Retargetable Code Optimization for Predicated Execution

M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, and H. Meyr
Institute for Integrated Signal Processing Systems
RWTH Aachen University, Germany

Gerrit Bette
Associated Compiler Experts bv
Amsterdam, The Netherlands

Balpreet Singh
NXP Semiconductors
Eindhoven, The Netherlands

Abstract

Retargetable C compilers are key components of today's embedded processor design platforms for quickly obtaining compiler support and performing early processor architecture exploration. The inherent problem of the retargetable compilation approach, though, is the well known trade-off between the compiler's flexibility and the quality of generated code. However, it can be circumvented by designing flexible, configurable code optimization techniques applicable to a certain range of target architectures. This paper focuses on target machines with predicated execution support which is wide-spread in deeply pipelined and highly parallel embedded processors used in next generation high-end video, multimedia and wireless devices. We present an efficient and quickly retargetable code optimization technique for predicated execution that is integrated into an industrial retargetable C compiler. Experimental results for several embedded processors demonstrate that the proposed technique is applicable to real-life target machines and that it produces significant code quality improvements for control intensive applications.

1 Introduction

Over the past few years, the complexity of embedded System-on-Chip (SoC) designs has been increasing due to the continually growing performance requirements of next generation high-end video, multimedia and wireless applications. Contemporary embedded processors must provide very high processing performance at low power and at low costs along with programmability and flexibility. Therefore, system designers employ more and more Application Specific Instruction Set Processors (ASIPs) [1, 2, 3] as building blocks in such systems, since they tend to meet these constraints well. Consequently, the amount of ASIP-related products has grown significantly in the past years. Companies like e.g. CoWare (Processor Designer) [7] or Tensilica (Xtensa) [26] offer platforms for ASIP architecture exploration and design. These platforms build around *retargetable* software development tools (C-Compiler, Simulator, Assembler,etc.) in order to explore different ASIP design alternatives within a short amount of time for a given application domain. Among these tools the *retargetable C compiler* plays an important role for attaining high software development productivity and to cope with the ever growing complexity of today's applications. Unfortunately, such compilers are often hampered by their limited code quality as compared to hand-written compilers or assembly code due to a lower amount of target specific optimizations. While such optimizations are a necessity to generate high code quality, however, this would be counterproductive to achieve the required flexibility to adapt quickly to different ASIP designs. Consequently, once the ASIP architecture exploration phase has converged and an initial working compiler is available, it must be *manually* refined to a highly optimizing compiler. This is both time-consuming and error prone. One way of overcoming this dilemma is to design *retargetable optimizations* for those architectural features that are often recurring in ASIP design, thus achieving *retargetability and high code quality* for a whole *target processor class*. An example is the *retargetable SIMD* support for the class of multimedia processors recently proposed for the CoSy compiler development platform [4, 6].

This paper focuses on another class of target processors, namely those equipped with *Predicated Execution* (PE). It refers to the conditional execution of instructions based on the value of a boolean source operand p, called the *predicate*.

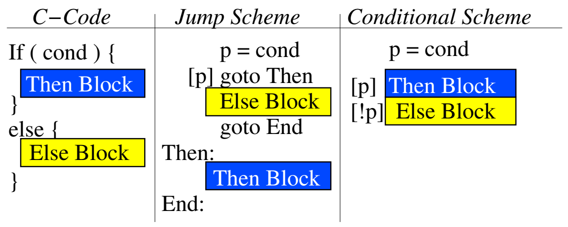


Figure 1. *Jump instructions / predicated instructions*

PE allows compilers to convert control dependencies into data dependencies, also referred to as *if-conversion* [5]. Fig. 1 shows two possible implementation schemes for an *if-then-else* (ITE) statement. Predicated instructions are marked by the prefix "[p]" or "[!p]" (negated predicate). The conditional scheme predicates the `then` block with the result of the if-statement's condition and the `else` block with the negation thereof respectively. This allows on Instruction Level Parallelism (ILP) processors the parallelization of the still mutually exclusive `then` and `else` blocks. Additionally, PE enables more aggressive compiler optimizations which are often limited by control dependencies. For example software pipelining, which is crucial to achieve high performance for ILP processors, can be substantially improved

978-3-9810801-3-1/DATE08 © 2008 EDAA

by PE [13].

