement.

5.5.1 Scope of Resource Management

A resource manager instance is always connected to a scheduling scope. A single resource manager may only be able to handle resource assignment in the scope of a basic block, but depending on the implementation, it may also span multiple basic blocks.

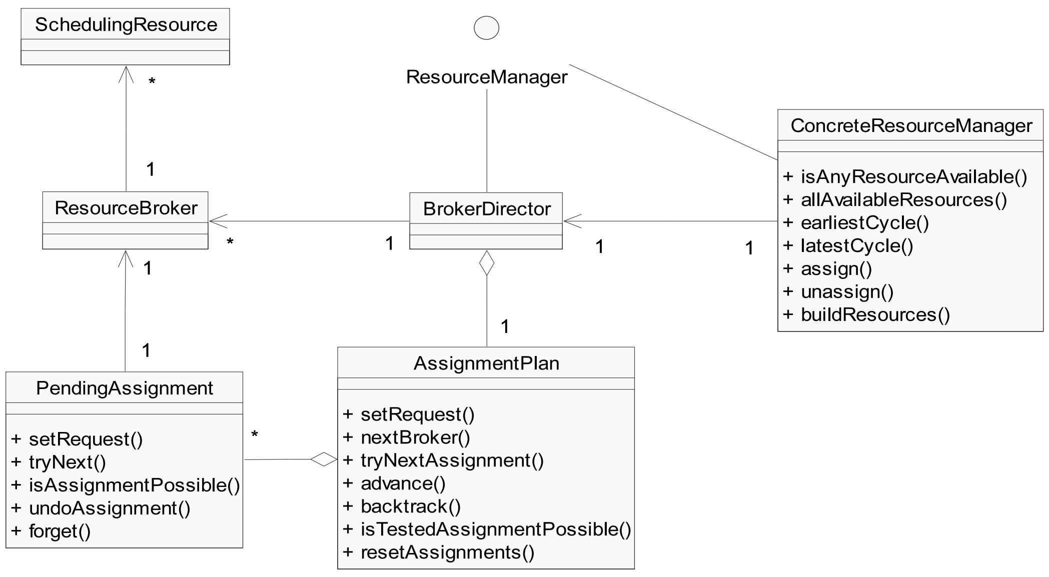

Figure 5.9: Associations between classes in the resource manager module.

5.5.2 Resource Manager Interface

The basic set of services provided by the resource manager interface is the following:

1. Tell whether a given move can be assigned all required resources without conflicts in the given cycle or not.
2. Assign resources to a move in the given cycle.
3. Unassign all resources assigned to a move.
4. Tell what is the earliest or the latest clock cycle, where all the required resources can be assigned to a move without conflicts.

This is the minimum interface required to be implemented by each resource manager, that is, a subclass of the *ResourceManager* base class. It is still missing for example a full support for external resource assignment and probably some other useful services dependent on the instruction scheduler implementation. The users are encouraged to extend this interface if required in their concrete implementations to provide finer control over the resource assignment process.

For a more detailed description of the interfaces of the resource manager, see [21].

The class diagram in Figure 5.9 depicts the associations between the resource manager and its helper classes which are described in the following sections.

5.5.3 Broker Director

The broker director is the main controller in the process of allocating and assigning resources to a move. It implements the same basic resource manager interface as the concrete resource manager, because the resource manager delegates all allocation and assignment requests to its broker director.

With the help of the assignment plan, the broker director decides in which order the resource-specific brokers are called to carry out their part in the assignment. It also implements the actual algorithm for allocating resources on a single move.

The broker director is inspired by the Mediator design pattern [22]. It does not hard-code dependencies between brokers, and the broker invocation sequence is not dependent on any broker implementation. That is why resource brokers can be replaced with another implementation without modifications to the broker director.

5.5.4 Resource Brokers

Resource brokers are responsible for managing assignment of groups of resources of the same type. There is a broker for each type of resource. Clients that need to assign a certain resource type to a move need to communicate with the appropriate brokers. The brokers also maintain links from resource objects to concrete machine parts.

