IR-Level Versus Machine-Level If-Conversion for Predicated Architectures

Alexander Jordan Nikolai Kim Andreas Krall
Institute of Computer Languages,
Vienna University of Technology
{ajordan,kim,andi}@complang.tuwien.ac.at

ABSTRACT

If-conversion is a simple yet powerful optimization that converts control dependences into data dependences. It allows elimination of branches and increases available instruction level parallelism and thus overall performance. If-conversion can either be applied alone or in combination with other techniques that increase the size of scheduling regions. The presence of hardware support for predicated execution allows if-conversion to be broadly applied in a given program. This makes it necessary to guide the optimization using heuristic estimates regarding its potential benefit. Similar to other transformations in an optimizing compiler, if-conversion inherently suffers from phase ordering issues. Driven by these facts, we developed two algorithms for if-conversion targeting the TI TMS320C64x+ architecture within the LLVM framework. Each implementation targets a different level of code abstraction. While one targets the intermediate representation, the other addresses machine-level code. Both make use of an adapted set of estimation heuristics and prove to be successful in general, but each one exhibits different strengths and weaknesses. High-level if-conversion, applied before other control flow transformations, has more freedom to operate. But in contrast to its machine-level counterpart, which is more restricted, its estimations of runtime are less accurate. Our results from experimental evaluation show a mean speedup close to 14% for both algorithms on a set of programs from the *MiBench* and *DSPstone* benchmark suites. We give a comparison of the implemented optimizations and discuss gained insights on the topics of if-conversion, phase ordering issues and profitability analysis.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*Optimization*

General Terms

Algorithms, Performance

Keywords

if-conversion, VLIW architectures, phase ordering

ODES'13 , February 23 - 24 2013, Shenzhen, China

1. INTRODUCTION

In many embedded applications, very long instruction word (VLIW) architectures are popular due to their reduced hardware complexity (compared to superscalar alternatives) combined with potential runtime and resource efficiency. However, the performance of a VLIW machine greatly depends on the compiler. It has to produce highly optimized code in order to avoid performance bottlenecks. Therefore, sophisticated optimization approaches are required when targeting a VLIW architecture. This is especially true for the instruction scheduling phase, which needs to hide the delay of branches and other long-latency instructions. Code enlargement techniques are beneficial for improving the efficiency of the scheduler, but inherently expose a higher register usage. This, in turn, can cause the register allocator to insert spill code, thus having an adverse effect.

In this paper we describe and compare against each other, two if-conversion variants implemented within the LLVM compiler framework [18]. This modern framework provides a wide collection of optimizations on its intermediate representation together with a relatively flexible back-end code generator structure. This design makes it easy to add new architectures, but on the other hand restricts the scope of optimizations. As a result, some common code transformation passes in LLVM are implemented in two different versions, targeting different code representations.

For implementation and evaluation we targeted the Texas Instruments (TI) TMS320C6000 DSP family [19] and concentrated especially on C64x and C64x+ VLIW CPUs. All presented benchmarks were evaluated using TI's instruction set simulator configured for cycle accuracy, but without cache simulation (i.e. perfect instruction and data caches are assumed). Even if primarily targeting a VLIW architecture, we believe however, that our implementations are general enough to be adaptable to any predicated architecture with only minor modifications.

This paper is organized as follows. In the next section we present a selection of works published in the past and related to this paper. Section 3 introduces the target architecture, section 4 first depicts the particular problem more concretely and then presents details of the algorithms we have developed. The evaluation environment is addressed in section 5, which also shows and relates results for our collection of benchmarks. Section 6 concludes and looks at issues related to if-conversion that we intend to study closer in the future.

2. RELATED WORK

If-conversion is related to region formation and region enlargement techniques. These share the goal of extending the scope of scheduling beyond a single basic block to a region, ideally, as large as possible. The first region scheduling technique was trace scheduling introduced by Fisher in [7]. A trace is a cycle-free sequence of blocks along a frequently executed path, with multiple entrances and exits. Due to its flexibility, trace formation is straightforward, but maintaining data consistency (especially at side entries) is complicated. Traces were followed by superblock scheduling, which restricts entrances to a single entry into the first block. Superblocks and their related region enlargement technique tail duplication, are described in a paper by Hwu et al. [8]. Mahlke et al. [11] combine the idea of if-conversion with superblock formation to produce a hyperblock. Doing so eliminates side entrances and leads to an increase in region size and thus scheduling efficiency, while avoiding the complexity of trace scheduling.

