Dataflow processor architecture implementation survey

Moritz Faust
University of Kaiserslautern, Embedded Systems Group
m_faust12@cs.uni-kl.de

Abstract

In this review a short overview over three different types of dataflow processor architecture will be given. There will be given a short introduction in the basic architectures “VLIW”, “Superscalar”, “CMP”, “RAW Processor”, “Dataflow Machines” and “Systolic processors”. Followed by a closer look at the “MIT Tagged-Token Dataflow Architecture”, the “TRIPS Architecture”, as well as the ”WaveScalar Architecture”. These three architectures will be explained by their basic hardware setup and by an example program running on the architecture showing the advantages of the same.

1 Introduction

Effort spent for improving the parallelism in machines to reach performance on an level that is far beyond the state of the art as there is not much potential to improve the hardware architecture anymore by miniaturizing or increasing of the clock frequency. We will have a look at some basic architectures which are necessary to know as fundamental knowledge for understanding the studied architectures.

VLIW - Very long instruction word is a processor architecture to improve the performance by executing operations in parallel. Therefore, the number of ALU is giving the maximum number of operations of one VLIW word. Operations in one word to be solved in parallel is specified by the compiler. In VLIW, the instruction logic loads the VLIW instructions in a cycle from memory to the ALU’s. The instruction logic in VLIW is quite simple, short cycle length and word length of 128 bit. Every VLIW task is given as a single instruction for a sequential processor.

Superscalar processors are able to execute more than one instruction at the same time. Superscalar processors use pipelines to execute the instructions. The maximum number of instructions executed at one time is limited by the number of pipelines. If the pipeline is asynchronous the performance will be higher since fast instructions do not have to wait for slower instructions to be executed. Furthermore, superscalar architectures need some sub units, which provide the data transfer from cache to any necessary unit. Also there is a unit checking all the instructions and decide which instruction is executed and which has to wait since it has dependencies from other instructions.

CMP - Chip Multicore Processors are designed with the driving force that performance gains of either wider issues width or deeper pipelines only would be marginal since we have delays and longer access times for larger structures. The solution for CMP is to bring some new features like “off-chip memory bandwidth” and a shared memory hierarchy, so processors are forced to share some low level cache and the on-chip links between those.

RAW processors are mainly designed to expose wire delay at the ISA level. This allows the compiler to explicitly manage datatransports. RAW processors also provide a direct parallel interface to all of the chip resources like gates, pins and wires. The usage of those gates is managed by using a tightly coupled network communication mechanism in the ISA.
Dataflow Machines are the most common alternative architecture to "von-Neumann-Architecture". Dataflow machines use parallelism to execute multiple concurrent operations and utilize them to solve any task. Dataflow machines require an own programming language like for example Id. Each program code written for such a machine is modeled as a dataflow graph. A dataflow graph describes concurrent processes which can be solved by a dataflow machine in parallel. The main motivation for those machines is multithreading. Since every dataflow machine got a bad performance if it is forced to solve tasks in a single sequential thread.

We can see there are many different ideas trying to solve the problem that is caused by not having that much potential to improve the hardware architecture anymore. The three architectures we are taking a closer look at (“VLIW”, “Dataflow-Architecture”, “TRIPS”) have totally different ways to handle this problem and trying to solve it. So this review is structured as follows: The first architecture we are taking a detailed look at will be the “MIT-Tagged-Token Dataflow Architecture”. We will discuss it by taking a look at the advantages and disadvantages. The general architecture of the hardware and later on we will analyze the language Id, as well as a dataflow graph by a few example programs. Afterwards we take a closer look on “Wavescalar Architecture” in Chapter 3. Here we will figure out how it works in general, what a wave is and how the whole architecture works shown in an example. As last architecture we take a detailed look on “TRIPS Architecture” in Chapter 4. Here we will take advantage of “direct instruction communication”. We will look out for block compilation and show how compiling is realized in TRIPS. Also we will ensure the advantages of hyperblock. The last section of the review called “Conclusions, Results, Discussion” will be used to discuss all those three architectures, mention their advantages and disadvantages and take a closer look which architecture is dominating in different areas of tasks and how the future for each architecture could look like.

