A

Integer Linear Programming Based Scheduling for Transport Triggered Architectures

TOMI ÄIJÖ, PEKKA JÄÄSKELÄINEN, TAPIO ELOMAA, HEIKKI KULTALA, JARMO TAKALA, Tampere University of Technology, Finland

Static multi-issue machines such as traditional VLIW architectures move complexity from the hardware to the compiler. This is motivated by the ability to support high degrees of instruction-level parallelism without requiring complicated scheduling logic in the processor hardware. The simpler control hardware results in reduced area and power consumption, but leads to a challenge of engineering a compiler with good code generation quality.

Transport triggered architectures, and other so called exposed datapath architectures, take the compiler-oriented philosophy even further by pushing more details of the datapath under software control. The main benefit of this is the reduced register file pressure, with a drawback of adding even more complexity to the compiler side.

In this article, we propose an *Integer Linear Programming (ILP)* based instruction scheduling model for transport triggered architectures. The model describes the architecture characteristics, the particular processor resource constraints, and the operation dependencies of the scheduled program. The model is validated and measured by compiling application kernels to various transport triggered architectures with a different number of datapath components and connectivity. In the best case, the cycle count is reduced to 52% when comparing to a heuristic scheduler. In addition to producing shorter schedules, the number of register accesses in the compiled programs is generally notably less than those with the heuristic scheduler; at the best case, the ILP-scheduler reduced the number of register file reads to 33% of the heuristic results and register file writes to 18%. On the other hand, as expected, the integer linear programming based scheduler uses distinctly more time to produce a schedule than the heuristic scheduler, but the compilation time is within tolerable limits for production code generation.

Additional Key Words and Phrases: Code generation, Integer linear programming, Transport triggered architectures, Exposed datapath, Instruction-level parallelism

ACM Reference Format:
Tomi Äijö, Pekka Jääskeläinen, Tapio Elomaa, Heikki Kultala, and Jarmo Takala, 2015. Integer Linear Programming Based Scheduling for Transport Triggered Architectures. *ACM Trans. Architec. Code Optim.* V, N, Article A (January YYYY), 23 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000
[1]

1. INTRODUCTION

Static multi-issue machines such as traditional *Very Long Instruction Word (VLIW)* architectures move complexity from the hardware control logic to the compiler [Fisher 1983]. The compiler-oriented architectures are motivated by their ability to support high degrees of instruction-level parallelism without requiring a major part of the

[1]New Paper, Not an Extension of a Conference Paper.

This work was funded by Academy of Finland (funding decision 253087), Finnish Funding Agency for Technology and Innovation (project ”Parallel Acceleration”, funding decision 40115/13), and ARTEMIS JU under grant agreement no 621439 (ALMARVI).
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© YYYY ACM. 1544-3566/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

chip area dedicated to a complicated scheduling logic in the processor hardware. The simpler control hardware results in reduced area and power consumption, but leads to a challenge of engineering a compiler that can utilize the processor efficiently.

Transport Triggered Architectures (TTA) and other so called exposed datapath architectures [Jääskeläinen et al. 2015], reveal even more details of the datapath under the control of the software. For TTA, the main benefits of this are in reducing the register file pressure, which allows more energy efficient and scalable designs due to the reduced general purpose register file bottleneck. Clearly, the exposed datapath adds even more complexity to the compiler side due to the additional scheduling freedoms. The challenges to exploit the datapath automatically from higher level software descriptions has in part hindered the appeal of these machines, despite their excellent potential for low power high performance designs.

The code generation phase in compilers for exposed datapath cores is commonly implemented using heuristic methods as they produce results in a reasonable time. The quality of the produced code, however, is usually suboptimal. Code generators based on mathematical models can be used to produce optimal results, but they are notorious for their superpolynomial complexity, leading to compile time explosion with larger programs. Therefore, although the computational power that can be utilized for solving the models has increased and keeps increasing dramatically, it is still the case that mathematical models are used only for scheduling hot spots of the program such as heavily executed inner loops. In order to make the compilation time feasible, the less frequently executed parts can be processed with a heuristic code generator as their code quality is not as significant to the total execution time.

In this article, we propose an *Integer Linear Programming (ILP)* based mathematical model that supports the unique scheduling freedoms presented by the TTA programming model. To the best of our knowledge, this is the first publication of such a working model. We show that the proposed model can provide significant improvements compared to a scheduler based on heuristics, with the compilation time still being reasonable for production code generation. The contribution has major benefits due to the challenges of efficiently compiling high-level programs for irregular processors with high degree of programmer control such as reduced connectivity network TTAs.

The rest of this article is organized as follows. In order to understand its additional challenges placed to the scheduling problem, Section 2 introduces the TTA and its programming model. Section 3 describes the proposed ILP model. The model is evaluated in Section 4 using two distinctly different TTA machines and a set of benchmark programs. Previous similar code generation work is revised in Section 5. Some of the planned future work is discussed in Section 6 before concluding the article in Section 7.

2. TRANSPORT TRIGGERED PROGRAMMING MODEL

The classical VLIW is a processor architecture style that is designed to take advantage of instruction-level parallelism in programs without requiring complex control unit hardware by making the parallelized operations explicit to the programmer [Fisher 1983]. However, there are scalability issues with the common VLIW style; when the number of function units is increased to support more instruction-level parallelism, the interconnection network and the register file complexity of a VLIW processor (due to the need to assume worst case register file accesses) grows dramatically. More complex bypass network and the large number of register file ports lead to increased power consumption, increased chip area, and reduced clock frequency [Hoogerbrugge and Corporaal 1994].

Transport triggered architecture is a processor design style which has been proposed to alleviate some of the bottlenecks of VLIW-style processor designs [Corporaal 1995].

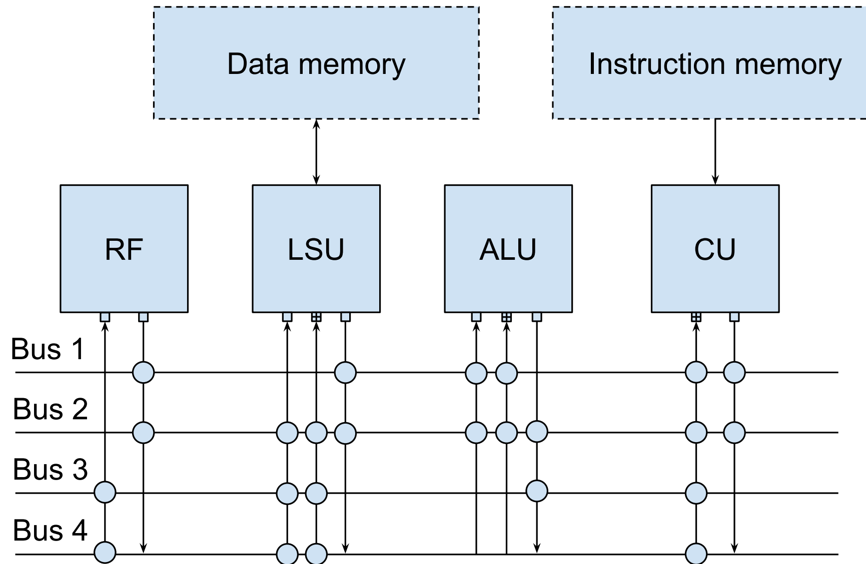

Fig. 1: A simple TTA processor with four buses and four units. Socket-bus connections are illustrated with circles over intersections. The arrows in the sockets indicate the data transfer direction. The ports of the units are visualized with the squares below the units. The highlighted ports are trigger ports.

TTA generalizes VLIW in a sense that the interconnection network is visible to the programmer, which even further moves complexity from the hardware to the compiler. As a side effect, it alleviates the interconnection network and register file bottleneck by means of a more feasible interconnection (or bypass) network customization and reduced number of simultaneous general-purpose register file accesses. This translates to energy savings in comparison to traditional VLIW designs due to general purpose register accesses often consisting a significant part of the total power consumption.

In the TTA processor template targeted by the model in this article [Cilio et al. 2006], the TTA interconnection network is composed of *sockets* that are attached to *buses*. Sockets can be either unidirectional or bidirectional depending on the ports attached to them. Figure 1 shows an example TTA processor consisting of four different units: an *Arithmetic Logical Unit (ALU)*, a *Load-Store Unit (LSU)*, a *Register File (RF)*, and a *Control Unit (CU)*. The units can be either *Function Units (FU)* that execute operations or general-purpose RFs that store intermediate values. In the example, CU, ALU, and LSU are all classified as FUs and there is only one RF.

Each unit port has a socket attached to it, and these sockets are further attached to one or more buses. In Fig. 1, the socket-bus connections are illustrated with circles over the intersection of a bus and a socket. The arrows at the sockets indicate the direction of data transfers. Although the example TTA in the figure has only one ALU, a single LSU, and a very simple RF with only two ports, the TTA processors supported by the referred template can have an arbitrary number of computational resources to support different degrees of instruction-level parallelism available in applications of interest. Furthermore, the connectivity and the number of buses is also completely customizable in the processor template, imposing an additional challenge to the proposed model.

