Generic Software Pipelining at the Assembly Level

Markus Pister
Saarland University & AbsInt GmbH
Saarbrücken, Germany
pister@cs.uni-sb.de

Daniel Kästner
AbsInt GmbH
Saarbrücken, Germany
kaestner@absint.com

ABSTRACT

Software used in embedded systems is subject to strict timing and space constraints. The growing software complexity creates an urgent need for fast program execution under the constraint of very limited code size. However, even modern compilers produce code whose quality often is far away from the optimum. The PROPAN system is a postpass optimization framework that enables high-quality machine-dependent postpass optimizers to be generated from a concise hardware specification. The postpass approach allows to enhance the code quality of existing compilers and offers a smooth integration into existing development tool chains. In this article we present an adaptation of the modulo scheduling software pipelining algorithm to the postpass level. The implementation is fully retargetable and has been incorporated in the PROPAN system. The differences of postpass modulo scheduling compared to the standard version of the algorithm are outlined. Experimental results conducted on the Philips TriMedia TM1000 processor demonstrate that modulo scheduling can be applied at the postpass level and allows to achieve a significant code speedup with moderate code size increase.

Categories and Subject Descriptors

D.2.13 [**Software engineering**]: Reusable software—*reusable libraries*; D.3.4 [**Programming languages**]: Processors—*code generation, compilers, optimization, retargetable compilers*; C.1.1 [**Processor architectures**]: Single data stream architectures—*VLIW architectures*

General Terms

Algorithms, Performance

Keywords

PROPAN, Postpass optimization, Software pipelining, Modulo scheduling

1. INTRODUCTION

During the last years, embedded systems have become nearly omnipresent in everyday life. Embedded processors are used in a variety of application fields: healthcare technology, telecommunication, automotive and avionics, multimedia applications, consumer electronics, etc. Common characteristics of many applications are that high computation performance has to be obtained at low cost and low power consumption. Moreover many applications have safety-critical characteristics and must satisfy real-time constraints. This leads to an additional requirement to be respected in embedded systems design: the requirement of predictable performance. It is not enough for microprocessors to yield high peak performance, but it should also be possible to statically guarantee their worst-case performance. Contemporary superscalar architectures are characterized by deep complex pipelines, often with features like out-of-order execution, branch prediction, and speculative execution which make determining the guaranteed performance of an application a difficult task [16].
Instruction-level parallel architectures have the advantage that they are statically scheduled, i. e. under the control of the compiler, which significantly improves their predictability. They typically have multiple execution units and provide multiple issue slots (EPIC, VLIW). However, since the amount of parallelism inherent in programs tends to be small [33], it is a problem to keep the available execution units busy. For architectures with static instruction-level parallelism this problem is especially virulent, since if not enough parallelism is available the issue slots of the long instruction words are filled with nops. For embedded processors this means a waste of program memory and energy.
A common way to improve the efficiency of embedded processors is the incorporation of application-specific functionality. The consequence is often a highly irregular hardware design which makes code generation a difficult task. Still today, the code quality of traditional high-level language compilers is often far from satisfactory [41, 25]. Generating efficient code for irregular architectures requires highly optimizing techniques that

have to be aware of specific hardware features of the target processor. Traditional legacy compiler systems also are not designed to take full advantage of instruction-level parallel architectures.

The Propan system [18, 21, 19, 20] has been developed as a retargetable framework for high-quality code optimizations and machine-dependent program analyses at assembly level. From a concise hardware specification a machine-sensitive postpass optimizer is generated that especially addresses irregular hardware architectures. The generated optimizer reads assembly programs and performs efficiency-increasing program transformations. One advantage of applying an extra optimization pass on assembly or object code is that the post-pass optimization usually has a larger scope: it can handle libraries, mixed-language code applications and hand-written assembly [10]. Another motivation for the postpass approach is that industry-standard embedded compiler tool chains often do not generate satisfactory code quality, both in terms of performance and code size. However, changing an compiler tool chain requires modifications to the established development process, which, especially for embedded systems, is a long and very costly process. Postpass optimizers can easily be integrated in existing tool chains. Since they work at the machine code level, they can be applied in the context of traditional development environments consisting of CASE tools, compilers and linkers. Only small changes to the software development process are required.

This article deals with the incorporation of the modulo scheduling software pipelining algorithm in the Propan framework. For statically scheduled architectures with a high degree of instruction-level parallelism it is extremely important to enhance the available parallelism of the program[1]. Modulo scheduling is a static global cyclic instruction scheduling technique which improves the execution times of loops by overlapping the execution of different iterations of the original loop. Since typically most of the program execution time is spent in loops, modulo scheduling can cause significant performance improvements. To enable its integration in the Propan framework we have adapted the modulo scheduling algorithm to the postpass level and implemented it in a fully retargetable way. In this article we demonstrate that applying modulo scheduling at the postpass level is a reasonable approach which can give rise to a significant performance increase. The experiments are conducted on the Philips TriMedia TM1000 multimedia processor.

This paper is structured as follows: Sec. 2 gives an overview of related work and Sec. 3 shortly summarizes the Propan framework. The basic concepts of the modulo scheduling algorithm are introduced in Sec. 4; Sec. 5 characterizes our adaptation to the postpass level. The TriMedia Tm1000 VLIW processor used for our implementation is described in Sec 6. The experimental results are presented in Sec. 7 and Sec. 8 concludes.

[1]The available parallelism inside of basic blocks is usually no larger than 2 or 3[33].