2 MIT Tagged-Token Dataflow Architecture

2.1 Introduction

The idea behind MIT Tagged-Token dataflow architecture as shown in the Paper Executing a program on the MIT-Tagged-Token dataflow architecture is not to go with a conventional sequential language. It comes with Id, which is a high level language with fine grained parallelism. The program written in Id is getting compiled into a “dynamic dataflow graph” which is executed on the MIT Tagged-Token dataflow architecture (TTDA). In Section 2.2 “id language” we are going to have a closer look at how Id works, and what the benefits of Id are. Afterwards we will see in Section 2.3 “Dataflow graph” how the Id programs get compiled into dataflow graphs. And the last section for MIT Tagged Token dataflow architecture deals with how the processing elements work.

2.2 Id language

The main motivation to develop a language like Id was that a parallel programming language has to provide a few characteristics. First of all the programmer has to be isolated from any details about the machine itself. He need not to think about how many cores the processor has or how fast it is. Also the communication network or the network topology should not matter for the programmer. The language also should have parallelism in the operational semantics, so the programmer need not identify parallelism explicitly. Hence the objective is to create a language where a programmer can write his code without paying attention to the machine it is running on. The code shall run on any machine meeting the minimum requirements. Id is a high-level functional programming language. It is augmented with
a deterministic, parallel datastructuring mechanism called I-Structures. Also we have a function called "currying", which means we are able to give meaning to any expressions.

\[
\text{Def } vsum \ A \ B = \\
\{ \ C = \text{array} \ 1 \ n \}; \\
\{ \text{For } j \text{ From } 1 \text{ To } n \text{ Do} \\
\]

In Figure 1 we have a function \( vsum \ A \) we can give it the meaning \( \text{move} \ . \point \) just by defining it as: \( \text{move} \ . \point = vsum \ A \). If we did so the application \( \text{move} \ . \point \ p \) is equivalent to the application \( vsum \ A \ p \). The function \( vsum \ A \ p \) will take a point \( p \) and create a new point which is exactly a distance \( A \) away from the old point \( p \). We can also see how parallelism works here. Since the iterations of the loops are independent they can be solved in parallel. In any block, the return expression and the statements will all be executed in parallel as far as data dependencies allow it. Right now we will have a look for an example code in Id which will give the inner product of two vectors:

\[
\text{Def } ip \ A \ B = \{ \ s = 0 \\
\text{In} \\
\{ \text{For } j \text{ From } 1 \text{ To } n \text{ Do} \\
\quad \text{Next } s = s + A[j] \times B[j] \\
\quad \text{Finally } s \} \}
\]

Figure 1: Function \( vsum \) that can add 2 Vectors

In Figure 2 first of all we use currying to define the name to “ip”. Ip has 2 input vectors called A and B. The next thing happening is having a sum “s” which is initially set to “0”. During the jth iteration of the loop, the “s” for the next iteration is bound to the sum of s for the current iteration and the product of the jth elements of the vectors. The value of the s after the nth iteration is returned as the value of the loop, block and function.

Another feature mentioned before is the I-Structure semantics. I-Structures are array-like data structures related to terms in logic programming languages. What I-Structure does is for example eliminating read-write races. This means that it is possible to read elements even before they have been defined, this is also called "nonstrict". Nonstrictness highly increases the opportunity for parallelism.

2.3 Dataflow graph

To understand dataflow graphs we will start by introducing them with an example dataflow graph of the “inner product of two vectors” program mentioned before in Figure 2.