Resource brokers do not communicate direcly with each other. If information exchange is required, it is done through their controller class, *BrokerDirector*.

Conventionally the resource brokers are named after the main scheduling resource they manage. For example, a *BusBroker* manages assignment of transport buses.

All concrete resource brokers are derived from the *ResourceBroker* base class. The implementations provided by TCE are briefly described in the following.

InputFUBroker. This broker is responsible for assigning function unit inputs. It maps operation inputs to input ports and sockets, and when assigning an opcode-setting operation input, operations to execution pipelines.

OutputFUBroker. This broker is responsible for assigning function unit outputs. It maps operation outputs to output ports and sockets.

InputPSocketBroker. The *InputPSocketBroker* maps a write access to a unit or a register file through one of its ports and the associated socket.

OutputPSocketBroker. The *OutputPSocketBroker* maps a read access from a unit or a register file through one of its ports and the associated socket.

ExecutionPipelineBroker. The *ExecutionPipelineBroker* maps the resource usage of executed operations to the pipeline resources of the associated function unit.

IUBroker. This broker maps long immediates to the registers of an immediate unit.

BusBroker. This broker maps data transports to buses.

ITemplateBroker. The *ITemplateBroker* is responsible for assigning instruction templates on instructions for each cycle in its scope.

Each resource broker implementation can be replaced with another without any modifications to the controller classes provided that the resource types are compatible and the implemented interface remains the same.

This becomes sensible if enhanced or case-dependent heuristics are required for selecting resources. For example, the *FUBroker* implementations provided with TCE use a simple first-fit algorithm for selecting function units for assignment. That is, the first unit found that is available and supports the given operation is selected and assigned. The user may want to give more intelligence to the broker so that it compares all available units that support the given operation, and according to their properties, for example other supported operations or connections to other units, chooses the best one in the current situation.

5.5.5 Assignment Plan

The *AssignmentPlan* class is a central helper class used in resource assignment. It determines in which order resources are assigned to a move and stores the current state of the process of assigning resources to a move.

It keeps track of tentative and potential assignments for each resource type. These assignments are represented by an ordered sequence of pending assignments. The assignment plan implements advancing and backtracking assignments for each resource type.

The *PendingAssignment* class represents the temporary state of the assignment process of a single resource type. It contains a set of potential assignments of resources and records the currently chosen assignment. Every resource broker has one associated pending assignment object. The main responsibility of the pending assignment is to manage the candidate resource assignments found by the broker and keep a record of already tried and discarded assignments.

