A Backtracking Instruction Scheduler using Predicate-based Code Hoisting to Fill Delay Slots

Tom Vander Aa
IMEC, Belgium
Tom.VanderAa@imec.be

Bing-Feng Mei
IMEC, Belgium
bennet@imec.be

Bjorn De Sutter
IMEC, Belgium
Bjorn.DeSutter@imec.be

ABSTRACT

Delayed branching is a technique to alleviate branch hazards without expensive hardware branch prediction mechanisms. For VLIW processors with deep pipelines and many issue slots, the instruction scheduler faces the difficult problem of filling the many delay slots. This paper proposes two solutions: a code hoisting technique that produces more candidate operations to be put in the delay slots and an adapted backtracking instruction scheduler that is capable of efficiently placing these candidate operations in the delay slots.

We have demonstrated that the two mechanisms work well on various multimedia and SPECINT2000 benchmarks. The code hoisting technique reduces the schedule length of a traditional scheduler without backtracking by 18%. Using the backtracking scheduler, this amount increases to 24%.

Categories and Subject Descriptors

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

General Terms

Algorithms, Languages

Keywords

VLIW Scheduling, Predication, Code Hoisting

1. INTRODUCTION

Programs that run on mobile systems are becoming increasingly complex. Today you can play games or record video on your cell-phone or PDA.

To be able to run these applications on battery-operated devices, high performance, low-power embedded systems are needed. Furthermore, the design and manufacturing cost has to be kept as low as possible, meaning the system needs to have simple hardware and should be easy to program.

Typically the cores of such systems are programmable processors. VLIW ASIPs in particular are known to be very effective in achieving high performance with reasonably low power consumption[7]. Unlike superscalar processors, VLIW have a simple hardware structure and rely on the compiler to extract the instruction level parallelism (ILP) from the program [13]. They usually do not support hardware branch prediction, but instead feature delayed branches. If a VLIW has two delay cycles, this means two instructions after the branch are always executed, independent of the branch outcome, thus avoiding that the pipeline is flushed when a branch is taken. Figure 1 shows a branch with two delay cycles. Although the two add operations in the schedule appear *after* the branch instruction statically, they are executed before the branch instruction, as the control flow transfer itself occurs only after cycle 2.

Since the compiler is responsible for filling the delay slots, it has to solve the following two problems:

1. The compiler has to find enough operations than *can* be scheduled in a delay slot. The operations need to be completely independent of the branch, meaning that we should be able to schedule them before the branch but they cannot produce a result that is used by the branch.

2. The compiler has to ensure that the instruction scheduler schedules the candidate operations in the delay slot. Since in the unscheduled program, the candidate operations precede the branch, a traditional scheduler will schedule them before it schedules the branch. Next the branch needs to be scheduled, but often it will occur that the resources in the cycle where the branch should be scheduled, are already consumed by the already scheduled operations. In that case, and unless backtracking is used, the only solution is to schedule the branch after the already scheduled operations, leaving the delay slots empty.

This paper proposes solutions for both problems. Sec-

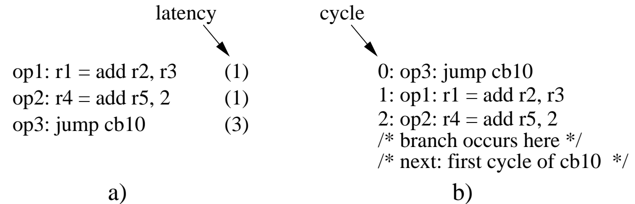

Figure 1: Branch with two delay cycles. Unscheduled code (a) and scheduled code (b).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'07, September 30–October 3, 2007, Salzburg, Austria.
Copyright 2007 ACM 978-1-59593-826-8/07/0009 ...$5.00.

tion 2 explains a simple mechanism that hoists operation from after the branch to be put in the delay slot. Next, Section 3 outlines an algorithm that schedules the branch operation before the candidate operations are scheduled and thus allows for a better filling of the delay slots. Section 4 treats related work. After that, results are presented and discussed in Section 5 by applying the two algorithms on a set of benchmark applications. Finally Section 6 contains conclusions and future work.

2. PREDICATE-BASED CODE HOISTING

In this section discusses a code transformation called *predicate-based code hoisting* that creates freedom for the instruction scheduler to fill the delay slots better. First, the characteristics of the branch instructions used are discussed. Next, the superblock concept is presented as the scope for the transformation. Finally the transformation itself is outlined.

2.1 Simplified Hardware

VLIW processors are used in embedded systems because they achieve relatively high performance with reduced hardware complexity (and thus reduced power consumption) compared to superscalar out-of-order processors. The VLIW processor that is used in this paper (called ADRES [12]) uses two mechanisms to simplify the hardware:

1. **Delayed branching**: Instead of spending expensive hardware and energy on branch prediction, delayed branching is used. Branches on ADRES normally have two delay cycles. This means two instructions that are after the branch in the program are always executed, independent of whether the branch is taken or not. For this paper we have also done experiments with zero and one delay cycles (see Section 5.1). Note that even with zero delay cycles, there are slots in the cycle of the branch instruction itself that need to be filled.