What we can see here in Figure 3 is that each dataflow graph consists of operators connected by directed arcs. These directed arcs represent data dependencies between operators. Thus for example we see the multiplier which needs as inputs both output arcs of the two select operators. The values between the operators that move along the arcs are carried on “tokens” have got the address of the operator as destination at the end of the arc. Every operator can only execute when it received tokens on all its
Figure 3: Dataflow graph for the inner product of two vectors

input arcs. The condition when all inputs are having tokens is called “ready to fire”. Next is that the operator as soon as it executes ("fires") takes the tokens on all the inputs and generates the tokens for all its outputs. Therefore we can see that in one dataflow graph it is possible to have multiple operators firing at the same time. For any function we compiled into a dataflow graph, we will get an input arc for each formal parameter of the function and an output arc for each result of the function. Functions are able to call other functions and get the result returned from that. This is realized by a call/return scheme which only supports nonstrict functions. Important is to take a closer look at the above already mentioned I-Structure. Since I-Structure resides in the global memory tokens only carry a descriptor for a pointer to an I-Structure. So if we have a fork we do not need to duplicate the I-Structure, we just have to duplicate the token and the pointer by association. Now we look at the I-Structure storage. It is a memory module which contains a controller that handles the I-Structure read and write requests and initializes the storage. Beside the I-Structure storage we have the I-Structure select operation which gives the opportunity for split-phase reads. This means the request and the reply are not synchronous so the processor can execute any number of other enabled dataflow instructions during the memory fetch. The I-Structure assignment handles the distribution of the memory to the I-Store operators. I-Store operators generate the store tokens as well as a signal token which is necessary to detect the termination of the function. Here it is important to know that a signal token does not mean that the write to memory has been done. Last thing we discuss is the I-Structure allocation. It has three important tasks:

- allocates a free area of I-Structure memory
- initializes all locations to the absent state
- sends the descriptor to the instruction

I-Structure allows us to implement easily other functions that our Id programs may require to be more efficient.

After we finished bringing all the basic knowledge about dataflow graphs it is time to go a bit more in detail. Within a dataflow graph we have different units called codeblocks. Every code block is a user defined compiled function written in Id. Similarly every loop within another loop (also called inner loop) is defined as a separate code block. In the code a function call is nothing different than in the dataflow graph executing one of those units. Every unit can have multiple output arcs but has to have one termination signal output which shows the end of the active execution of the unit. As soon as this signal is sent the unit is vacant for the next execution.
Final two things to be mentioned for finishing the dataflow graphs will be loops and managers. Loops are an efficient implementation of tail recursive functions. The Id compile is able to transform tail recursive forms as soon as it recognizes them into loops. For better understanding we will now go back to Figure 2 the “inner product of two vectors” program and demonstrate it based on a “while loop” and the corresponding dataflow graph.

\[
\text{Def } \text{ip } A \ B = \{ \text{s = 0 ;} \\
\text{j = 1} \\
\text{In} \\
\{ \text{While } (j \leq n) \text{ Do} \\
\text{Next } j = j + 1 ; \\
\text{Next } s = s + A[j] \cdot B[j] \\
\text{Finally } s \} \}
\]

In the dataflow graph there are 4 input arcs: A,B,s,j. These are equally defined as variables A,B,s,j in the body of the loop. The next call in the loop shows that the two variables s and j are bound on every iteration of the loop again. That is why they are called circulating variables. For every circulating variable (s,j) the dataflow graph got also an output arc. A and B are called “loop constant” since they do not change their value. The dataflow graph shows us that s and j will circulate as long as the operation \( j = j + 1 \) is true. Since for parallelism all the executions can be solved asynchronous it can happen that the j tokens for example circulate way faster than the s tokens. This would mean that it could happen for an operator there can be different tokens generated by j so he would have to decide which one to take.

Solving this is pretty simple managed by giving the tokens information about the iteration. Therefore we got the D and D-reset operator, while D changes the context of its input token to a new context, D-reset resets it to the context of the entire loop.

Another big point in loops is to regulate the amount of loops existing simultaneously. This is done by controlling all the inputs of the loop body by the bank of switch operators on top of the loop. Switch operators themselves are controlled by a single boolean value from the loop predicate. Now we are able to regulate the rate of the unfolding loop. In Figure 6 we can see how our dataflow graph changed for including throttling of the loops. Now we got the synchronization block which generates the boolean value for control the input variables of the loop.

What we can see here is that all the select statements were packed up into the “loop body”, too.