2. RELATED WORK

In the area of code generation for general-purpose processors there are numerous retargetable systems, e. g. PO [9] and its descendents Vpo [3] and `gcc` [34], `lcc` [12], Marion [5], or Suif [35]. Retargetable code generation and optimization systems for irregular architectures, mostly digital signal processors are Mimola [26], Record [25], Cbc [11], CHESS [24], Flexware [29], Spam [37], Express [14], or Aviv [15]. While these systems do not support transformations on assembly- or executable code, Salto [4] is a retargetable system designed to support implementing tools for analyses and transformations of assembly code. However, generating program optimizers from the machine description is not supported. Retargetable postpass frameworks focusing on code size reduction are the link time optimizers Diablo [6], Squeeze++[38], and the aiPop system which is partially based on Propan [10].

Software pipelining aims at improving the execution times of loops by overlapping the execution of different iterations to increase the available parallelism. It is typically applied at a mid-level or low-level intermediate representation of the compiler. The most common software pipelining techniques can be classified into two categories, namely *kernel recognition* and *modulo scheduling* techniques. The idea behind kernel recognition schemes is to simultaneously unroll and schedule the loop until the rest of the schedule would be a repetition of an existing portion of the schedule. Then the process is terminated by generating a branch back to the repetitive portion. Common kernel recognition techniques are, e. g. *perfect pipelining* [1], the *petri net model* [2] and *Vegdahl's technique* [40, 39]. Kernel recognition techniques usually have to maintain complex information to be able to recognize the repetitive code pattern. In contrast to that *modulo scheduling* techniques directly construct the pipelined loop without unrolling, based on an initiation interval (cf. Sec. 4). Well-known software pipelining approaches are [22], *iterative modulo scheduling (IMS)* [31], *slack modulo scheduling* [17], *swing modulo scheduling* [7] and *integrated register-sensitive software pipelining* [8]. Especially the computation of operation priorities and the scheduling direction (top-down vs. bidirectional) are important differences among these approaches [7]. Another more complex software pipelining technique is *enhanced pipeline scheduling* [2] which can schedule loops with conditional jumps and arbitrary iteration counts.

An implementation of modulo scheduling for the TI TMS320C6x is described in [36]. The implementation is not retargetable but specifically tailored to the C6x, the description focusing mostly on the characteristics of the C6x. The authors use slack modulo scheduling [17] and present adaptations, mostly concerning modifications of the scheduling direction and priority function of slack modulo scheduling.

3. PROPAN

Propan (*Postpass-oriented retargetable Optimizer and Analyser*) is a retargetable framework for machine de-

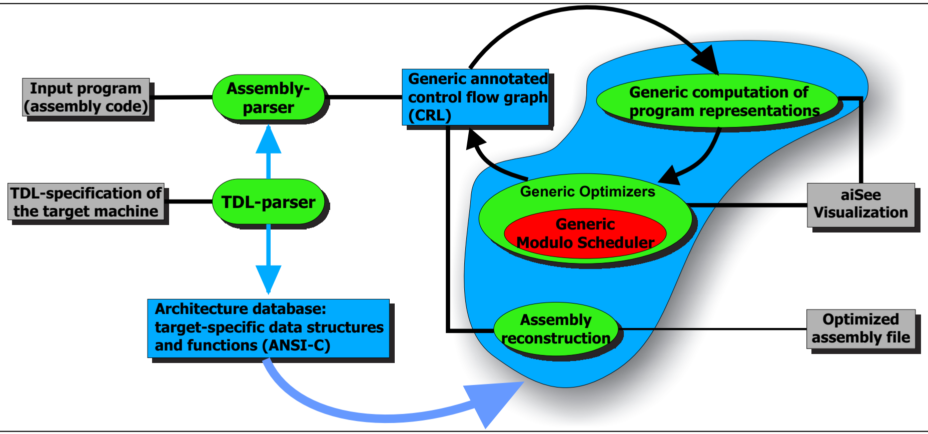

Input program (assembly code)
Assembly-parser
Generic annotated control flow graph (CRL)
Generic computation of program representations
TDL-specification of the target machine
TDL-parser
Generic Optimizers
Generic Modulo Scheduler
aiSee Visualization
Architecture database: target-specific data structures and functions (ANSI-C)
Assembly reconstruction
Optimized assembly file

Figure 1: Structure of the Propan-System

pendent high quality code optimizations and analyses at the assembly level. The structure of the PROPAN-Framework is shown in Fig. 1. Inputs are a specification of the target processor and the assembler file to be optimized. The target specification is written in the *Target Description Language* TDL [19]. It contains definitions of all functional units, the instruction set including timing behavior and semantics, descriptions of irregular hardware constraints and a specification of the assembler syntax. From the TDL file an assembly parser is automatically generated which translates the input program into the intermediate representation of PROPAN, called CRL (*Control Flow Representation Language*) [23]. Moreover a hardware architecture database is generated along with generic access functions such that all specified information about the target processor can be retrieved in a generic way from the core system. The core of PROPAN is fully generic, i. e. processor independent. It is linked with the generated target specific files to yield a dedicated hardware sensitive postpass optimizer [21].