An important embedded processor class are VLIW architectures. Such architectures are well suited for today's C compiler technology. Thus, high software development productivity comes along with very high processing performance. Since the performance of such processors [8, 10] strongly relies on the available ILP, PE is almost a standard feature in VLIW processors. However, it is by far not limited to highly parallel and deeply pipelined processors. Even though less beneficial, standard embedded processors like ARM [12] or configurable cores [11] are equipped with this feature, too. Therefore, support for PE in retargetable compilers is of high interest.

PE utilization can already be found in several target specific C compilers, but it is still very weakly supported in ASIP compilers. For the use in this domain, *retargetable predicated execution optimizations* are required. Therefore, in this paper, we propose a novel concept for retargetable code optimization for ASIPs with PE, and we prove this concept by an implementation within an existing, well-tried retargetable compiler framework and an experimental evaluation for several real-life embedded processors.

The rest of this paper is arranged as following. In section 2 related work is discussed. The system overview is given in section 3. Section 4 describes our retargetable PE optimization and section 5 the code generation flow. Section 6 presents results for several embedded processors with PE support. Finally, we summarize the contribution of this paper and point to some future avenues of work.

2 Related work

Many compilation techniques for PE are based on the work by Mahlke et al. [14]. It describes the formation of so called *hyperblocks*, an extended basic block concurrently executing multiple threads of conditional code. The decision whether to include a basic block in a hyperblock is based on the criteria of execution frequency, block size and instruction characteristics. Since it does neither take the degree of ILP into account nor the dependencies between different blocks, scheduling for machines with few issue slots increases the resource interference and thus, results in performance degradation. August et al. [15] improved this work by allowing the scheduler to revise decisions on hyperblock formation. But this leads to a complicated scheduler implementation. Additionally, it extends the previous work by partial if-conversion: in many cases, including only a part of a path may be more beneficial than including or excluding the entire path. Smelyanskiy et al. [20] try to solve the resource interference of Mahlke's approach by a technique called predicate-aware scheduling. However, they state that an architecture that supports their optimization proposal does not exist yet. All hyperblock-based approaches optimize the average execution time. Further approaches exist, but they suffer from code size overhead [18] or increased register pressure [16], both are issues in the embedded domain. Another approach supports only out-of-order architectures [19] which is a non-typical embedded processor design.

ASIP design platforms comprising retargetable C compilers include Processor Designer [7], Expression [23], Mescal [1], Trimaran [25] and ASIPMeister [24]. Except for Trimaran, which is limited to a narrow range of architectures, no PE support has been reported for those tools yet. In the domain of "general purpose" retargetable compilers, the *gcc* [21] supports if-conversion, but *gcc* is generally known as being difficult to adapt efficiently to embedded processor architectures.

In summary, a number of techniques for PE are available, most of which are adapted for a certain target machine. Porting one of them to a new target machine is still a tedious manual process. Therefore, our approach emphasizes *efficient utilization of PE* and *compiler retargetability* at the same time. We used the approach from [22] as starting point for our work. It focuses especially on embedded processors and optimizes the worst-case execution time. In contrast to previous works it is capable of handling complete (possibly nested) ITE statements with multiple basic blocks at a time. The approach introduces several ITE implementation schemes depending on the nesting level and used instructions (conditional jumps or predicated instructions). The optimization algorithm is based on dynamic programming. In the first phase, the costs (i.e. the worst case execution time) for each implementation scheme are calculated. This is based on static formulas which incorporate the ILP degree of the target processor. The second phase selects the implementation schemes. The algorithm guarantees an optimal solution for nested ITE statements provided that the static formulas give an accurate estimation - which might not always be the case.

As outlined in the next sections, we developed an efficient concept for *retargetable PE* support. To the best of our knowledge, our effort offers the first general treatment of PE that is applicable across different architectures. Apart from our main contribution, *retargetability*, we also extended [22] in several directions. More specifically, we use *scheduler feedback* for a precise cost computation, account for the `then`/`else` block order and added support for *partial if-conversion* and *transition probabilities*. Furthermore, we developed a new technique called *ITE splitting*. It can achieve significant speedups in case if-conversion fails. The amount of required target specific information is quite limited, so that most of it can also be extracted automatically from high-level processor models.