Fang [6] explains the idea of predication and presents an if-conversion algorithm along with an optimization that aims to assign predicates early. However, the paper contains no results and implicitly suggests the algorithm to be run after register allocation. Park et al. [12] describe a similar if-conversion algorithm, formally prove its correctness and point out some of the performance degrading pitfalls. Warter et al. propose reverse if-conversion [16] that makes it possible to undo some of the earlier conversion steps, countering the greedy behaviour of forward if-conversion. A framework incorporating both, forward and backward if-conversion mechanisms is presented by August et al. in [2]. The authors advise to if-convert aggressively and early, by using profitability estimation heuristics. Later in the optimization chain some of these conversions are undone, in case the profitability has been overestimated. The verification of the estimation together with the if-reversion is driven by the instruction scheduler. This suggests both steps to be integrated. While sophisticated and effective, such a framework also exposes a much higher complexity compared with non-integrated approaches. Leupers [10] describes an efficient two-pass, dynamic programming inspired algorithm for if-conversion. His approach targets fully predicated VLIW architectures, works on a high-level IR and optimizes for worst-case execution time. Stoutchinin and Gao [14] also suggest to run if-conversion early. They extend single assignment form (SSA) to create ψ-SSA, which allows application of traditional SSA optimizations after if-conversion. However, no concrete implementation is proposed and only results about code size increase are presented. Bruel [3] also describes an SSA-based implementation targeting a partially predicated VLIW machine. A key aspect is the preprocessing of regions to eliminate side entries by duplicating blocks and thus increasing the potential for if-conversion. While phase ordering is not addressed, this technique results in substantial speedup for the presented benchmarks. Ebner et al. [5] also target a VLIW architecture and describe a leveraging approach, but perform if-conversion after the register allocation as a late optimization. Winkel [17] combines global code motion with decisions about speculation and predication for the Itanium processor. By formulating the problem as an integer linear program, his solutions are optimal, but limited to regions with a size of up to 200 instructions.

Different approaches have been proposed to tackle the problem of phase ordering in traditional compilers that apply optimizations in a sequence of analyses and transformations. A class of meta optimizers attempts to improve the ordering of optimizations in a program-specific way. For example, this can be done by either searching the space of compilation sequences [1], using genetic algorithms [4], or by machine learning [13]. Systems like Denali [9] and equality saturation [15] approach the problem differently. Instead of applying transformations in a sequence, they represent blocks or programs and their various alternative optimizations in a graph and try to select a near-optimal (as far as the cost model goes) set of transformations from it.

3. TARGET ARCHITECTURE

As mentioned above, the Texas Instruments C64x and C64x+ processors served as a target for our implementations. Within the family of 32-bit C6000 VLIW processors, the C64x+ has an instruction set that contains DSP instructions, a SIMD instruction subset and hardware loops. Additionally to the normal 32-bit wide instructions, the C64x+ also has 16-bit wide compact instructions. C6000 is a dual-cluster architecture. That means, to increase the number of functional units without adding a significant amount of interconnect logic, registers are only connected to a subset of the available units. Each cluster contains four functional units: L, S, M, and D. Units L and S are the integer arithmetic logic units (ALUs); S additionally performs control flow changes. Loads, stores and complex address computations are handled by the D unit. Unit M is exclusively used for all simple and complex (e.g. dot product) multiplications. The units of each cluster are connected to their own register file containing 32 general purpose registers.

An instruction packet can contain up to 8 single operations, in the best case utilizing all available functional units in parallel. All units are fully pipelined, i.e. a new instruction can be issued every cycle. Operations within an instruction packet are subject to various structural restrictions. For example, the number of simultaneous reads from the same register is limited to two and only one cross-cluster register read may occur per cluster. With one exception, all instruction latencies are explicit and it is the job of the programmer or compiler to account for the required delay slots and fill them with useful instructions. 16-bit multiplies have a latency of two, 32-bit multiplies take four, loads take five, and branches take six cycles to complete. Most other instructions have one cycle latency. The exception to the statically exposed latency of the instruction types above, is the one cycle long inter-cluster cross path stall, which is dynamically inserted by the hardware. It occurs since results need to complete the write-back stage in the pipeline, before they can be read from the other cluster. All latencies described above are due to the micro architecture of the C64x and happen within the CPU. Delays from external memory, which may occur when loading instructions or data, are implementation specific. These external latencies depend on the properties and availability of caches and are commonly orders of magnitude higher than internal ones.

The C64x+ is fully predicated. Essentially every instruction that is not a compact instruction and not a call, can be predicated with a register and boolean flag that may negate the predicate. As a subset of the general purpose registers, three registers per cluster (six in total) can be used as predicates.