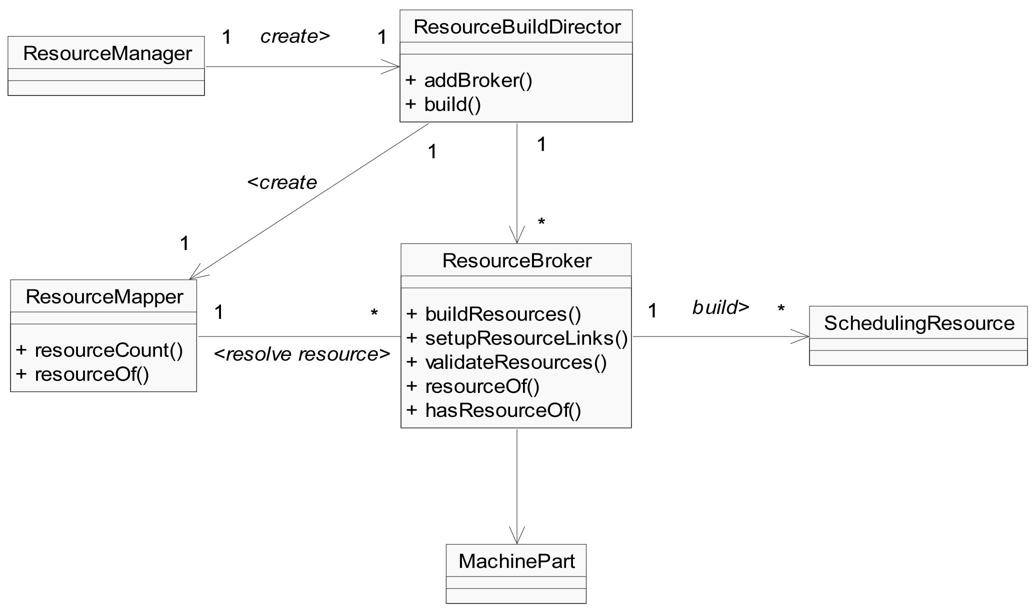

Figure 5.10: Associations between classes involved in resource model construction.

5.5.6 Resource Model Construction

The resource model itself does not provide a mechanism for building the model for example from a MOM instance. It is the responsibility of the resource manager to construct the model according to the properties of the concrete resource types and the dependencies between them. This job is delegated to resource brokers, that know the concrete type of each resource object and its relation to the parts of the target machine.

Because resources have dependencies and relations to each other, and some of the related resource instances may not exist at the time the other related object is constructed, the resource model construction is divided in two phases. In the first phase, each resource broker constructs the primary resource objects it manages. In the second phase, each broker sets up links to all dependent and related resource objects which all are now constructed.

Two classes help in coordinating the resource model construction: the *Resource-BuildDirector* is a main coordinator in the process and the *ResourceMapper* maps resource objects to concrete machine parts as required in the second phase. The classes involved in resource model construction are depicted in Figure 5.10.

5.5.7 Resource Assignment

The resource allocation and assignment process is started by constructing a resource manager instance. Then, from the object model of the target machine (MOM), the

```
function ASSIGN(move, cycle)
    plan.setRequest(move, cycle)               ▷ prepare plan to assign resources for
                                               ▷ the given move on the given cycle
    success := false
    while not success do
        if not plan.isTestedAssignmentPossible() then
            if current broker is the first then
                plan.resetAssignments()        ▷ no resource assignment found
                return false
            else
                plan.backtrack()
            end if
        else
            plan.tryNextAssignment()
            if current broker is the last then
                success := true                ▷ valid resource assignment found
            else
                plan.advance()
            end if
        end if
    end while
end function
```

Figure 5.11: Resource assignment algorithm.

resource manager builds the resource model it uses in resource bookkeeping. The initialization of the resource manager also includes forming an assignment plan, which is then passed to the broker director.

When the resource manager (or the broker director) is asked to assign resources to a move, it goes through the brokers according to the assignment plan and requests each broker to assign a resource of the appropriate type to the move. As described earlier, the assignment plan keeps track of the valid assignments for a given resource type and records which of these have already been evaluated.

Finally, if a valid assignment is found for every applicable type of resource, the resource assignment for the given move is completed. On the contrary, the resource assignment will fail, if it is not possible to find a valid assignment for some resource type even after backtracking and trying other possible combinations of assignments.

The resource assignment algorithm used is represented in Figure 5.11.

5.6 Customization and Maintenance

In this section, the main characteristics of the framework from the flexibility point of view are discussed and explained by giving a few typical use case examples and describing what kind of modifications or additions are needed in each case.

Adding a new processor resource type. When a new hardware resource type that the instruction scheduler must take into account is introduced, a couple of extensions are needed in the resource model and the resource manager implementations.

First, the new resource requires a class that implements the *SchedulingResource* interface to enable state bookkeeping for the resource. Similarly, a specialized *ResourceBroker* is needed to take care of selecting resources of this type for assignment. Finally, the resource manager implementation needs to be aware that there is a new broker capable of assigning the new resource type. The resource manager instantiates the broker and registers it to its resource build director to add the resource in the resource model at the time it is constructed. The new broker is also registered in the broker director to include it in the assignment process.