Last thing to be mentioned for dataflow graphs are “Managers”. For the fact that any machine needs to allocate and deallocate resources dynamically like memory or frame and heap for dataflow graphs there are resource managers. They have to manage all the resources needed for any part of the program.
and they also got to manage this service for multiple processes since we are talking about parallelism. To provide this first of all each resource manager must have private data structures that are shared across all calls to that manager. Next thing is that multiple calls of a manager need to be serialized so the manipulation of the data structures is done consistently. In Id it is possible to provide this by coding the manager as an ordinary Id program which runs continuously and concurrently with the main program we are executing. The manager program we run uses privileged instructions that allows it to manipulate the state of the machine. This gives us the opportunity for example to allocate and deallocate memory. Finally managers allow the programmer to code independently from the hardware, thereby fullfilling the main aim of the Id language (mentioned in Section 2.2).

2.4 TTDA processing element overview

In this section it is time to take a closer look at the processing elements (PE) in the Tagged-Token dataflow architecture. Therefore, first of all a single processor operation is analyzed and later on the capability to multi processor is shown up. In the graphic ?? a processing element and all its different units are shown. Below a description of all the different units will be given.

The **Wait-Match Unit** is a memory containing a pool of waiting tokens. The tokens stay in the “Wait-Match” unit until all tokens with the same tag arrived. Then they are given to the “Instruction-Fetch Unit”.

In the **Instruction-Fetch Unit** the tag on the operand tokens identifies an instruction to be fetched from the program memory. This is managed by using a code block register (CBR). In the instruction that became fetched, a literal or an offset into the constant area and a list of destination instruction offsets can be included. The tag also specifies a database register (DBR). All those informations are passed to the next stage: the “ALU” and “Compute-Tag Unit”.

**ALU and Compute-Tag Unit** receive the data and those two ALUs get different jobs. The ALU is a conventional one which takes the operand/literal/constant data values and the opcode, performs an operation and produces a result. Meanwhile (since they work in parallel) the compute-tag unit takes the code block register and the database register numbers as well as the instruction offsets for the destinations
to compute the tags for the output of the operator. When both results are available they are passed to the “Form-Tokens Unit”. The **Form-Tokens Unit** is taking both results, the data from the “ALU”, and the tags from the “Compute-Tag” and combines them into new tokens called the “result tokens”. Last unit is the **Control Unit**. It receives special tokens which can manipulate any part of the processing elements state. These tokens are generated by various managers to for example initialize code-block registers or store values.

After understanding how a single processing element works it comes up that the changes for multi-processor operations are not that big. The important thing is to consider that all the memories (program, constant, I-Structure) are addressed globally. As this is given we see that for a multi processor system any PE can do any execution that is needed. Many iterations of loops for example can be solved parallel. It is no problem if all the memories are available to solve them on different processors. The token structure as well as the I-Structure allows bringing all the results together and deliver a final result for any program written in \Id.

### 3 Wavescalar Architecture

#### 3.1 Introduction

Wavescalar as mentioned in the Paper “Wavescalar” is getting interesting concerning one big problem: it is not enough to scale up current architectures to improve the performance. The fundamental problem is that it is not enough to try improving the Superscalar processor designs, way more it is necessary to try to reduce the wire delays and improve the broadcast networks. The Wavescalar model is an
alternative to Superscalar design. It is a dataflow instruction set architecture and with its execution model designed for scalable low-complexity/high-performance processors. Wavescalar provides the unique feature to run programs written in any language without sacrificing parallelism. Also it provides the minimization of long wired, high latency communication by co-locating computation and the data in physical space. Finally it includes a completely decentralized implementation of the “token-store” of dataflow architectures. So the key feature is: Wavescalar supports traditional “von-Neumann” memory semantics in form of a dataflow model. Until now every dataflow model provided an own dataflow language and memory style which makes it difficult to use them as a general solution for improvement. It is important to understand how the wavecache works when talking about wavescalar, since it provides the ability to cache any instructions and execute them. So all the instructions are executed in-place in the memory system and their results are sent to the dependents. One of the reasons for creating such a system was “dataflow locality”. Dataflow locality describes the predictability of instruction dependencies through the dynamic trace of an application. Right here we have to distinguish between static dataflow locality and dynamic dataflow locality. Static dataflow locality exists since in the absence of control, the consumer and producer of register values is already known, dynamic dataflow locality arises from the branch predictability. This means from knowing which instruction is used it is highly predictable which instructions will be communicated next. If it would be possible to ensure that necessary inputs are close to the using units the architecture would be improved significantly.