As TTAs are in-order processors without instruction scheduling hardware, the control unit only fetches instructions from memory, decodes them to control signals, and implements control flow affecting operations such as *call* and *jump*.

Although the TTA approach clearly has it benefits, TTAs are not considered the best option for general purpose operating systems or for reactive applications because exceptions and fast interrupts are expensive to implement in TTAs due to the abundance of programmer visible architecture context data. The authors have found the best uses for TTA-style processors in customized programmable designs that execute data oriented kernels and might have high performance requirements combined with a strict power budget, or at the another end of the spectrum; ultra low power IoT scenarios which need some computational performance. For handling exceptions and fast interrupts, other style cores can be used either in an assisting role to a TTA master, or in a more traditional host-slave offloading setup with a general purpose core as a master.

For these reasons, there are no fabricated TTA general purpose processors available in bulk, but TTAs are used as accelerators in customized SoCs of which internals are often company secrets. In addition to various academic TTA designs, the authors know about two that are advertised in commercial products [KDPOF 2014; Maxim Integrated Products, Inc. 2004], and of multiple designs fabricated to test chips but not yet sold or published. Furthermore, several commercial designs contain other variations of the "exposed datapath" concept which can benefit from the model described in this article [Jääskeläinen et al. 2015].

2.1. Programming Model

The VLIW processor instruction stream is explicitly parallel, consisting of sets of operations that are executed at each cycle. The operations describe the operation codes to execute, the input operands and the result register as general-purpose register indexes. The hardware then decodes these instructions to control signals that perform movements of data in the interconnection network of the datapath at correct time slots.

The programming model of TTA increases the visibility of the datapath to the programmer by making the interconnection data transports explicit. TTA programs consist of instructions which describe parallel data transports, later referred to as *moves*, between the units on the interconnection network. In this programming style, the concept of a bus is tightly bound to the concept of a *move slot* in the instruction word. A bus can transfer a single value between two connected *sockets* at each instruction cycle. The source and destination of the transfer is encoded in the instruction word in a move slot that controls a single bus.

A *program operation* is divided into a set of *operand* and *result* moves. The actual operations are launched as a side effect of these data transports into the units, hence the name *transport triggered*. In the used TTA template, exactly one of the operand moves is a *trigger move* that starts the execution of an operation in a function unit.

The template supports constant values or *immediates* in two ways; encoded into the move operation source field (*short immediates*) or encoded to occupy one or more move slots in the instruction word (*long immediates*). The latter is controlled using separate template bits in the instruction word, which also describe the destination *Immediate Unit (IU)* where the decoder should transfer the constant when encountering the instruction in the stream.

As a concrete example of the programming model, let us consider a program operation of two inputs `ADD x,y` $\rightarrow$ `z`. It is described as a TTA program by means of two operand moves: `x` $\rightarrow$ `ALU.in` and `y` $\rightarrow$ `ALU.t.ADD`, and a single result write move `ALU.out` $\rightarrow$ `z`. This naming convention is typical when working with the referred TTA template: `in` $(1...n)$ are input ports, `t` is the trigger port, and `out` $(1...n)$ are output ports for reading the results.

Input operands of the addition operation are transported through the connections provided by the interconnection network to a compatible unit. When compiling from a high-level language, the selection of which function unit to use for the operation is the responsibility of the instruction selector or, as in our case, the instruction scheduler.

After the operation latency, the result is read from the ALU unit and moved to the register `z`. The latency depends on the implementation of the operation in the chosen FU. For example, it can be a multicycle implementation, a pipelined multicycle implementation, or a single cycle implementation. In addition, the operation can share resources with other operations in the same FU, implying additional scheduling constraints.

An additional challenge that was placed to the proposed instruction scheduler model is that the timing of the result reads is explicit in the used template; reading an FU port before the result is ready causes reading of the previous value, and reading it too late returns a more recently computed value.

2.2. TTA Specific Optimizations

The transport triggered architecture programming model allows unique software optimizations. Perhaps the most well-known one is *software bypassing* which is an optimization that is traditionally performed at run-time by the processor hardware's register file bypassing or "forwarding" logic in case of dynamic processors [Corporaal and Hoogerbrugge 1995]. In software bypassing, the program transfers a result of an operation directly from an FU producing it to the inputs of FUs that need the value. This alleviates the pressure placed on the general-purpose registers and especially the ports of the register files. Furthermore, bypassing may eliminate some false dependencies between operations that otherwise would need to share a general-purpose register, increasing the instruction-level parallelism.

If all the reads of the result can be bypassed to the destination by the program, software bypassing may result in an unnecessary register file write move. This removal of result moves to general purpose registers that are never read is called *Dead-result (move) elimination* optimization.

For example, consider a TTA program containing two dependent operations `ADD x,y → z` and `SUB z,a → b`. The latter operation uses the result of the former operation, thus the latter depends on the former. These subsequent moves produce the following TTA moves:

```
x → ALU.in
y → ALU.t.ADD
ALU.out → z
z → ALU.in
a → ALU.t.SUB
ALU.out → b
```

If the machine has a connection between the `ALU.out` and `ALU.in`, the result of the addition operation can be bypassed directly to `ALU.in`. After the bypass, move `ALU.out → z` can be eliminated as there are no other uses for the result:

```
x → ALU.in
y → ALU.t.ADD
ALU.out → ALU.in
a → ALU.t.SUB
ALU.out → b
```

Apart from handling software bypassing and dead result elimination, the TTA programming model presents the timing of the operand and result transfers as a new

scheduling freedom. While the timing of data transfers is fixed in traditional VLIWs, the TTA model allows choosing the timing for the transfers freely, which further reduces the RF port bottleneck, but presents yet another parameter that must be controlled efficiently using the scheduling model.

3. ILP FORMULATION FOR TTA INSTRUCTION SCHEDULING

In this section, we introduce shortly the basics of the involved ILP concepts and more detailed discussion can be found e.g., from [Rossi et al. 2006]. Then we describe the proposed ILP model, focusing on the challenges presented by the unique degrees of freedom in programming TTAs.

3.1. Integer Linear Programming Fundamentals

A *Constraint Satisfaction Problem (CSP)* $\mathcal{P}$ is a triple $\mathcal{P} = \langle X, D, C \rangle$, where X is an n-tuple of variables $X = \langle x_1, x_2, ..., x_n \rangle$, D is an n-tuple of domains $D = \langle D_1, D_2, ..., D_n \rangle$ and C is an m-tuple of constraints $C = \langle C_1, C_2, ..., C_m \rangle$. Each variable x_i is in the corresponding domain, $x_i \in D_i$. A constraint C_j is a pair $\langle X, S_j \rangle$, where S_j is a relation on the variables.

A *feasible solution* to a CSP is an assignment $\underline{x}$ of values to the variables in X, where the assigned values appear in their respective domains D, that satisfies the constraints in C. There might be no solution, an unique solution, or multiple solutions. Typically the applications are *under-constrained*, that is they have multiple solutions which can be arranged according to some property of the solution. These kinds of problems are called *Constraint Optimization Problems (COP)*. In addition to the constraints, they have an objective function f which is to be either minimized or maximized. An *optimal solution* $\underline{x}^*$ is a feasible solution such that for any other feasible solution $\underline{x}'$, $f(\underline{x}^*) \leq f(\underline{x}')$ given that the objective function is to be minimized.

Integer Linear Programming problems are a subset of constraint satisfaction problems in which the variables X are restricted to be integers, and the objective function f as well as the constraints D are linear. Moreover, $0-1$ integer programming is a special case of integer linear programming problems where the variables are required to be binary. ILP has been applied widely to "real world" problems such as operations scheduling, artificial intelligence, and resource allocation [Jünger et al. 2010].

Constraint satisfaction problems are typically NP-hard [Mackworth 1977], and many important constraint satisfaction problems have been proven to be NP-complete [Rossi et al. 2006]. In particular, integer linear programming is NP-hard [Jongen et al. 2004] and $0-1$ integer programming is NP-complete [Karp 1972]. Therefore, compiler algorithms utilizing ILP are not generally applicable, but they are usually used for only the most important parts of the program with feasible size.

Constraint satisfaction problems are typically solved using some sort of search exploring all the possible assignments of the variables. Typical techniques are backtracking and constraint propagation. There are numerous algorithms to solve integer linear problems exactly. A popular solver algorithm is *Branch and Bound (BB)* algorithm [Land and Doig 1960] which is used also in our work. Today there are sufficient commercial ILP solvers available (e.g. CPLEX, Gurobi, XPress optimization suite), which promote the use of ILP in code optimization.