4. MODULO SCHEDULING

Modulo Scheduling is a common framework in which software pipelining algorithms can be defined [2]. To achieve an overlapping of consecutive loop iterations the algorithm computes the so-called *initiation interval (II)*, which defines the number of cycles between the start of two consecutive iterations. Based on the initiation interval a *modulo schedule* is constructed by dividing the original loop body into *stages* of *II* cycles each. *Stages* from different loop iterations can be executed in parallel. This yields a new (shorter) loop body called the *kernel*. Since one iteration of the kernel executes multiple original iterations it is necessary to insert acyclic code sequences to fill/drain the software pipeline, i. e. to execute the required number of iterations before/after the execution of the kernel (see Fig. 2). The computation of the kernel from the original loop schedule is divided into three phases. First a lower bound for the initiation interval is computed, called *minimum initiation interval (MII)*. This is the phase with the highest

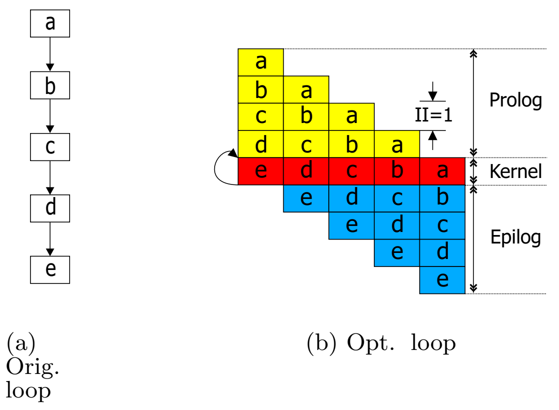


Figure 2: Structure of a pipelined loop

computational complexity, namely $\mathcal{O}(n^3)$, where n is the number of operations contained in the loop. Note that a SCC[2]-based computation as mentioned in [31] could reduce the computational costs to $\mathcal{O}(n^3)$, where n is the number of operations in the largest SCC in the data dependence graph of the loop. We do not rely on a SCC-based computation in aid for the reuse of intermediate data of this phase (cf. [31]). In the second phase, the actual scheduling phase, the algorithm tries to find a feasible kernel starting with the computed *MII*. In *Iterative Modulo Scheduling (IMS)* this phase is iterative because each operation can be scheduled and unscheduled at several time slots [7], which is the main characteristic property of *IMS*. If the search for a kernel fails, the *initiation interval* is increased and the scheduling phase is restarted. This process is iterated until a feasible kernel has been found. In spite of the exponential complexity of scheduling in general, the run time of this scheduling phase is bounded to $\mathcal{O}(n^2)$ (according to [31]) by heuristics.
Our implementation is based on *IMS* [31] since it has

[2]Strongly connected component

been extensively studied and is a common phase in advanced optimizing compilers. In the following we will describe how the three phases are computed by the *IMS* approach and briefly sketch the generation of prologue and epilogue code. Afterwards, in Sec. 5 we show the characteristic properties evolving for the application of *IMS* on assembly code.

4.1 Computing the MII

The problem of computing the *MII* is split into two subproblems: the computation of a *resource based initiation interval* (MII_{res}) and a *data dependence based initiation interval* (MII_{dep}). MII_{res} and MII_{dep} represent two upper bounds for the number of iterations that can be overlapped; the computed *MII* is the maximum of both values.

MII_{res} computes a lower bound for the delay between the start of two consecutive iterations by only taking the resource constraints of the target processor into account. Data dependences are ignored. The result is the minimal number of instructions[3] needed to execute all operations of the loop without violating any resource constraints.

In contrast to that, MII_{dep} only considers the data dependencies in the loop. Let $e = (i_1, i_2)$ be an edge in the data dependence graph (DDG), i. e. i_2 is data dependent on i_1. Then $Delay(e)$ denotes the minimal number of cycles between the start time of i_1 and i_2 that is necessary to preserve the data dependence. In cyclic control flow there are elementary cycles[4] in the DDG which impose delays between the start of two consecutive iterations. Moreover we have to distinguish between dependencies within the same iteration of a loop (*loop-independent*) and dependencies that span iteration boundaries (*loop-carried*). Therefore we denote the number of iteration boundaries that are spanned by the particular data dependence e with $Dist(e)$. Each elementary cycle c then imposes the following constraint for the MII_{dep}:

$$MII_{dep} * Dist(c) \geq Delay(c).$$

$Delay(c)$ resp. $Dist(c)$ are defined as the sum of the delays resp. distances of all edges in c. If c spans k iterations, the delay of c can be at most k times the MII. The following formula defines how MII_{dep} can be computed:

$$MII_{dep} = \max_{c \in \mathcal{C}} \left(\left\lceil \frac{Delay(c)}{Dist(c)} \right\rceil \right)$$

4.2 The Scheduling Phase

In the scheduling phase the initiation interval is used to compute a so-called *flat schedule*. The flat schedule is an acyclic schedule of the operations of the loop that takes also cyclic precedence and resource constraints into account. The final kernel can be derived from the flat schedule by simple modulo arithmetic (cf. Sec. 4.3). The central idea of the algorithm is to start with an empty schedule and add operations to it maintaining the property that any partial schedule represents a feasible partial schedule of the loop. That means that neither acyclic nor cyclic resource or precedence constraints are violated.

4.2.1 Computing Scheduling Priorities

Before scheduling, the order must be defined in which the operations are added to the partial schedule. *Iterative Modulo Scheduling* uses a priority function similar to the *highest level first (hlf)* priority known from *list scheduling*. This priority function prefers operations that are on the critical path[5] in the program [31]. In order to cope with cyclic control flow and to take the computed initiation interval into account the *hlf* priority scheme has to be modified [31]. The resulting priority for an operation p is denoted as:

$$HeightR(p) = \max_{q \in Succ(p)} (HeightR(q) + EffDelay(p, q))$$

where $Succ(p)$ is the set of all successors of p in the DDG and $EffDelay(p, q) = Delay((p, q)) - II * Dist((p, q))$.