Changing the behavior of an existing processor resource. If the behavior of an already existing hardware resource is changed, it only affects the lowest levels of the resource model and the resource manager, that is, the scheduling resource object that represents the corresponding machine part and probably its broker.

If the change has an effect only on the bookkeeping of the internal state of the resource, only the resource class should be modified. On the other hand, if the change affects the resource selection and assignment processes, then also the resource broker should be modified accordingly.

Improving resource selection heuristics. If the user of the framework wants to improve the algorithm that selects a resource from a set of available candidates, either the existing broker of the corresponding resource should be modified or a new specialized implementation of the broker should be inherited.

If the resource selection algorithm in the already existing broker is modified, no other changes in the framework are needed. If the current broker implementation is replaced by a new one, the resource manager needs to also be aware that the old broker should be replaced.

Writing a new code transformation or optimization pass. Two write a new code transformation or optimization pass and include it in the scheduling chain, the user needs implement an appropriate part of the base plug-in interfaces: if the pass is independent, it should implement *StartableSchedulerModule* or if it implements a subtask of another module, it should specialize *HelperSchedulerModule*. The pass needs then to be added to the scheduler configuration in the desired position. The code transformation algorithm may also implement a part of the scheduler pass hierarchy, depending on its required inputs.

6. INSTRUCTION SCHEDULING ALGORITHM IMPLEMENTATIONS AND VERIFICATION

This chapter describes how the instruction scheduler framework was put to the test and verified by implementing a couple of "proof-of-concept" instruction scheduling algorithms. Some benchmark results gathered using the TCE testbench are also presented in the end of the chapter.

6.1 Proof-of-Concept Algorithm Implementations

To verify that the framework meets the requirements, is as flexible as planned and generally makes sense, two instruction scheduling algorithms were written for the framework. The implemented algorithms use the interfaces provided by the framework as much as possible so that any design flaws and inflexibilities concerning the interfaces would be revealed. Some minor design changes were indeed made during the process of writing these algorithms.

Some benchmarks were also run using these algorithms. The purpose of these benchmarks was not to evaluate the performance of the implemented algorithms, but to verify that the implemented algorithms produce a correct schedule and show how the used algorithm and the input parameters, such as the target application and the given architecture, affect the produced schedule. The benchmarks also give an early perspective on the potential of the framework and scheduling algorithms written on it.

This verification process as a whole covers most of the framework use cases and possible user needs: developing new algorithms, scheduling for a fixed target and evaluating effects of different algorithms and parameters on the produced schedule (see Sections 4.2 and 4.3). Therefore, the successful completion of these benchmarks is considered a thorough verification.

The first instruction scheduling algorithm implementation, called the *sequential first-fit resource mapper*, can not perform proper instruction scheduling, but uses the resource manager interface to directly map the provided intermediate program representation to concrete processor resources of the target architecture. The second algorithm implemented is a complete basic block instruction scheduler. The implementations of both the algorithms are presented in the following sections.

6.1.1 Sequential First-Fit Resource Mapper

The sequential first-fit resource mapper “schedules” the given IR for the target architecture without any parallelization or other optimizations by simply assigning required processor resources to the program. In case an operation latency is larger than one cycle, the algorithm generates no-operations (NOPs) to wait for the result. The resource assignment algorithm assumes registers pre-assigned by the front-end.

The resource mapper consists of a single scheduler pass written according to the plug-in module interfaces of the framework described in Section 5.1.1. The class that implements the resource assignment pass is *SequentialFirstFitResourceAllocator*, a subclass of *StartableSchedulerModule*. For resource bookkeeping, the resource mapper uses a *SimpleResourceManager* instance, which is a simple implementation of the resource manager interface of the instruction scheduler framework (see Section 5.5).

6.1.2 Basic Block Scheduler

The basic block scheduler implemented on the scheduler framework is a full-blown instruction scheduler capable of scheduling and assigning resources either to a whole program or to a single basic block.