3.2. Model Formulation

The local instruction scheduler is modeled as a $0-1$ ILP problem. The input to the scheduler consists of sequential operation moves generated from the program operations in the compiled high-level language software, and a *Data Dependency Graph (DDG)* built from the moves belonging to a basic block. The sequential intermediate representation has registers preallocated to physical machine registers, and it uses

only operations that are found in the target machine, but it does not yet have the operations assigned to function units, nor does it use software bypassing nor exploit instruction-level parallelism. The input format can be thought of as targeting a simple virtual scalar TTA machine that can trigger one instruction at a time. The DDG includes true and false dependencies due to register usage and memory accesses as edges connecting nodes that bundle together moves belonging to program operations such as ADD and MUL.

Initially, for each move there may exist multiple destination and source ports, and similarly there might be more than one connection between a pair of destination and source port on the interconnection network. Immediate values only have a destination port and an assigned bus. After scheduling, this sequential intermediate format must have all the moves assigned to the interconnection network onto a single cycle (instruction), a connection, and consequently the operations mapped to function units.

In order to integrate the software bypassing decision to the model, in addition to the moves generated from the compiled program, bypass move is created for each result move that is candidate for bypass optimization given the interconnection network. This move, the *bypass move*, is generated by taking the source from the *bypass candidate* move and the destination from the *bypass result* move.

The decision variables of the model specify the connection to be used, and the cycle the move is to be executed. We associate an indicator variable

$$M_{i,t,c} = \begin{cases} 1 & \text{if move } i \text{ is assigned to connection } c \text{ at cycle } t, \\ 0 & \text{otherwise,} \end{cases}$$

to describe a possible move assignment, indicated as a Boolean value.

The interconnection network confines a set of possible connections C_i for each move i. For each move there exist variables for all possible connections C_i, and cycles in the range $[t_{min}, t_{max}]$. This range depends of the input program, and the TTA machine in question. The length of the range greatly increases the model size and in turn the required solving time. A decent outcome can be achieved by setting the range from the DDG critical path length to heuristically scheduled program cycle count. Let P be the set of all moves in the considered TTA program.

3.3. Constraints

The following constraints are used in the proposed model to produce valid TTA schedules.

All Non-Bypass Related Moves Must be Assigned. All the moves that are not bypass moves, moves candidate for bypassing, or bypass result moves must be assigned exactly once to a connection and a cycle:

$$\forall i : \sum_{c \in C_i} \sum_{t=t_{min}}^{t_{max}} M_{i,c,t} = 1. \tag{1}$$

Dependencies Between Moves. The DDG defines the legal orders for execution of moves in a given program. Let $M_{i,c,t}$ be an indicator associated with an arbitrary move and $M^D_{i',c',t'}$ be its source dependence. The following constraint must be satisfied

$$M^D_{i',c',t'} + \sum_{t=t_{min}}^{t'+l} M_{i,c,t} \leq 1, \text{ for } t' \in [t_{min}, t_{max}]. \tag{2}$$

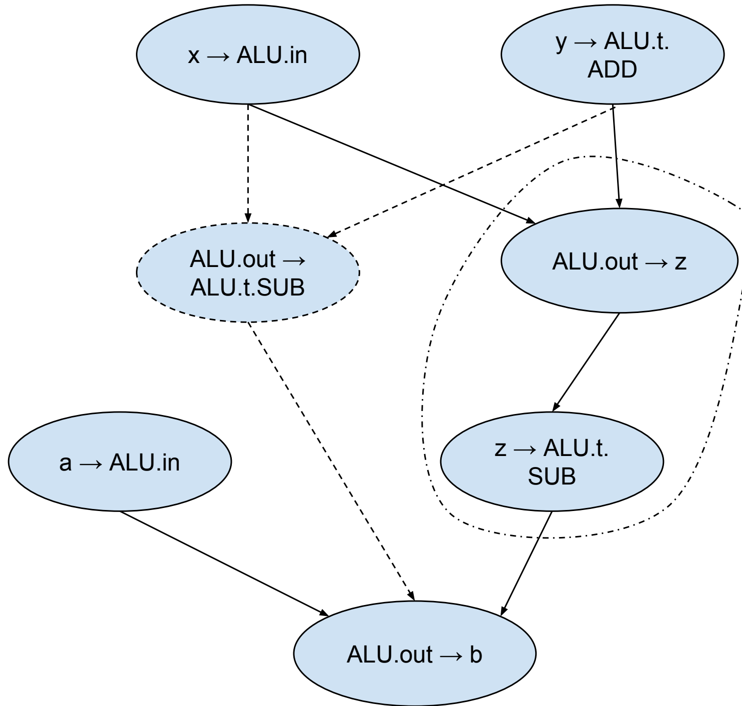

Fig. 2: Data dependence graph showing a bypass opportunity for move `z → ALU.t.SUB`. The dashed subgraph illustrate the variant after applying bypassing. The Nodes and dependency edges that are dashed indicate the potential bypass path.

The constraint requires for each t' that the dependent move must not be executed within time interval $[t_{min}, t' + l]$, where l is latency of the executed operation.

Bypassing and Dead-Result Elimination. The temporary bypass moves generated from the bypass candidate moves are alternative to each other. The bypass move inherits the dependencies from the source move, and each move that is dependent of the bypassed move is dependent of the bypass move as well, except for the possible register antidependencies that do not exist in the bypassed case. Fig. 2 presents a data dependence graph with an added bypass move, highlighted by the dashed line. The bypass source `ALU.out → z` and bypass candidate move `z → ALU.t.SUB` are emphasized with the dashed-dotted poly-line. The bypass move contains the incoming edges of the bypass source `ALU.out → z` and, respectively, the outgoing edges of the bypass candidate `z → ALU.t.SUB`.

Dead-result elimination allows the removal of the result move in case a move is bypassed to all the consumers of the result. For example, in the situation presented in Fig. 2, the result move `ALU.out → z` can be eliminated as the only outgoing edge is to the bypassed move `z → ALU.t.SUB`.

In summary, bypassing together with dead-result elimination results in two possible cases: (a) the result move only has a single outgoing edge. In this case either the bypass move (bypass and eliminate), or the bypass candidate and the result move shall be assigned (no bypassing nor elimination) and (b) the result move has multiple out-

going edges. In the latter case, the result move is assigned and the bypass candidate and the bypass move are alternative to each other. In case all the bypass moves are assigned, the result move can be eliminated post solving. This case leads to latency reduction when moves are bypassed, and energy savings in case the result move can be eliminated. This slightly limits the scheduling freedom, but makes the model more understandable and less demanding in terms of solving complexity.

The former case results in a constraint

$$\sum_{c \in C_{\text{candidate}}} \sum_{t=t_{min}}^{t_{max}} M_{\text{candidate},c,t} + \frac{1}{2}\left(\sum_{c \in C_{\text{bypass}}} \sum_{t=t_{min}}^{t_{max}} M_{\text{bypass},c,t} + \sum_{c \in C_{\text{result}}} \sum_{t=t_{min}}^{t_{max}} M_{\text{result},c,t} \right) = 1, \quad (3)$$

where $M_{\text{candidate}}$, M_{bypass} and M_{result} are the bypass candidate, the bypass move, and the result move, respectively. The latter condition requires two constraints. The bypass candidate and the bypass move must be alternative to each other. Thus we require that

$$\sum_{c \in C_{\text{candidate}}} \sum_{t=t_{min}}^{t_{max}} M_{\text{candidate},c,t} + \sum_{c \in C_{\text{bypass}}} \sum_{t=t_{min}}^{t_{max}} M_{\text{bypass},c,t} = 1. \quad (4)$$

Also the result moves have to be assigned yielding

$$\sum_{c \in C_i} \sum_{t=t_{min}}^{t_{max}} M_{\text{result},c,t} = 1. \quad (5)$$

Register File Port Constraints. Register files have a number of registers which can be read through output ports. Even if an output port was connected to multiple buses, only one register can be read through a single port at the same cycle. In other words, the number of output ports bounds the number of concurrent register reads from a register file.

Similarly, each input port can write into a single register at a time. To limit the outbound moves at each cycle to one for each port we require that

$$\sum_{M_i \in P} \sum_{c \in C_i^{\text{rf}}} M_{i,c,t} = 1, \text{ for } t \in [t_{min}, t_{max}], \text{rf} \in R, \quad (6)$$

where R is a set of all RF ports on the given TTA processor, and C^{rf} is a set of all connections that originate from a register file port rf.