4.2.2 Computing the Slot Window

Adding an operation to the partial schedule without violating data dependences requires all data dependences among the operations that are contained in the partial schedule to be taken into account. The data dependences determine how early resp. how late an operation can be scheduled within the partial schedule. This results in a *slot window* relative to the operations contained in the partial schedule. A slot window is defined by two values: the earliest possible issue time, *early start (EStart)*, and the latest possible issue time, *late start (LStart)*. *Estart* is computed as

$$EStart(p) = \max_{(q,p) \in E_{dd}} (\quad \max(0, SchedTime(q) + EffDelay(q, p))),$$

where E_{dd} is the set of all edges in the data dependence graph and $SchedTime(p)$ is the time at which p is scheduled. Note that this formula is only valid if the predecessors q are contained in the current partial schedule. In the case that p has no predecessor or no predecessor of p has already been scheduled at all, i. e. is not contained in the partial schedule, $Estart(p) = 0$. $LStart(p)$ is computed analogously by considering the successors of p.

4.2.3 Scheduling Conflicts

It is possible that no feasible instruction can be found in the computed slot window – this is called a *conflict*. There are two difference conflict scenarios: first the computed slot windows might be infeasible, i. e. $Lstart < EStart$. This follows from the property of modulo scheduling that some successors in the DDG of an operation p can already be scheduled while trying to

[3]We use the terminology that a (VLIW) instruction is composed of several micro-operations that are issued simultaneously

[4]An elementary cycle is a cycle with pairwise disjoint nodes.

[5]The critical path is the longest acyclic path in the DDG.

find an issue slot for p. Second, the computed slot window may be feasible, but the algorithm does not find an instruction not causing a resource conflict. To solve such a conflict for an operation i, a forced control step c_f is computed where $c_f = \max\{c_{old}+1, EStart(i)\}$, if i had been scheduled before to control step c_{old}, and $c_f = EStart(i)$ otherwise. Then i is scheduled to c_f, and all previously scheduled operations that cause conflicts with i are removed from the partial schedule. The removed operations possess higher priorities than any operations not scheduled before. The computation of c_f ensures termination since an operation is never scheduled twice at the same control step.

4.3 Computing the Kernel

The kernel can be computed directly from the flat schedule. The scheduling position of an operation in the kernel is given by its control step in the flat schedule modulo the initiation interval. This directly induces an overlapping of different loop iterations.
After generating the kernel we have to check whether the maximal life span of a register in the flat schedule is longer than the length of the kernel itself. In that case we need to perform *Modulo Variable Expansion* [22]. In the postpass scenario modulo variable expansion has to be applied on physical registers; Sec. 5.3 elaborates.

4.4 Prologue and Epilogue

Since the kernel is computed by taking the control steps of the flat schedule $\mathcal{F}$ modulo the minimum initiation interval MII, the flat schedule is divided into *stages* of MII cycles each. In the computed kernel all stages are executed in parallel. The number of iterations of the original loop represented in the kernel is given by the *stage count* $SC = \left\lceil \frac{|\mathcal{F}|}{II} \right\rceil$ with $|\mathcal{F}|$ denoting the number of instructions in $\mathcal{F}$. Since the kernel represents SC iterations of the original loop and since one iteration of the original loop is started by each execution of the kernel, prologue and epilogue must start/end $SC - 1$ iterations of the original loop. Extensions are required if there are *early exits* in the original loop [32].

5. POSTPASS MODULO SCHEDULING

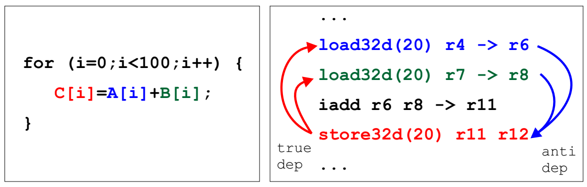
A main characteristics of our approach is that we apply modulo scheduling on assembly code. This has several consequences: one difference to using modulo scheduling inside a compiler on some (low-level) IR is that the amount of information available to the modulo scheduler is smaller. Also the degree of freedom available to the modulo scheduler is limited since there are decisions made in code generation stage that cannot be undone. However, all transformations required for modulo scheduling can still be applied at the postpass level, although some modifications are required. The main limitations result from control flow and data dependence characteristics of assembly programs. In the remainder of this section we describe the necessary control flow reconstruction stage and investigate the consequences of the data dependence structure of typical assembly code. Sec. 5 concludes with a description of postpass modulo variable expansion.

5.1 Handling Control Flow

In assembly programs the control flow is implicitly defined by the semantics of unstructured control flow instructions like jump, branch, call and return instructions. Especially information about the loops in the program is not directly available. Thus, the control flow graph has to be explicitly reconstructed. PROPAN uses a generic control flow reconstruction algorithm [20] which is based on an extended program slicing mechanism that is able to cope with unstructured control flow instructions.
The major problem in the reconstruction of the CFG are indirections of the control flow. These are computed jumps and computed calls, i. e. , jumps and calls whose argument is not an assembly label but a register value. In order to determine targets of control flow indirections static information about register contents are computed. Program slicing allows to restrict the required evaluation of machine instructions to those instructions that immediately determine the targets of control flow indirections [20]. In cases where the control flow cannot be precisely reconstructed, user-annotations are provided. They can be used to classify branch instructions used as returns or name the actual targets of computed branches.
The postpass approach does not only require special care when reconstructing the control flow; also after applying modulo scheduling some transformations are required. The control flow of the input program is represented by a set of unstructured control flow instructions. After modulo scheduling the targets of these branches are not valid anymore. Thus, the branches have to be updated so that the new loop structure is represented, i. e. with the prologue as the new loop entry, the kernel as the loop body and the epilogue to finish and leave the loop. Moreover it has to be ensured that all branches to the loop body target the new loop body. These branch target modifications apply to all *back edges*, all *loop-exit branches* and all *loop-entry branches*.

