TAMPERE UNIVERSITY OF TECHNOLOGY
Degree Programme in Information Technology

Viljami Korhonen

TOOLS FOR FAST DESIGN OF APPLICATION-SPECIFIC PROCESSORS

Master of Science Thesis

Examiners: Prof. Tommi Mikkonen and
Dr. Pertti Kellomäki
Examiners and subject approved by
Department Council
3rd December 2008

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Degree Programme in Information Technology
Korhonen, Tuomas Viljami: Tools for Fast Design of Application-Specific Processors
Master of Science Thesis: 47 pages
January 2009
Major subject: Software Engineering
Examiners: Prof. Tommi Mikkonen and Dr. Pertti Kellomäki
Keywords: transport triggered architecture, compiled simulation, eclipse plugin

In the modern computing world, there is always an endless demand for more effective processors, be the effectivity in processor speed, power-consumption or some other area. Most of the microprocessors used nowadays belong in appliances that run only a limited set of software. This is especially true in embedded devices. When striving to develop the most effective processors possible for them, the a priori knowledge of the software can be used to tailor the processors to fit better for the applications. These kind of processors are called application-specific processors or ASPs.

Transport Triggered Architecture (TTA) is a processor design paradigm which can be used for quickly developing application-specific processors. TTA-based Codesign Environment (TCE) is a design environment developed at the Tampere University of Technology. Its primary goal is to produce a full-fledged software kit that can be used for creating TTA processors and compile applications for them. TCE consists of a number of programs such as a processor designer, a compiler, a design space explorer, and a simulator.

In this thesis, a new compiled simulation engine for the simulator is introduced. It can be used for simulating TTA applications faster than with the previous simulation engine. Simulation speed is essential when one wants to simulate long-lasting test cases such as audio or video decoders.

In addition to the compiled simulator, graphical tools for fast manual processor design space exploration are introduced. These include an experimental TCE Eclipse plugin which can be used for quick creation and simulation of TTA applications using the Eclipse IDE, and further enchancements to the processor designer application.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
Korhonen, Tuomas Viljami: Työkaluja Nopeaan Sovelluskohtaisten Prosessorien Kehittämiseen
Diplomityö: 47 sivua
Tammikuu 2009
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Tommi Mikkonen ja TkT. Pertti Kellomäki
Avainsanat: transport triggered architecture, kääntävä simulaatio, eclipse-liitännäinen

Nykyään tietokonemaailmassa on aina tilaa tehokkaammille prosessoreille, tarkoitettiinpa tällä tehokkuudella sitten prosessorin nopeutta, virrankulutusta tai jotain muuta sen ominaisuutta. Useimmat mikroprosessorit kuuluvat nykyään sellaisiin sovellusalueisiin, jotka suorittavat vain hyvin rajattua ohjelmistoa, kuten erityisesti sulautettujen laitteiden tapauksessa. Kun halutaan kehittää mahdollisimman tehokkaita prosessoreita tällaisille sovellusalueille, tulee myös niillä suoritettava ohjelmisto ottaa huomioon prosessoria kehitettäessä. Tällaisia prosessoreita kutsutaan sovelluskohtaisiksi prosessoreiksi (application-specific processor, ASP).

Transport Triggered Architecture (TTA) on eräs sovelluskohtaisille prosessoreille hyvin soveltuva suunnitteluparadigma, jonka avulla uusia ASP:tä voidaan suunnitella nopeasti ja edullisesti. TTA-based Codesign Environment (TCE) on kehitysympäristö, joka on kehitetty Tampereen Teknillisessä Yliopistossa. Sen päätavoitteena on tarjota täysipainotteinen ohjelmisto TTA-prosessorien ja -sovellusten suunnittelua varten. TCE koostuu monesta ohjelmasta, kuten prosessorien suunnitteluohjelmasta, kääntäjästä, suunnitteluavaruuden tutkimisohjelmasta ja simulaattorista.

Tässä diplomityössä on rakennettu uusi simulaatiomoottori nykyiseen TTA-simulaattoriin, jonka avulla voidaan simuloida TTA-sovelluksia nopeammin kuin aikaisemmalla simulaatiomoottorilla. Simulaation nopeus on olennaista, kun halutaan simuloida pitkäkestoisia testitapauksia, kuten esimerkiksi äänen- tai videon purkusovelluksia.

Kääntävän simulaattorin lisäksi työssä esitellään graafisia työkaluja manuaaliseen prosessorien suunnitteluavaruuden tutkimiseen. Näitä ovat kokeellinen TCE Eclipse-liitännäinen, jota voidaan käyttää nopeaan TTA-sovellusten luomiseen ja simuloimiseen Eclipse-ympäristössä, sekä laajennokset prosessorinsuunnitteluohjelmaan.

PREFACE

The work for this thesis was done in the Department of Computer Systems of Tampere University of Technology (TUT) in 2007-2008 for a codesign software implemented as a part of the Energy-Efficient Computing in Ubicom Systems (ECUUS) project funded by the National Technology Agency of Finland.

I would like to thank professor Jarmo Takala for giving me a chance to work on this interesting project. I am very grateful for Pertti Kellomäki, Pekka Jääskeläinen and Tommi Mikkonen for their endless supply of ideas on how to improve my work. I would also like to thank all the hard working people of the TCE project for creating such an excellent working atmosphere.

Thanks to my friends for not forgetting me while I was trying to work on the thesis. Finally, I would like to thank my family for their support throughout my life.

Tampere, January 25, 2009

Viljami Korhonen

CONTENTS

LIST OF ABBREVIATIONS

ADF	Architecture Definition File
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Set Processor
ASP	Application-Specific Processor
DAG	Directed Acyclic Graph
DSDB	Design Space Database
HDB	Hardware Database
HLL	High Level Language
IDF	Implementation Definition File
ILP	Instruction Level Parallelism
JNI	Java Native Interface
LLVM	Low Level Virtual Machine
MAU	Minimum Addressable Unit
MOM	Machine Object Model
OSAL	Operation Set Abstraction Layer
OTA	Operation Triggered Architecture
PIG	Program Image Generator
POM	Program Object Model
SWT	Standard Widget Toolkit
TCE	TTA-based Codesign Environment
TPEF	TTA Program Exchange Format
TTA	Transport Triggered Architecture
VHDL	Very high speed integrated circuit Hardware Description Language
VLIW	Very Long Instruction Word
XML	Extensible Markup Language

1. INTRODUCTION

Since computers have become very common items in our daily lives, one could have a crude estimate of how vast the total amount of microprocessors is. This estimate is usually wrong because there are way more microprocessors than those that reside inside our desktop computers. Microprocessors have endless number of applications in other areas such as in cell phones, DVD players, refrigerators and cars, just to name a few.

Microprocessors in these appliances are usually limited to running a very specific piece of software which raises questions: *if* and *how* the processors can be optimized to fit better for their targeted software. Optimization of the microprocessor towards the application is more or less important depending on the usage of the microprocessor. In embedded devices, customization can often be a deciding factor for the success of the product whereas in desktop computers it can be practically impossible because of the amount of different software. Application-specific integrated circuits (ASIC) could be used but their design time is usually long which makes them expensive to develop and their risks higher than when using general purpose processors which on the other hand are not nearly as effective as an ideal ASIC could ever be.

Transport Triggered Architecture (TTA) is a novel processor design paradigm similar to VLIW that can be used for quickly developing application-specific processors for virtually any kind of high level language application, so it can be defined as an application-specific instruction set processor (ASIP). TTA offers easy customization of the processor through basic building components such as register files, function units, buses, and through the usage of custom operations. When designing effective TTA processors for large (high level language) applications, there has to be good tools for the job. TTA-based Codesign Environment (TCE) is a toolset developed since 2003 at the Tampere University of Technology and it aims to provide a wide range of tools for developing TTA processors. The toolset starts from a C language program and ends up with a finished bit image of the optimized TTA program and a VHDL description of the TTA processor.

A good processor design toolset includes a processor simulator that is capable of simulating any kind of application or processor that is possible to develop with the toolset. For improving the speed of the existing TTA simulator, a new compiled simulator engine was developed. With the compiled simulator, it becomes possible to execute long applications in a reasonable time frame.

Finally, graphical tools for manual processor design space exploration are introduced. These include an experimental Eclipse plugin, which can be used for creating, simulating and evaluating TTA applications using the Eclipse IDE. Some improvements to the existing processor designer application are also introduced.

The purpose of this thesis is to describe the TCE Compiled Simulator and work as a documentation material for it. Chapter 2 describes application-specific instruction set processors in general as well as TTA and its programming capabilities. Chapter 3 is an introduction for TCE and describes most of the TCE tools that are related to the compiled simulator. Chapter 3 also describes the most important TCE code structures that were used while creating the compiled simulator. Chapter 4 describes processor simulation principles and the TCE Compiled Simulator design and implementation. Chapter 5 is for verifying and benchmarking the compiled simulator using the test cases found in TCE. Chapter 6 describes tools developed for fast manual design space exploration such as the experimental TCE Eclipse plugin and improvements to the processor designer application. Chapter 7 is left for future improvement ideas for the compiled simulator and the TCE Eclipse plugin. Chapter 8 concludes the thesis and gives some last thoughts of the development.

2. APPLICATION-SPECIFIC INSTRUCTION SET PROCESSORS

Application-specific processors or ASPs by definition are processors designed to run only a limited set of applications with an instruction set specifically tailored for the application(s). Application-specific instruction set processors or ASIPs are a special case of ASPs that have the minimal functionality to run any high level language (HLL) program. In this thesis we focus solely on ASIPs because of their suitability for any kind of application (and because TTA can be classified as a one). [1]

2.1 Overview

A common example of an ASP or an ASIP would be a signal processing CPU (see Figure 2.1) running a set of instructions on the input signal and then producing a modified signal as an output. In this case, there is only one application to be run, but there can be different kind of data as input as well as output.

Application-specific instruction set processors can be very efficient when compared to traditional general purpose processors because the problem domain, that is, the application or a set of applications is usually well known in advance and therefore the processor can be designed to fit precisely for the applications it is supposed to run. Everything unneccessary for the problem domain of the application can be taken away from the processor architecture. For example, if the application does not use any or only a little floating point operations, there is no need to implement them in hardware, but rather emulate on software. If the application never uses integers larger than 8 bits, then all the 16- and 32-bit registers and operations can be safely removed from the processor.

Finally, one of the most important optimization techniques specific to ASIPs is that they can implement so called custom operations for the most commonly used parts of the program or for some specific algorithms. For example, if the code consists of many additions (ADD) and subtractions (SUB) like the following, it

input signal
digital signal processor (DSP)
output signal

Figure 2.1: **A Digital Signal Processor.**

could be replaced with a single call of a special operation “ADDSUB”.

```
ADD C, A, B     // C = A + B
SUB D, A, B     // D = A - B
```

The same addition and subtraction optimized using special operation “ADDSUB” that has two inputs and two outputs:

```
ADDSUB C, D, A, B         // C = A + B and D = A - B
```

This makes it possible to calculate the addition and the subtraction simultaneously inside a function unit potentially reducing the overall latency of the operations. It also makes the code smaller in terms of instruction calls.

2.2 Toolset-Assisted Processor Design Space Exploration

The efficiency of ASIPs is based on their high specialization to the application and its problem domain. However, their specialization does not come without a tradeoff. Designing ASIPs from the beginning can be a very time-demanding and expensive process because of the non-recurring engineering costs such as the costs of research, designing and testing of the product. The time it takes to develop an ASIP can be very long, and because the annual performance increase in general purpose processors is much higher, the outcome might not be as good as expected. As such, designing ASIPs from the beginning is only practical for large-volume products and sales and even then there is a risk of not achieving the required correctness or performance levels. Toolset-assisted processor design space exploration aims to ease and speed up this process. [1]

Processor design space exploration is the process of finding out a suitable processor for the application. This suitability could be different depending on the target usage of the ASIP. For example in mobile devices, power consumption and size requirements often outweigh the performance requirements of the chip whereas on desktop computers, speed is usually the most wanted property.

The design space can be potentially very large, so there has to be effective tools for reducing it to a reasonable size without sacrificing too much in quality or extent. Processor design space exploration tools can be roughly divided in levels of automation into manual, semi-automatic and fully automatic.

Manual processor design space exploration. In manual processor design space exploration, the processor designer deals directly with the processor components and attributes such as register files, function units, buses, implemented operations and so on. To help in this task, there can be tools for quickly adding and removing components and modifying the processor parameters. For measuring the relative speeds of the modified processors and for finding out the best one, there has to be a processor simulator capable of running the evaluated application on each processor implementation variation. In addition, there could be tools for estimating the processor charasteristics for qualities such as energy consumption and die surface area requirements, depending on the target usage of the processor. The decision in which direction to proceed next in the processor design space is always up to the processor designer himself.

Automatic processor design space exploration. Automatic processor design space exploration tools leave very little or nothing left for the processor designer to configure by hand, except for the initial parameters of the exploration. These parameters are goals, requirements and limitations for the exploration and could include things such as processor performance requirements (for some specific application), maximum exploration runtime, memory limitations and iteration counts.