3.2 Wavescalar ISA

With Wavescalar there is a new approach to dataflow computing. A Wavescalar binary is pretty much the dataflow graph of any executable program. It is placed in the memory as a collection of intelligent instruction words. This is given by a “wavecache” which holds the current working set of instructions and executes them right in place. A new feature for Wavescalar compared to the common dataflow is the special instructions for managing the control flow it provides. When talking about “Dataflow Instructions” we always talk about the need of converting control dependencies into data dependencies. To provide this there are two solutions available:

- First a conditional selector which takes two input values and a boolean selector input. The selector boolean will determine which input value is given to the output.
- The second solution for this is a conditional split. This one will take an input value and a boolean output selector and passes the input through one of the two paths depending on the output boolean.

Next to be mentioned in detail is “Waves”. Waves are generated by breaking the control flow graph of an application into pieces. A Wavescalar processor is able to execute one wave at a time. Each wave has three main properties:

- each time a processor executes a wave, each instruction is executed once
- there are no loops in the instructions since they are ordered partially
- the control can enter only at one point

What the Wavescalar compiler does is mostly to split each application into a maximum amount of waves and to add some wave management instructions. For the coordination and the handling of the complexity of all this not only tokens are used by Wavescalar. Here are some new sorts of tokens introduced called “Wave Numbers”. While every data value carries a tag, wave numbers are introduced to manage the tags across the waves. This means for example in a loop one processing element handles the instructions for all the iterations and wave numbers make sure that in every iteration only values of the iterations are
used. One of the key features about those waves numbers is that the wave numbers management can be
totally distributed to software control. The usual dataflow tag management only allows to either partially
distribute or completely centralize the tag management. **“Indirect Jumps”** offer the option for indirect
function calls. It is realized by adding a new instruction called “INDIRECT-SEND”. “INDIRECT-SEND” has three inputs:

- a data value
- an address
- an offset

All that “INDIRECT-SEND” does is to send the value to the consumer instruction located at the ad-
dress plus the offset. While understanding all this it is necessary to mention **“Memory Ordering”**. The Wavescalar architecture brings load-store ordering to dataflow computing while using wave ordered
memory. This means that wave ordered memory annotates each memory operation with its location in
its wave and its ordering relationships with other memory operations in the same wave. Since those
annotations travel with the memory request after execution it is guaranteed that the system can apply
the memory operations in the correct order. As soon as a memory instruction is executed the following
information is sent:

- its link
- the wave number (taken from an input value)
- an address
- data (for a store)

That information is used by the memory system to assemble the loads and stores in the right order for
not having any gaps in the sequence. To eliminate possible gaps in the sequence there is a “no gap rule”:

” No path through the program may contain a pair of memory operations in which the first operation’s
succ value and the second operation’s pred value are both ‘?’”

Wave ordered memory is the key for executing programs written in conventional languages efficiently
since it allows to separate memory ordering from the control flow. In the next Section ?? we will take a
look at a simple Wavecache and see how it is structured.

### 3.3 Wavecache

Right now it is time to just give an simple example of how such a Wavecache looks like. In Figure 7 a
simple Wavecache is shown. It is a grid of processing elements arranged into clusters of 16 each. The
processing elements contain:

- logic to control instruction placement and execution
- input and output queues for instruction operands
- a communication unit
- a functional unit