5.2 Data Dependences

In a postpass scenario the register allocation of the input program has already been done. Thus the program operates on physical registers, and not on the original program values. Since during register allocation the values of the program have been mapped to a finite number of physical registers, register reuse has been introduced. This register reuse causes spurious data dependences which are not induced by the semantics of the program but are caused by the actual register assignment. It is possible to retransform the input program in a low-level intermediate representation where physical register are replaced by virtual registers [21] so that the spurious data dependences can be eliminated.
In assembly programs accesses to structured data types are represented by sequences of memory referencing machine instructions. As an example consider Fig. 3. At the source level, it is easy to establish that the array elements being accessed represent different memory lo-



```
for (i=0;i<100;i++) {
    C[i]=A[i]+B[i];
}
```

```
...
load32d(20) r4 -> r6
load32d(20) r7 -> r8
iadd r6 r8 -> r11
store32d(20) r11 r12
...
```

(a) C-Code (b) Assembly Code

Figure 3: Data Dependencies at the Assembly Level.

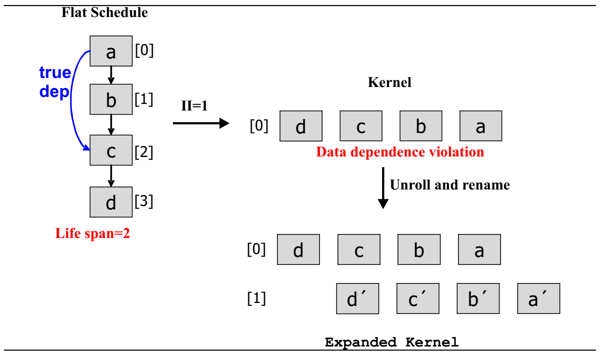


Figure 4: Modulo Variable Expansion

cations. Thus, there are no loop-carried data dependences. Inside a compiler this information is available. However, at the postpass level the array accesses are represented by a sequence of different memory referencing instructions. Statically proving that such a set of memory references can be divided in groups accessing disjoint structured memory locations in general is not possible. In our implementation we conservatively assume a potential aliasing relation between any two memory accesses, i. e. there is a data dependence between each memory access.
In consequence the available parallelism of the program is reduced and the scheduling freedom is restricted. Advanced alias analyses allow to increase the available parallelism by disambiguating memory accesses, e. g. by computing values analyses [27] for address registers. However, this is not part of our current implementation.

5.3 Register Allocation and Modulo Variable Expansion

Processing a register allocated input program means that the decision which values are held in registers and which values are stored in memory has been made by the compiler. Undoing such allocation decisions at the postpass level would require extensive memory disambiguation and alias analyses without being generally applicable. However, the register assignment is still amenable to a postpass optimizer [18].
An important component of the modulo scheduling algorithm is *modulo variable expansion* (MVE). Without MVE the kernel computed by modulo scheduling will contain overlapping life spans of physical registers if there are register life spans in the flat schedule whose length exceeds the kernel length. Repeatedly executing the kernel then causes live register values to be overwritten. Modulo variable expansion avoids this by unrolling the kernel while appropriately renaming the used registers.

Since MVE only affects the register assignment, it is possible to provide modulo variable expansion by register renaming at the postpass level. To prevent live registers from being overwritten, the kernel $\mathcal{K}$ has to be unrolled so that the unrolled kernel is as least as long as the maximal life span of a register in the flat schedule.

The unroll factor is computed by

$$u_{min}(\mathcal{K}) = \max_{r\in\mathcal{R}}\left(\left\lceil \frac{LifeSpan_{\mathcal{F}}(r)}{II}\right\rceil\right),$$

where $\mathcal{R}$ denotes the set of all registers and $LifeSpan_{\mathcal{F}}(r)$ the life span of register r in the flat schedule $\mathcal{F}$. Then all loop-variant registers used in the unrolled copies of the kernel are renamed using the following renaming scheme: all definitions of such a register r are changed to $r^\star$, where $r^\star$ is a new free register. All uses of r before its first redefinition[6] are changed to r', where r' is the register used for r in the preceding copy. All uses of r after the first redefinition are changed into $r^\star$. Because of the cyclic use of registers this renaming scheme simulates the behavior of *rotating register files* [17]. The decision which registers are free and can be used for the renaming are based on liveness analysis [28] for the values contained in physical registers.

6. TRIMEDIA TM1000

The TriMedia TM1000 [30] is a *Digital Signal Processor (DSP)* for high performance multimedia applications processing high quality video and audio streams. It consists of a 100MHz VLIW-Core (32 Bit) with 128 32-bit general purpose registers and 26 parallel functional units. Tab. 1 gives an overview of the available types of functional units and lists the number of cycles used to execute an operation on each particular functional unit type. The instruction set does not contain special return operations; instead all returns are implemented by branch operations. Since nearly all control flow transfers are realized by indirect branches many user annotations are required for control flow reconstruction, e. g. all returns have to be explicitly annotated (see Sec. 5.1).
The TriMedia TM1000 issues one VLIW instruction per cycle, each one consisting of five *issue slots*. An issue slot contains one machine operation so that up to five operations can be executed in parallel. The functional unit binding is heterogeneous (cf. Sec. 5), which is typical for DSP processors. A branch operation, e. g. , can

[6]All read accesses to r in a VLIW instruction also containing definitions of r are considered to be executed *before* the definitions.