A simple example of automatic processor design space exploration could be to have a minimal working processor architecture and then gradually increase the number of components and operations until their addition no longer improves the performance of the processor. Workflow in this kind of automatic exploration could be as follows:
Prior to exploration: set up the initial exploration parameters as described before.

1. create a minimal processor
2. modify the processor by adding components and operations
3. generate an updated version of the processor
4. simulate the application(s) on the generated processor
5. estimate the size, speed and power consumption of the processor
6. go to 2 until N number of iterations are done or if the exploration goals have been achieved.

Even though the total exploration time might be long depending on the simulation time of the application and the number of iteration steps, the simulation speed is not as important as in manual exploration, where the processor designer often has to check whether his changes made any difference to the outcome and decide the actions. Automatic exploration could easily take hours or even days without requiring additional input from the user. Another advantage in automatic processor design space exploration is that it is less error-prone because there is less user interaction than in manual exploration (of course setting good initial parameters for the exploration is more necessary than ever!). The downside of using automatic exploration is that if there is no decent exploration algorithm available, the processor variations generated by the explorer application might be too similar to each other to make any difference to the outcome. There is also a risk of missing some good spots in the design space.

Semi-automatic processor design space exploration. Semi-automatic processor design space exploration combines parts from both of the methods described before. The processor designer is again responsible for choosing in which direction to proceed next but this time has more automation to help in this task than in plain manual exploration. Workflow in semi-automatic processor design space exploration could be as follows:

1. add N new components / add a new custom operation.
2. create a new processor
3. simulate the new processor
4. estimate the new processor
5. jump to 1 until required cycle count is achieved
6. continue exploration manually.

2.3 Transport Triggered Architecture (TTA)

One of the most important ways to improve processor performance is to add more concurrency into processing of instructions. In superscalars [2] (such as in Pentiums ever since Pentium Pro and Pentium II), instructions that are possible to execute in parallel are detected by hardware during run time. On each cycle, the processor tries to fetch, decode and execute as many successive instructions as possible while preserving dependencies between the instructions. While easy to program, superscalars have a problem with costs it takes to detect the dependencies between instructions. The more there are instructions running simultaneously, the faster the costs of detection grows. [1]

In Very Long Instruction Word (VLIWs) architectures, this concurrency checking is done by the compiler during compile time. The VLIW compiler statically schedules and packs multiple operations into single instructions and guarantees their independence. This has the scalability advantage of not being dependent on complex hardware to detect the instruction dependencies during run time. The problem

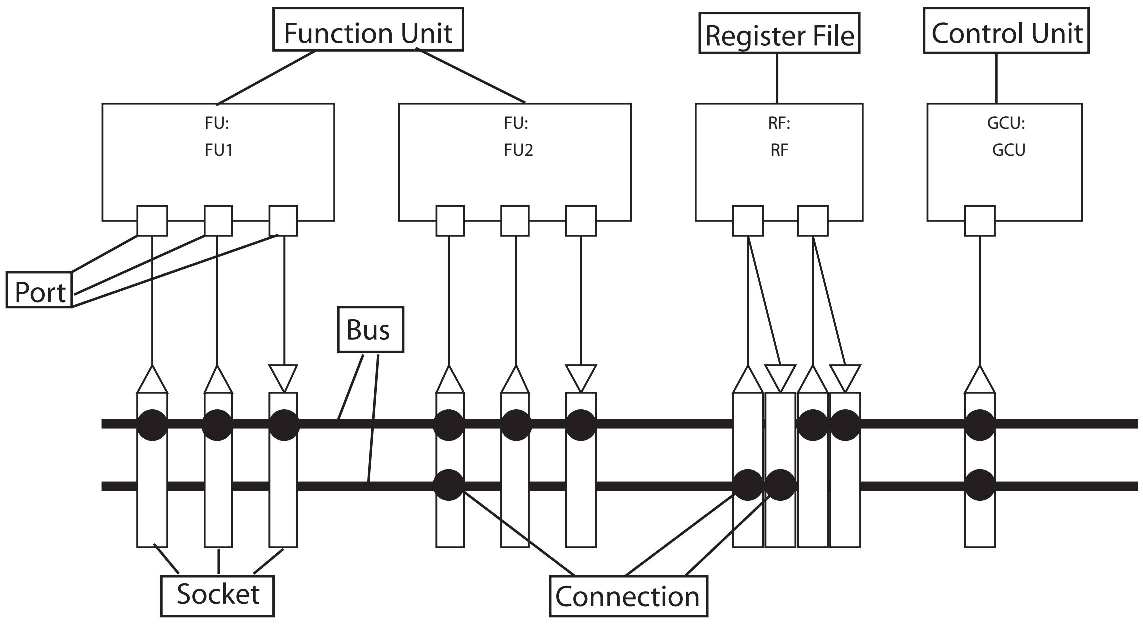


Figure 2.2: **An example of a two-bus TTA machine. [3]**

with VLIWs is that when adding more function units to gain more parallelism, the maximum amount of data needed to transport grows fast because of potentially many concurrent data transports between register files and function units. Data transports that are done via an interconnection network are needed more due to the maximum possible number of simultaneous operations, but in many cases, the network is still being underutilized when there are not enough concurrent operations going on. [1]

Transport Triggered Architecture or TTA is a relatively new processor architecture style similar to VLIW. TTAs can be constructed of the following building blocks: a control unit, function units, register files and an interconnection network consisting of one or more buses. An example of a two-bus TTA machine is shown in Figure 2.2. The big difference between TTA and VLIW comes when managing operation executions; In VLIWs operations are packed into one big instruction and when executing it, each output and input have to be transported via buses. Because of potentially high number of simultaneous data transfers in VLIW, the amount of register file ports and buses has to be very high to support them. TTA takes a different approach: the only real instruction available in TTA is a “Move” instruction, which moves data between TTA components such as register files and function units. Operations are triggered as a byproduct of these moves when a value is written to the trigger port of the function unit, hence the architecture name “Transport Triggered”. Data transports can also be done between ports of different function units so that no intermediate values need to be stored in register files between consecutive operation calls. Scalability in TTAs is better than in VLIWs because the buses can be utilized more effectively due to smaller amount of data needed to transport simultaneously. That is, TTA offers more scheduling freedom for operand and result transfers. Because TTAs can be constructed of the same basic building blocks as traditional processors and VLIWs, it is very modular and therefore suitable for various kind of ASIPs. Additional customization in TTAs can be achieved by creating function units with custom operations. [1]

TTA processors are programmed by directly controlling data transports between TTA components. Whenever a value is written to the trigger port of a function unit, the operation specified by the port gets triggered. After a specified operation latency, the result values of the operation are ready to be gathered from the output ports of the function unit.

Consider the previous example of an addition and a subtraction. On a traditional operation triggered architecture (OTA), it could look as follows:

```
ADD C, A, B     // C = A + B
SUB D, A, B     // D = A - B
```

With a two-bus TTA machine, the previous example could look as follows (in TTA assembly):

```
1.  RF1.1 -> FU1.operand.1       ...
2.  RF1.2 -> FU1.trigger.add     ...
3.  FU1.result.1 -> RF1.3        RF1.2 -> FU1.trigger.sub
4.  FU1.result.1 -> RF1.4        ...
```

In instructions 1 and 2, operands from the register file RF1 ports 1 and 2 are copied to the input operand ports of the function unit FU1. When writing the second operand in instruction 2, the operation “ADD” is triggered and its result is stored in the output port “result.1” of FU1 which is read on instruction 3. Note that in this example, operation latency for both “ADD” and “SUB” is assumed to be 1 which means that the operation result is ready on the next cycle. In instruction 3, a basic example of instruction level parallelism (ILP) can be seen at work. In this instruction, a second bus was utilized for moving values simultaneously to the first bus.

Since the operation latency is known to be 1, a new operation can be executed at the same time because the function unit result will not be ready until the beginning of the next cycle and the current return value does not get lost. Note how operand 1 was needed to be written only once to the FU1.operand.1 operand port. This is called “operand sharing”, which is an optimization technique used for reducing unneccessary moves between TTA components. Whenever there is an operation with a latency more than one, there exists an opportunity for the compiler to fill in the delay slots with moves belonging to either earlier or upcoming instructions.

Control flow operations such as “JUMP” and “CALL” are implemented in the same way as other operation calls except by using a special function unit called the control unit. A “CALL” on OTA could look like the following:

```
CALL function_1
  <some code>
function_1:
  <function_1 body>
RET
```

In TTA, this could be implemented as a series of moves like the following:

```
function_1 -> GCU.Trigger.CALL    // calls function_1
    <some code>
function_1:
    <function_1 body>
GCU.Return_Address -> GCU.Trigger.JUMP  // returns from function_1
```

Software bypassing [4] is another optimization technique possible to implement on TTAs. In software bypassing, operation results are moved directly from function unit to another so that no register files are needed for storing intermediate values. This reduces the overall delay between dependent operations. The following example uses software bypassing for quickly setting an offset (RF1.1+100) for a memory access operation (“LDW”, “Load Word”).

```
RF1.1 -> FU1.operand.1              ...
100 -> FU1.trigger.add              ...
FU1.result.1 -> FU2.trigger.LDW     ...
```

Conditional execution in TTA can be done with so called guards. A move can be “guarded” with the value of a register file port or a function unit port. If the value is non-zero (or zero in case of inverted guards) then the move following the guard gets executed. Consider the following example:

```
CMP r1, 0x01
JE  0x100        // jump to 0x100 if r1 equals 1
    ...          // not equal
```

In TTA, when using guarded moves, the conditional execution example could look like the following:

```
RF.1 -> FU1.operand.1           ...
0x01 -> FU1.trigger.cmp         ...
```

```
FU1.result.1 -> boolean0.0      ...
?boolean0.0 0x100 -> GCU.Trigger.JUMP // if boolean0.0 equals 1, jump
    ...          // Delay slots for operation "JUMP"
    ...
    ...          // boolean0.0 not equal to 1
```

In this example, the boolean register boolean0.0 is being used as a guard register. If the value is non-zero, then the move triggering a jump to address 0x100 is executed. If not, then the code execution continues from the next instruction.

3. TTA-BASED CODESIGN ENVIRONMENT (TCE)

TTA-based Codesign Environment (TCE) [5] is a suite of applications aiming to provide a reliable and efficient toolset for designing application-specific processors based on the TTA processor template. TCE has been developed at the Tampere University of Technology since 2003, and it currently consists of more than 400,000 lines of C++ code.

3.1 TCE Design Flow

The TCE design flow starts from a program written in a high level language. Currently C is the only programming language fully supported although many parts of C++ can be used already. The C program is compiled to an intermediate bitcode (.bc) format using the LLVM [6] compiler. The bitcode is then compiled to a target TTA processor using an internal compiler of the TCE backend. The output of this process is a TTA Program Exchange Format (TPEF) binary file which has to be converted to a raw bit image using the Program Image Generator (PIG) before it is runnable and ready to be uploaded on the TTA processor. The bitcode produced by LLVM is reusable for different TTAs as long as the program can be compiled for the given TTA. This makes it possible to reuse the bitcode files when exploring suitable TTA processors for the program in the processor design space exploration phase.[3]

The TTA architecture definition file (ADF) describes the characteristics of TTA in a high-level XML, which by itself is not enough to generate a real working processor. The ADF file contains processor component information visible to the assembly programmer or compiler so that programs can be compiled and simulated without needing the actual processor implementation data, which belongs in its own implementation definition file (IDF). The Processor Generator in TCE reads in an architecture definition file (ADF) and an implementation definition file (IDF) and uses a Hardware Database (HDB) to produce a VHDL implementation of the processor.

TCE consists of many different applications sharing the same codebase to achieve their corresponding goals. For creating and editing ADF files, there is a visual processor designer tool called 'ProDe'. For architectural simulation, there is a command line tool 'ttasim' as well as a graphical one called 'Proxim'. For processor design space exploration, there is 'explore' and so on.