4. IF-CONVERSION

Among compilation techniques for VLIW architectures if-conversion with the help of predication plays an important role. Not only is it able to remove hard to predict conditional branches, but also, more importantly, it helps increasing the size of basic blocks. This gives instruction scheduling, which in many compilers is not able to schedule beyond basic block boundaries, more freedom to reorder instructions and thus utilize the hardware more efficiently.

The use of predication for if-conversion has been described in [12] and [6]. The basic idea is to encode a special flag into a machine instruction, that determines at runtime whether the instruction is to be executed or ignored. An instruction set architecture (ISA) can either be predicated fully, which means that practically every instruction supports predication, or partially, which restricts the use of predication to a few special operations (e.g. conditional move instructions).

```
int foo (int z, int val) {
  int x = val;
  if (z > 10) x = x + 1;

  return x * x;
}
```

Figure 1: Simple C-code example.

```
foo:
   move val, t1
   cmp_leq z, 10, t2
   branch_eq t2, 1, tail

body:
   add t1, 1, t1

tail:
   mul t1, t1, t1
   ret t1
```

```
foo:
        move val, t1
        cmp_leq z, 10, p
  [!p] add t1, 1, t1
        mul t1, t1, t1
        ret t1
```

Figure 2: Two variants of (pseudo) assembler code, left using a conditional branch, right using a predicate.

Figure 1 shows a simple C function using an if-statement. Two corresponding (pseudo) assembler variants are shown in Figure 2. The left variant uses a conventional conditional branch instruction (together with two fallthrough transitions), while the right one incorporates predication. As can be seen, the predicated variant has a lower instruction count, no control flow between basic blocks (there is actually only one basic block) and presents a larger scope of 5 instructions to the scheduler. Such a code transformation is achieved by the proper replacement of conditional statements with predicate definitions and uses and is widely known as (forward) if-conversion.

4.1 Implementation

The main difference between the if-conversion implementations we compare, is when they take place during compilation (see Figure 3). Early if-conversion targets the LLVM intermediate representation (LLVM IR), which itself is architecture independent. On the other hand, machine-level if-conversion occurs after instruction selection, but before register allocation. In both cases, if-conversion itself is specifically scheduled early, so that subsequent optimization passes can take advantage of the larger blocks and simplified control flow.

Machine-level if-conversion is implemented in a straightforward manner. Since it modifies the target specific instructions directly, predicate registers can be defined and used anywhere in the function. Also, machine instructions provide definitive information (such as instruction size, latency, possible side effects), which has to be estimated on the intermediate level, and thus allow a more exact cost-benefit analysis.

Due to the lack of predication in the LLVM IR, adding IR-level if-conversion to LLVM is more complicated and involves several phases. Speculation without predication is supported with the built-in *select* instruction, but to extend if-conversion to instructions that may cause side effects and thus need to be predicated, our optimization pass has to make use of target specific *intrinsics*[1]. It is only during the *lowering* phase (when IR instructions are *lowered* to target specific machine code), that predicates get eventually applied to instructions based on the predication intrinsics. The reason why it still makes sense to implement if-conversion at the IR level is the flexibility it gains. All optimizations available for the intermediate representation can be applied to the result of if-conversion. Furthermore, as an IR-level pass, it can be scheduled before other transformations that might inhibit if-conversion. Virtual registers, which occur in both representations, have the advantage of being more flexible to handle during conversion; retaining SSA form greatly simplifies implementation and allows existing SSA transformations to be used after if-conversion. A major disadvantage of any conversion using virtual registers is however, that the register pressure cannot be computed exactly and needs to be estimated heuristically. This creates a potential for 'over-converting', i.e. if-converting too aggressively, which leads to a performance degradation.

Tail duplication makes it possible to remove side entries into regions that would otherwise be profitable to convert. While duplicating instructions has to be done under consideration of code size enlargement, it increases the scope of convertible patterns by some degree. Ultimately it allows, if desired, more aggressive code transformations. We only perform tail duplication on the machine level, because of the coarse estimation of code size that is available on the IR level.

4.2 Iterative if-conversion

Both algorithms process the control flow graph of a function in a bottom-up manner. Basic blocks of a given function are analyzed, information about contained machine instructions is extracted and recorded. After analysis, each basic block is associated with a record that contains information including whether its instructions can be speculated (in SSA form) or need to be predicated because of side effects. Note that, because of the overhead involved, function calls are never considered for predication. Additionally, the number of instructions and the estimated execution cycles for a basic block are recorded.

Given this information, the algorithms extract patterns suitable for conversion by inspecting structural relationships

[1]Intrinsics can be used to model features of a specific architecture within the otherwise target-independent LLVM IR.