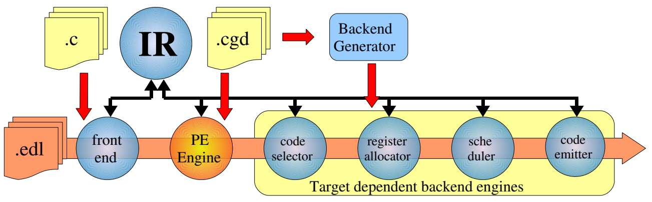


Figure 2. *CoSy compiler generation with PE support*

3 System overview

We employ the CoSy Compiler Development Platform [6] as the retargetable C compiler. ASIP design platforms like CoWare's Processor Designer [7] use it as "backend" to generate the compiler executable. As illustrated in Fig. 2, CoSy compilers are composed of so called *engines* which work on the Intermediate Representation (IR) of the input program. There is a dedicated *Engine Description Language (EDL)* that describes the dynamic calling sequence and calling parameters of the engines. CoSy takes *Code Generator*

Description (CGD) files as input and generates most of the target dependent compiler backend components (code selector, scheduler, etc.) from it. Due to its modular architecture and wide range of available engines, a CoSy compiler using a variety of standard optimizations can be built in short time and with minimum effort. Adding new optimization engines can be done in a plug-and-play fashion. Hence, we added a retargetable *Predicated Execution* engine (plus several auxiliary engines) into the CoSy framework that implements the techniques described next.

4 Predicated Execution

As previously mentioned, PE allows *if-then-else* (ITE) statements to be implemented without jump instructions, i.e. the mutually exclusive `then` and `else` blocks are conditionally executed. As introduced in section 2 the optimization algorithm is based on the costs for different implementation schemes. Thus, we present firstly in section 4.1 how the costs are calculated using *scheduler feedback* information and *transition probabilities*. Then we introduce the new *splitting mechanism* in section 4.2. Thereafter, section 4.3 describes the required retargeting information and section 5 the PE code generation flow.

4.1 Cost Computation

A triplet $S = (cond, B_T, B_E)$ defines an ITE statement, where $cond$ is the condition and B_T and B_E the `then` and `else` blocks, respectively. Let $T(B_x)$ denote the execution time of block B_x, and J_x the target dependent *jump penalty* or *jump delay slots* for a conditional jump taken (J_t), a conditional jump not taken (J_{nt}) and an unconditional jump (J_{unc}). They denote the number of cycles, the pipeline needs to be stalled in order to prevent execution of incorrectly prefetched instructions (i.e. a control hazard). Taking the implementation schemes in Fig. 1 as example, the worst case execution time of the jump scheme is

$$T(S) = \max \begin{cases} T(B_T) + J_t, \\ J_{nt} + J_{unc} + T(B_E) \end{cases}$$

and for the conditional scheme

$$T(S) = T(B_T \circ B_E)$$

where $B_T \circ B_E$ denotes the joint execution of all instructions in the `then` and `else` blocks, i.e. both blocks are merged. Especially the computation of $T(B_T \circ B_E)$ is difficult. In prior work [22] this value is modeled by a static formula which takes the execution times of the individual blocks, the ILP degree and possible resource conflicts into account. In certain cases performance degrades due to inaccurate estimation. Figure 3 illustrates this. Suppose the static formula presumes the schedule in the left column for the jump scheme and the schedule in the middle column for the conditional scheme. As result, the latter will be selected because of the lower $T(S)$. The real schedule for the conditional scheme (right column) though results in a higher $T(S)$ and thus, in a performance degradation. We reduced the number of such cases by coupling the cost computation to the scheduler. It has, naturally, concrete information about the resource usage, thus a more accurate execution time of the block merger can be obtained. Since the scheduler description is already part of the compiler backend, no additional retargeting effort