Furthermore, each processing unit contains buffering and storage for 8 different instructions. The
input queue are indexed relative to the current wave while a small multi-ported RAM holds full empty bits
for each entry in the input queue what maximizes the efficiency of the input queue. Within each cluster
of 16 processing elements the communication is performed in a set of shared buses. What we can see
here is that if the cluster size gets too big the wire delay will stop any increase of performance. Besides the buses for communication within a cluster there are dynamical routed on-chip networks used for inter-cluster communication. A characteristic is that each hop crosses one cluster and takes exactly a single cycle. These facts lead to the conclusion that a good instruction placement is the key feature for optimal performance. Bad instruction placement would lead to high wire delays and higher communication latency than necessary. Finally when we talk about Wavecache we have to mention store buffers. All the memory requests for a wave bound to a store buffer go directly to it. As soon as the store buffer completes, it signals the store buffer for the next wave to proceed. This scheme leads to a logically centralized store buffer which can migrate around the whole Wavecache as needed. At the moment there are two common solutions for the problem of binding a wave onto a store buffer:

1. adding an instruction called “MEM-COORDINATE” which acquires a store buffer on behalf of his wave and passes the name of the store buffer to the memory instructions of the wave

2. using a hash table kept in main memory that maps wave numbers onto store buffers. Every memory instruction sends their request to the nearest store buffer. So if the wave does not have a store buffer yet, the store buffer uses the cache coherence system to get exclusive access to the waves entry and fills in his own location.

### 3.4 Example for Wavescalar

In this section we will look at a simple code fragment in Figure 8 which copies all the values from “in” into “out”. This is realized by using a do-while loop:

Next there will be shown in Figure 9, the RISC assembly (a) for the program shown in figure 8. In the center there is the Wavescalar version of it (b) and on the right is the Wavescalar version mapped onto a small Wavecache (c).
Figure 8: Code example to copy “in” into “out” realized with a “do-while” loop

4 TRIPS Architecture

4.1 Introduction

The TRIPS architecture is based on the EDGE instruction set architecture. The motivation to develop TRIPS is to bring a few new characteristics to provide a lot of new potential to improve the performance of a machine. Therefore, in the paper "Scaling to the End of Silicon with EDGE Architecture" four major characteristics are mentioned:

1. The depth limit of pipelines are achieved hence other fine-grained concurrency mechanisms are needed to further improve the performance.
2. Power limits are hastened by the extreme acceleration of clock speed. Therefore, it is necessary to set a power ceiling and try to obtain as much performance as possible without breaking it. In other words "power efficiency" becomes very important.
3. Increasing resistive delays through global on-chip wires means that future ISAs must be amenable to on-chip communication-dominated execution
4. Polymorphism is getting necessary. Polymorphism is the ability to use the execution and memory units in different ways and modes to run diverse applications.

To fix those issues as good as possible there are different approaches. The ISA of an EDGE architecture supports “direct instruction communication”. This feature allows it to deliver the output of an producer’s instruction directly to the input of a consumer instruction. The advantage in “direct instruction communication” is that there is no “write back to” for example a register file necessary. A few of the four above mentioned improvement points are tried to be solved in EDGE. EDGE already has a higher exposed concurrency and more power-efficient execution. There is also a better interface between the compiler and the microarchitecture. As a consequence there is no need for the hardware to rediscover data dependencies at the runtime since this is solved already by having the ISA directly expressing the dataflow graph generated by the compiler. In Section 4.2 the TRIPS architecture basics” the architecture of trips will be explained. Afterwards “Block Compilation” is shown in Section 4.3. The next thing will be taking a closer look at is the “Compiling for TRIPS” in Section 4.4. In the last Section 4.5 “Supporting parallelism” a general look at the different options to support parallelism in TRIPS is taken.
4.2 The TRIPS architecture basics

In the introduction above the four major characteristics were mentioned that are necessary to improve efficiency of a machine. TRIPS is trying to cover those characteristics:

1. **Increase concurrency**: to cover this the TRIPS ISA includes an array of concurrently executing ALUs that provide scalable issue width as well as scalable instruction window size.

2. **Power-efficient high performance**: to attain “power-efficient high performance” the architecture works with large 100-plus instruction blocks instead of overheads like in the “von-Neumann” semantic.

3. **mitigate communication delays**: are given by compile-time instruction placement which minimizes the physical distance that operands must travel for dependent instruction chains across the chip. Minimizing the physical distance leads to minimizing the execution delay.