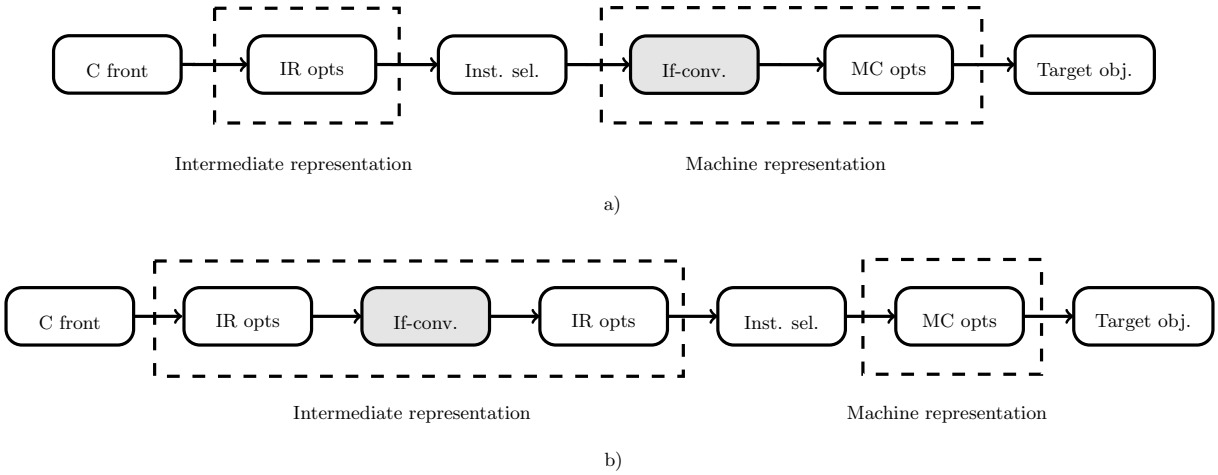

Figure 3: Simplified LLVM compiler pass pipeline including machine-level (a) and IR-level (b) if-conversion.

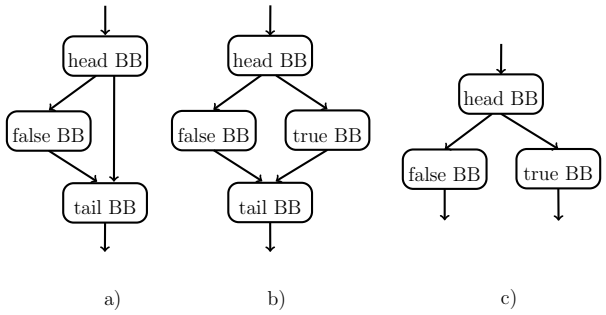

Figure 4: Basic patterns used for if-conversion: a) if-then triangle, b) if-else diamond, c) open if-then pattern.

(such as dominance and successor relationships) between basic blocks. Figure 4 shows the basic patterns used for conversion. Applied iteratively, these serve as building blocks, which together with tail duplication cover all conditional-branch-based control flow in a program. Extracted patterns are stored within a list of candidates and ordered descending by the execution frequency of the head block. This frequency is either supplied by the profiling information or computed statically by an estimator.

After extraction, candidate regions are examined for their conversion profitability (see section 4.3 for details) and eventually converted. Both algorithms are organized as an iterative process, which runs until a specified threshold is reached or there is nothing more to convert. Thus, a given sequence of basic blocks can be converted more than once. When if-conversion needs to handle instructions or whole basic blocks that already make use of predicates, a common solution is to connect these by logical operations, which depend on the original control semantics.

Algorithm 1 describes the general structure of the if-conversion used for the machine level and Figure 5 illustrates an example. Assuming blocks *for.cond* and *land.end* use variables p and q as branching conditions respectively, iterative if-conversion first collapses the diamond containing *for.cond*, *land.rhs*, *BB#6*, and *land.end*. Here (assuming *land.rhs* to be the *true* destination of the corresponding branch), predicate p is assigned to *land.rhs*, while instructions contained in *BB#6* are assigned the inverted predicate *!p*. The second step assigns predicate q or *!q* (depending whether *for.body* is the *true* successor of *land.end*) to *for.body* and produces a single basic block (*for.cond)* for the whole loop.

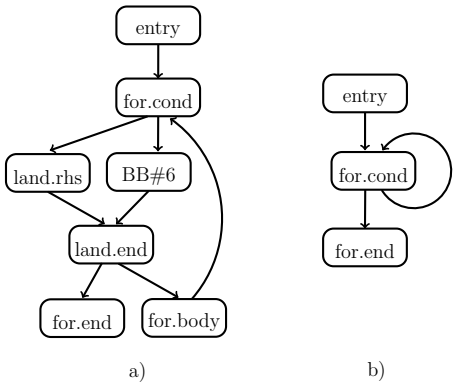

Figure 5: Iterative if-conversion: a) original CFG fragment, b) resulting CFG after two conversions.