3.2 TTA Processor Simulator

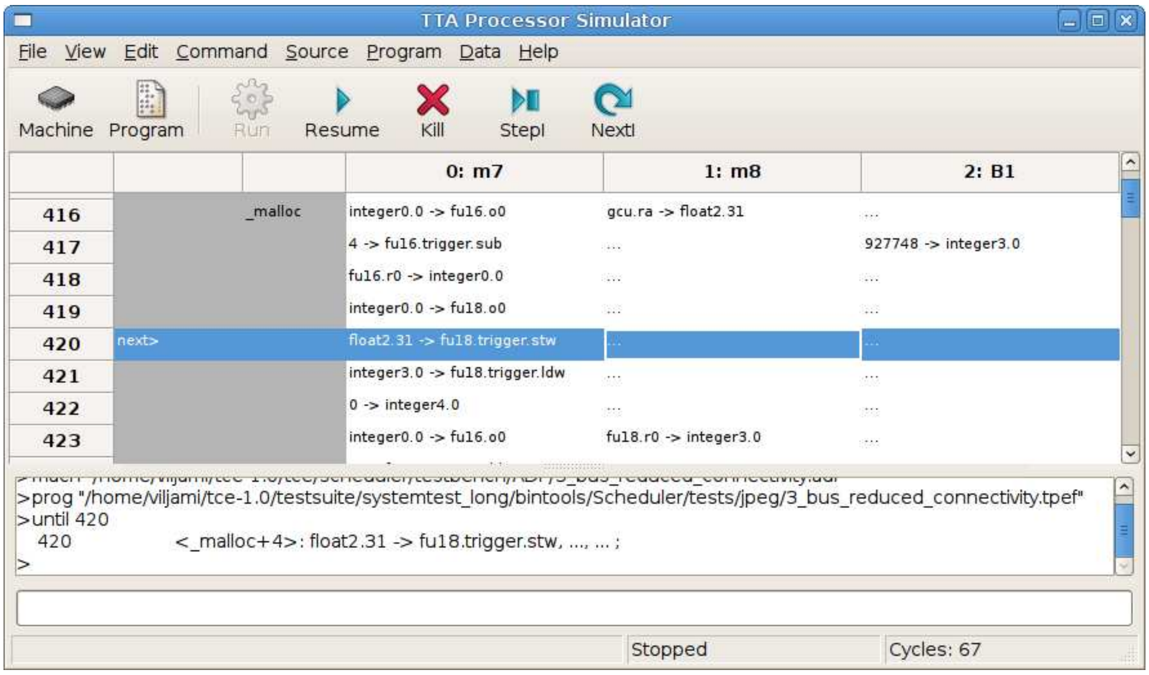
The TTA Processor Simulator in TCE is designed to be able to simulate any TTA processor created in TCE with any kind of operation set defined by the processor designer. The simulator operates at a cycle-accurate level and simulates all components of the TTA processor but only from the viewpoint of the processor architecture programmer. Internal structure of TTA components is not exposed or simulated any further than what is necessary for application simulation and processor architecture evaluation. Latencies for operation calls need to be taken into account for accurate simulation as well as cycle counts and processor utilization data, which both are needed in the evaluation of the processor. [7]

TTA applications that can be simulated are not direct binary images of the final program, but essentially a higher-level object model of the assembly program, that is, the program object model or simply the POM [8]. Using the program object model makes it possible to entirely remove the instruction decode part from the simulation so that fewer host processor cycles are needed to simulate one instruction. This is possible because in TCE, instruction and data memories are assumed to be like in Harvard architecture, that is, they are located in separate memory spaces and the instruction memory is not accessible from the machine code as a data [7]. This makes it impossible to simulate self-modifying code or “programs that generate programs”, but it makes the implementation of compiled simulation much easier.

The TTA Simulator offers various commands for debugging and controlling the application flow. For example, there are commands for single cycle stepping, setting of breakpoints and for inspecting memory spaces and visible TTA components. TCE offers two different TTA simulator frontends: ttasim and Proxim. Both of them use the same interpretive simulation engine on the background. In ttasim, a new simulation engine for running static and dynamic compiled simulations is also introduced. Compiled simulation principles and the TCE compiled simulator engine are described in Chapter 4.

The program 'ttasim' is a command line interface for simulating TTA applications in TCE. Its main features [7] are:

- running TTA applications from the command line,
- provide commands for controlling the simulation,
- work as a debugging interface for TTA applications,



```
                               0: m7                          1: m8                     2: B1
416           _malloc   integer0.0 -> fu16.o0         gcu.ra -> float2.31       ...
417                     4 -> fu16.trigger.sub         ...                       927748 -> integer3.0
418                     fu16.r0 -> integer0.0         ...                       ...
419                     integer0.0 -> fu18.o0         ...                       ...
420  next>              float2.31 -> fu18.trigger.stw ...                       ...
421                     integer3.0 -> fu18.trigger.ldw ...                      ...
422                     0 -> integer4.0               ...                       ...
423                     integer0.0 -> fu16.o0         fu18.r0 -> integer3.0     ...
```

```
>prog "/home/viljami/tce-1.0/testsuite/systemtest_long/bintools/Scheduler/tests/jpeg/3_bus_reduced_connectivity.tpef"
>until 420
  420           <_malloc+4>: float2.31 -> fu18.trigger.stw, ..., ... ;
>
```

```
Stopped    Cycles: 67
```

Figure 3.1: **Proxim main view.**

- provide a TCL script interpreter for automatizing simulations and for adding additional simulator logic if needed, and
- work as a frontend for all simulation engines.

Proxim is a graphical interface for the TTA Simulator. Because it uses the same interpretive simulation engine as ttasim, it has all the capabilities ttasim has except for the compiled simulation engine which is only available in ttasim at the moment (although it could be implemented in Proxim as well). Proxim is best used as a tool for graphical debugging and evaluation of short-length TTA programs. An example of using Proxim as a debugging tool is displayed in Figure 3.1. Proxim makes it easier to understand and manipulate the simulation program flow by providing a code disassembly view and control buttons for commonly used simulator commands. Proxim also has an ability to highlight utilization statistics of TTA components in different colours for manual design space exploration.

3.3 Design Space Explorer

The processor design space explorer or simply the 'Explorer' in TCE is an application that acts as a frontend for various automatic and semi-automatic exploration algorithm plugins [3]. The Explorer is used from the command line where the user can choose an exploration algorithm and give parameters to it. All the exploration algorithms are loaded as separate plugins which makes it possible to extend the Explorer with third party algorithms without having to recompile TCE.

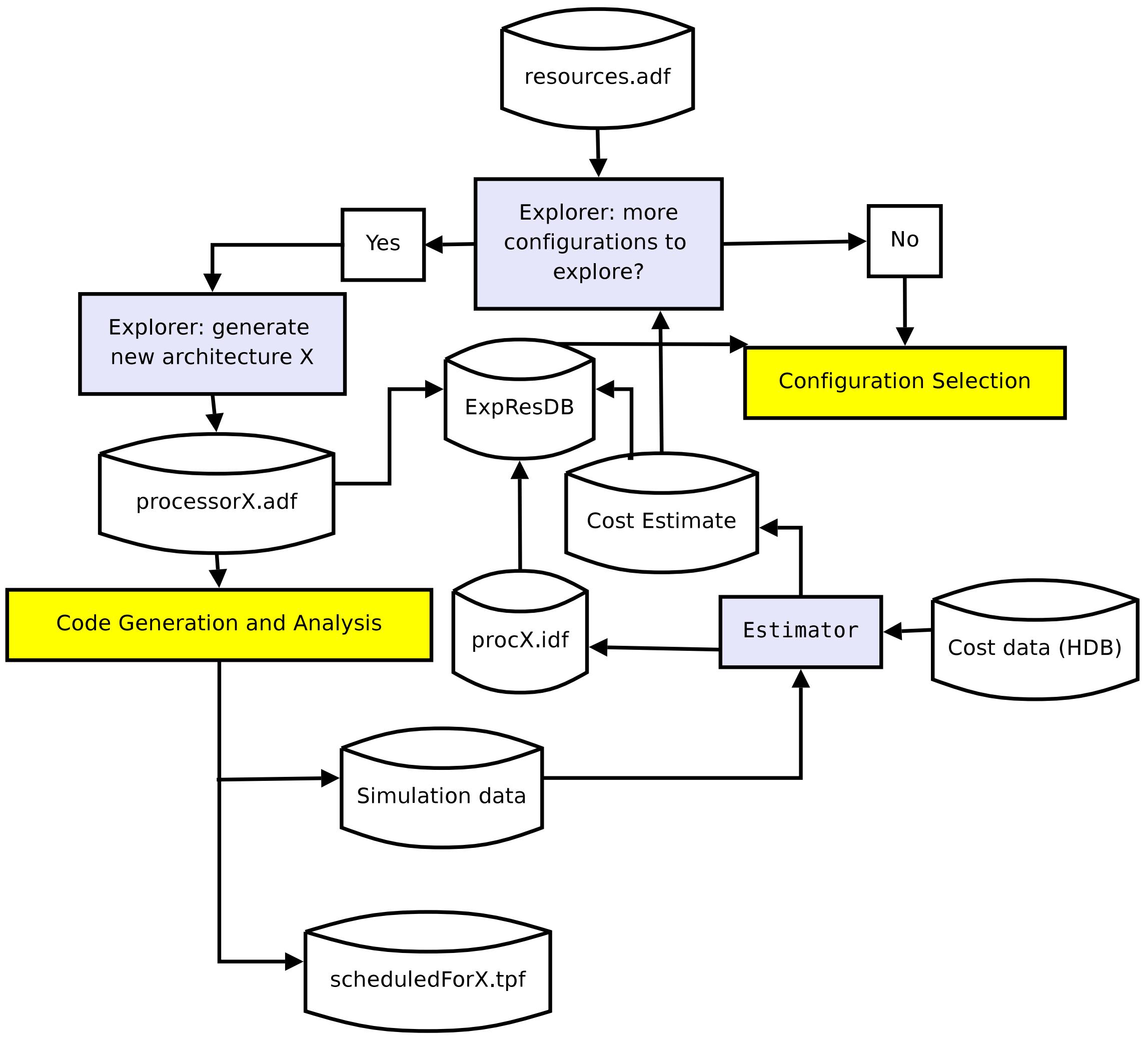


Figure 3.2: **Processor Design Space Exploration in TCE [9].**

The Explorer workflow described in Figure 3.2 proceeds by maintaining a Design Space Database (.DSDB file) which holds data about the exploration results for each different configuration (i.e. architecture and implementation definition pairs). After successfully running Explorer with the specified exploration plugin, Explorer stores exploration data into a new configuration row in the DSDB. The actual exploration data that gets stored per exploration configuration are simply the processor utilization statistics and simulated cycle counts gathered by the simulator and processor cost estimates gathered by the Estimator. A simple exploration database is shown in Figure 3.3.

After finishing the exploration of several configurations, the only thing left to do is to select the most suitable processor and generate a processor image (in VHDL) out of the architecture (ADF) and implementation (IDF) specified by that configuration.

Configuration ID	**Application path**	**cycle count**	**energy estimate**
1	tmp/test1.tpef	513	101
2	tmp/test1.tpef	478	87

Figure 3.3: **An example of a design space database.**

3.4 TCE API Used in the Compiled Simulator

This section describes briefly the most important TCE code structures used in the Compiled Simulator.

3.4.1 Machine Object Model (MOM)

Machine Object Model or simply MOM [10] in TCE is an architecture consisting of various C++ classes. The purpose of MOM is to represent a single target TTA processor by representing all of the components it consists of. In MOM, there is a composite class TTAMachine::Machine, which holds information of all the machine components and operations supported by the machine. The composing classes are the classes that represent the corresponding TTA components such as Port, Bus, RegisterFile, ControlUnit, FunctionUnit, all belonging to the TTAMachine:: namespace.

MOM is designed to be serializable into .XML files. Initially, MOM is created in the Processor Designer GUI (ProDe) and saved into an XML file (.ADF) by the user. The resulting file can be later used for re-creating the same MOM, for example, when needed to simulate or estimate the processor.

3.4.2 Program Object Model (POM)

Program Object Model or POM [8] is very similar to MOM in terms of class design except that it represents a TTA assembly program instead of a TTA processor. The main class in POM is TTAProgram::Program which constructs a hierarchy of program part classes such as Procedure, Instruction and Move, very much the same way as TTAMachine::Machine does for TTA machine components.

Depending on the size of the program, POM can contain a very large amount of data in terms of procedures, instructions and moves, so it is not feasible to store such information into an .XML file. POM data is stored instead into a .TPEF (TTA Program Exchange Format) file, which is the binary file format for TTA programs.

```
<operation>
  <name>ADD</name>
  <inputs>2</inputs>
  <outputs>1</outputs>
  <in id="1">
    <can-swap> <in id="2"/> </can-swap>
  </in>
  <in id="2">
    <can-swap> <in id="1"/> </can-swap>
  </in>
  <out id="3"/>
</operation>
```

Figure 3.4: Static properties of operation ADD [3].

3.4.3 Operation Set Abstraction Layer (OSAL)

Because custom operations are an essential optimization technique in ASIPs, TCE needs to support them as well. TCE has a so called Operation Set Abstraction Layer (OSAL), which allows the processor designer to create new operations for TTAs. Operation properties defined in OSAL can be divided into two categories: static and dynamic. Of these, the static properties (see Figure 3.4) of the operation are saved in an .XML file. These include information such as the number of input and output operands and other details which are required by the compiler backend to be able to produce semantically correct and optimized programs. [3]

The dynamic part is what is more interesting from the simulation point of view. Operation behaviours (see Figure 3.5) which are used for defining what the operation does and "how it behaves" are stored in dynamically loadable .opb plugin files. When the simulator needs to simulate a triggered TTA operation, it uses the implementation found from these plugin files. The operation behaviours are programmed in C by using OSAL macros found from OSAL.hh.

```
#include "OSAL.hh"

OPERATION(ADD)
TRIGGER
    IO(3) = INT(1) + INT(2);
    RETURN_READY;
END_TRIGGER;
END_OPERATION(ADD);
```

Figure 3.5: Simulation behavior definition for operation ADD [3].