At the highest level, the class *BasicBlockSchedulerPass* implements a startable scheduler module based on the plug-in interface of the framework (see Section 5.1.1). It takes the source program and the target architecture as input and instantiates a *BasicBlockScheduler*, which is a class that implements the actual instruction scheduler by specializing multiple parts of the scheduler pass hierarchy described in zSection 5.3. Because of this polymorphism, the class is able to take many different scopes and program representations as input: a whole program or a single procedure as POM, a CFG or a DDG, or just a single basic block.

The scheduling process is started by finding the basic blocks in the program. This is done by constructing a CFG for each procedure in the program. At this point, a procedure-wide DDG is built, which is later used also for filling delay slots of control flow operations. Then, the CFG’s are scheduled node by node, nodes being the found basic blocks.

For each basic block, a DDG (a subgraph of the procedure-wide DDG) and a resource manager instance is constructed, which are then used to schedule the moves in the given basic block. *SimpleResourceManager* is used in resource bookkeeping and assignment, just like in the sequential resource mapper.

The basic block scheduler is a local scheduler in the sense that most optimizations are only run in the scope of a single basic block. Only the delay slot filler is able to import operations from one basic block to another.

Copying delay slot filler. The TTA has code-visible jump latency, which means that branching results in additional delay slots after the jump. The delay slot filler tries to "hide" these slots by filling them with instructions from successive basic blocks. If the jump is conditional, the moves in the instruction are also given the same condition.

The delay slot filler is run after a whole procedure has been scheduled. It checks all basic blocks that are jump targets for code that could be moved to the delay slots of the jump. All moves keep their relative positions. Transport buses need to be reassigned for these moves, but other resources remain intact.

Register copy adder. A sensible target architecture usually has limited connectivity. The scheduler must cope with cases when there is no direct connection from the source of a move to the destination. For example, the output port of a function unit that is assigned to perform an operation may not be directly connected to the input port of the previously assigned register file. The basic block scheduler deals with these cases by adding temporary register moves.

This involves finding a register that is connected to both the source and destination of the original move and then inserting an additional data transport through this temporary register. Multiple temporary register moves may be required if connectivity is very limited.

Adding temporary register moves increases the cycle count of the output program but is necessary to produce executable code.

The register copy adder is a pass that handles required temporary register moves in the basic block scheduler. In case of missing connectivity in operand moves of operations, it tries to add the minimum required number of extra data transports through registers in the program.

In addition, it annotates the moves of the operation with missing connectivity with a best FU assignment, that is, the assignment that requires least temporary moves. Long immediates are also annotated with the best IU assignment. In this way, the register copy adder "guides" the resource manager in assigning resources.

After executing this pass, the scheduler may assume, that there is enough connectivity available to produce a valid schedule for the program.

Software bypasser. This helper class applies software bypassing as described in Section 2.4. While scheduling an operation, it sees whether it can transfer the input operands of the currently scheduled operation directly from the outputs of the operation on which it is data dependent.

After scheduling the result reads of the operation, dead-result elimination is applied. If possible, the bypassed and thus obsolete result move is removed from the

Architecture	ALU FU's	Registers	Buses
Minimal	3	1x8	1
Restricted	3	1x8	2
4-bus	3	1x16	4
Huge	20	8x32	32

Table 6.1: Benchmark machine configurations.

schedule.

Long immediate handling. Some immediate values in the program may be large enough to exceed the width of the source field of the move slot in the used instruction encoding. If the scheduler cannot find a suitable encoding for the immediate of the required width, the so called long immediate needs to be transported through an immediate unit. The simplest way for the scheduler to handle this is to insert an instruction before the instruction containing the original immediate transport, and in that instruction, define a transport to an immediate unit. The original immediate transport is then replaced by a read from the register of the immediate unit.

A more advanced algorithm that looks for a space for the long immediate definition in the earlier clock cycles without inserting a new instruction would be preferred. This method is used in the basic block scheduler.