Unit	Operation Class	Cycles
CONST	immediate	1
ALU	integer arithmetic , logical operations and (De-)compression	1
DSPALU	DSP arithmetic	2
DSPMUL	DSP multiplications	3
DMEM	load/store	3
DMEMSPEC	cache operations	3
SHIFTER	shift operations	1
BRANCH	control flow operations	3
FALU	floating point arithmetic	3
IFMUL	integer and floating multiplications	3
FCOMP	floating point comparisons	1
FTOUGH	floating point divisions	17

Table 1: Functional units and timing

only be executed in issue slots 2, 3, or 4, arithmetic operations like an addition can be assigned to any of the five issue slots. There are five *write-back busses* so that at most five results can be written per cycle. Since different machine operations have different execution times, the write back bus has to be modeled explicitly in the scheduling phase.

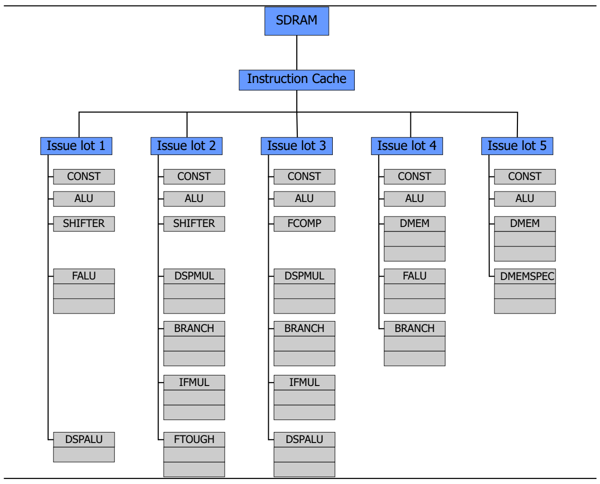


Figure 5: Functional unit binding of the TriMedia TM1000

7. EXPERIMENTAL RESULTS

The generic modulo scheduler has been integrated into the Propan-System and has been retargeted to the TriMedia Tm1000 processor. In this section we present our experimental evaluation of the postpass modulo scheduling algorithm on the TriMedia Tm1000. Our input program comprise files from the *DSPSTONE* [41] and the *MiBench* [13] benchmark suites. In addition two hand-written assembly files (`chain`,`chain2`) are investigated that are dominated by linear dependence chains preventing acyclic parallelization. The files are compiled using the *Philips* `tmcc` compiler in the version V5.5.4 (compile switches: `-O2 -Xunroll=0`) which doesn't perform software pipelining.
The results are measured by several metrics: the gained performance achievements, the code size increase, the feasibility of the computed MII and the runtimes of the software pipelining process.

7.1 Performance

In order to measure the gained performance increase we determine the number of instructions needed for 100 loop iterations. In case of the original loop, this is 100 times the number of the instructions of the loop body. Since in the pipelined loop the *prologue* and *epilogue* code sequences are executed exactly once and each iteration of the kernel finishes one original iteration, the number of iterations is

$$|\mathcal{P}| + |\mathcal{E}| + \left(\frac{100 - SC - 1}{u_{min}} * |\mathcal{K}|\right),$$

where $|.|$ denotes the length (i. e. the number of instructions) of each code sequence, respectively. The variable u_{min} is the factor of kernel unrolling used in modulo variable expansion (see Sec.4.3).
Tab. 2 shows that our implementation achieves a speedup by a factor of up to 3.13 (column Δv), compared to the schedule produced by the `tmcc` compiler. The average speedup is 1.81. *IMS* works best in the presence of dependency chains where acyclic scheduling techniques cannot exploit instruction level parallelism at all (cf `chain1,chain2`). If the input programs already exhibit a high degree of instruction level parallelism, there may be no further parallelization opportunities. For that reason we could not optimize the performance of three programs `dfir`, `dlms` and `sha`). In Tab. 2 the average

Program	$\mathbf{I_{in}}$	$\mathbf{I_{opt}}$	Δv	$\mathbf{P_{in}}$	$\mathbf{P_{opt}}$
adpcm	2800	1596	1.75	2.1	3.8
bitstrng	1000	798	1.25	2.2	2.8
blowfish	4000	2092	1.91	1.5	2.9
chain	800	292	2.74	1.0	5.0
chain2	900	295	3.05	0.8	5.0
crc32	1000	403	2.48	1.5	3.0
dfir	1200	1200	1.00	3.3	3.3
dijkstra	2100	1194	1.76	1.8	3.2
dlms	1000	1000	1.00	3.6	3.6
dmat1x3	2500	799	3.13	1.0	3.1
dmatrix1	4000	1985	2.02	1.0	3.1
FFT	5400	2386	2.26	1.2	2.8
gsm	2000	1692	1.18	2.9	3.4
isqrt	1100	796	1.38	2.6	3.6
patricia	1600	701	2.28	1.5	3.4
pgp	1400	698	2.01	1.7	2.7
sha	800	800	1.00	2.4	2.4
Ø			1.81	2.0	3.3

Table 2: Performance Increase.

number of operations per VLIW instruction before and after modulo scheduling are compared as a measure of the instruction level parallelism exploited. We can see that the maximum parallelism achieved (column P_{opt}) is 5.0 operations per instruction (for `chain`, `chain2`). The issue width of the TriMedia Tm1000 is 5 instructions, but due to the heterogeneous issue slot assignment, a parallelism of 5 operations per instructions mostly cannot be achieved. To that effect, the achieved average operations per instruction of over 3.0 (for `adpcm`, `dfir`, `dijkstra`, ...) are quite well. The average parallelism of all tests is 3.3 operations per instruction, compared to an average parallelism of 2.0 in the input programs (column P_{in}).