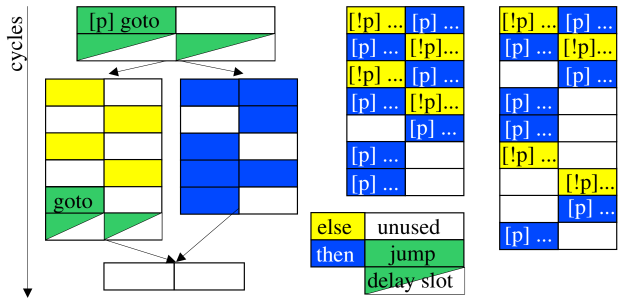


Figure 3. *Scheduling for a two issue slot processor*

is required. However, one should notice that registers are not allocated in that stage of the compiler and hence the values are still estimates. Since we integrated the optimization for the average execution time as well we created new cost formulas to regard the transition probabilities, $P(B_T)$ and $P(B_E)$, for the blocks. They can be provided via pragmas to override the default values for the worst case optimization. Additionally, we added support for *if-then* (IT) statements and *partial if-conversion*. The latter leads to the following new implementation schemes:

```
 //   only Then              or only Else
         p = cond                p = cond
     [p] B_T                 [!p] B_E
     [p] goto End            [!p] goto End
         B_E                      B_T
End:                        End:
```

The execution time for predicating only `then` can be calculated as (for `else`, B_T and B_E need to be exchanged):

$$T(S) = T(B_T) + \begin{cases} \Delta(J_t, B_T) & P(B_T) > P(B_E), \\ T(B_E) + \Delta(J_{nt}, B_T) & P(B_E) > P(B_T) \\ \max \begin{cases} \Delta(J_t, B_T), \\ T(B_E) + \Delta(J_{nt}, B_T) \end{cases} & \text{worst case.} \end{cases}$$

where $\Delta(J_x, B_T)$, in case the `then` block complete fits into the jump's delay slots, regards the unused delay slots:

$$\Delta(J_x, B_T) = \begin{cases} J_x - T(B_T) & J_x > T(B_T), \\ 0 & \text{else.} \end{cases}$$

This technique proves advantageous in case of uneven long blocks. Suppose the `then` block is much shorter than the `else` block. It might be worthwhile to execute both blocks conditionally, because it is likely that the instructions of the `then` block fit into free instruction slots of the `else` block. Regarding the worst case execution time this argument is correct. However, optimizing for the average execution time the actual performance can degrade. For instance, if we assume the transition to the ITE blocks is equiprobable then applying if-conversion means an increase of the execution time in 50% of all cases. If $P(B_T) > P(B_E)$, it is even worse. In those cases executing only the `then` block conditionally might be beneficial. As stated earlier, further implementation schemes exist to deal with nested ITE statements and different order of `then` and `else` blocks (i.e. which block comes first after the condition evaluation).

Apart from the jump penalties J_x, certain schemes must account for target dependent setup costs. For instance, some architectures might need an additional instruction to calculate the negation [!p] for certain schemes in case it is not directly supported. Thus, our approach retargets the cost formulas according to the target configuration (see section 4.3).

4.2 Splitting Mechanism

The *splitting mechanism* is a technique which uses PE to fill delay slots of conditional jumps. It targets ITE statements to which if-conversion cannot be applied. For various reasons the cost computation might decide against if-conversion, one ITE block might have multiple incoming control flow edges or one or both ITE blocks might contain hampering elements, e.g. non predicable instructions. The mechanism is illustrated in Fig. 4. It alternately selects instructions from the then and else block (i.e. T.1, E.1, T.2 in the example) and moves them into the delay slots of the conditional jump where they are predicated. An instruction is considered movable, if it can be predicated and does not change the control flow. Furthermore, it must not write a predicate which is used as condition of the jump or as guard of an ITE block (in case of partial if-conversion). Moreover, it must not depend on an instruction which is non-movable to simplify the dependency analysis. If a non-movable instruction is found in one block it proceeds with instructions from the other block. The algorithm stops either if no more movable instructions are found or if a configurable threshold (3 in the example) is reached.

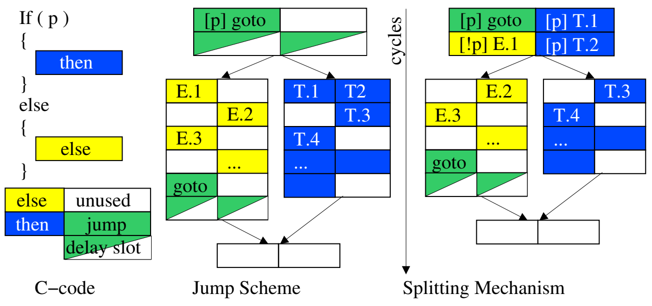


Figure 4. *Splitting mechanism, two issue slots*

4.3 Retargeting Formalism

An evaluation of several processors for different domains showed that processors featuring PE can be grouped accordingly to the location the guard is stored in. We obtained the following three categories which are all supported by our approach:

1. Processors using general purpose registers.
2. Architectures using dedicated registers and
3. those that use condition flags stored in a status register.