4.3 Profitability heuristics

To estimate profitability of conversion, both implementations rely on simple heuristics that take into account the size of basic blocks and their estimated execution time based on the instruction latencies (*cycles*). Additionally, execution frequencies of basic blocks and edges between them contribute to the estimate as *weights*. These are either obtained from profiling information or can be statically estimated.

To decide whether it is profitable to perform if-conversion, the cost of a candidate region C with n blocks (which are all convertible) is estimated as

$$Cost(C) = branch\_cost(C) + \sum_{i=1}^{n} cycles(Block_i) * Weight_i$$

Algorithm 1 Iterative if-conversion

```
while true do
    patterns ← initialize as empty list
    for each block BB in the current function do
        if BB is head of a pattern then
            P ← pattern for BB
            if P is convertible then
                patterns ← patterns + P
            end if
        end if
    end for
    Sort patterns by execution frequency
    converted ← false
    for each pattern P contained in patterns do
        if is profitable to convert P then
            if P has side-entries then
                Run tail-duplication on P
            end if
            head ← head basic block of P
            preds ← branching predicates for head
            Predicate blocks of P using preds
            Merge blocks of P
            if profile information available then
                Update profile information
            end if
            converted ← true
        end if
    end for
    if converted is false then
        break
    end if
end while
```

and compared to an estimate that takes if-conversion into account:

$$Cost_{conv}(C) = cycles(\bigcup_{i=1}^{n} Block_i) * Weight_C$$

Cycle estimations on both levels suffer from incomplete information. For example, since register allocation happens at a very late stage, most instructions operate on virtual registers and thus, neither algorithm can take register dependencies or spill cost into account. Furthermore, since IR-level if-conversion precedes instruction selection, instruction properties (i.e., the concrete number and their latency) are also based on estimates there. Ultimately, both algorithms make use of constant factors to balance their estimated cost against real-world experience. These constant factors are based on the behavior of small test programs and inherently depend on how effective the code generator can exploit ILP in the converted basic blocks. Thus, any major change in the code generator would mean that the constant factors need to be adapted as well. To generally avoid adverse effects from resource conflicts (registers or functional units) within converted regions, the size of the resulting block needs to be controlled. In case of the C64x+ architecture, the number of predicate registers is an important limit. Thus our implementation rejects conversions that require more than three predicates at the same time.

5. EVALUATION

5.1 Benchmark environment

Our evaluation results are based on well-known benchmark collections such as *MiBench*, *DSPstone* and *Olden*. Additionally, two micro-benchmarks (both sort algorithms) and two programs from *BenchmarkGames* have also been tested. Especially *MiBench* presents a wide set of benchmarks covering automotive, security, office and telecommunication application domains. Some tests had to be omitted however, due to headers or system calls being unavailable in the used standard library. Also, we excluded tests that exhibit an extremely simple structure and thus low cycle count (below a threshold of 1000 cycles). An `adpcm` benchmark is contained in the *DSPStone* and *MiBench* suites. Although the programs are different, the results only include the latter one, to prevent skewing of the average. For larger test programs and when possible, the small input data set was used. But due to the limited throughput of the simulator, the size of the input had to be further restricted in some cases. Note, that with some benchmark programs (e.g. `MB-qsort`), computation is mainly performed within standard library calls. Since we cannot optimize the standard library itself, these programs are expected to have low potential for if-conversion.

All results were obtained using TI's instruction set simulator, configured for cycle-accurate timing but without modeling of caches. To investigate a possible impact on instruction cache misses, we measured the code size increase instead.

For benchmarks, both presented if-conversion implementations make use of execution frequencies that we obtained by profiling the respective full input data set. The profiling framework provided by LLVM targets the intermediate representation and could be used by our if-conversion algorithm at that level directly. Using the same profile information at the machine level turned out to be troublesome and led to significant imprecisions in the execution frequencies for some benchmark programs. At the time of LLVM version 2.9, machine-level profile information is available, but not accurately maintained by some transformation passes that change the CFG. As a workaround, additional instrumentation and profile loading passes have been added at the machine level.