4. **Memory flexibility**: is given by configural memory banks which provide a general-purpose, highly programable spatial computing substrate.

TRIPS supports conventional languages like C, C++, Fortran by using **block-atomic execution**. This means that the compiler is grouping single instructions into blocks of instructions also called “hyper-blocks”. Each of those blocks is fetched, executed, and committed automatically. It is important that a block can only be committed entirely or needs to be rolled back. It is impossible to only commit parts of the block. One of those blocks can contain up to 128 instructions which are getting mapped to an array of execution units by the compiler. Within each executing block the hardware uses fine-grained dataflow model with direct instruction communication to execute as efficient and quick as possible. Fetching and mapping of a block is realized by fetching the instructions in parallel by the processor and loading them into the instruction buffers at each ALU in the array. By this block wise mapping and execution model it is ensured that there is no need to go through any fully shared structures like register files. This could only occur if there is an instruction communicating across distinct blocks. Besides that loads and stores must access to the data cache and memory ordering hardware.
### 4.3 Block compilation

For understanding the “Block compilation” the example code in Figure 10 is getting compiled and the resulting dataflow graph in Figure 11 and afterwards the TRIP’s instruction block is shown. This code example is taking an input “y” and writing it into a register. Next we will have a look at the TRIPS’ dataflow graph which is constructed by the TRIPS’ compiler.

![Figure 10: Code example which allocates the input integer “y” into a register](image1)

On the dataflow graph we can see two read instructions obtaining the register values and forwarding them to their consuming instructions which are mapped on the execution array. The dotted lines are control dependent instructions predicated on the result of a test instruction by the compiler. In figure e, the instruction block of the code a TRIPS compiler would generate is shown. Characteristically instructions do not contain their source operands, they only contain the physical location of their dependent consumers.

![Figure 11: Dataflow graph and instruction block for the code example in Figure 10](image2)

### 4.4 Compiling for TRIPS

Compiling is a function which is very close to the microarchitecture since any architecture works only in an optimized way if the strength and capabilities of the compiler and the microarchitecture are used in the best way. EDGE is already at a state where a proper division between the compiler and the architecture matching their responsibilities to their capabilities is given. Dividing in this case means that all the
data dependencies are expressed by the compiler through the ISA. Meanwhile the hardware execution model has to handle dynamic events. Thereto we count variable memory latency or conditional branches as well as the issue order of instructions without needing to reconstruct any compile time information. Compared to the compilers already known by RISC or VLIW an EDGE compiler has to handle two new responsibilities:

1. forming hyperblocks for TRIPS architecture
2. spatial schedule of those blocks by assigning instructions from a block into execution arrays of the ALU’s.

On the code example in Figure 12 a simple code fragment is given to show how the compiler splits it up to construct the control flow graphs, the hyperblocks and later on the generated and scheduled code.

In Figure 13 the 5 “compiling steps” from control flow graph to scheduled code are shown:

a) **Control flow graph** after the first parsing of the code by the front end.

b) **Unrolled control flow graph:** Here the compiler has unrolled the inner “do-while” loop three times

c) Here the **hyperblocks** are generated. So all the basic blocks from the unrolled and “if-converted” loop (it was a “do-while” before) are placed inside a single hyperblock now

d) The code generated by the compiler. In this step also the registers were allocated. In picture d) the full **dataflow graph of the hyperblock** is shown. In this step the compiler splits the block into multiple hyperblocks if there are any illegal hyperblocks discovered

e) The last step is the final **scheduled code**. At this step the scheduler has to balance between two competing goals:
   1. placing independent instructions on different ALUs to increase concurrency
   2. placing instructions near one another to minimize routing distances and communication delays

### 4.5 Supporting parallelism

When talking about parallelism for the TRIPS architecture we have to distinguish three different parallelism classes.
Figure 13: 5 steps from control-flow-graph to scheduled code

1) **Instruction level parallelism**: is supported by TRIPS with minimal additional hardware support. It also supports loop driven data-level codes and threaded parallelism.