The first retargeting step is to configure the cost computation. Three boolean parameters for the PE engine specify to which of the above classes the target architecture belongs. Another boolean parameter indicates whether the architecture directly supports negated conditions. Furthermore, the jump penalties J_x need to be provided.

Moreover, some of the architectures can execute a wide subset of their instruction set conditionally while others offer only for a few instructions a predicated version. In order to determine whether an instruction or a basic block can be conditionally executed by the target processor, we employ the generated tree covering based code selector [28]. In CoSy, the code selector description consists of so called code selection rules. Basically, each rule describes how a certain IR operation is mapped to the target assembly code. For retargeting the PE, each rule of the code selector that can emit code which is conditionally executable, has to be annotated. Fig. 5 shows two examples for the TriMedia [8] processor. The rule covering a plus node can be conditionally executed (denoted by peinclude). The other rule is missing that annotation and thus, is assumed to be not conditionally executable by default. Consequently, if one of the rules covering the then or else block is missing that annotation, if-conversion cannot be applied to the corresponding if-statement. Furthermore, the instructions of such a rule cannot be moved by the splitting mechanism.

```
RULE mirPlus(s1:reg_nt,s2:reg_nt) -> d:reg_nt;
CLASS peinclude;
EMIT {
 print_with_condition("iadd  %s   %s -> %s",
       REGNAME(s1),REGNAME(s2),REGNAME(d));
}
RULE o:mirIntConst -> d:regi;
EMIT {
 print("uimm( %s ) -> %s ",o.Value,REGNAME(d));}
```

Figure 5. *Annotated TriMedia code selector rule*

```
// Register r0 is always zero and r1 always one
INSTRUCTION peSetCondition (cond:reg_nt) -> d:reg_nt;
EMIT {
 print("IF %s  iadd  r1   r0 -> %s",
        REGNAME(cond),REGNAME(d));
}
INSTRUCTION peResetCondition (cond:reg_nt) -> d:reg_nt
EMIT {
 print("IF %s  iadd  r0   r0 -> %s ",
       REGNAME(cond),REGNAME(d));
}
INSTRUCTION peNegateCondition (s:reg_nt) -> d:reg_nt;
EMIT {
 print("IF r1 bitinv %s -> %s",REGNAME(s),REGNAME(d));
}
INSTRUCTION peBranchAlways(label:BBlock);
EMIT {
 print("IF r1 ijmpi ( %s )",label);
}
INSTRUCTION peBranchCond (cond:reg_nt,label:BBlock);
EMIT {
 print("IF %s  ijmpi ( %s )",REGNAME(cond),label); }
```

Figure 6. *PE instruction rules for the TriMedia*

For the code generation, the emitter must take care to print the correct assembly syntax (see Fig.5) in case the rule is used in a predicated block. The if-statement rules emit the code for the selected ITE scheme. All ITE schemes can be generated using the following instructions:

peSetCondition conditionally sets a predicate to true
peResetCondition conditionally sets a predicate to false
peNegateCondition conditionally inverts a condition
peBranchAlways unconditional jump instruction
peBranchConditional conditional jump instruction

We provide rule templates for these instructions which have to be filled in with the assembly code that has to be emitted. Fig. 6 shows the filled templates for the TriMedia processor. No other information, apart from those described above, needs to be provided to retarget the extension. Thus, PE can be quickly integrated into any CoSy based compiler with minimum effort.

5 Code Generation Flow

Due to the modular concept of CoSy, we can intertwine the standard backend components (tree pattern matcher, scheduler, register allocator) with the PE modules. Fig. 7 depicts the backend of a CoSy compiler with PE support.