The following sections describe how the functionality of the framework was verified by running selected benchmarks using the previously presented instruction scheduling algorithm implementations and a set of target architectures.

6.2 Test Cases and Benchmarks

The instruction scheduling algorithm implementations were verified and benchmarked using a set of real-life test cases and four different 32-bit TTA machine configurations. The processor resources in each architecture are presented in Table 6.1. In addition to the listed resources, each machine contains a load/store-unit, an immediate unit and a global control unit. They all have a fully connected IC.

The purpose of the *huge* machine is to provide the scheduler with as much ILP as possible. Using this machine we are able to determine how well the scheduler can exploit available ILP in the software when processor resources are not limited. Using the constrained machines, *minimal*, *restricted* and *4-bus* which contain less duplicated resources we can see how well the scheduler utilizes more scarce resources while still trying to exploit as much ILP as possible.

Each test case was first transformed to LLVM bytecode using the TCE front-end. The bytecode was then scheduled for all target architectures using the sequential

SEQUENTIAL	**Minimal**	**Restricted**	**4-bus**	**Huge**
Cycle count	718096	718096	476809	439708
Register reads	246798	246798	145417	145097
Register writes	201939	201939	133364	133172
Operations/cycle	0.27	0.27	0.23	0.25
BB SCHEDULED				
Cycle count	572515	349458	198085	197346
Register reads	203556	147142	83057	71265
Register writes	167476	119775	92572	86644
Operations/cycle	0.34	0.56	0.55	0.55

Table 6.2: ADPCM benchmark results.

first-fit resource mapper, referred to as "Sequential" in the benchmark results tables. Correctness of the produced schedule was verified by simulating each test case with the TCE simulator [3] and comparing the produced output to the expected correct output. Some key performance figures were also gathered to see the effect different target architectures had on the schedule.

The same procedure was then repeated using the basic block scheduler as the scheduling algorithm. The same performance data was collected to see how the selected algorithm affected the schedule. The benchmark results produced using the basic block scheduler are referred to as "BB Scheduled" in the comparison tables.

In the comparisons we use the following performance figures: *cycle count*, *register reads and writes* and *operations per cycle.* Cycle count is the most fundamental measure, since it tells exactly how long the program takes to execute. Register reads and writes measure how well the scheduler can bypass unnecessary register accesses, and finally, operations per cycle indicates how well the program was parallelized. All types of operations (arithmetic as well as memory operations) are included in this count.

The test cases and produced results are presented in the following.

ADPCM. The first test case is a short ADPCM encode/decode routine from the DSP-Stone test suite [23]. The benchmark results are shown on Table 6.2.

FFT. The second test case performs a 1024-point in-place radix-4 decimation-in-time (DIT) FFT on the given input. The benchmark results are presented on Table 6.3.

JPEG. This test case decodes a 227 x 149 pixel JPEG image. The results are presented on Table 6.4.

SEQUENTIAL	**Minimal**	**Restricted**	**4-bus**	**Huge**
Cycle count	4996643	4996643	2996533	2562265
Register reads	2000520	2000520	1155980	980615
Register writes	1420840	1420840	846132	719413
Operations/cycle	0.31	0.31	0.27	0.26
BB SCHEDULED				
Cycle count	3861263	2203976	1218412	762193
Register reads	1435000	1066880	815878	437815
Register writes	870935	540399	552440	307620
Operations/cycle	0.40	0.69	0.73	0.87

Table 6.3: FFT benchmark results.

SEQUENTIAL	**Minimal**	**Restricted**	**4-bus**	**Huge**
Cycle count	76210262	76210262	41434170	35067561
Register reads	31373900	31373900	16911700	14206800
Register writes	22448000	22448000	12312500	10096800
Operations/cycle	0.32	0.32	0.29	0.27
BB SCHEDULED				
Cycle count	61017751	32888917	16023927	9109674
Register reads	25441900	18803400	11962800	6027600
Register writes	16704300	10486000	8273490	4422240
Operations/cycle	0.40	0.73	0.74	1.03

Table 6.4: JPEG decoding benchmark results.