Function Unit Constraints. Operations executed at function units consist of multiple operands, of which one is a triggering operand that starts the operation execution. All other operands shall be written to FU input ports at any cycle before or at the same time as the triggering operand. Trigger move constrains the latest cycle on which other operands must be written. There might be multiple function units that can execute each operation, and each operand might have multiple connections to an FU. Therefore,

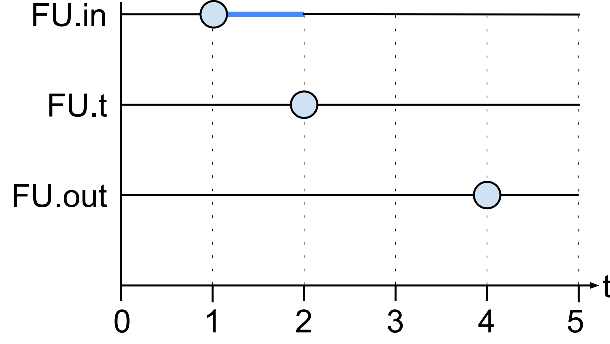

Fig. 3: Function unit port reservations until the triggering operand is written. The first operand, `FU.in`, reserves the corresponding input port until the execution is triggered to start from port `FU.t` (illustrated with the blue line).

$$\sum_{c \in C^{\mathrm{f}}_{\text{trigger}}} M_{\text{trigger},c,t} - \frac{1}{|O|} \left(\sum_{M_i \in O} \sum_{c \in C^{\mathrm{f}}_i} \sum_{t'=t-e}^{t} M_{i,c,t'} \right) \leq 0, \quad \text{for } t \in [t_{min}, t_{max}], \forall \mathrm{f} \in \mathbf{FU}, \tag{7}$$

where FU is a set of all function units that can execute the considered operation, C^f is a set of all connections that can transport given move to an appropriate port on f, M_{trigger} is the trigger move, O is a set of operand moves that relate to the triggering move M_{trigger} on the FU of the port f. The variable e is so called *operand slack limit*, a model parameter that restricts the operand move transport to occur at most e cycles before the triggering move to limit the model size.

No other operand moves must be written to operand ports between cycles $[t', t]$, where t and t' are cycles in which the triggering move and the operand move is being written, respectively. In other words, an operand port is reserved to the program operation until the triggered execution starts after the trigger operand is written. This is illustrated in Fig. 3 where an operation with two operands is being transferred to an FU. The port reservation status after an operand write is illustrated with blue line, which spans until the triggering port $FU.t$ is written into. This yields a constraint

$$\sum_{c \in C^{\mathrm{f}}_{\text{trigger}}} M_{\text{trigger},c,t} + \sum_{c \in C^{\mathrm{f}}_{\mathrm{o}}} M_{\mathrm{o},c,t'} + \frac{1}{|K|} \left(\sum_{M_i \in K} \sum_{c \in C^{\mathrm{f}}_i} \sum_{t''=t'}^{t} M_{i,c,t'} \right) \leq 2, \quad \text{for } t \in [t_{min}, t_{max}], \forall t' \in [t-e, t], \forall \mathrm{f} \in \mathbf{FU}, \forall M_o \in O, \tag{8}$$

where M_o is the current operand node, and K is a set of other operand moves that might be assigned to same port as M_o. By setting the left-hand side equal to two, both the triggering move and the current operand move can be equal to one, and all the other operand moves must be zero.

The model assumes that the results of a program operation are available for reading at the output ports after an operation latency has passed counting from the time instance the triggering move has been scheduled until the time a new triggered opera-

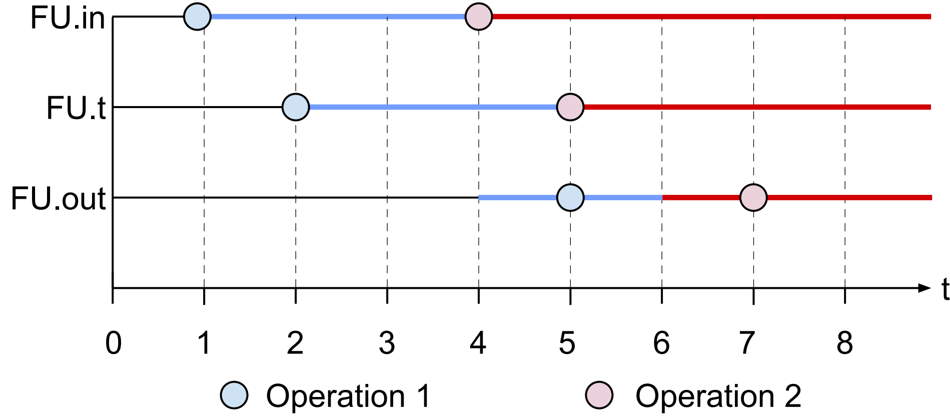

Fig. 4: An example of legal schedule of two operations to a single FU. The trigger cycle of Operation 1 determines when its results are available. Similarly, the cycle of its last result read move determines when the execution of the Operation 2 can be started. The colored lines illustrate the possible schedules for the different moves.

tion overwrites the result. This adds a constraint that all result reads of an operation must be read some time after the results are ready, and no other program operation shall overwrite the result before all the result reads have been scheduled. However, at the same time, pipelined execution of multiple program operations in a same FU must be supported for maximum operation throughput.

As an example, Fig. 4 illustrates the port reservation for a legal schedule of moves from two separate program operations in an FU where `FU.t` is the triggering port and both operations have a latency of two cycles. The first operation result is available at $t = 4$, that is, after the operation latency has passed. Since the result of the first operation is read only at $t = 5$, the execution of operation 2 shall not start before $t = 4$.

In order to enforce all result moves to occur at correct interval,

$$\sum_{c \in C^{\mathrm{f}}_{\mathrm{trigger}}} M_{\mathrm{trigger},c,t} - \frac{1}{|R|} \sum_{M_i \in R} \sum_{c \in C^{\mathrm{f}}_i} \sum_{t'=t+l}^{t+l+y} M_{i,c,t'} \leq 0, \text{ for } t \in [t_{min}, t_{max}], \forall \mathrm{f} \in \mathbf{FU}, \quad (9)$$

where R is a set of all results reads of the program operation that M_{trigger} triggers, l is the latency of the operation, and y is *result read slack*, a maximum cycle count that any result can be stored in an output port.

In addition, other triggering moves from all other program operations must be prevented to overwrite the result too early

$$\sum_{c \in C^{\mathrm{f}}_{\mathrm{trigger}}} M_{\mathrm{trigger},c,t} + \sum_{c \in C^{\mathrm{f}}_{\mathrm{r}}} M_{\mathrm{r},c,t'} + \frac{1}{|T|} \left(\sum_{M_i \in T} \sum_{c \in C^{\mathrm{f}}_i} \sum_{t''=t'-d}^{t'} M_{i,c,t'} \right) \leq 2,$$
$$\forall t \in [t_{min}, t_{max}], \text{ for } t' \in [t-e, t], \forall \mathrm{f} \in \mathbf{FU}, \forall M_r \in R, \quad (10)$$

where T is a set of other trigger moves to function unit f besides M_{trigger}, d is the latency of the operation of $M_{i,c,t'}$, and R is a set of result moves of M_{trigger}. The reasoning behind the constraint is identical to that of Equation 8.

No proof of completeness is provided at this time, but the correct results were validated using simulations and static resource constraint checking.

3.4. The Objective Function

The objective function of the ILP model can be varied depending on the intended requirements the application, the processors, and the software running on it implement. In this article, we focus on minimizing the execution time of the compiled program while other interesting goals could be, e.g., to minimize the energy consumption of the program while meeting a maximum execution time constraint.

Minimizing the execution time of the schedule can be formulated as minimizing the latest execution time of all moves of the scheduled program. This can be expressed in the model as an objective function that attempts placing the moves in leaf nodes of the DDG as early as possible. This implies that all the other nodes of the graph are pushed to be scheduled earlier as they all are descendants of the leaf nodes.

The objective function to minimize the schedule length is

$$\min : \sum_{M_i \in L} \sum_{c \in C^b} \sum_{t=t_{min}}^{t_{max}} t \cdot M_{i,c,t}, \tag{11}$$

where L is a set of moves in the leaf nodes of the DDG. It should be noted that the model must minimize the issue time of all the leaf nodes and not just the ones on the critical path, because otherwise there might be "outlier leaf nodes" that "stretch" the schedule and increase the cycle count of the basic block.

4. EVALUATION

The integer linear programming model for the TTA scheduling problem was integrated in the compiler of a design and programming tool, *TTA-based Co-design Environment (TCE)* [Esko et al. 2010].

TCE's compiler utilizes LLVM [Lattner and Adve 2004] and Clang version 3.6 for language support, intermediate representation optimizations, and parts of the code generation. Before the code is passed to the ILP-based scheduler, the standard level three optimizations of LLVM are executed on the intermediate representation. In addition,the registers were allocated and operations were selected using default LLVM code generation passes.

Compilation was benchmarked on a server that has a 6-core (12 hardware threads) Intel i7-3930K CPU with a clock speed of 3.2 GHz, and 16 GB of RAM. The models were optimized with *Solving Constraint Integer Programs (SCIP)* mixed integer programming solver, which is one of the fastest non-commercial solvers available [Achterberg et al. 2008]. All basic blocks of the test programs were scheduled using the ILP-based scheduler. The operand slack and the result read slack limits were both set to 5 for 5 for the experiments to decrease the computational complexity of the models. We also run the most complex architecture and programs combinations with higher values, but that did not result in improvements.