Operation Directed Acyclic Graph (Operation DAG). OSAL provides operation directed acyclic graphs or operation DAGs, which are used for representing operation dependencies and data flow of the operation, starting from the input operands. An operation DAG for the operation "ADDSUB" is shown in Figure 3.6.

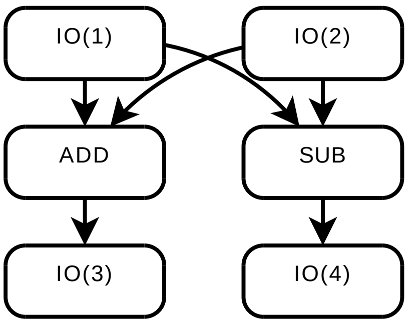


Figure 3.6: A directed acyclic graph (DAG) for operation ADDSUB.

The behavior for operation ADDSUB (defined in the .opp static properties file) could be constructed as follows, using the existing behaviors of ADD and SUB:

```
<trigger-semantics>
      EXEC_OPERATION(add, IO(1), IO(2), IO(3));
      EXEC_OPERATION(sub, IO(1), IO(2), IO(4));
</trigger-semantics>
```

In the previous simulation engine, operations are simulated via a function call to the simulateTrigger() function defined by the operation. Operation DAGs can be used in the compiled simulator for generating C source code for fast simulation of operations found in the base operation set. The C source code is used for inlining operation calls so that no additional function calls (such as the simulateTrigger() call) are needed when simulating operations. At the moment of writing, the operations that have the C simulation behavior code defined are all the operators that are supported by the C language. If the operation does not have a DAG, then it cannot be necessarily inlined and the operation behavior has to be called through the simulateTrigger() function which calls the behavior found from the .opb file.

4. PROCESSOR SIMULATION IN TCE

When developing new ASPs, there is often a need for simulating the processor before it is built to ensure that it functions correctly and it will be suitable for the applications it is supposed to run. Processors can be simulated on numerous different abstraction levels depending on the required simulation accuracy. The simulation performance between different abstraction levels can be huge so it is important to find out the best accuracy level for different situations.

In TCE, VHDL simulation (using the GHDL simulator [11]) can be done on the final TTA bit image to verify the processor correctness. In addition, TCE includes its own higher level simulator that can be used for simulating and debugging TTA assembly programs.

4.1 Interpretive Simulation

In traditional interpretive assembly program simulations, the simulation progresses as described in Figure 4.1. For every simulated instruction of the program, the simulator has to fetch the instruction, decode it to an executable format and finally execute it. If needed, each processor cycle can be simulated as a separate case from each other, which provides very good simulation accuracy (cycle-accurate) at the expense of some simulation overhead.

The decoding part in interpretive simulation can potentially have a very large overhead because of the number of conditional statements the simulator has to execute due to the number of opcode and operation combinations that have to be distinguished [12]. The overhead that adds up is considerable when simulating

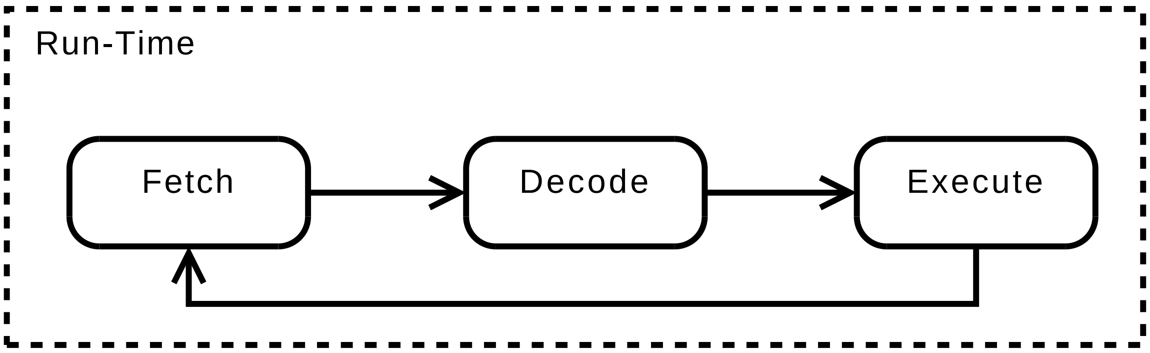


Figure 4.1: **Traditional Interpretive Simulation Flow.**

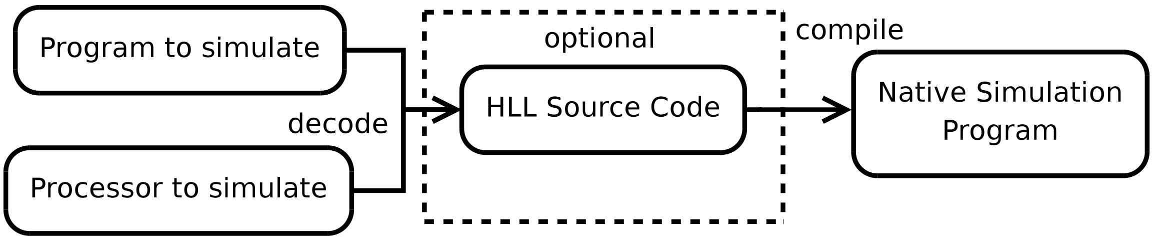


Figure 4.2: **A Compiled Simulation Process.**

a single processor cycle, let alone when simulating a program that can consist of billions of simulation cycles before finishing execution. It gets even worse if the simulator does things such as range checking or maintaining trace databases which are often needed in debugging simulators.

4.2 Compiled Simulation

Compiled simulation is usually the ideal way for achieving as fast simulation speed as possible. It works by eliminating the decoding phase of the simulation entirely, thus reducing the overall overhead of the simulation, even if some accuracy (machine state during cycle N) or error tolerance (such as checking for valid memory addresses) is lost during the process. For example, the compiled simulator in TCE functions at a basic block granularity instead of being cycle-accurate like the interpretive simulator. This means that it is not possible to pause the simulation between single cycles but only between basic blocks. Basic block granularity makes it possible to run the decoded and compiled code blocks effectively as single code units with no breaks or jumps inside or into the middle of the block. However, it is still possible to calculate accurate cycle counts (cycle-count accuracy) by keeping track of the executed instructions during running of the basic block.

In compiled simulation (see Figure 4.2), the simulated program is first decoded and then compiled to native assembly code that can be run either directly as an executable or as a plugin loaded by the simulator during runtime. There are two interesting questions regarding this process:

1. How to convert the simulation program into assembly code native to the simulator host?
2. How much simulator control code is needed to be included in the generated simulation code?

Answer to question 1) is actually quite simple. The program to be simulated can be converted into a high level language (HLL) such as C. After conversion, the resulting C program can be compiled directly to an executable or to a simulator plugin, depending on how the simulation is supposed to work [12]. In this conversion,

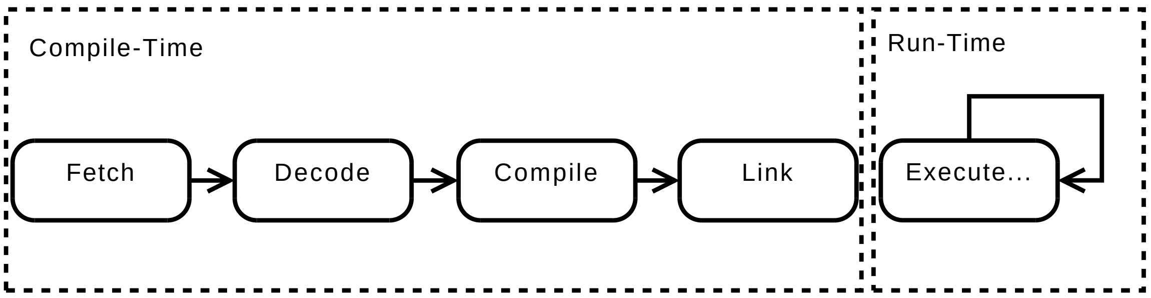


Figure 4.3: **Static Compiled Simulation.**

each instruction has to be converted to C code first. This can be done by having a (inlined) function per each instruction variation. However, this can get complex, because there can be instructions with varying numbers and kinds of parameters but it can be worked around for example, with customized C++ templates like used in IS-CS [13]. The TTA Compiled Simulator does not have this problem because there is only one real instruction: "move". On the other hand, triggered operations in TTA do have to be taken into account in a similar way, either by using an operation DAG or a simulateTrigger() function call. After all the used instructions and operations have their C simulation counterpart, the actual code generation begins. For each instruction call in the simulated program, code for calling the converted instruction will have to be generated.

Code generation in compiled simulation can be either static or dynamic. In static compiled simulation as seen in Figure 4.3, the whole simulated program is loaded in memory at once and then re-generated and compiled as a (C) program instruction by instruction, after which it is ready to be run. While the code generation can be relatively fast, the compilation can turn out to be the biggest time consumer because the amount of C code generated can be huge depending on the size and complexity of the simulated program and the processor architecture. In some cases, linking and compiling can easily take more time than when running the simulation with an interpretive simulator. For example, if the program is otherwise very small but has much code that does not even get executed, the unused parts of code will still have to be generated and compiled because it is not known at compile-time which code parts get executed and which do not.

Dynamic (or just-in-time) compiled simulation (see Figure 4.4) works by splitting the program into smaller units such as single instructions or basic blocks and then generating and compiling simulation code on-need during runtime. The advantage in this technique compared to static compiled simulation is that no code is unneccessarily generated or compiled but everything is done only by demand. The simulator can even use an interpretive engine and decide which parts of the application need to be compiled and which can be simulated by interpretation. Usually only the

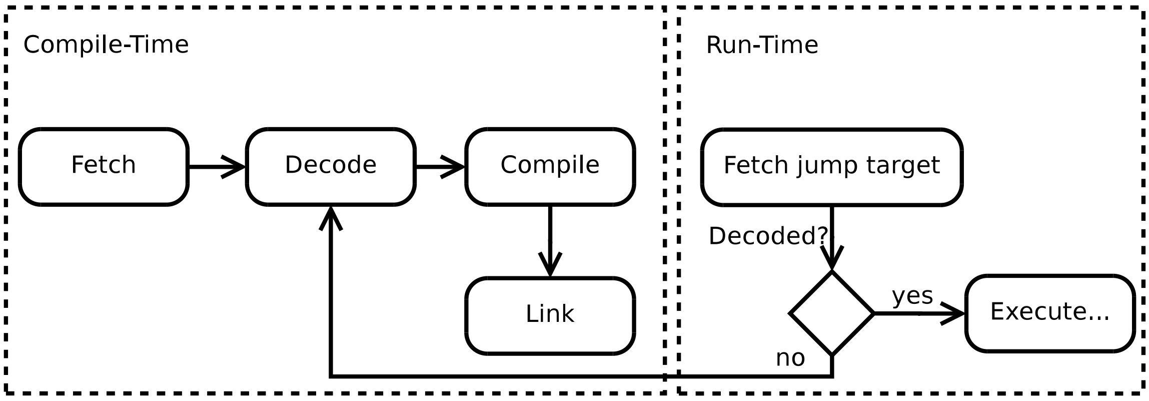


Figure 4.4: **Dynamic (or JIT) Compiled Simulation.**

most executed code units need to be compiled and the rest can be interpreted. The downside is that there are less possibilities for the compiler to optimize things as the code units have to be stored usually behind a function pointer, to avoid compiling and generating the same code over and over again. More system calls are also needed because each dynamically generated unit has to be loaded separately instead of loading them all at startup like in static compiled simulation.

Additional problem with dynamically compiled simulations is that they cannot take advantage of multiple CPU cores by compiling the source files simultaneously unless while compiling there are already more than one source file ready. However, dynamic compiled simulation can be enchanced further by using proactive code generation and compilation. In this technique, the goal is to predict where the program is going to proceed next and compile that part of code in advance. Control flow graphs (CFG) can be used to aid in this. The code parts can be compiled with different optimization flags depending on the number of executions or other heuristics. [14]

As for question 2) how much simulator control code is required to be included in the simulation code, it depends on the target usage of the simulator. If nothing else is required by the simulator than to be able to run the application, then the simulator engine, simulated application and the simulated processor can be compiled together as a static native executable and be run as such. If the simulation needs to have any control logic such as breakpoints or cycle stepping, then a reasonable way is to use a plugin loadable on runtime instead so that no simulator code has to be recompiled per each simulation variation. Depending on the required detail of the simulation, it may be necessary to generate more simulator engine calls into the compiled simulation code. For example, the TCE compiled simulator can generate additional function calls for TTA operation call traces and for function unit conflict detection (i.e. run time checks for detecting if a function unit can perform operations scheduled for it in time).

4.3 TCE Compiled Simulator Design

In this section, a new compiled simulation engine for the TTA Simulator is described.