Tremor. This test case decodes a 40kB Ogg Vorbis file and is quite a lot larger case than the previous ones when it comes to total cycle count. The benchmark results of this case are presented in Table 6.5

Summary of results. The results show that the used target architecture does have an effect on the produced schedule. Duplicating processor resources generally reduced the number of cycles required to execute the program. When using the sequential scheduling algorithm, the most notable change in the cycle count was achieved by adding registers. The 4-bus machine has double the number of registers compared to the restricted machine. This reduced the amount of additional spill code required for correct operation, thus also reducing the total cycle count. Cases that use a lot of registers, such as the JPEG-case, did furthermore benefit from the extra registers in the huge machine. The basic block scheduler was able to parallelize the spill code on the multi-bus machines so that the effect of adding registers was not so clearly visible.

SEQUENTIAL	**Minimal**	**Restricted**	**4-bus**	**Huge**
Cycle count	1928470768	1928470768	1087130275	823249470
Register reads	799720000	799720000	452490000	351344000
Register writes	564574000	564574000	318757000	240659000
Operations/cycle	0.31	0.31	0.28	0.26
BB SCHEDULED				
Cycle count	1529302891	838487211	425143835	214394914
Register reads	631424000	479453000	331498000	160835000
Register writes	404367000	263318000	224326000	122223000
Operations/cycle	0.40	0.72	0.72	1.01

Table 6.5: Tremor benchmark results.

Additional resources also had an effect on the number of register reads and writes. Reduced spilling resulted in less total register accesses in the sequentially scheduled code. The number of register accesses in the basic block scheduled code was also reduced when moving from minimal to huge configuration. The basic block scheduler was able to bypass registers more effectively when there were less hardware constraints to cope with.

In some cases, the basic block scheduler produces more register writes than reads. This does not mean that some values are written to registers and never read, but because guard writes could not be differentiated from regular register writes in the simulator output data, the statistics become somewhat biased. Guard reads are not included in register read statistics.

Since the amount of available ILP in the scope of single basic blocks is limited, the results were not outstanding. A lot better scheduling results should be achieved by introducing a global scheduling algorithm and using additional optimization passes in the back-end as well as in earlier phases of compilation. A detailed performance analysis of the used scheduling algorithms and optimization passes is out of the scope of this document.

7. CONCLUSIONS

This thesis presented a software framework that implements a crucial part of code generation for TTA processors: instruction scheduling. The framework is a part of the compiler back-end in a TTA-based co-design environment. In addition to instruction scheduling, any architecture-dependent code transformations can be applied on the source program using the framework. It may be used as a stand-alone tool or as part of another application.

As inputs, the framework takes the unscheduled source application translated to generic bytecode by the compiler front-end and a description of the target architecture. Following the user-defined code transformation and scheduling chain, the scheduler optimizes and compiles the source program to a form executable on the given target architecture.

The main requirements set for the framework were easy configurability and flexibility. The requirements were met by designing an object-oriented and modular architecture for each part of the framework, in addition to a plug-in interface that allows the user to write and setup scheduler passes without recompiling the whole framework. This helps users in experimenting with different optimizations and parameters and in evaluating their effect on target application performance. The user is also able to plug-in and try out different scheduling algorithms easily in attempt of finding the most effective schedule for the given target architecture. In addition, researchers may take the flexibility points of the framework in use when developing new instruction scheduling algorithms and code optimization techniques.

The framework was verified and put to test by writing two "proof-of-concept" algorithms according to the provided interfaces and running a set of real-life benchmarks with them using the TCE testbench suite. This process as a whole covered most of the framework use cases and was therefore considered a thorough verification of the framework concept and functionality.