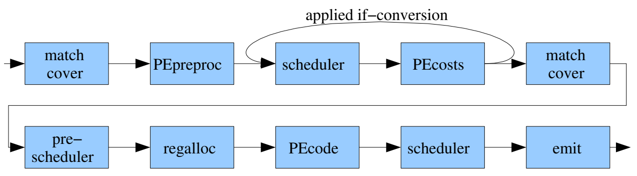


Figure 7. *CoSy compiler backend with PE support*

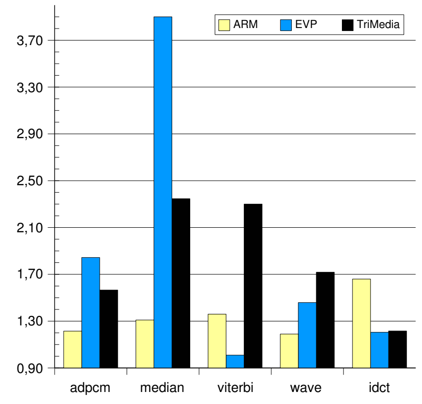
After an initial code selection with the standard tree pattern matcher (match, cover) the engine *PEpreproc* builds ITE trees (representing the structure of nested ITE statements) and determines those if-statements to which if-conversion can be applied. Reasons for an exclusion can be multiple incoming control flow edges of the `then` or `else` block as well as non predicable code in an ITE block. The latter is detected utilizing the already described rule annotations. If a basic block is covered by a rule emitting non conditionally executable code, an infinite cost value is assigned to the PE schemes of the corresponding if-statement. Then the costs of the different schemes are calculated and the scheme selection is performed by the engine *PEcosts* (section 4.1). This engine is coupled to the normal scheduler of CoSy. In the first iteration, the scheduler calculates the execution times of each basic block. These are used to compute the costs for the implementation with jump instructions. Afterwards, *PEcosts* instructs the scheduler to merge the `then` and `else` blocks of the innermost statements. The scheduler parallelizes them and provides cost estimates of the block merger. Thereafter, *PEcosts* selects the schemes according to the calculated costs. After the final code selection and register allocation the engine *PEcode* generates the code for the chosen schemes using the above mentioned instructions. The code does not only depend on the scheme but also on the order of the `then` and `else` block. The splitting mechanism operates at the last scheduler run and targets all if-statements to which if-conversion could not be applied. Apart from the compiler's dataflow information, it uses the annotations by the tree pattern matcher whether an instruction is predicable or not. Finally, the code is emitted.

Our approach requires limited retargeting information, also due to the coupling to existing compiler backend modules. These are typically part of any retargetable compiler. Consequently, our approach can be easily incorporated into other compiler platforms as well.

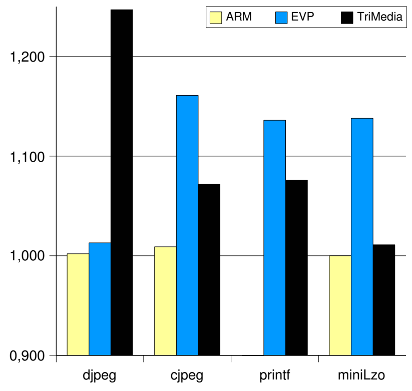
6 Experimental Results

The proposed technique was successfully integrated into CoSy compilers for the Adelante^{TM}VD32040 Embedded Vector Processor (EVP) [9] and the TriMedia from NXP Semiconductors [8] as well as the ARM9 [12]. The required retargeting effort for PE support was one day for each compiler. All three architectures can execute almost all their instructions conditionally. The TriMedia can use any of its 128 general purpose registers to store the predicate, whereas the EVP features eight dedicated predicate registers. The negated predicate has to be computed explicitly for both processors. The ARM uses condition code flags for predication. It can store one condition at a time in the status register and supports negation. The maximum VLIW parallelism available in the EVP equals five vector-, four scalar-, three address-operations and loop-control. The TriMedia can process up to five operations in parallel. The EVP jumps have 5-7 delay slots while the TriMedia jumps have two. In contrast, the ARM is a RISC like core. Since the ARM has no delay slots the splitting mechanism was disabled. The only benefit by PE for the ARM lies in the elimination of jump instructions.