4.3.1 Design Goals

There are several major design goals for the compiled simulator in TCE. The first and most important design goal is that the new simulation engine must be able to execute any given TTA program as fast as possible. This includes high performance requirements in all of the compiled simulation phases: code generation, compilation, linking, plugin loading and execution of the simulation.

Another goal for the compiled simulator engine is that it has to support basic commands for controlling the simulation such as instruction stepping and running of N number of instructions. It also needs to support inspection of the used TTA components such as register files and function units because it is necessary in debugging and validation of the program execution. Because of this, a design decision was made that the compiled simulator uses the same simulator frontend as the already existing interpretive one so that no redundant simulation control code would have to be created for command interpreting or anything else.

Additional notable design decision was to use C++ instead of C for the generated simulation code. This was decided because the original interpretive simulator is programmed in C++ and it has some important classes such as SimValue (a class for handling simulation values such as integers and floats) that were not feasible to convert to C at the time of writing. Because of this, there was a risk for serious simulation overhead during runtime, because of virtual functions and C++ "code bloat". In addition, some overhead exists during compile-time, because it is more expensive to compile C++ code than plain C [15]. Basically, the generated simulation code should be as C-like as possible except for simulator-specific code such as the SimValues mentioned before. Even the SimValues could be optimized in some simple cases where we could treat them simply as C structs containing variables. It may still be possible to modify the code generator so that it generates pure C code, but at the moment, it is left for future improvements.

The simulator needs to be able to at least produce cycle counts and processor utilization statistics because both of these information are used in processor design space exploration. The idea was to be able to use the compiled simulator in the TCE Explorer so that processor design space exploration becomes feasible for applications running "very long" such as full video codec implementations. Finally, the compiled simulator needs to support two different types of simulation modes: statically and dynamically compiled simulations. Static compiled simulation is to be used when the application executes most of the code anyway so that the dynamically compiled

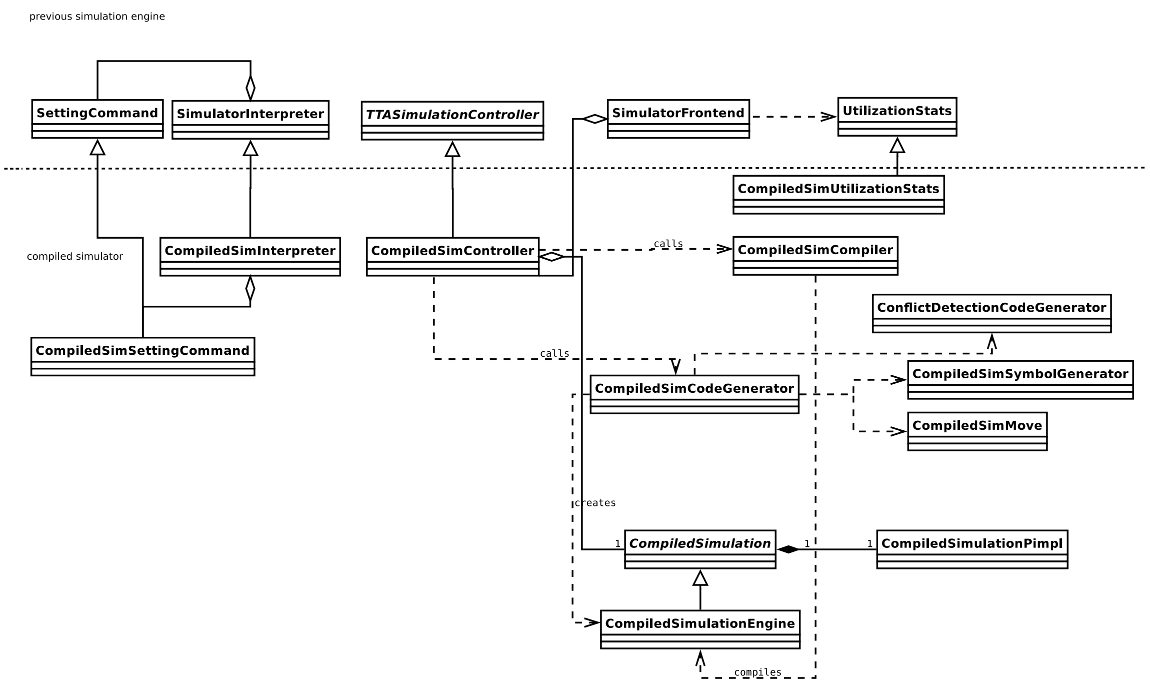


Figure 4.5: **Compiled Simulator Class Diagram.**

simulation would offer little or no benefit. Dynamic compiled simulation is needed in opposite cases where only a small percentage of the total code gets executed but is executed many times, otherwise a simple interpretive engine would suffice.

4.3.2 Design Overview

The Compiled Simulator in TCE is constructed around the existing TTA simulator frontend [7] as an independent engine that is selectable from the command line via command line parameter “-q” as in “quick”. After selecting the use of compiled simulator, the user can choose from the simulator settings whether to use dynamic or static compiled simulation mode. Because the functionality of the compiled simulator is a subset of the interpretive simulator, some simulator settings are not available in the compiled simulator. Most of the removed settings are related to computationally expensive traces (such as register file tracing) which would require more detailed simulation in the compiled simulator at the expense of simulation performance.

A class diagram of the compiled simulation and its integration to the old simulation engine can be seen in Figure 4.5. Original simulator classes that are extended in the compiled simulator are displayed over the dotted line and the new compiled simulation part is displayed below the line. The compiled simulator attempts to reuse as much of the original simulator classes as possible, and it manages to do so reasonably well. New functionality is added by deriving from base classes and introducing some new classes to aid in code generation and compiled simulation creation.

When the user first decides to use a compiled simulator engine, the SimulatorFrontend class creates an instance of CompiledSimController, which implements methods described in an abstract class TTASimulationController. Originally this class was named simply as "SimulationController", but after a compiled simulation controller was required, the class interface was moved higher so that both the compiled (CompiledSimController) and interpretive engines (SimulationController) could use the same interface.

After a CompiledSimController instance is created and the user has given the Simulator a TTA machine (MOM) and a program (POM) to simulate, the CompiledSimController calls CompiledSimCodeGenerator for helping to generate the code. The CompiledSimCodeGenerator generates a new C++ plugin class named CompiledSimulationEngine that implements the interface CompiledSimulation which acts as an interface for all generated compiled simulation plugin classes. After the C code has been generated, CompiledSimCompiler, which works as a wrapper class around the GCC compiler [16], compiles the generated simulation code files either simultaneously or one at a time depending on simulation settings.

All of the simulated basic blocks are generated as independent C functions that get the class CompiledSimulationEngine as a parameter. This way they can access the simulated TTA machine components (stored in CompiledSimulationEngine) such as register files and function units but the CompiledSimulationEngine header file does not get cluttered with endless amounts of forward declarations of these functions. It is important to keep the CompiledSimulation- and CompiledSimulationEngine header files as clean as possible because for each simulation file, the same engine header file has to be included, so the amount of code needed to be parsed only because of headers could become huge. For example, if the compiled simulation code consists of 100 basic blocks, each having 100 code lines and saved in separate source files, and if the simulator header code would consist of 2000 lines, the total amount of header code included would be 2000 x 100 = 200,000 when the actual simulation code would only be 100 x 100 = 10,000 code lines! This is unacceptable when we want as fast C++ code parsing and compilation as possible.

The code bloat that comes from include files can be minimized by using a "private implementation" or "PIMPL" idiom [17]. It was used heavily in every class included either directly or indirectly in the CompiledSimulationEngine. The purpose of this idiom is to minimize compile-time header dependencies by moving the private implementation of class into a new class so that the original class can simply forward declare the private implementation class and then refer to its contents inside a .cpp (C++) source file.

4.3.3 C/C++ Code Generation

The heart of the TCE compiled simulator is a C++ class CompiledSimCodeGenerator. It is responsible for generating C/C++ simulation code out of a TTA program object model (POM) and a TTA machine object model (MOM). The code generator works exactly the same way for dynamic and static compiled simulations i.e. the code generator generates all of the simulation code at once. This means that the dynamic compiled simulator in TCE is not "truly dynamic" because it does not allow modifying code on the run. However, it has some big benefits such that it could allow compiling multiple source files in parallel and even proactively while running the simulation, as described in Chapter 4.2. Code generation does not take nearly as much time as compiling or linking, so it does not add too much overhead into dynamic simulation anyway. The CompiledSimCodeGenerator workflow is as follows:

1. Create a temporary directory for generated code.
2. Create a makefile.
3. Create simulation code source (.cpp) files.
4. Create header (.h) and source (.cpp) files for the CompiledSimulationEngine plugin class.

The CompiledSimCodeGenerator works by running through a single main loop when generating the code. A pseudocode of the code generation process looks like the following:

```
Find all basic blocks for each procedure.
Find all exit points of the program.

For each Procedure do:
  Open a new source file for writing

  For each Instruction in Procedure do:
    If a new basic block?
      If in dynamic simulation mode? -> Open a new source file
        Start a new simulation function
    Endif

    If enabled, generate advance clock code for conflict detectors
    Generate immediate assignments for FU ports
    For each Move in Instruction do:
      - If guarded move? -> Generate guard code
      - Generate move code
      - If triggering move? -> Generate Operation Simulation code
    Generate Immediate assignments for everything else
    Generate cycle ending code
      - If traces are enabled? -> Generate a simulator frontend call
    If instruction is an exit point? -> Generate shutdown code
    If a jump is delayed here? -> Generate jump code

Generate shutdown code after the last instruction.
Generate simulation plugin getter code.
```

The code generator first splits the program into basic blocks which each get their own C function and a function pointer to it. These function pointers are stored in the compiled simulator controller into a lookup table for quick reference when a jump or a procedure call occurs. The simulation code is generated in a temporary folder (user-specified or /tmp/) where it also gets compiled when the simulation is running. The files can be separated so that each procedure or a basic block is placed in their own file. This makes it easier to find, debug and even modify the compiled simulation code by hand if needed.

Because TTA has only one instruction ("Move"), generating simulation code for moves is almost as simple as generating an assignment of two SimValues. If the move source and destination are of the same width, then the assignment can be done directly via SimValue members instead of C++ object copying. If a move triggers an operation call, then some additional logic for calling the operation has to be done. Operations in TTA have their implementation defined in Operation Set Abstraction Layer (OSAL) .opb plugin files which are loaded during runtime by the simulator. When an operation is calculated to be triggered, a call for OSAL DAG is generated with the SimValues obtained from the simulated machine. If no DAG is available, then the operation is simulated using the simulateTrigger() functions found from OSAL but these are to be avoided because a function call itself always has some overhead.

When a jump occurs, the jump target is stored into a register in the GCU. If the jump is actually a "CALL" operation, then the return address is also stored. The jump code generated is simply a C++ "return" call which returns control for the jump dispatcher found in CompiledSimulationEngine.

The jump dispatcher is initialized by setting up the jump dispatcher table. Addresses for each simulation function (that represent a simulatable basic block) are stored in the table at corresponding address index, or set to zero if there is no basic block start at the given address. For example, the code that stores pointers to basic block simulation functions could look like the following in C++:

```
memset(jump_table, 0, sizeof(jump_table)); // set table elements to 0
jump_table[0] = &simulate_0; // set simulate function at address 0
jump_table[13] = &simulate_13; // set simulate function at address 13
jump_table[22] = &simulate_22; // set simulate function at address 22
```

A pseudocode of the jump dispatcher is as follows:

```
If jump_table[address] != 0
    Return jump_table[address]
Else If dynamic compilation
    compile_and_load_function(address)
    If jump_table[address] != 0
        Return jump_table[address]
    Else
        Throw exception "cannot jump to address:" + address
```

Conditional execution in TTA is done via guards. If a guard has a value of true, then the move following the guard gets executed. The CompiledSimCodeGenerator converts guarded moves into C code by adding an if()-statement in front of the move code. Guarded jumps are handled by adding another if()-statement, that has the jump code in it, after the delay slots.

In TTA, a conditional jump could look as follows:

```
?boolean0.0 90 -> gcu.trigger.jump
```

Register file port boolean0.0 is used as the condition variable and 90 is the immediate address to jump into (if the condition matches). When generated as C simulation code, the guarded jump looks like the following:

```
const bool guard_0 = engine.RF_boolean0_0 != 0;
if (guard_0) { engine.GCU_gcu_trigger = 90; }
if (guard_0) {
    engine.jumpTarget_ = engine.GCU_gcu_trigger;
}
```

The actual jump happens after delay slot instructions so we have to generate additional code for returning out of the basic block simulation function like the following:

```
if (guard_0) { engine.programCounter_ = engine.jumpTarget_; return; }
```

4.3.4 TTA-specific Optimizations

There are some interesting optimizations that are possible to do in the TCE Compiled Simulator due to the nature of how TTA works. First of all, the code generator in the Compiled Simulator attempts to optimize away all bus moves by assigning the moves directly from source to destination. This is possible whenever there are no other moves affecting the move source. Consider the following example:

```
1 -> RF.1 ... RF.1 -> FU.in
```

The move RF.1 -> FU.in can not be assigned directly without simulating a bus because otherwise the original value of RF.1 would get overwritten before it is written in FU.in. The bus to be simulated is simply another variable (of the same width as the bus) that is used in cases like this but nowhere else. Because the buses exist only as temporary variables and only in rare occasions, accurate bus traces cannot be produced in the TCE Compiled Simulator. Temporary variables for buses are not always used even if the compiler were able to optimize them away because:

1. The compiled simulation code is not necessarily compiled with optimization flags -O2 or -O3.
2. The optimization is easy to do during code generation.
3. No additional code bloat is introduced by random (if the compiler was not able to optimize it away for some reason).
4. There is less code to be compiled which means faster compile times.