The performances of the produced schedules were compared to the heuristic operation-based list scheduler that tcecc uses by default. In addition to the cycle count, for each input program, the number of register accesses was inspected to evaluate the efficiency of applying the TTA-based optimizations that have the capability to reduce register file accesses.

All the execution statistics of the compiled programs were obtained using the instruction cycle accurate architecture simulator bundled with TCE. In order to assess the compile time of a known slower ILP-based scheduler, we also compared the CPU

time and the wall clock time spent during the compilation to measure the practical compilation time. The GNU time utility (version 1.7) was used for these measurements.

We used five benchmark programs from the DSPstone suite [V. Zivojnovic and Meyr 1994]: *Infinite Impulse Response (IIR)* filter, *Finite Impulse Response (FIR)* filter, dot product, convolution, and complex number arithmetics. These workloads are typical kernels in digital signal processing applications and contain a reasonable amount of instruction-level paralellism while being small enough to be feasible for optimal ILP-based scheduling.

The following subsections present the results for two architectures with a very different number of architectural resources.

4.1. Minimalist Architecture

The Minimalist architecture is a machine resembling a simple RISC core in its resources. It consists of a register files (RF), a predicate register file (BOOL), arithmetic-logical unit (ALU), load-store unit (LSU), control unit (CU), and multiplication unit (MUL). There are three buses with aggressively reduced connectivity while it still provides sufficient connections to execute multiple operations concurrently. The ALU can transport results back to its input ports, enabling bypassing between arithmetic operations. In addition, it is possible to bypass results from the MUL unit to the ALU, and the other way around. LSU results and inputs can be bypassed directly from both the ALU unit and the MUL unit.

The architecture is shown in Figure 5.

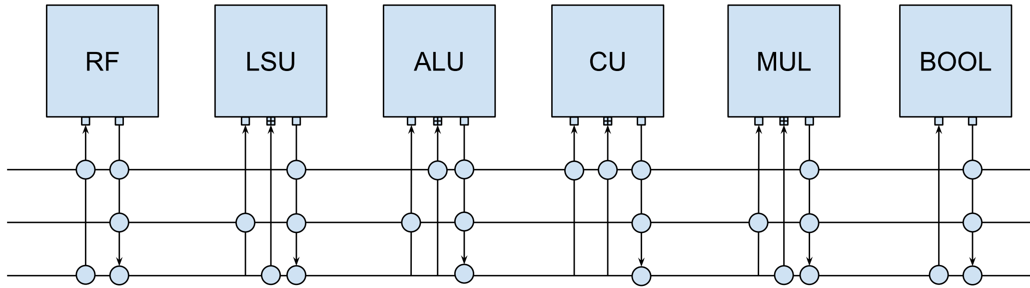

Fig. 5: Minimalist architecture with two register files, RF and BOOL, an ALU, a LSU, a CU, and a function unit MUL providing the multiplication operation.

The ILP-scheduler is able to reduce the cycle count into 52% of the cycles compared to the heuristic scheduler in the best case, and 97.5% at the worst case. The average cycle count equals 70.0% of the cycle count of the heuristic scheduler. The relative cycle counts for all benchmarks are shown in Fig. 6.

The register accesses of the ILP-scheduler are presented in Fig. 7. The reduction of register reads is still apparent although the objective was only to minimize the cycle count. In some cases (the IIR filter and the dot product calculation), the ILP-scheduler was able to eliminate all the result reads. The schedule produced for the convolution program had slightly more register reads than the heuristic scheduler.

On average, the ILP-scheduler performed 64.7% less register file reads compared to the schedule of the heuristic scheduler. In addition, the ILP-scheduler was able to eliminate 25.1% of the register writes on average. In the case of the convolution program, the amount of register writes was equal to that of the heuristic scheduler.

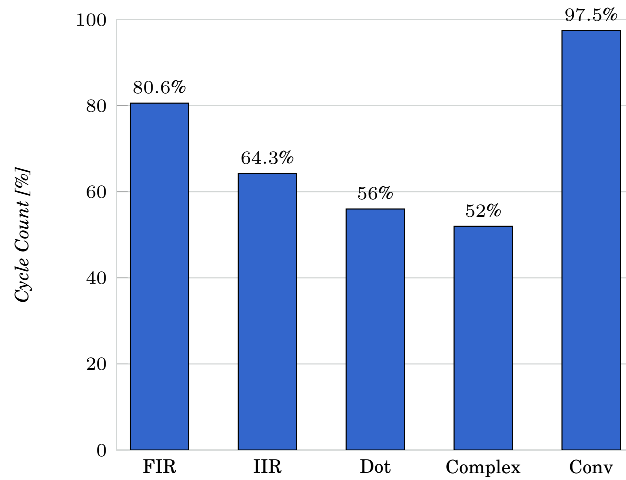

Fig. 6: Minimalist architecture: ILP-scheduler cycle counts relative to that of the heuristic scheduler. The percentages are obtained by dividing the cycle count of the integer linear programming scheduler by the cycle count of the heuristic scheduler.

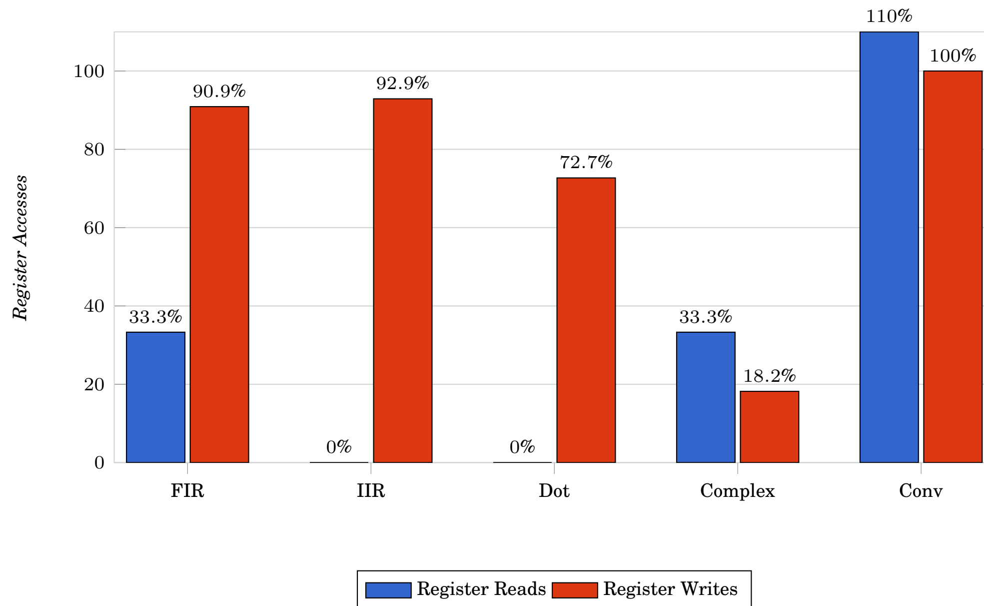

Fig. 7: Minimalist architecture: The ratio of register reads and register writes between the ILP-scheduler and the heuristic scheduler.

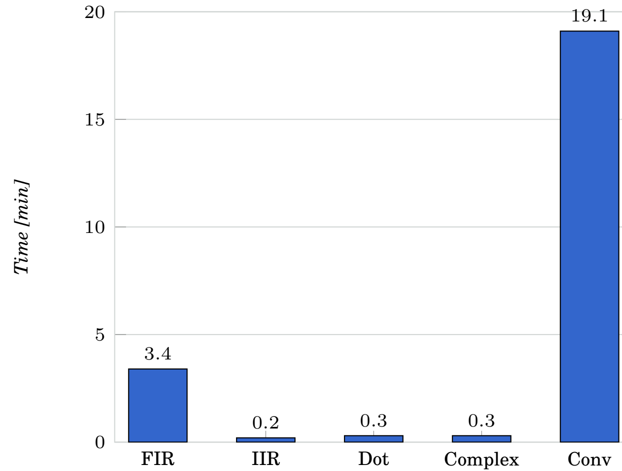

Fig. 8: Minimalist arhitecture: Compilation times for the ILP-scheduler. The heuristic scheduler used less than one second to compile any of the programs.

	Integer Linear Programming Scheduler						
Program	**Cycle Count**		**Register Reads**		**Register Writes**		**Compilation Time [min]**
FIR filter	25	(80.6%)	5	(33.3%)	10	(90.9%)	3.4
IIR filter	18	(64.3%)	0	(0.0%)	13	(92.9%)	0.2
Dot product	14	(56.0%)	0	(0.0%)	8	(72.7%)	0.3
Complex	13	(52.0%)	3	(33.3%)	2	(18.2%)	0.3
Convolution	78	(97.5%)	33	(110%)	35	(100.0%)	19.1

Table I: Simulation results for the minimal architecture with the integer linear programming scheduler for the benchmark programs. In the integer linear programming columns, the percentage in the parentheses shows the difference with respect to the corresponding value of the heuristic scheduler.