7.2 Code Size

In embedded systems code size plays an important role. Software pipelining usually causes code size to grow since it is necessary to add prologue and epilogue code. In order to be useful for embedded systems it is essential to only get a moderate code size increase.
Tab. 3 illustrates the code size of the investigated programs before applying modulo scheduling (S_{in}) and afterwards (S_{opt}); as a measure for code size the number of instructions is given. Our results show that the average increase of code size is about 50% (column S_{incr}). The maximal code size increase is 120% (`bitstrng`). However, the number of instructions in the loop body, the kernel, is significantly reduced since more instruction level parallelism could be exploited. The size of almost half of the loop bodies could be reduced by more than 50%. Thus, compared to *loop unrolling* techniques , the code size increase is moderate (50% on average) and due to the higher instruction level parallelism the size of the loop bodies can even be *shortened* (to up to 22% of the original size).

Program	S_{in}	S_{opt}	S_{incr}	MII	II
adpcm	28	60	114%	12	12
bitstrng	10	22	120%	5	6
blowfish	40	76	111%	13	13
chain	8	10	25%	1	1
chain2	9	7	- 22%	1	1
crc32	10	23	30%	3	3
dfir	12	12	0%	8	12
dijkstra	21	32	52%	9	9
dlms	10	10	0%	8	10
dmat1x3	25	39	56%	5	5
dmatrix1	40	45	13%	12	12
FFT	54	58	7%	19	19
gsm	20	43	115%	12	12
isqrt	11	20	82%	6	7
patricia	16	29	81%	5	5
pgp	14	26	86%	5	5
sha	8	8	0%	8	8
Ø			50%		6%

Table 3: Code Size Increase.

7.3 Feasibility of the MII

Calculating the *MII* as a lower bound of the kernel length has a high computational complexity. Thus it is interesting to see how far this lower bound is away from the actual *II* for which a feasible kernel could be found. Our experiments show that in 58% of the test files the computed *MII* is already feasible. The average deviation in the other tests is only about 6%. This justifies the rather high computational complexity of computing the MII.

7.4 Runtime

The experiments are conducted on a Linux system with a Intel® Pentium^{TM} 4 (3.06 GHz and 512MB DDR Memory). Tab. 4 shows the computation time of the modulo scheduling for all test programs. The computation times for computing the MII, the prologue, the kernel, the epilogue, and the reintegration of the pipelined loop into the surrounding control flow are listed separately. Column *Total* shows the overall runtime for the whole optimization. The computation time for the input programs ranges from 0.5s to 8.2s, which is acceptable for industrial tool chains. On average about 97% of the time is spent in computing the MII.

8. CONCLUSION

The growing complexity of embedded software creates an urgent need for fast program execution under the constraint of very limited code size. However, industry-standard embedded compiler tool chains often do not generate satisfactory code quality, both in terms of performance and code size. However, changing a compiler tool chain requires modifications to the established development process, which, especially for embedded systems, is a long and very costly process. The Propan system is a postpass optimization framework that enables high-quality machine-dependent postpass optimizers to be generated from a concise hardware specification. The postpass approach allows to enhance the code quality of existing compilers and offers a smooth integration into existing development tool chains.
Subject of this article is the incorporation of modulo scheduling into the Propan framework, which specifically aims at improving code quality for statically scheduled instruction level parallel architectures. Compared to superscalar pipelined architectures such processors have the advantage of a better predictability of performance. This is essential for real-time and safety-critical systems. Modulo scheduling improves the execution times of loops by overlapping the execution of different iterations of the original loop. To enable its integration in the Propan framework the modulo scheduling algorithm is adapted to the postpass level.
The main difference to using modulo scheduling inside a compiler is that the amount of information available and, in consequence, the scheduling freedom is smaller. However, in our work we could demonstrate that all transformations required for modulo scheduling can still be applied at the postpass level. To assess the quality of our postpass modulo scheduling framework we con-

Program	MII [ms]	Prologue [ms]	Kernel [ms]	Epilogue [ms]	Integration [ms]	Total [ms]
adpcm	6740.77	1.60	231.95	0.90	0.25	6977.10
bitstrng	266.99	0.25	6.53	0.25	0.14	275.14
blowfish	6561.12	1.96	302.89	1.21	0.31	6957.81
chain	3.02	0.12	0.33	0.07	0.11	4.27
chain2	3.15	0.06	0.31	0.03	0.11	4.22
crc32	81.02	0.33	2.70	0.24	0.14	85.22
dfir	1574.55	0.05	33.03	0.11	0.04	1608.51
dijkstra	1559.27	0.38	48.05	0.38	0.17	1609.20
dlms	1244.43	0.41	25.76	0.14	0.14	1271.78
dmat1x3	442.20	0.60	10.70	0.29	0.18	455.39
dmatrix1	6371.95	0.62	169.06	0.50	0.22	6544.03
FFT	8231.24	0.86	263.55	0.54	0.26	8498.72
gsm	5546.25	0.58	153.37	0.58	0.19	5702.27
isqrt	671.14	0.29	22.73	0.25	0.13	695.41
patricia	367.13	0.29	10.86	0.52	0.16	379.91
pgp	380.34	0.43	14.22	0.33	0.15	396.31
sha	160.95	0.24	4.14	0.18	0.13	166.35
Ø	2194.83	0.51	70.33	0.36	0.16	2271.94

Table 4: Computation Time for Modulo Scheduling.

ducted experiments for the Philips TriMedia multimedia processor. The TriMedia is a five-issue VLIW processor with restrictions on the assignment of operations to issue slots of the instruction word. Our experiments show that in spite of the limitations of the postpass approach a significant speedup over version V5.5.4 of the Philips tmcc compiler can be achieved with only a moderate increase of code size.