Immediate assignments (moves with constant values as source) are another special case, which has to be taken into account when generating code without buses. A bus in TTA can have two sign extension modes: sign extension or zero extension. When generating simulation code for immediate moves, the sign extension mode of the bus is used for calculating the final value that is assigned to the move destination.

TTA operation calls are optimized away by using a DAG as described before. For example, a call to operation "ADD“ looks like the following:

```
integer0.0 -> FU.0
1 -> FU.trigger.add
```

Instead of calling the expensive simulateTrigger() function, this can be optimized (or inlined) away in the generated simulation code by doing something like the following:

```
SimValue result;
result = engine.fu_FU_0.value + engine.fu_FU_trigger.value_;
```

Memory operations are optimized for the following operations in OSAL:

- Stores: STW, STH, STQ.
- Loads: LDW, LDH, LDQ, LDHU, and LDQU.

For these operations, faster memory accessing functions are used which do not perform any range checking. The minimum addressable unit (MAU) sizes are known for these operations so the fastest and most direct functions to read or write several MAUs at once are used.

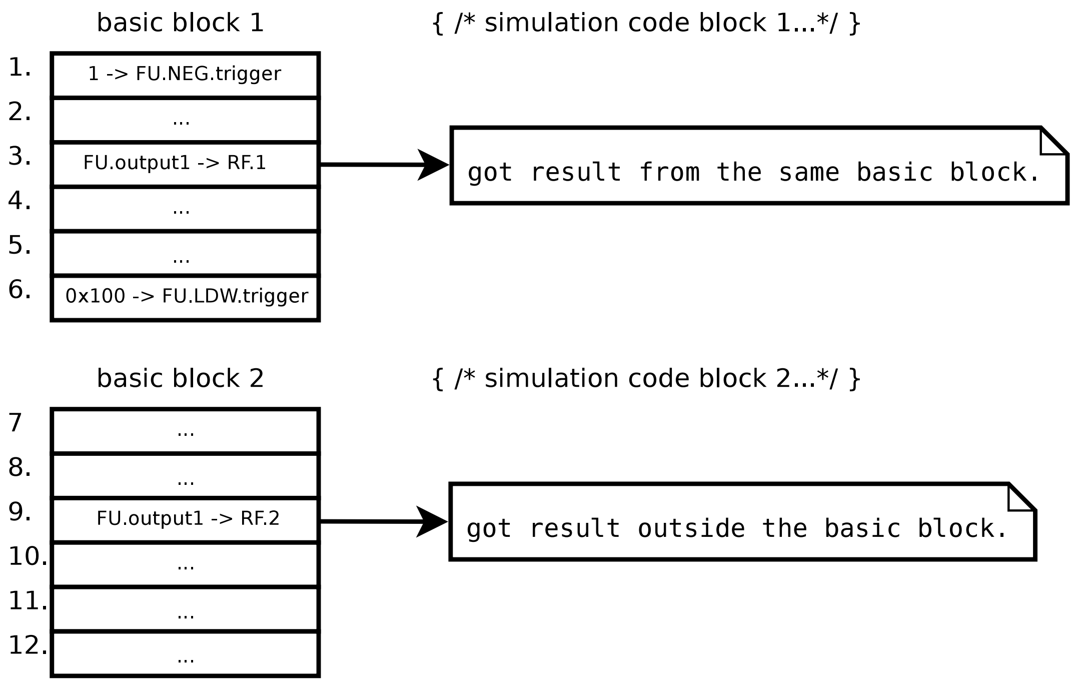


Figure 4.6: **Operation Latency Simulation Between Basic Blocks.**

4.3.5 Operation Latency Between Basic Blocks

Probably the trickiest part in code generation is a case where a function unit writes its output value after the operation delay in another basic block (see Figure 4.6). The output cannot be written directly to the output register because of the operation latency and a temporary variable cannot be used because the variable would leave out of scope in between the basic block functions. In this case, a special ring buffer (one per function unit) is used for adding result values into the buffer. When reading output from the function unit, a new value is obtained from the ring buffer if it is ready, otherwise the previously ready value is picked and so on. The ring buffer maintains itself by quickly clearing it whenever the newest value is read and no more values are awaiting in the buffer.

Each element in the ring buffer has the following attributes:

- time (in processor cycles) when the result is ready
- result value (a SimValue), and
- a flag for determining if the element is used or not.

When adding a new result to the ring buffer as seen in Figure 4.7, the current time in cycles and the operation latency are used for calculating when the result will be ready.

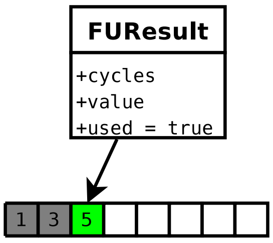


Figure 4.7: **Adding a new element to the ring buffer.**

When getting the result value (see Figure 4.8), the current simulation time is used again but this time it is used for checking which of the ring buffer elements has the latest result that is ready. All of the used elements will have to be checked for finding out the best match.

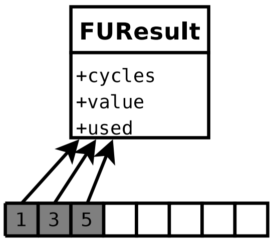


Figure 4.8: **Getting latest results from the ring buffer.**

4.3.6 Trace and Statistics Generation

The TCE Compiled Simulator is capable of creating various statistics and traces for the executed simulations. The most important ones are simulated cycle counts and TTA component utilization statistics. The cycle counts are calculated simply by increasing a cycle counter every time a cycle is simulated. These additions can be merged together by the compiler easily. The utilization statistics require exact execution counts for every simulated move. These are calculated in two phases:

1. basic block execution counts

2. guarded move counts.

If the basic block execution count is known, then the execution counts for moves without a guard in the basic block are also known. For guarded moves, we have to have separate counters because there is no way to know the guard value at the time of simulation code generation. The utilization statistics are generated from previously calculated move counts on the fly in the class CompiledSimUtilizationStats. The final utilization statistics calculation can be somewhat slow because of possibly large number of moves and basic blocks in the application but it is done only if the user asks to do so, so no real overhead is created during simulation runtime. The move and basic block counting did not add any noticeable overhead when testing.

The TCE Compiled Simulator also has a support for generating the following traces by using the already existing trace system from the TCE Simulator:

- basic execution trace, and
- program procedure execution paths.

Using these traces adds a huge overhead to the simulation because each time a simulated cycle ends, an additional call for SimulatorFrontend has to be made. This is the main reason why traces are not enabled by default. If the user wants to use them, then the simulation code has to be re-generated with the frontend calls in it.

TTA bus traces were not implemented in the TCE Compiled Simulator because it does not fully simulate buses and because of performance reasons.

4.4 Speedup Tips

In this section, some performance improvement tips for the TCE Compiled Simulator are introduced as well as an analysis of the current bottlenecks.

4.4.1 Multiprocessing

Originally, there was an idea to implement a multi-threaded TTA Simulator to the already existing interpretive simulator engine. Because all of the TTA moves and simultaneously executed operations are independent to each other, it could have potentially increased the speed of the simulator by the number of buses. However, it also requires there to be as many threads as there are buses, and each thread would have to poll to the main thread until all the other threads have executed the current move and after that can proceed to the next instruction.

The multi-threaded simulator idea was eventually discarded because the current interpretive simulator does a lot more things when performing a move, than simply copying values from bus to another, so that lock-free and thread-safe implementation becomes really difficult to implement. Multi-threaded simulator was successfully tested only for function units performing simultaneous operations but because of lack of available processor cores, it was not proven to be any faster than the single-threaded simulator because most of the CPU time went for polling and idling when the function unit had nothing to do.

However, TCE Compiled Simulator does benefit from multiple processor cores. When using static compiled simulator, the Simulator can be given an environment variable TTASIM_COMPILER_THREADS, which speficies the number of threads to be used when compiling the generated compiled simulation basic block files. On low level, this flag is passed to generated Makefile and the make parameter "-j“ which controls the concurrency level for make. Dynamic compiled simulation does not support this flag yet because it compiles only one file at a time, even though this could be implemented somewhat easily. The main problem in implementing this would probably be to figure out effective heuristics for deciding which basic block simulation file(s) should get be compiled next or during simulating of the currently running basic block.

4.4.2 Distcc and ccache

Distcc [18] is a program which distributes compilation across several computers over a network. Static compiled simulator can benefit from this the most because all of the generated simulation code is compiled at once during simulation startup.

Distcc is controlled using the make flag "-j“ which should be set as high as there are CPU cores available in the distcc cluster. Distcc can be utilized in the TCE Compiled Simulator by setting the environment variable TTASIM_COMPILER to match the compiler of choice. For example if we want to use both ccache and distcc, we set TTASIM_COMPILER="ccache distcc“.

CCache [19] is a C/C++ compiler cache which works by caching a hash sum of the preprocessed C/C++ source files (.cpp) and their corresponding object files (.o). When the compiler detects that a source file to be compiled is already in the cache and it is being compiled using exactly the same compile flags as specified before, the compiler loads the existing object file from the cache thus saving a lot in compile time.

CCache is best used with the compiled simulator when running the same simulation program multiple times, for example, when simulating the program with different input data or parameters. This can greatly reduce the total simulation time when compilation is the greatest time consumer. Unlike in distcc, this optimization can also be used in dynamic compiled simulation since it does not require multiple code files to be compiled at once.

CCache can be enabled in the TCE Compiled Simulator simply by installing it

Function name	**Cost relative to parent basic block function (%)**
SimValue::Operator=(SimValue const&)	13-23
DirectAccessMemory::fastRead4MAUS	5-17
SimValue::SimValue()	7-11
SimValue::Operator=(int const&)	4-8
CompiledSimulation::clearFUResult()	3-5

Table 4.1: **Combined profiling data for both of the tests.**

on the system and making sure it is enabled by default when compiling.

It is important to note that all of the previously described optimizations can be utilized together at the same time to gain maximum possible simulation speed in different simulation conditions.

4.5 Profiling

For profiling the performance and for identifying bottlenecks of the TCE Compiled Simulator, several simulations were run with valgrind [20] –tool=callgrind. Valgrind is a command line tool for profiling and debugging Linux programs. When ran with the command line parameter "–tool=callgrind", Valgrind generates a call graph hierarchy for easily measuring out the most used functions of the program. Valgrind usually slows down program execution by a factor of 1-100 so it is not suitable for everyday testing purposes but only for specific profiling and debugging cases.

Two profiling tests were ran with the TCE v1.0 Compiled Simulator using simulator flags "-O0" and "-g" so that no code is optimized away by the compiler. This way we can find out the most used functions by name. The percentages shown in Table 4.5 are the minimum and maximum costs (execution count and time) for the most executed functions in the most executed basic blocks of the simulated programs.

Both of the simulations spent most of the time in SimValue copying and creation. This was expected because TTA assembly is practically just moving values from one TTA component to another. The ring buffer implementation in addFUResult() and clearFUResult() functions is also somewhat costly and could be part of the reason why there exists so many SimValue creations and copying.

What was surprising was that no noticeable overhead was coming from the simulation of TTA operations (simulateTrigger() functions) except for the memory operation fastRead4MAUS(). This was probably because most of the operations could be inlined effectively using Operation DAGs.

5. VERIFICATION AND BENCHMARKING

The Compiled Simulator functionality was tested and verified using the interpretive simulator and native test executables as test oracles (applications that provide the expected results for test cases). All of the test cases produce output data which can be reproduced on different platforms and compared to the original data. The behavior of the Compiled Simulator was tested by ensuring that the produced output, procedure transfer traces and final cycle counts match with the data got from the interpretive simulator. The most complex application that has been run with the Compiled Simulator is the DENBench benchmark suite, which consists of about 150,000 lines of C code. It has been tested with several TTA machines to make sure the simulator performs correctly.

5.1 Testing

All of the testing were done on an Intel Core 2 Duo computer running at 2.4 GHz with 2 GB RAM using 32-bit Ubuntu [21] Hardy Heron (Ubuntu 8.04.2, Linux kernel 2.6.24) as the operating system. GCC [16] version 4.2.4 was used as the compiler when running compiled simulation performance tests. No ccache or distcc were utilized when running the tests to maintain stability in simulation speeds.