Fig. 8 presents the compilation times. The heuristic scheduler was able to schedule all programs in one second or less. It is unsurprising that the ILP-scheduler takes distinctly more time to schedule the programs, 4.7 minutes on average and 19.1 minutes in the worst case.

Table I presents the simulation results for the minimal architecture for the benchmark programs. In addition, Table II introduces the corresponding integer linear programming model characteristics.

4.2. Clustered Architecture

Clustered architecture is a clustered VLIW-like architecture divided into separate increased connectivity clusters that are interconnected with a sparse connection network. In this machine, three clusters are formed of register file and ALU pairs. In addition, the architecture provides a single MUL unit. Although its interconnection network with 17 buses is quite reduced, the architecture provides a high level of concurrency for the operations.

The architecture is illustrated in Figure 9.

	Integer Linear Programming Scheduler			
Program	**Constraints**	**Variables**	**Basic Blocks**	**DDG Nodes**
FIR filter	398120	142478	22	813
IIR filter	142478	13424	13	187
Dot product	10496	2070	3	37
Complex	13333	2520	3	44
Convolution	20705	4530	3	76

Table II: Integer linear programming model sizes for the minimal architecture.

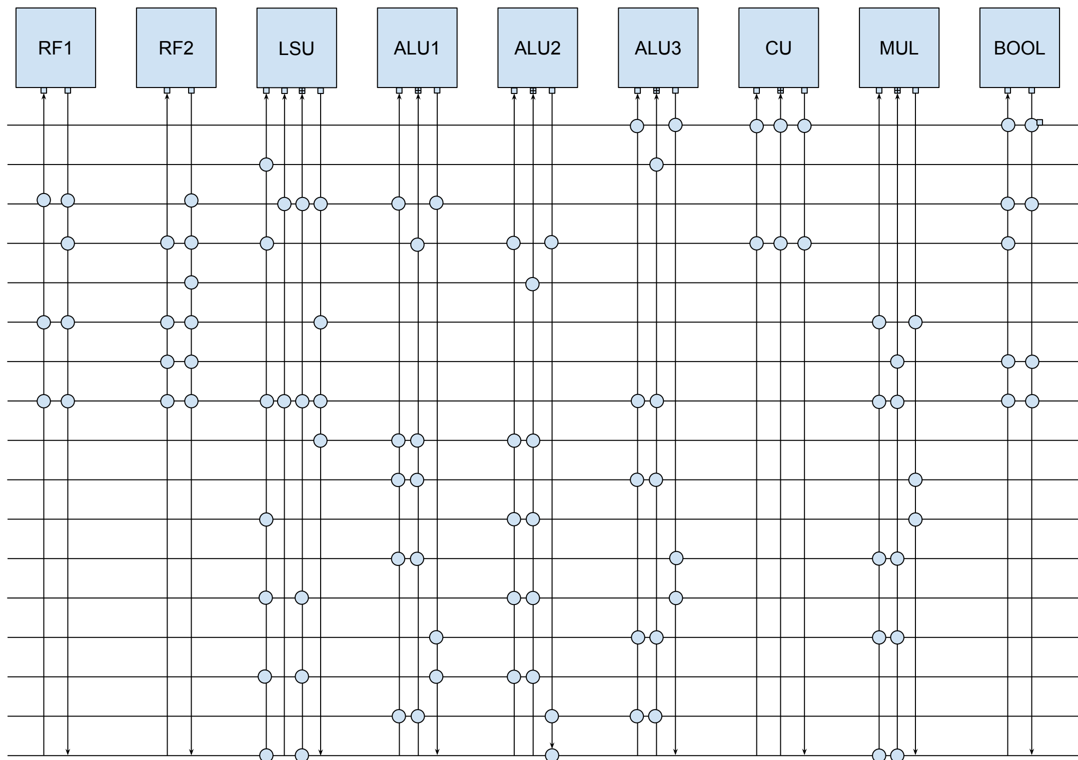

Fig. 9: Clusted architecture consisting of three separate computing clusters, a multiplication unit, a CU, a LSU, and a boolean register file.

The cycle counts with the ILP-scheduler are presented in Fig. 10. On average, the ILP-scheduler decreased the execution time by 19.6% on average. The most notable reduction was in the case of the complex number arithmetics program, in which the number of cycles was 43.7% less than that of the heuristic scheduler.

Register accesses of the programs scheduled with the ILP-scheduler with the clustered architecture are illustrated in Fig. 11. The schedules produced with the ILP-scheduler executed 45.7% less register reads compared to the heuristic scheduler. In the case of the IIR filter, the ILP-scheduler was able to eliminate all register reads.

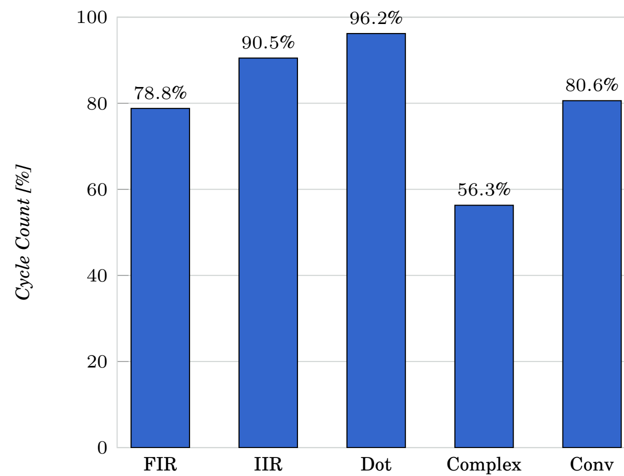

Fig. 10: Clustered architecture: ILP-scheduler cycle counts relative to that of the heuristic scheduler. The percentages are obtained by dividing the cycle count of the ILP-scheduler by the cycle count of the heuristic scheduler.

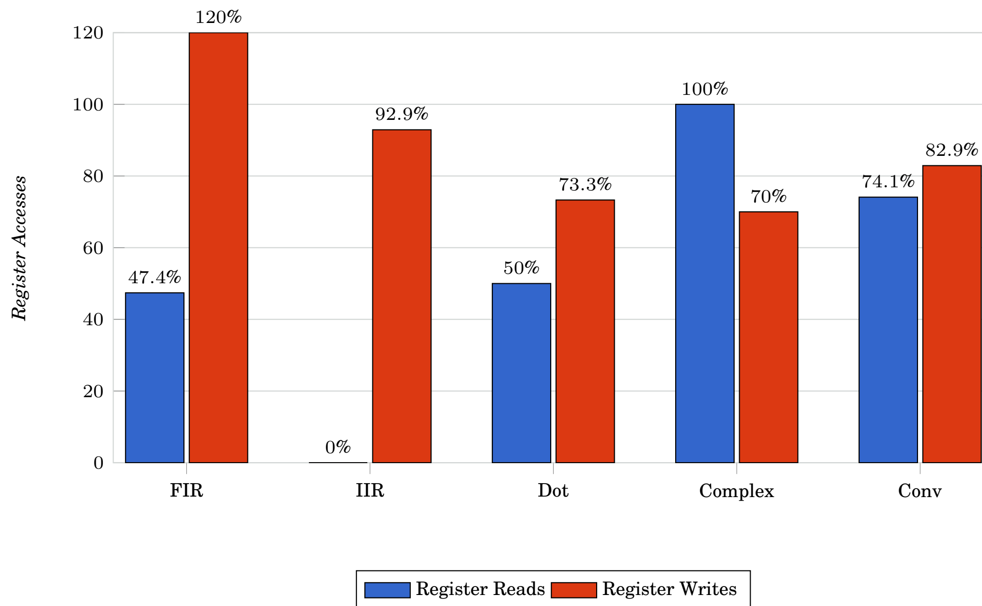

Fig. 11: Clustered architecture: The ratio of register writes and register reads between the schedules produced by the ILP-scheduler and the heuristic scheduler.

Furthermore, there were on average 12.2% less register writes compared to the heuristic scheduler.

Fig. 12 shows the compilation times for the ILP-scheduler with the Clustered architecture. The execution times are even greater than those of the minimalist architec-

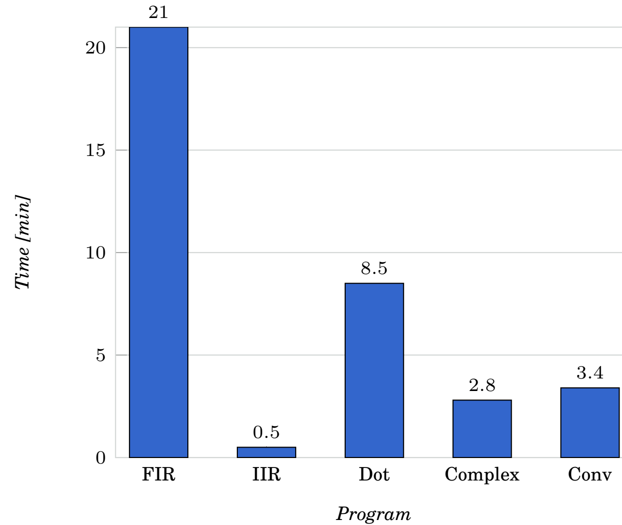

Fig. 12: Execution times for the integer linear programming instruction scheduler.