2) **Data-level parallelism**: short DLP is well suited by TRIPS. This is realized because the compiler can map the processing pipelines for multiple data streams to groups of processing elements. Besides that the compiler is trying to bring one loop body into one single block, this signifies that the processor only needs to fetch these instructions once from the cache.

3) **Thread-level parallelism**: short TLP is one of the parts with the best effort/efficient balance for optimization. While the TRIPS architecture supports up to four threads per processing core in most of the times the highest performance is reached by executing a single thread per processing core. For the processor’s control logic it is necessary that each processor got a separate copy of an architectural register file for each activated thread as well as some per-thread identifiers augmenting cache tags and a few other state information.

5 Conclusions, Results, Discussion

For all processor architectures, it is important to remember that today the problem of improving a machine and especially high speed general purpose parallel computing can only be solved by improving all the corresponding parts of it. For processor architectures this means that the synergy between the language, the compiler, and the architecture itself has to be maximized. After introducing all three different architectures we discuss here advantages and disadvantages of the mentioned architectures. The first going to talk about will be the “MIT Tagged-Token Dataflow Machine”. One of the advantages of it is its language Id which makes it possible to convert nearly every other language code into Id. This means that every common program can run on it. Dataflow graphs are the next step in this architecture since they enable a quite easy understanding of the program as well as the feature that any programmer does not need any knowledge about the machine itself to use parallelism. The architecture itself offers a quite simple but really good way of parallelism for the year 1990 since the program parts can be solved in parallel on any process element with vacant capacities. At the time when it came up this was a really well planned and reasoned solution but has big trouble to dominate against other architectures since it seems to be quite complex to bring the Id language and the I-Structure in perfect collaboration. The memory organization is well planned and has a high potential to be used in further years if it is extended in a more
complex way. A huge improvement of the dataflow machine can be observed within the wavescalar architecture. Nevertheless, wavescalar leaves still room for optimization. Possible starting points therefore are:

- **Cluster Size:** The clusters are one of the main characteristics of the Wavescalar architecture. The limitation of the cluster is its size: if the cluster gets too large the clock needs to be slowed down and the wire delay is getting bigger. Here is a lot of potential in finding the right clustersize for the task being solved and develop techniques to decrease the delays.

- **Cache replacement:** The problem occurring at this point is finding the right cache size for the running applications. If the cache is chosen too big there are delays as well as if chosen too small. There is even more improving potential in cache replacement by dynamic instruction placement and optimization for the dataflow locality.

- **Input queue size:** The queue size only affects the performance if it is chosen small. In this case the performance decreases. Therefore an optimization could only be reached by adjusting the queue size to the needs of the programs (mainly for programs needing a small queue size).

- **Control speculation:** Here we are effectively talking about branch prediction. Since this is a feature Wavecache is not using yet it is a great opportunity to improve here.

- **Memory speculation:** The essential part here is that a processor can execute memory accesses out of order even if they are at different locations. So if it is possible to use this effectively wavescalar can be improved significantly. But we should remember that this would bring only a high increase in performance if it is combined with branch prediction.

Compared to the superscalar architecture, wavescalar is outperforming in every stage. More interesting is the comparison of wavescalar to TRIPS. While wavescalar uses waves, TRIPS works with hyperblocks. When optimized for a specific program both architectures can reach nearly the same performance. However wavescalar outperforms TRIPS when it comes to complex programs due to its stronger memory ordering structure. Overall both architectures have potential for improvement. Still wavescalar has the better memory optimization opportunities and its tweaks are easier to access. TRIPS got the stronger optimization opportunities regarding to parallelism. Here the block-based approach gives a looser hardware binding than other architectures which would definitely need much more hardware support to achieve the same results. Refering to the architecture standard in Year 2003/2004 parallelism is just coming up and will be one of the most focused improvements for the hardware companies. So this can be the point where TRIPS maybe is able to out scale wavescalar if there will not be huge improvements in this section. So what TRIPS is doing well already is providing a solid base for single processors and still effectively exploit in DLP and TLP as soon as the software can discover it.

6 Bibliography

References