Three test cases were tested with the different simulation engines (interpretive, static and dynamic compiled simulation) to get real world statistics on how well the Compiled Simulator performs in different situations. The JPEG and Tremor test cases were run with two different TTA machines to see how much difference the number of buses can make to the simulation speed. In static compiled simulation tests, only the run time is taken into account when calculating the simulation speed. The compile times are still included in the table to give some perspective on how long it takes to compile larger test cases. Static compiled simulations were tested with no compile time optimizations (-O0) and with full optimizations (-O3) to see how much they can affect the compile times.

The test cases are the following:

JPEG. The JPEG test case consists of decoding a JPEG image of size 227x149 (in pixels) and outputting it to a .BMP file. The test case is approximately 24 million cycles long on a 1-bus machine and 7 million cycles on a 10-bus machine.

Tremor. The Tremor test case has a short (size 40kB) OGG vorbis file which is decoded and exported to a RAW format. The test case is approximately 574 million cycles long on a 1-bus machine and 175 million cycles on a 10-bus machine.

Xvid. The Xvid test case consists of decoding a short MPEG4 stream and exporting it to a PGM format. The test case is approximately 831 million cycles long on a 3-bus machine.

5.2 Interpretive Simulation

In Table 5.1 are the simulation time statistics for the old interpretive simulator. They can be used as a reference to see how much TTA simulation speed has improved since the introduction of the compiled simulator.

Test Case	**Cycles**	**Run Time**	**Total Speed (MHz)**
JPEG (1 bus)	24,308,743	50 s	0.486
JPEG (10 buses)	6,818,276	21 s	0.325
Tremor (1 bus)	573,732,612	11 min 18 s	0.846
Tremor (10 buses)	175,314,125	5 min 19 s	0.550
Xvid (3 buses)	830,634,029	14 min 31 s	0.954

Table 5.1: **Simulation statistics for interpretive simulation.**

5.3 Static Compiled Simulation

The following statistics were generated without compile time optimizations (-O0):

Test Case	**Compile Time**	**Run Time**	**Simulation Speed (MHz)**	**Total Speed (MHz)**
JPEG (1 bus)	6 s	<1 s	24.3	4.05
JPEG (10 buses)	5 s	<1 s	6.82	1.36
Tremor (1 bus)	38 s	17 s	33.75	10.43
Tremor (10 buses)	22 s	36 s	7.97	3.02
Xvid (3 buses)	1 min 41 s	26 s	31.9	6.54

Table 5.2: **Simulation statistics for static compiled simulation (-O0).**

The following statistics were generated with all optimizations enabled (-O3):

Test Case	**Compile Time**	**Run Time**	**Simulation Speed (MHz)**	**Total Speed (MHz)**
JPEG (1 bus)	22 s	<1 s	24.3	1.10
JPEG (10 buses)	29 s	<1 s	6.82	0.24
Tremor (1 bus)	2 min 50 s	8 s	71.7	3.22
Tremor (10 buses)	3 min 28 s	10 s	17.5	0.80
Xvid (3 buses)	7 min 8 s	13 s	63.9	1.89

Table 5.3: **Simulation statistics for static compiled simulation (-O3).**

5.4 Dynamic Compiled Simulation

The following statistics were generated with dynamic compiled simulation with no optimizations (-O0). Compile time is not included, because all of the compiling is done at run time.

Test Case	**Run Time**	**Total Speed (MHz)**
JPEG (1 bus	9 s	2.7
JPEG (10 buses)	9 s	0.758
Tremor (1 bus)	1 min 5 s	8.83
Tremor (10 buses	1 min 6 s	2.66
Xvid (3 buses)	1 min 7 s	12.4

Table 5.4: **Simulation statistics for dynamic compiled simulation (-O0).**

Following are the statistics for dynamic compiled simulation using all optimizations available. (-O3).

Test Case	**Run Time**	**Total Speed (MHz)**
JPEG (1 bus)	36 s	0.675
JPEG (10 buses)	46 s	0.148
Tremor (1 bus)	4 min 17 s	2.232
Tremor (10 buses)	5 min 20 s	0.548
Xvid (3 buses)	2 min 32 s	5.465

Table 5.5: **Simulation statistics for dynamic compiled simulation (-O3).**

Test Case	**static (O0)**	**static (O3)**	**dynamic (O0)**	**dynamic (O3)**
JPEG (1 bus)	8.33	2.26	5.56	1.39
JPEG (10 buses)	4.18	0.69	2.33	0.46
Tremor (1 bus)	12.33	3.80	10.44	2.64
Tremor (10 buses	5.49	1.45	4.84	0.99
Xvid (3 buses)	6.86	1.98	12.99	5.73

Table 5.6: **Compiled Simulator vs Old Simulation Engine.**

5.5 Results Analysis

Performance increments of the Compiled Simulator compared to the original simulation engine are shown in Table 5.6. Each number in the table represents improvement (or disimprovement) in total simulation speed.

The best simulation speeds were reached with the static compiled simulator using compile flag -O3. The biggest improvement was noticed in the Tremor test case, which originally took over 11 minutes with the interpretive simulator, but with the compiled simulator could be run in only 8 seconds. When the number of buses on the simulated machine was increased from 1 to 10, the performance of the simulator dropped to about 1/4. This happens because the more there are concurrent moves per TTA instruction, the longer it takes to execute it, unless executing in parallel which at the moment is not possible.

The usage of -O3 comes with a great cost. The simulation speed was only doubled with it, but the compile times were more than quadrupled in the worst cases. The usage of compile flag -O3 should be considered only when the simulation run time is expected to be way longer than the compile time, and in cases where we want to simulate the application in “real time”, but this has not been needed yet in our testing purposes. If the compile times are not an issue, for example, when using distcc and ccache, then the flag -O3 should be used.

When we want to get the best simulation performance overall, then the static compiled simulator with compile flags -O0 should be used. This allows quick compiling of the entire application (using the speedup methods described in Section 4.4) while preserving moderate simulation speed. Dynamic compiled simulator should be used with programs that have much code that does not get executed, like it seems to be with the Xvid test case, for which the overall best simulation time was got indeed with the dynamic compiled simulator.

6. TOOLS FOR MANUAL PROCESSOR DESIGN SPACE EXPLORATION

The TCE Compiled Simulator was originally planned to be used heavily in manual processor design space exploration, where simulation speed is most important because of constant need for redesigning and tweaking the processor by the processor designer. In this chapter, two new graphical tools usable for manual processor design space exploration are introduced.

6.1 Eclipse Plugin

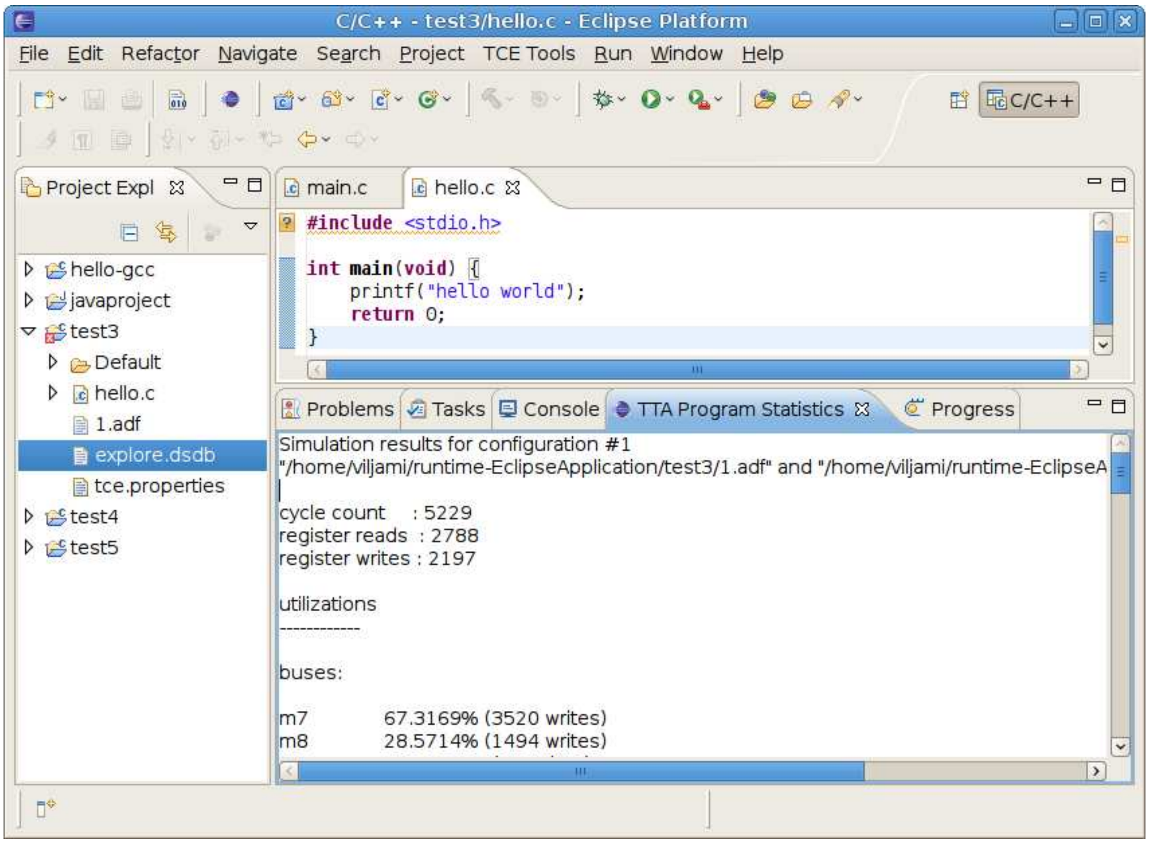
An experimental Eclipse [22] plugin for TCE was developed. It is capable of being used as a standalone IDE for rapid development of TTA applications (in C/C++ language) and simulating them using the compiled simulator. The compiled simulator is integrated to the Eclipse plugin via its command line interface ttasim. It can be used for simulating and verifying TTA applications and outputting utilization statistics of the used TTA components (see Figure 6.1). The TCE Eclipse plugin also has quick launch icons for all the other GUI tools available in TCE. This makes it possible to easily launch the required tools. For example, when needed to modify the processor, a processor designer tool could be launched and so on.

The supported features of the TCE Eclipse Plugin are:

- C/C++ toolchain for CDT using the TCE C Compiler
- TCE tool launcher, and
- TTA Application simulation using the compiled simulator.

6.1.1 Plugin Design

Eclipse plugin creation starts by defining a plugin.xml XML manifest file. This file contains all the information necessary for Eclipse to be able to load and run the plugin. It also holds information of the extension points used and offered for further development. The TCE Eclipse plugin does not offer any extension points outside but it does extend several classes found from both the Eclipse Framework and the Eclipse CDT 5.0 [23].



```
#include <stdio.h>

int main(void) {
    printf("hello world");
    return 0;
}
```

```
Simulation results for configuration #1
"/home/viljami/runtime-EclipseApplication/test3/1.adf" and "/home/viljami/runtime-EclipseA

cycle count    : 5229
register reads  : 2788
register writes : 2197

utilizations
------------

buses:

m7        67.3169% (3520 writes)
m8        28.5714% (1494 writes)
```

Figure 6.1: **Processor simulation using the TCE Eclipse plugin.**

In the TCE Eclipse Plugin, Eclipse UI Plugin is extended for modifying and creating new visible Eclipse UI parts such as windows, menus and perspectives. For example, an entire new "TCE Tools" menu was introduced, which contains launchers for all TCE GUI tools available at the moment. User interfaces are created with the SWT (Standard Widget Toolkit) library, which is an open source GUI toolkit developed especially for Eclipse.

Eclipse CDT was extended for creating a new compilation toolchain which uses the TCE C/C++ Compiler (tcecc) as the compiler of choice for creating TTA applications using the Eclipse IDE. It was also extended for giving some additional TTA-related information to the C perspective, such as the "TTA Program Statistics" tab which displays statistics of the last simulation run.

6.1.2 Extending the Eclipse Framework

Most Eclipse plugins need to extend the existing Eclipse Framework to a certain degree and TCE Eclipse plugin is no exception. Eclipse plugins can be extended and new functionality be added to them through the usage of so called extension points. [24]

Eclipse UI Plugin (org.eclipse.ui) offers many classes for various UI components such as menus, configurations, perspectives and views. These classes can be extended by using the extension points defined which are visible for new plugin developers. Extension points are used by configuring them in the plugin.xml file of the plugin project.