Program	Integer Linear Programming Scheduler						
	Cycle Count		Register Reads		Register Writes		Compilation Time [min]
FIR filter	26	(78.8%)	9	(47.4%)	12	(120.0%)	21.0
IIR filter	19	(90.5%)	0	(0.0%)	13	(92.9%)	0.5
Dot product	25	(96.2%)	7	(50.0%)	11	(73.3%)	8.5
Complex	9	(56.3%)	5	(100.0%)	7	(70.0%)	2.8
Convolution	29	(80.6%)	20	(74.1%)	29	(82.9%)	3.4

Table III: Simulation results for the clustered architecture with the integer linear programming scheduler for the benchmark programs. In the integer linear programming columns, the percentage in the parentheses shows the difference with respect to the corresponding value of the heuristic scheduler.

ture. This is due to the fact that the clustered architecture is much larger and provides much more opportunities for data transports between the units, thus presents more degrees of freedom and options to the scheduler. However, the growth of compilation time is much less than the increase in the number of resources when compared to the Minimalist architecture. The average compilation time for the clustered architecture is 7.2 minutes. In the worst case, the proposed scheduler uses 21 minutes to produce a schedule for the FIR filter program.

Table III presents the simulation results for the clustered architecture. and Table IV contains the corresponding integer linear programming model characteristics.

5. RELATED WORK

Integer programming and the branch-and-bound solver algorithm has been used extensively in the past for producing optimal solutions for various optimization problems. A general solution for the scheduling problem was proposed in the context of operations research [Greenberg 1968]. This paper presents a generic mixed integer formulation

	Integer Linear Programming Scheduler			
Program	**Constraints**	**Variables**	**Basic Blocks**	**DDG Nodes**
FIR filter	398453	147428	22	820
IIR filter	40458	17024	13	187
Dot product	28424	8646	3	37
Complex	33844	4368	3	62
Convolution	34070	11910	3	76

Table IV: Integer linear programming model sizes for the clustered architecture.

for scheduling n jobs to m machines with objective functions that minimize either the total time to process the jobs or the machine idle time.

Integer programming was for the first time used to optimize machine code in [Arya 1985], which considered a class of register-based vector processors (with Cray-1 used as an example), and provided separate formulations for both acyclic code and loops.

Integer linear programming based simultaneous instruction scheduling and register allocation for multi-issue machines is proposed in [Chang et al. 1997]. In this work, two separate solutions were proposed; one that can handle spill code, and another one assuming that no spill code is needed. The former is much more complex and should be used only in case assuming the available registers are not sufficient for the program's live variables. Their solution makes several simplifying assumptions such as assuming that live ranges of variables do not span across multiple basic blocks and that the scheduled instructions always take only one cycle to execute. In our model, we make no such assumptions in order to make the model generally usable for different programs and targets with multiple cycle function units. Especially important for performance is our model's capability to handle pipelined execution of operations in function units.

Similarly, [Kästner and Langenbach 1999] propose utilizing ILP for integrated scheduling and register allocation. This work targets an irregular DSP target and also does approximations to produce good enough schedules in faster compilation time. The compilation time problem of ILP-based instruction scheduling is addressed also in [Wilken et al. 2000] which presents model size reductions by means of data dependence graph simplifications. These techniques are generic enough to be adapted in our model to reduce the compilation time.

Work in [Kästner and Winkel 2001] targets IA-64, an architecture with *instruction bundles* that attempt to reduce the NOP waste of VLIW architectures. Due to the bundle selection affecting the size of the resulting instruction stream, this paper attempts to optimize the program bit size as a secondary goal to the schedule length. The two problems of scheduling and bundling are done in separate stages, thus this does not always lead to optimal solutions, but allows the compilation to complete in much shorter time. Our model takes into account the *instruction templates* (which describe the NOP slots, *move* slots, and *immediate pieces* in the instruction word) during the scheduling process as one optimized parameter. This enables including the bit size minimization as an optimization goal in case of variable length instruction targets.

In addition to combining register allocation and instruction scheduling to the same ILP formulation, [Bednarski and Kessler 2006] integrate also the instruction selection phase for implementing all the major phases of VLIW code generation using the same mathematical model. The model is extended to support clustered VLIW architectures in [Eriksson et al. 2008] and modulo scheduling in [Eriksson and Kessler 2009; 2012]. These present one of the most complete published solutions to clustered VLIW code generation using ILP.

Constraint Programming (CP) is another combinatorial approach with a higher abstraction level than integer programming. In comparison to ILP, it appears to be a "cleaner" approach as the models do not need to be formulated as linear equations and inequalities that might not be natural for the problem domain. A major difference to linear programming is that CP is meant for producing *feasible* solutions that satisfy a set of constraints whereas integer programming gives one or more *optimal* solutions which are known to produce the minimal value for a desired objective function directly. CP has been used also for implementing code generators. A notable recent work is Unison [Lozano et al. 2012; Lozano et al. 2013; Lozano 2014], a retargetable CP-based code generator that combines register allocation and instruction scheduling. Formulating the TTA scheduling problem as a CP problem is an interesting alternative which we plan to study more in the future.

The closest published work to ours is [Guo et al. 2006]. It proposes an ILP-based compiler backend for *Synchronous Transfer Architectures (STA)*. The STA approach is a simplification of the TTA we target in our work [Cichon et al. 2004]. Interestingly, STA also supports software bypassing (authors refer to it as *direct data routing*) which is included in their ILP model. However, STA is not transport programmed, but is more like a standard VLIW with exposed function unit ports. It has a separate opcode field in the instruction word (like standard VLIWs) to start the operation execution for each function unit. Thus, it does not present operand/result scheduling freedom to the compiler, simplifying the scheduling and the instruction word, but missing the register file pressure reduction available from freely scheduled operand data transports [Jääskeläinen et al. 2015]. In addition, their ILP model limits *bypass distance* to be equal to zero. In other words, direct data routing is done only in case the result of an operation is ready on the exact cycle it is needed. This reduces the impact of bypassing as is shown in [Guzma et al. 2008].

Based on our related work search, the software controlled datapath interconnection network and the related unique software optimizations in the TTA programming model bring additional scheduling challenges that are not published in earlier work. The model we presented in this article is novel in its capability of utilizing reduced interconnection network TTA machines efficiently when starting from higher-level language software descriptions such as C, which is challenging for heuristic approaches.

6. FUTURE WORK

The sequential program input to the scheduling phase in the used compiler framework has some of the code generation phases done in order to exploit generic register allocators and instruction selectors, and to limit the model size. Clearly, the best quality code would be generated when all the decisions of the code generation, that is, instruction selection, register allocation/partitioning, and instruction scheduling/bundling were handled simultaneously by the model. However, our experience indicates that the additional antidependency removal benefits from integrated register allocation are probably not very high due to the model already using dead result elimination and software bypassing which eliminate some of the antidependencies resulting from register reuse. On the other hand, integrating the variable to register file assignment (operation/variable partitioning) is expected to bring more code quality benefits with clustered machines.

When using a code generator based on a mathematical model, one has to be very careful when expanding the controlled parts as the compile time can explode very easily, making the scheduler usable only for ever smaller pieces of code. Further study of integrating the different phases of a TTA code generator in the model, their compile time impact and the possible schedule quality maintaining model simplifications (such as those proposed in [Wilken et al. 2000]) are left as a future work. As a higher priority,

we plan to integrate modulo scheduling to the model as data oriented processors such as TTAs greatly benefit from software pipelining. In addition, we are experimenting with alternative objective functions that better exploit the energy saving optimizations of the TTA programming model.

7. CONCLUSIONS

Static processor architectures, especially so called exposed datapath architectures such as TTA are notoriously difficult compiler targets. Code generators based on heuristics produce schedules reasonably fast, but usually lead to suboptimal results and resource underutilization.

In this paper we proposed an integer linear programming based scheduler for TTA processors. The scheduler can exploit the important TTA-specific software optimizations efficiently and leads to much better quality schedules. The model was verified by simulating the produced code and benchmarked against a heuristic scheduler which showed performance improvements. In the best case, the cycle count was reduced to 52% of the one with the heuristic scheduler. Dramatic reductions were measured also in register file accesses: At best, the ILP-scheduler reduced the number of register file reads to 33% of the heuristic results and register file writes at best to 18%.

The longer compilation time of the ILP-scheduler (up to 21 min in the presented benchmarks) is tolerable when producing the final code for production, but a heuristical scheduler is more usable in the iterative “compiler-in-the-loop” design space exploration where hundreds of architecture variations are compiled for to find the best alternative. Interestingly, we found that the architectures which are more challenging to the heuristic scheduler are often nicer targets to the ILP-scheduler due to the lower degree of freedom. For example, TTAs with highly reduced connectivity network also have reduced model size, making ILP-based scheduling especially interesting for aggressively optimized production architectures.