Within LLVM, most of the built-in optimizations are done on the IR level. We apply standard optimizations (e.g. inlining, redundancy elimination) independent of the if-conversion algorithm used. The back-end eventually performs cluster assignment and scheduling using a unified assign-and-schedule (UAS) algorithm. Target specific loop optimizations, such as software pipelining, are not available and thus have no impact on the results. One of LLVM's standard passes, which tries to eliminate unnecessary branches in the CFG (`-simplifycfg`) is specifically omitted, since it performs limited if-conversion using speculation and the *select* instruction. It is thus superseded by our if-conversion.

5.2 Results

Table 1 gives an overview of the benchmark results. The geometric mean of the speedup achieved over all benchmark programs is 13.82% for the IR-level and 14.38% or 13.85% for the machine-level if-conversion, when using profile information or relying on a profile estimator respectively. This speedup is compared to a baseline version compiled with the

same optimization passes but without if-conversion. Compared to related work, the lower speedup can be explained due to our back-end code generator, which is at an experimental stage and cannot fully exploit the higher potential instruction level parallelism (ILP) that if-conversion provides. Bruel in [3] reports a mean improvement of 25% on a comparable, but narrower (4-issue, non-clustered) VLIW architecture. In [5], Ebner et al. achieve a speedup of 18% also on a 4-way VLIW processor, but performing their if-conversion after register allocation. In both reports the speedup of the `adpcm` benchmark is significantly higher than that of all other benchmarks (factor 3 and factor 2.82 respectively). We also see `adpcm` as an outlier in our results and know from `simplifycfg`, which only yields a significant improvement of 35% for this benchmark, that it has a strong disposition to the conversion of its small conditional blocks. Thus `adpcm` is, in our opinion, a better indicator for ILP-related optimizations in the back-end code generator than for accurate if-conversion decisions.

Even though the average speedups of both if-conversion implementations are very close, the difference for individual programs goes up to 28 percent points in one case (`MB-CRC32`). To gain more insight we turn to the details given in Table 2. Its columns list the number of basic blocks before if-conversion, number of converted (and thus removed) blocks, as well as tail duplications performed by the IR-level and machine-level if-conversion respectively. Contrary to what one might expect, the lack of speedup the IR-level if-conversion exhibits in `MB-CRC32`, actually is a case of optimizing too aggressively. `MB-stringsearch` behaves differently: while at the machine level 16 conversions are being applied, the IR-level if-conversion only performs a single one. All other possible if-conversions (51 decisions in total) are not applied, as they are prevented by the cost function. We can also see in Table 2 that a single optimization decision can have substantial impact on the outcome. Larger benchmark programs like `susan` show that the number of blocks and opportunities to perform if-conversion is higher at IR level. Upon reaching the machine-level if-conversion, LLVM's code transformations implicitly change the control flow of a program. Working on the IR-level thus increases the chances of finding profitable regions for if-conversion. On the other hand it is prone to underestimate the overhead of predication and, as can be seen in Table 2, sometimes performs too many conversions. In the case of the IR-level optimized `MB-CRC32`, the presence of one adverse schedule completely cancels out any benefit, while the machine-level if-conversion is able to make the right choice.

We do not compare our results to those of the limited IR-level if-conversion performed by LLVM's `simplifycfg`-pass in detail. Since the latter does not target a specific hardware architecture, it can only make conservative decisions and does not use predication at all. Thus `-simplifycfg` performs no conversion for the majority of our benchmarks and only achieves 3.6% average speedup.

Regarding code size, we noticed that even for the largest benchmark programs (see Table 2) it does not increase (it actually decreases slightly), even in the presence of tail duplication. It appears that with if-conversion being as conservative as it is in our case, increased instruction cache misses are no issue.

Another observation we made is that for the majority of presented benchmarks, the quality of if-conversion decisions is not strongly degraded by using static estimates instead of dynamic profiling information. In Table 1 we also give estimator-based results for machine-level if-conversion, for which it is more difficult to obtain accurate profiling data. Three benchmarks (`DSP-fir2dim`, `DSP-matrix2`, `BG-fannkuch`) even perform better when using the estimator. In these cases, the profitability heuristic lacks aggressiveness by either overestimating predication overhead or underestimating scheduling benefits of larger code regions.