For example, creating a new menu item to the Eclipse Workbench window can be done with the extension of the org.eclipse.ui.actionSets extension point. The menu items are configured by setting a name, icon, id, label and an action class that has a callback function which is called every time the menu item is selected. The following code sample for starting the Processor Designer (ProDe) from a menu item is directly from the plugin.xml file of the TCE Eclipse plugin:

```
<action
    class="fi.tut.cs.tce.actions.ProdeAction"
    icon="icons/prode.gif"
    id="fi.tut.cs.tce.actions.ProdeAction"
    label="Processor Designer"
    menubarPath="tceMenu/tceTools"
    tooltip="Starts the TTA Processor Designer">
</action>
```

The menu item action class needs to implement IWorkbenchWindowActionDelegate interface defined in org.eclipse.ui which is as easy as implementing “runnable” method:

```
public void run(IAction action);
```

6.1.3 Extending Eclipse CDT

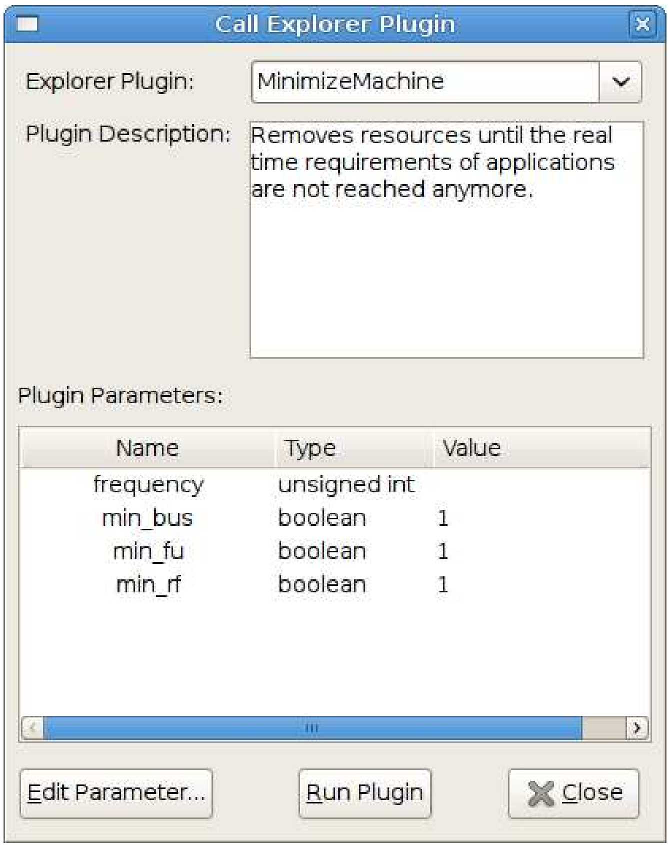
Extending the Eclipse CDT plugin happens very much the same way as with the Eclipse Framework, through the usage of its own extension points. Since CDT version 5.0, much has changed in the API and because of lack of recent documentation (on extending CDT), it has been very difficult to figure out which components can be extended and how.

The main package in CDT which can be used for extending CDT tools is org.eclipse.cdt.managedbuilder.core. It has extension points for creating new toolchains, make, builders and error parsers. The C perspective can be extended with org.eclipse.cdt.ui.CPerspective. New debugger interfaces could be added through the extension point org.eclipse.cdt.debug.core.CDebugger. For example, if we wanted to integrate ttasim as the debugging interface for Eclipse, this extension point should be used.

6.1.4 Integration to Other TCE Tools

The TCE Eclipse plugin relies much on other TCE applications such as the TCE Compiler and the Explorer. At the moment, all communication with these applications is done internally through the command line which is used for opening new applications, sending parameters and receiving output data from the used applications. The biggest problem with this kind of a system is that no TCE (library) code may be called or reused unless it is possible to do so using the command line options of the application.

Ideally, the communication should be done with JNI [25] or a similar interface that would allow direct calling of TCE code from Java but unfortunately this has not been implemented yet. As a result, the TCE Eclipse plugin is extremely sensitive to all external changes in TCE tools, especially in the Compiler and the Explorer. These changes should be avoided or at least kept to minimum until JNI calls are implemented properly.



Explorer Plugin: MinimizeMachine

Plugin Description: Removes resources until the real time requirements of applications are not reached anymore.

Plugin Parameters:

Name	Type	Value
frequency	unsigned int	
min_bus	boolean	1
min_fu	boolean	1
min_rf	boolean	1

Figure 6.2: **Calling an Explorer plugin (MinimizeMachine) using ProDe.**

6.2 Manual Processor Design Space Exploration Support for ProDe

Another tool that was implemented to aid in graphical processor design space exploration is a GUI frontend for calling individual exploration plugins of the TCE Explorer. This was implemented as a part of the already existing processor designer application (ProDe).

The tool displays a list of available exploration plugins (found from the default plugin search paths). After the user has selected which plugin to use, the dialog displays a plugin description and the initialization parameters of the plugin which can be modified if needed. When the plugin is commanded to run, the dialog passes all the parameters internally to the TCE Explorer.

An example of the tool is shown in Figure 6.2 where the user is about to call a plugin for minimizing the TTA machine.

7. FUTURE IMPROVEMENTS

Not all work was finished in the TCE Eclipse plugin or even in the compiled simulator. The compiled simulator could still use more performance improvement in some places because now it seems that even the simplest code generation changes such as reducing a few SimValue (copy) constructor calls can make a noticeable difference to the simulation performance.

Although functional and capable for simple TTA application programming, the experimental TCE Eclipse plugin could be improved greatly by integrating it better with other TCE tools and the Eclipse CDT.

7.1 Compiled Simulator

The TCE Compiled Simulator has been proven to be significantly faster than the interpretive simulator so its primary goal of increasing simulation speed has been achieved. However, as was seen in Section 4.5, there are still some bottlenecks that could be removed and therefore increase the overall performance of the simulator further.

Excessive usage of the SimValue class should be investigated more. There are probably still a few situations remaining where SimValues are copied and created unneccessarily, especially when simulating operation calls (with or without a DAG) and memory operations. The ring buffer which is used for simulating operation latency could also be improved to store plain values rather than (copies of) SimValues to avoid unneccessary object creation and copying.

Compile speed could probably be improved further by changing the generated code to be plain C language rather than a mix of C/C++, but this change is slightly more difficult to implement because all of the simulator code is in C++ and some callback functions are still needed in the compiled simulation code. The compilation process should be profiled to see which parts of the compiling takes the most time. Compilers other than GCC (like the Intel C++ Compiler) could also be investigated to see if they can be more effective in code generation or compilation. Dynamic compiled simulation could also be improved greatly by the usage of multi-threaded and proactive code compilation.

At the moment, the compiled simulator works only from the command line but technically there are no limits why it should remain that way. It could also be integrated to the Simulator GUI (Proxim) which would be more intuitive to use for most people. The biggest work left to do for achieving this would be to disable some controls and options in Proxim that cannot be implemented in the compiled simulator, such as the ability to step single cycles or trace register reads and writes.

7.2 TCE Eclipse Plugin

The experimental TCE Eclipse plugin could be improved greatly. It would need to be integrated better with TCE as a whole to be more useful. This could be achieved with JNI for instance but it would require more work (and preferably more people with previous JNI experience!). If JNI were not to be used, integration with TCE tools would have to be continued to be done on command line which means that the tool interfaces would have to be very expressive and complex, yet remain stable enough so that nothing breaks when altering them.

The TCE Eclipse plugin should make better use of the tools and API's CDT provides. For example, the TCE C/C++ toolchain could be improved to have code syntax checking using the TCE C compiler. A debugging interface using CDT and ttasim could also be implemented somewhat easily although it may be best to use Proxim for that and continue its development instead.

Even though the original idea of manual processor design space exploration was changed and that part was moved to Processor Designer (ProDe) application, some ideas left for it could still be implemented in Eclipse. For example, tools for quickly creating new custom operations and testing them using the compiled simulator could be implemented in the plugin.

8. CONCLUSIONS

In this thesis, a new compiled simulator engine for simulating TTA applications and an Eclipse plugin for TCE tools were introduced. In ideal conditions, the compiled simulator engine was proven to be about 84 times faster than the interpretive engine so the original design requirement of "increasing the performance of the simulator" seems to have been achieved rather well. The speed improvement is essential for the TTA Simulator so that it becomes able to simulate longer, real life applications such as Tremor (OGG decoder) or XVid (MPEG4 decoder) in a reasonable time frame.

The compiled simulator engine ran at best at about 72 MHz on a 2.4 GHz computer so it can even be considered a "real-time" simulator (at least for low-frequency TTAs). However, there is definitely more room left for speed improvement, most notably in the compile times. The best simulation speeds so far were reached with TTA machines with only one bus. The number of buses seems to directly affect the performance of the simulation. For example, when increasing bus count from 1 to 10, the simulation performance dropped to about 1/4. This was expected because the number of buses can affect directly to the amount of work needed to be done on a single simulation cycle. With more buses, the same work is done except in parallel, and this parallelism has to be emulated with simulator software instead of being run on a real TTA processor.

Optimization flags for the compiled simulator can improve simulation speed further but at the expense of longer compile times. Most of the time we need only fast simulation without any real-time requirements so the flag -O0 should be used for gaining shorter compile times. If the compile times are not an issue, then the optimization flag -O3 should be used for gaining as fast simulation execution speed as possible.

The design of the experimental TCE Eclipse plugin has gone through some major changes, most notably with the changes to the manual processor design space exploration part. Currently, the plugin functions as a simple C/C++ IDE for creating TTA applications with TCE tools. The plugin can also be used for simulating TTA applications quickly with the compiled simulator although it is not integrated nearly as well with the Eclipse IDE as it could be. Further development of the plugin would require some major structural changes like the usage of JNI so they were not implemented at the time.

BIBLIOGRAPHY

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

[2] J. E. Smith and G. Sohi, “The microarchitecture of superscalar processors,” *Proceedings of the IEEE*, 1995.

[3] 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, http://tce.cs.tut.fi/.

[4] V. Guzma, P. Jääskeläinen, P. Kellomäki, and J. Takala, “Impact of software bypassing on instruction level parallelism and register file traffic,” in *SAMOS*, ser. Lecture Notes in Computer Science, M. Bereković, N. Dimopoulos, and S. Wong, Eds., vol. 5114. Springer, 2008, pp. 23–32.

[5] Tampere Univ. of Tech., “TCE project at TUT,” 2009 (accessed January 25, 2009), http://tce.cs.tut.fi. [Online]. Available: http://tce.cs.tut.fi

[6] The LLVM Team, “The LLVM Compiler Infrastructure Project,” 2009 (accessed January 25, 2009), http://llvm.org. [Online]. Available: http://llvm.org

[7] 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, http://tce.cs.tut.fi/.

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

[9] Tampere University of Technology Department of Computer Systems, “TTA codesign environment v1.0 user manual,” Project Document, Tampere University of Technology, Tampere, Finland, 2008.

[10] A. Oksman, “Machine Object Model,” Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2004-2005.

[11] T. Gingold, “Ghdl,” 2009 (accessed January 25, 2009), http://ghdl.free.fr. [Online]. Available: http://ghdl.free.fr

[12] J. Glossner, S. Dorward, S. Jinturkar, M. Moudgill, E. Hokenek, M. Schulte, and S. Vassiliadis, “Sandbridge software tools,” 2005, pp. 269–278. [Online]. Available: http://dx.doi.org/10.1007/11512622_29

[13] M. Reshadi, P. Mishra, and N. Dutt, “Instruction set compiled simulation: a technique for fast and flexible instruction set simulation,” in *DAC ’03: Proceedings of the 40th conference on Design automation.* New York, NY, USA: ACM, 2003, pp. 758–763.

[14] W. Qin, J. D’Errico, and X. Zhu, “A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation,” in *CODES+ISSS ’06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis.* New York, NY, USA: ACM Press, 2006, pp. 193–198.

[15] Warwick Irwin and Neville Churcher, “A Generated Parser of C++,” *N.Z. Journal of Computing*, vol. 8, 2001.

[16] Free Software Foundation, “GCC, The GNU Compiler Collection,” 2009 (accessed January 25, 2009), http://gcc.gnu.org. [Online]. Available: http://gcc.gnu.org

[17] H. Sutter, “Pimples—beauty marks you can depend on,” in *More C++ gems.* New York, NY, USA: Cambridge University Press, 2000, pp. 407–416.

[18] Fergus Henderson, “distcc,” 2009 (accessed January 25, 2009), http://distcc.org/. [Online]. Available: http://distcc.org/

[19] Andrew Tridgell, “ccache,” 2009 (accessed January 25, 2009), http://ccache.samba.org/. [Online]. Available: http://ccache.samba.org/

[20] The Valgrind Developers, “Valgrind,” 2009 (accessed January 25, 2009), http://valgrind.org/. [Online]. Available: http://valgrind.org/

[21] Canonical Ltd., “Ubuntu,” 2009 (accessed January 25, 2009), http://www.ubuntu.com/. [Online]. Available: http://www.ubuntu.com/

[22] The Eclipse Foundation, “Eclipse,” 2009 (accessed February 5, 2009), http://www.eclipse.org/. [Online]. Available: http://www.eclipse.org/

[23] ——, “Eclipse C/C++ Development Tooling - CDT,” 2009 (accessed February 5, 2009), http://www.eclipse.org/cdt. [Online]. Available: http://www.eclipse.org/cdt

[24] E. Clayberg and D. Rubel, *Eclipse: Building Commercial-Quality Plug-ins (2nd Edition) (Eclipse).* Addison-Wesley Professional, 2006.

[25] S. Liang, *Java Native Interface: Programmer’s Guide and Reference.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.