REFERENCES

T. Achterberg, T. Berthold, T. Koch, and K. Wolter. 2008. Constraint Integer Programming: A New Approach to Integrate CP and MIP. In *Proc. Int. Conf. Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Paris, France, 6–20.

S. Arya. 1985. An Optimal Instruction-Scheduling Model for a Class of Vector Processors. *IEEE Trans. Comput.* C-34, 11 (Nov 1985), 981–995. DOI:http://dx.doi.org/10.1109/TC.1985.1676531

A. Bednarski and C. Kessler. 2006. Optimal Integrated VLIW Code Generation with Integer Linear Programming. In *Euro-Par 2006 Parallel Processing*, W.E.. Nagel, W.V. Walter, and W. Lehner (Eds.). Lecture Notes in Computer Science, Vol. 4128. Springer, Berlin, Germany, 461–472. DOI:http://dx.doi.org/10.1007/11823285_48

C.-M. Chang, C.-M. Chen, and C.-T. King. 1997. Using Integer Linear Programming for Instruction Scheduling and Register Allocation in Multi-Issue Processors. *Computers & Mathematics with Applications* 34, 9 (1997), 1–14. DOI:http://dx.doi.org/10.1016/S0898-1221(97)00184-3

G. Cichon, P. Robelly, H. Seidel, E. Matús, M. Bronzel, and G. Fettweis. 2004. Synchronous Transfer Architecture (STA). In *Computer Systems: Architectures Modeling and Simulation*, A. D. Pimentel and S. Vassiliadis (Eds.). Lecture Notes in Computer Science, Vol. 3133. Springer-Verlag, Berlin, Germany, 343–352. DOI:http://dx.doi.org/10.1007/978-3-540-27776-7_36

A. Cilio, H. Schot, and J. Janssen. 2006. Architecture Definition File: Processor Architecture Definition File Format for a New TTA Design Framework. Tampere University of Technology, Finland. (2006). http://tce.cs.tut.fi/specs/ADF.pdf

H. Corporaal. 1995. *Transport Triggered Architectures, Design and Evaluation*. Ph.D. Dissertation. TU Delft, Netherlands.

H. Corporaal and J. Hoogerbrugge. 1995. Code Generation for Transport Triggered Architectures. In *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens (Eds.). Springer, Heidelberg, Germany, 240–259. DOI:http://dx.doi.org/10.1007/978-1-4615-2323-9_14

M.V. Eriksson and C.W. Kessler. 2009. Integrated Modulo Scheduling for Clustered VLIW Architectures. In *High Performance Embedded Architectures and Compilers*. Lecture Notes in Computer Science, Vol. 5409. Springer, Berlin, Germany, 65–79. DOI:http://dx.doi.org/10.1007/978-3-540-92990-1_7

M. Eriksson and C. Kessler. 2012. Integrated Code Generation for Loops. *ACM Trans. Embed. Comput. Syst.* 11S, 1 (June 2012), 19:1–19:24. DOI:http://dx.doi.org/10.1145/2180887.2180896

M.V. Eriksson, O. Skoog, and C.W. Kessler. 2008. Optimal vs. Heuristic Integrated Code Generation for Clustered VLIW Architectures. In *Proc. Int. Workshop on Software Compilers for Embedded Systems*. Munich, Germany, 11–20.

O. Esko, P. Jääskeläinen, P. Huerta, C.S. de La Lama, J.Takala, and J. Martinez. 2010. Customized Exposed Datapath Soft-Core Design Flow with Compiler Support. In *Proc. Int. Conf. Field Programmable Logic and Applications*. Milan, Italy, 217–222. DOI:http://dx.doi.org/10.1109/FPL.2010.51

J.A. Fisher. 1983. Very Long Instruction Word Architectures and the ELI-512. In *Proc. Int. Symp. Comput. Arch.* Los Alamitos, CA, 140–150. DOI:http://dx.doi.org/10.1145/356757.356761

H.H. Greenberg. 1968. A Branch-Bound Solution to the General Scheduling Problem. *Operations Research* 16, 2 (1968), 353–361. DOI:http://dx.doi.org/10.1287/opre.16.2.353

J. Guo, T. Limberg, E. Matus, B. Mennenga, R. Klemm, and G. Fettweis. 2006. Code Generation for STA Architecture. In *Euro-Par 2006 Parallel Processing*, W.E. Nagel, W.V. Walter, and W. Lehner (Eds.). Lecture Notes in Computer Science, Vol. 4128. Springer, Berlin, Germany, 299–310. DOI:http://dx.doi.org/10.1007/11823285_31

V. Guzma, P. Jääskeläinen, P. Kellomäki, and J. Takala. 2008. Impact of Software Bypassing on Instruction Level Parallelism and Register File Traffic. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, M. Berekovic, N. Dimopoulos, and S. Wong (Eds.). Lecture Notes in Computer Science, Vol. 5114. Springer, Berlin, Germany, 23–32. DOI:http://dx.doi.org/10.1007/978-3-540-70550-5_4

J. Hoogerbrugge and H. Corporaal. 1994. Register File Port Requirements of Transport Triggered Architectures. In *Proc. Ann. Int. Symp. Microarchitecture*. San Jose, CA, USA, 191–195. DOI:http://dx.doi.org/10.1109/MICRO.1994.717458

P. Jääskeläinen, H. Kultala, T. Viitanen, and J. Takala. 2015. Code Density and Energy Efficiency of Exposed Datapath Architectures. *J. Signal Process. Syst.* 80, 1 (2015), 49–64. DOI:http://dx.doi.org/10.1007/s11265-014-0924-x

H.T. Jongen, K. Meer, and E. Triesch. 2004. *Optimization Theory*. Kluwer, Norwell, MA, USA.

M. Jünger, T. Liebling, D. Naddef, G. Nemhauser, W. Pulleyblank, G. Reinelt, G. Rinaldi, and L.A. Wolsey (Eds.). 2010. *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Springer, Heidelberg.

R.M. Karp. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, R.E. Miller, J.W. Thatcher, and J.D. Bohlinger (Eds.). Springer, US, 85–103. DOI:http://dx.doi.org/10.1007/978-1-4684-2001-2_9

D. Kästner and M. Langenbach. 1999. Code Optimization by Integer Linear Programming. In *Compiler Construction*, S. Jähnichen (Ed.). Lecture Notes in Computer Science, Vol. 1575. Springer, Berlin, Germany, 122–136. DOI:http://dx.doi.org/10.1007/978-3-540-49051-7_9

D. Kästner and S. Winkel. 2001. ILP-based Instruction Scheduling for IA-64. In *Proc. ACM SIGPLAN Workshop Languages Compilers Tools for Embedded Syst.* Snow Bird, UT, USA, 145–154. DOI:http://dx.doi.org/10.1145/384197.384217

KDPOF. 2014. *KD1000 Family Gigabit Ethernet POF Transceivers*. product brochure. KDPOF. http://www.kdpof.com/wp-content/uploads/2012/10/br001-kd1000_Family_Brochure-v1_10.pdf

A. H. Land and A. G. Doig. 1960. An Automatic Method of Solving Discrete Programming Problems. *Econometrica* 28, 3 (1960), 497–520.

C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation Optimization*. 75–87.

R.C. Lozano. 2014. Integrated Register Allocation and Instruction Scheduling with Constraint Programming. Licentiate thesis. KTH Royal Institute of Technology. (2014).

R.C. Lozano, G.H. Blindell, M. Carlsson, F. Drejhammar, and C. Schulte. 2013. Constraint-based Code Generation. In *Proc. 16th Int. Workshop on Software and Compilers for Embedded Systems (M-SCOPES '13)*. ACM, New York, NY, USA, 93–95. DOI:http://dx.doi.org/10.1145/2463596.2486155

R.C. Lozano, M. Carlsson, F. Drejhammar, and C. Schulte. 2012. Constraint-Based Register Allocation and Instruction Scheduling. In *Principles and Practice of Constraint Programming*, Michela Milano (Ed.). Springer Berlin Heidelberg, 750–766. DOI:http://dx.doi.org/10.1007/978-3-642-33558-7_54

A.K. Mackworth. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8, 1 (1977), 99–118. DOI:http://dx.doi.org/10.1016/0004-3702(77)90007-8

Maxim Integrated Products, Inc. 2004. *Introduction to the MAXQ Architecture*. Maxim Integrated Products, Inc. Application Note 3222.

F. Rossi, P. van Beek, and T. Walsh. 2006. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA.

C. Schläger V. Zivojnovic, J. Martinez and Heinrich Meyr. 1994. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proc. of ICSPAT'94 - Dallas*.

K. Wilken, J. Liu, and M. Heffernan. 2000. Optimal Instruction Scheduling Using Integer Programming. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*. Vancouver, BC, Canada, 121–133. DOI:http://dx.doi.org/10.1145/349299.349318