Benchmark	$\#BBs$		$\#BB_{conv}$		$\#BB_{dup}$	
	IR	ML	IR	ML	IR	ML
MB-adpcm	82	62	37	21	-	2
MB-bitcount	76	65	6	6	-	1
MB-CRC32	22	19	2	1	-	0
MB-dijkstra	75	57	13	8	-	3
MB-stringsearch	150	107	1	16	-	0
MB-susan	892	542	225	26	-	6

Table 2: Detailed statistics (IR versus machine-level)

6. CONCLUSION AND FUTURE WORK

Both of the described if-conversion algorithms perform reasonably well for the presented selection of benchmarks. The elimination of conditional branches reduces the number of pipeline stalls and merging basic blocks naturally enlarges scheduling regions. This leads to increased scheduling efficiency and resource utilization and results in a significant runtime speedup for the majority of our benchmarks. However, a few test programs do not profit from the if-conversions applied, and due to the imprecision inherent to the profitability heuristics, some even exhibit slight performance degradation. Performing if-conversion early (on the IR level) can in general compete with its machine-level counterpart, but may be too optimistic in some cases. Aggressive if-conversion, as it can be achieved in combination with tail duplication, requires exact scheduling estimates and is better suited for the machine level. This situation would seem to lend itself to a combined approach, which comes at the cost of maintaining separate estimation models and implementation details.

If-conversion as presented here integrates into LLVM's pass pipeline design without the need for comprehensive changes. The algorithms are general enough to support other architectures, regardless of the way predication is handled by the ISA. The most effort would be required for adapting the cost model. Also, with more VLIW specific features being introduced to the framework, we foresee even better support for if-conversion in future versions of LLVM.

We have seen that guiding if-conversion decisions using profile information does not considerably benefit results as long as cost estimation remains imprecise. It is clear that fine-tuning of optimization parameters is not the ultimate solution here and we are currently investigating to integrate if-conversion better with region formation and the scheduler. Another interesting approach, based on the idea by Tate et al. [15], would be to apply if-conversion as part of a non-destructive optimization framework.

Spill code can be another source for performance degradation. Therefore, a reliable method for estimating or computing register pressure is desirable in order to avoid conver-

Benchmark	Baseline (kCycles)	IR-ifconv prof. (kCycles)	ML-ifconv prof. (kCycles)	ML-ifconv estim. (kCycles)	IR-speedup prof. (%)	ML-speedup prof. (%)	ML-speedup estim. (%)
Simple-bubblesort	1.70	1.38	1.55	1.55	**18.46**	**8.37**	**8.37**
Simple-quicksort	1.41	1.17	1.26	1.26	**16.83**	**11.01**	**10.44**
DSP-fir2dim	6.20	5.50	5.66	5.33	**11.26**	**8.76**	**13.92**
DSP-lms	1.22	1.16	0.98	0.98	**5.16**	**19.98**	**19.98**
DSP-fft	90.07	89.42	89.07	89.41	**0.72**	**1.11**	**0.73**
DSP-matrix1	32.84	26.19	24.62	24.62	**20.26**	**25.02**	**25.02**
DSP-matrix2	29.28	22.79	24.09	23.90	**22.18**	**17.72**	**18.38**
DSP-n_complex_updates	1.42	1.42	1.30	1.30	**0.00**	**8.64**	**8.64**
DSP-n_real_updates	1.05	0.69	0.92	0.92	**34.09**	**12.63**	**12.63**
DSP-startup	7.26	5.44	5.68	5.68	**25.08**	**21.80**	**21.80**
MB-adpcm	1136.39	526.81	599.57	599.57	**53.64**	**47.24**	**47.24**
MB-basicmath	1919.05	1757.54	1919.05	1919.05	**8.42**	**0.00**	**0.00**
MB-bitcount	1002.55	820.11	810.59	810.59	**18.20**	**19.15**	**19.15**
MB-blowfish	393.38	395.25	370.44	370.44	**-0.48**	**5.83**	**5.83**
MB-CRC32	7376.18	7376.18	5310.91	5310.91	**0.00**	**28.00**	**28.00**
MB-dijkstra	87441.65	71757.59	79975.22	82901.63	**17.94**	**8.54**	**5.19**
MB-FFT	49896.15	50350.01	49910.29	49910.29	**-0.91**	**-0.03**	**-0.03**
MB-qsort	57888.96	56059.27	57788.96	57788.96	**3.16**	**0.17**	**0.17**
MB-stringsearch	5312.74	5304.64	4568.07	5246.85	**0.15**	**14.02**	**1.24**
MB-susan	51545.54	45264.09	43007.53	43007.53	**12.19**	**16.56**	**16.56**
Olden-bisort	48526.77	46456.66	45792.90	45792.90	**4.27**	**5.63**	**5.63**
BG-fannkuch	238.99	234.09	220.21	214.24	**2.05**	**7.86**	**10.35**
BG-nsieve-bits	32078.41	26262.03	24194.72	24194.72	**18.13**	**24.58**	**20.71**
Average					**13.82**	**14.38**	**13.85**