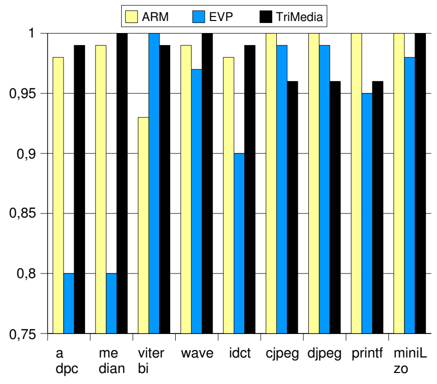
The benchmarks consist of some smaller, typical signal processing kernels (up to 30 ITE statements) as well as some larger and more complex applications (up to 2000 ITE statements). The total number of if-statements vary between the compilers due to their different design and integrated optimizations. If not stated otherwise, we used the test data that comes with these benchmarks for our measurements and optimized for the worst-case execution time. For the small benchmarks, *PEpreproc* determines that on average 80% of all if-statements can be considered for PE, the only exception being the `viterbi` [29] for the EVP with no predicable if-statements. Almost all these if-statements could finally be converted for the EVP, whereas the TriMedia could not convert all of them. This is mainly due to the higher degree of parallelism the EVP offers over the TriMedia. Thus, the chance is higher in TriMedia for resource conflicts resulting in longer schedules and hence, higher costs for predicated if-statements. Consequently, more if-statements are split for the TriMedia than for the EVP. Fig. 8(a) shows high speedups for the VLIW processors, whereas the ARM shows smaller speedups. The programs `cjpeg` and `djpeg` [30] feature a large amount of if-statements (around 2000), however only approximately 15% of them were recognized by *PEpreproc* for if-conversion. Finally, only $6 - 10\%$ of all if-statements could be converted by the compilers. Here, the splitting mechanism proves advantageous and handles nearly 80% (EVP) and 60% (TriMedia) of all if-statements. The ARM shows only marginal speedups due to the disabled splitting mechanism, but EVP and TriMedia show good speedups for both `cjpeg` and `djpeg` (Fig. 8(b)). The obtained speedups are less significant than for the small kernels since a large amount of cycles are spent in the runtime library for file operations. Considering the `printf` (implementation is shipped with CoSy) application, it contains many if-statements (around 100), approximately 17% are converted and around 60% are split by the EVP and TriMedia compilers. No results are reported for the ARM, since it could not be compiled due to a different runtime library setup. For `miniLzo` [31], although it contains around 80 if-statements, only a few could be converted. A look into the source code revealed that the if-statements either contain function calls or goto statements. Both is not allowed by *PEpreproc* and thus, no performance improvement can be obtained. However, except for the ARM, the splitting mechanism can be applied again and optimizes almost all if-statements.



(a) Speedup for small benchmarks



(b) Speedup for large benchmarks



(c) Code size factors

Figure 8.

On average, speedups of 1.2 for the ARM9, 1.5 for the EVP and 1.47 for the TriMedia can be obtained. For the code size, PE typically saves some instructions (jumps and nops), but may also generate new ones (e.g. negated conditions). In general, code size is slightly reduced (see Fig. 8(c)).

The optimization algorithm itself has linear complexity ($O(n)$ worst case for n ITE statements). Furthermore, it requires one additional tree pattern matcher pass and two additional scheduler passes (given the dataflow information, both have $O(n)$ worst case for n IR nodes in the ITE statements). Thus, the total complexity remains linear. In practice we found that the increase in compile time due to the additional passes is negligible.

7 Conclusions

In contrast to previous, largely target specific, code optimizations for Predicated Execution, we propose a *retargetable* approach in order to enable PE for a wide range of processor architectures at limited manual effort. This is achieved by a *retargetable Predicated Execution* extension for the CoSy compiler development system. This concept has been proven by generating PE enabled compilers for embedded processors with different PE configurations. For all processors, we generally achieved good speedups.

In future, we will integrate PE in the Compiler Designer tool [27] which is part of CoWare's Processor Designer to enable a complete and retargetable path from a single processor model, written in the LISA 2.0 Architecture Description Language, to a C compiler with PE optimization. Additionally, we will concentrate on further improvements in code quality. For instance, conditions of if-statements are often composed of expressions combined with boolean operations which is mapped onto several nested ITE statements. If the evaluation of the individual expressions is free of side effects, they can be evaluated in parallel. This idea could be implemented by a new scheme for the PE engines. Furthermore, a mechanism to enforce PE for certain ITE statements might be useful to enable other optimizations, e.g. software pipelining, which are blocked by control flow.