9. REFERENCES

[1] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In H. Ganzinger, editor, *ESOP'88, 2nd European Symposium on Programming*, volume 300 of *LNCS*, pages 221–235. Springer, 1988.

[2] V. Allan, R. Jones, R. Lee, and S. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[3] M. Benitez and J. Davidson. A Portable Global Optimizer and Linker. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, in SIGPLAN Notices*, 23(7):329–338, July 1988.

[4] F. Bodin, Z. Chamski, E. Rohou, and A. Seznec. *Functional Specification of SALTO: A Retargetable System for Assembly Language Transformation and Optimization, rev. 1.00 beta.* INRIA, 1997.

[5] D. Bradlee. Retargetable Instruction Scheduling for Pipelined Processors. Phd thesis, Technical Report 91-08-07, University of Washington, 1991.

[6] B. D. Bus, B. D. Sutter, L. V. Put, D. Chanet, and K. D. Bosschere. Link-time Optimization of ARM Binaries. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 211–220. ACM Press, 2004.

[7] J. Codina, J. Llosa, and A. González. A Comparative Study of Modulo Scheduling Techniques. In *ICS*, pages 97–106, 2002.

[8] A. Dani, V. Ramanan, and R. Govindarajan. Register-Sensitive Software Pipelining. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 194–198. IEEE Computer Society, March 1998.

[9] J. Davidson and C. Fraser. The Design and Application of a Retargetable Peephole Optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, Apr. 1980.

[10] B. De Bus, D. Kästner, D. Chanet, L. Van Put, and B. De Sutter. Post-Pass Compaction Techniques. *Communications of the ACM*, Aug. 2003.

[11] A. Fauth. Beyond Tool-Specific Machine Descriptions. In *[?]*, chapter 8, pages 138–152. Kluwer, 1995.

[12] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design And Implementation.* Benjamin/Cummings Publishing Company, Inc., 1995.

[13] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[14] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A

Language for Architecture Exploration through Compiler/Simulator Retargetability. *Proceedings of the DATE99*, 1999.

[15] S. Hanono and S. Devadas. Instruction Scheduling, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proceedings of the Design Automation Conference 1998*, San Francisco, California, 1998. ACM.

[16] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7), July 2003.

[17] R. Huff. Lifetime-sensitive modulo scheduling. *ACM SIGPLAN Notices*, 28(6):258–267, June 1993.

[18] D. Kästner. PROPAN: A Retargetable System for Postpass Optimisations and Analyses. *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, June 2000.

[19] D. Kästner. TDL: A Hardware Description Language for Retargetable Postpass Optimizations and Analyses. In *Proceedings of the Second ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE)*, 2003.

[20] D. Kästner and S. Wilhelm. Generic Control Flow Reconstruction from Assembly Code. *Proceedings of the ACM SIGPLAN Joined Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, June 2002.

[21] D. Kästner. *Retargetable Postpass Optimisation by Integer Linear Programming.* PhD thesis, Saarland University, Saarbrücken, 2000.

[22] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *ACM SIGPLAN Notices*, 23(7):318–328, July 1988.

[23] M. Langenbach. CRL – A Uniform Representation for Control Flow. Technical report, Universität des Saarlandes, 1998.

[24] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable Code Generation For Embedded DSP Processors. In *[?]*, pages 85–102. Kluwer, 1995.

[25] R. Leupers. *Retargetable Code Generation for Digital Signal Processors.* Kluwer Academic Publishers, 1997.

[26] P. Marwedel and W. Schenk. Cooperation of Synthesis, Retargetable Code Generation and Test Generation in the MIMOLA Software System. *European Conference on Design Automation*, pages 63–69, 1993.

[27] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, volume 2053 of *LNCS*, pages 155–172. Springer-Verlag, May 2001.

[28] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.

[29] P. Paulin, C. Liem, T. May, and S. Sutarwala. FLEXWARE: A Flexible Firmware Development Environment for Embedded Systems. In *[?]*, pages 67–84. Kluwer, 1995.

[30] Philips Electronics North America Corporation. *TriMedia* TM1000 *Preliminary Data Book*, 1997.

[31] B. Rau. Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[32] B. Rau et al. Code Generation Schema for Modulo Scheduled DO-Loops and WHILE-Loops. HP Labs Technical Report HPL-92-47, Hewlett-Packard Laboratories, 1992.

[33] B. Rau and J. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[34] R. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Cambridge/Massachusetts, 1998.

[35] Stanford Compiler Group. *SUIF Compiler System: The SUIF Library*, 1994.

[36] E. Stotzer and E. Leiss. Modulo Scheduling for the TMS320C6x VLIW DSP Architecture. In *Proceedings of the LCTES*, pages 28–34. ACM Press, 1999.

[37] A. Sudarsanam. *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors.* PhD thesis, University of Princeton, Nov. 1998.

[38] B. D. Sutter, B. D. Bus, and K. D. Bosschere. Sifting out the Mud: Low Level C++ Code Reuse. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 275–291. ACM Press, 2002.

[39] S. Vegdahl. *Local code generation and compaction in optimizing microcode compilers.* PhD thesis, CMU, 1982.

[40] S. R. Vegdahl. A Dynamic-Programming Technique for Compacting Loops. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 180–188, 1992.

[41] V. Živojnović, J. Martínez, C. Schläger, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *Proc. of ICSPAT'94 - Dallas*, October 1994.