Table 1: Benchmark results

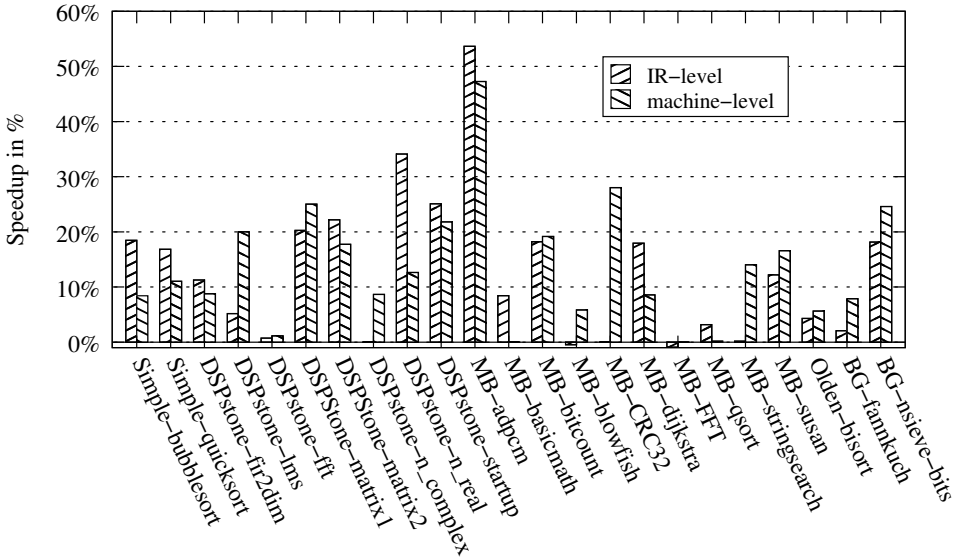

Figure 6: Benchmark results

sions which may increase spill code by an intolerable amount. Thus, we also intend to do a closer investigation on the interaction between if-conversion and the register allocator in the future. We also believe that better results can be achieved by improving the region selection of if-conversion. Increasing its scope creates additional optimization potential by giving more freedom to if-conversion.

Another interesting issue we want to address in the future is the question how much if-conversion influences instruction cache performance on our target. Eliminating control dependencies by removing conditional branches between basic blocks and merging or rearranging them properly increases spatial locality of the code, which in turn is profitable for an effective reduction of cache misses. This is an important issue, when the instruction cache has a predominant impact on the overall system performance.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive and detailed comments on this paper.

This work is supported by the Austrian Science Fund (FWF) under contract P21842, *Optimal Code Generation for Explicitly Parallel Processors*.

7. REFERENCES

[1] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '04, pages 231–239, New York, NY, USA, 2004. ACM.

[2] D. August, W.-M. Hwu, and S. Mahlke. The partial reverse if-conversion framework for balancing control flow and predication. *International Journal of Parallel Programming*, pages 381–423, 1999.

[3] C. Bruel. If-conversion ssa framework for partially predicated vliw architectures. In *ODES Workshop on Optimizations for DSP and Embedded Systems*, pages 5–13, 2006.

[4] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, LCTES '99, pages 1–9, New York, NY, USA, 1999. ACM.

[5] D. Ebner, F. Brandner, and A. Krall. Leveraging predicated execution for multimedia processing. In *IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*, 2007.

[6] J. Fang. Compiler algorithms on if-conversion, speculative predicates assignment and predicated code optimizations. In *LCPC '96 Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, 1996.

[7] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, 30:478–490, July 1981.

[8] W.-M. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The superblock: an effective technique for vliw and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, 1993.

[9] R. Joshi, G. Nelson, and Y. Zhou. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.*, 28:967–989, November 2006.

[10] R. Leupers. Exploiting conditional instructions in code generation for embedded VLIW processors. In *Proceedings of the conference on Design, automation and test in Europe - DATE '99*, pages 23–es, New York, New York, USA, Jan. 1999. ACM Press.

[11] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compier support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microachitecture*, pages 45–54, 1992.

[12] J. Park and M. Schlansker. On predicated execution. Technical report, Tech. report, HP laboratories, 1991.

[13] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Not.*, 38:77–90, May 2003.

[14] A. Stoutchinin and G. Gao. If-conversion in ssa form. In *Proceedings of Euro-Par*, pages 336–345, 2004.

[15] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. *SIGPLAN Not.*, 44:264–276, January 2009.

[16] N. Warter, S. Mahlke, W.-M. Hwu, and B. Ramakrishna Rau. Reverse if-conversion. In *PLDI '93 Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, 1993.

[17] S. Winkel. Optimal global scheduling for itanium processor family. In *Proceedings of the EPIC-2 Workshop*, number I, 2002.

[18] www.llvm.org.

[19] www.ti.com.