References

[1] M. Gries, K. Keutzer, H. Meyr, et al.: *Building ASIPs: The Mescal Methodology* Springer, 2005

[2] J.A. Fisher: *Customized Instruction Sets for Embedded Processors*, Design Automation Conference (DAC), 1999

[3] A. Oraioglu, A. Veidenbaum: *Application Specific Microprocessors (Guest Editors' Introduction)*, IEEE Design & Test Magazine, Jan/Feb 2003

[4] M. Hohenauer, C. Schumacher, R. Leupers, G. Ascheid, H.Meyr and H. van Someren, *Retargetable code optimization with SIMD instructions*, Proceedings of the 4th international conference on Hardware/Software Codesign and System Synthesis, 2006

[5] J. R. Allen and K. Kennedy and C. Porterfield and J. Warren, *Conversion of control dependence to data dependence*, Proc. of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1983

[6] Associated Compiler Experts ACE: CoSy compiler platform, www.ace.nl

[7] CoWare Inc.: Processor Designer, www.coware.com

[8] NXP Semiconductors: Nexperia PNX 1500 family and TriMedia media processors, www.nxp.com

[9] K. van Berkel, F. Heinle, P. Meuwissen, K. Moerman, M. Weiss, *Vector Processing as an Enabler for Software-Defined Radio in Handheld Devices*, EURASIP Journal on Applied Signal Processing, 2005

[10] Texas Instruments: The VelociTI architecture of the TMS320C6x, www.ti.com

[11] ARC International: www.arc.com

[12] Advanced RISC Machines Ltd: www.arm.com

[13] N. J. Warter, D. M. Lavery, and W. W. Hwu, *The benefit of Predicated Execution for software pipelining*,Proceedings of the 26th Hawaii International Conference on System Sciences, 1993

[14] S. A. Mahlke and D. C. Lin and W. Y. Chen and R. E. Hank and R. A. Bringmann, *Effective compiler support for predicated execution using the hyperblock*, 25th Annual International Symposium on Microarchitecture,1992

[15] D. I. August and W. W. Hwu and S. A. Mahlke, *A framework for balancing control flow and predication*, Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture,1997

[16] K. M. Hazelwood and T. M. Conte, *A Lightweight Algorithm for Dynamic If-Conversion during Dynamic Optimization*, International Conference on Parallel Architectures and Compilation Techniques,2000

[17] V. Bala and E. Duesterwald and S. Banerjia, *Dynamo: a transparent dynamic optimization system*, Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation,2000

[18] L.Carter and B. Simon and B. Calder and L. Carter and J. Ferrante, *Path Analysis and Renaming for Predicated Instruction Scheduling*, International Journal of Parallel Programming,2000

[19] W. Chuang and B. Calder and J. Ferrante, *Phi-Predication for light-weight if-conversion*, Proc. of the intern. symposium on Code generation and optimization (PLDI), 2003

[20] M. Smelyanskiy and S. A. Mahlke and E. S. Davidson and H. S. Lee, *Predicate-aware scheduling: a technique for reducing resource constraints*, Proc. of the intern. symposium on Code generation and optimization (PLDI),2003

[21] Free Software Foundation, *GNU Compiler Collection*,gcc.gnu.org

[22] R. Leupers, *Exploiting conditional instructions in code generation for embedded VLIW processors*, Proceedings of the conference on Design, automation and test in Europe (DATE), 1999

[23] P. Mishra, N. Dutt, A. Nicolau: *Functional abstraction driven design space exploration of heterogenous programmable architectures*, Int. Symp. on System Synthesis (ISSS), 2001

[24] A.Kitajima, M. Itoh, J. Sato, A. Shiomi, Y. Takeuchi, M. Imai: *Effectiveness of the ASIP Design System PEAS-III in Design of Pipelined processors*, Asian and South Pacific Design Automation Conference (ASP-DAC),2001

[25] Trimaran: An Infrastructure for Research in Instruction-Level Parallelism, http://www.trimaran.com

[26] Tensilica Inc.: Xtensa C compiler, www.tensilica.com

[27] M. Hohenauer, O. Wahlen, K. Karuri, et al.: *A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models*, Design Automation & Test in Europe (DATE), 2004

[28] S. S. Muchnik: *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, 1997

[29] DSPstone: http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE

[30] Mediabench: http://euler.slu.edu/ fritts/mediabench/mb1

[31] miniLZO: http://www.oberhumer.com/opensource/lzo