All the benchmarks could be compiled for the target architectures using the first written algorithm that merely mapped the source program on the target without applying any optimizations or parallelization. The TCE simulator was used to verify the correctness of the schedule by comparing the produced output to the expected correct output. The simulator was also used to gather performance statistics which illustrated the effect different scheduling parameters had on the produced schedule.

After the sequential benchmarks were done, the test cases were scheduled using a more advanced algorithm: the basic block scheduler.

The results showed that the target architecture had an effect on the produced schedule: the more duplicated resources, the less time it took for the compiled code to execute. As expected, the used scheduling algorithm also affected the schedule. The basic block scheduler could exploit available ILP in the target architectures as well as in the source program, thus producing lower cycle counts than the sequential scheduling algorithm. Better results are possible by introducing a global scheduling algorithm and more advanced code optimizations. However, writing the algorithms and utilizing them in the benchmarks proved that the framework meets the requirements and leaves space for future extensions and improvement. Implementing these should be easy because of the modular and flexible software architecture.

BIBLIOGRAPHY

[1] J. A. Fisher and B. R. Rau, "Instruction-level parallel processing," pp. 41–49, 1995.

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

[3] P. Jääskeläinen, "Instruction Set Simulator for Transport Triggered Architectures," Master's thesis, Department of Information Technology, Tampere University of Technology, Tampere, Finland, P.O.Box 553, FIN-33101 Tampere, Finland, Sep 2005, *See* `http://tce.cs.tut.fi/`.

[4] H. Corporaal and J. Hoogerbrugge, "Code generation for Transport Triggered Architectures," in *Code Generation for Embedded Processors.* Heidelberg, Germany: Springer-Verlag, 1995, pp. 240–259.

[5] J. Janssen, "Compiler strategies for transport triggered architectures," Ph.D. dissertation, Delft University of Technology, The Netherlands, 2001.

[6] Free Software Foundation, "Gcc, the gnu compiler collection," http://gcc.gnu.org. [Online]. Available: http://gcc.gnu.org

[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation and Optimization*, Palo Alto, CA, March 20–24 2004, p. 75.

[8] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Proc. Multimedia on Mobile Devices 2007*, 2007, pp. 65 070X–1 – 65 070X–11.

[9] L. Laasonen, "Program Image and Processor Generator for Transport Triggered Architectures," Master's thesis, Department of Information Technology, Tampere University of Technology, Tampere, Finland, P.O.Box 553, FIN-33101 Tampere, Finland, Apr 2007, *See* `http://tce.cs.tut.fi/`.

[10] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., 1986.

[11] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools.* Morgan Kaufmann, 2005.

[12] A. Cilio, H. J. M. Schot, and J. A. A. J. Janssen, "Architecture Definition File: Processor Architecture Definition File Format for a New TTA Design

Framework," Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2003-2006.

[13] P. Faraboschi, J. A. Fisher, and C. Young, "Instruction scheduling for instruction level parallel processors," in *Proceedings of the IEEE*, vol. 89, no. 11. Washington, DC, USA: IEEE Computer Society, 2001, pp. 1638–1659.

[14] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface.* San Francisco, US: Morgan Kaufmann, 1998.

[15] S. S. Muchnick, *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 3rd edition.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[17] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[18] A. Cilio, A. Metsähalme, and V. Guzma, "TTA Instruction Scheduler Functional Requirements," Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2004-2006.

[19] A. Cilio and A. Metsähalme, "Program Object Model," Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2004-2006.

[20] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual.* Addison Wesley, 2001.

[21] A. Cilio, A. Metsähalme, and V. Guzma, "TTA Instruction Scheduler Design Notes," Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2005-2007.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns.* Addison-Wesley, 1995.

[23] V. Zivojnović, J. M. Velarde, C. Schläger, and H. Meyr, "DSPSTONE : A DSP-oriented benchmarking methodology," in *Proceedings of the International Conference on Signal Processing and Technology(ICSPAT'94)*, Dallas, TX, 1994, pp. 715–720.