2. **Simplified instruction set**: ADRES only supports a simple RISC instruction set that is very similar to the EPIC instruction set [13]. Conditional branch instructions such as `bgt r1, r2, cb3` (branch to control block 3 if register 1 is greater than register 2) are not supported. Instead they have to be split in two instructions. The first instruction `pred_gt p1, r1, r2`, sets the boolean register (called a predicate) `p1` to the value of the condition. Next the predicate is used to guard an unconditional branch `<p1> jump cb3` (jump to cb3 if p1 is true). This simplifies the branch hardware, as no complex schemes are required to encode both the branch target address and the operands in a single instruction, and the comparison is performed in a separate cycle, this allowing us to save one delay cycle. Of course this only reduced the schedule length if the comparison instruction can be scheduled more than one cycle before the branch.

2.2 Superblocks

A superblock [5] is a program structure that is obtained by combining several basic blocks using code duplication. The combination is based on profiling information, and it enables the optimizer and scheduler to extract more ILP along the important execution paths by systematically removing constraints due to the unimportant paths. The resulting superblock has one entry point, but can have multiple exit branches. For the code hoisting transformations proposed in this paper, it is important to know that a superblock is constructed such that the conditional branches inside the block are most likely to fall through. Profiling information is used to calculate branch fall-through probabilities.

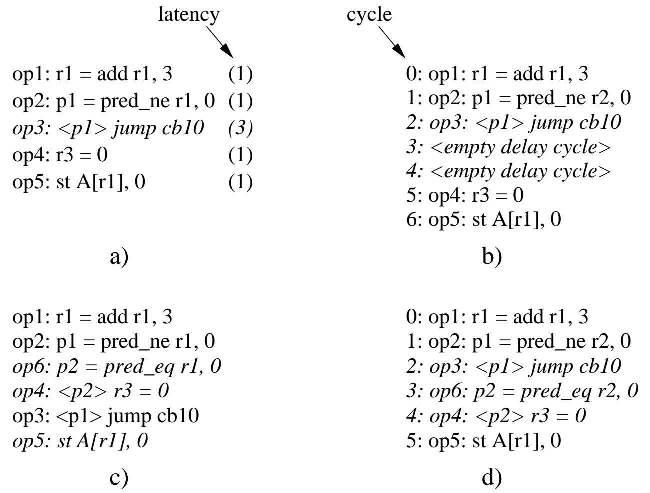

Figure 2: A small example superblock. Original unscheduled code (a) with branch op3. Original scheduled code (b). Transformed unscheduled code (c). Transformed scheduled code (d).

2.3 Code Hoisting Algorithm

Figure 2a shows a small example superblock. It contains one branch in the middle of the block (op3), and a fall-through path at the end. Every branch of the superblock, except the last one, has to be conditional. Otherwise the code after the branch would be unreachable. The scheduled block, shown in Figure 2b contains two delay cycles after op3 that cannot be filled due to the dependencies between op1 and op2 and between op2 and op3.

By computing the complementary predicate p2 of the branch (op6 in Figure 2c) and using this as a guard for op4, op4 can be hoisted across the branch.

The complete code hoisting algorithm as applied to a certain superblock is shown in Figure 3. On line 9 the `NumberOfDelaySlotsToFill` is computed as follows. A VLIW with N issue slots, and a branch delay of D has $N \times D - 1$ delay slots than can be filled: $N - 1$ slots in the cycle of the branch and N slots in every delay cycle.

The conditions in the algorithm (if statement on line 10) are:

- No more operations are hoisted than there are delay slots to be filled. This is a simple rule-of-thumb that works well in practice, as will be shown in the results section.

- Only operations that have a delay that is smaller or equal to the branch delay can be hoisted. This is because we do not allow operations to be in flight across branches.

Section 5 presents the results from applying this algorithm.

```
for CurrentOp = First to last operations in SuperBlock
do
  if CurrentOp defines a predicate then
    CurrentPredDef = CurrentOp
    P = Predicate computed by CurrentOp
    NotP = Inverse of P
    InversePredDef = Operation to compute NotP
  else if CurrentOp is a conditional branch then
    CurrentBranch = CurrentOp
    Compute NumberOfDelaySlotsToFill
  else if  (CurrentBranch is defined) and
        (NumberOfDelaySlotsToFill > 0) and
        (Delay(CurrentOp)  <  Delay(CurrentBranch))
  then
    if InversePredDef not yet inserted then
      Insert InversePredDef after CurrentPredDef
    end if
    Add NotP as a guard to CurrentOp
    Move CurrentOp before CurrentBranch
    Decrement NumberOfDelaySlotsToFill
  end if
end for
```

Figure 3: Code hoisting across branches using predication.

3. BACKTRACKING ASAP OPERATION SCHEDULER (ASAPBT)

This section looks at what is wrong with traditional list schedulers first, and then proposes an improved scheduling algorithm with respect to filling the delay slots of delayed branches.

3.1 Traditional list schedulers

Traditional ready-list schedulers have problems with negative latencies that occur when using delayed branching. The arrows in the example code in Figure 4a indicate dependency edges between operations. Note that in this example not all dependency edges are shown. Dependence constraints in number of cycles are annotated next to the edges. These numbers are calculated using the operation latencies, which are the numbers in brackets next to the operations. For example, the first add produces register *r2* that is used by sub. Since an add has a latency of one, the two operations should be at least one cycle apart. A branch operation in this example has a latency of three, meaning two delay cycles. As a consequence the dependency between the jump and preceding sub has a negative value. The sub should be scheduled at last -2 cycles before the jump, i.e., at last, two cycle after it.

A traditional list scheduler maintains a list of operation that are ready to be scheduled. This list is a subset of the operations yet to be scheduled. An operation can only be scheduled if it is in the ready-list, i.e., if all its predecessors have been scheduled. So in our example, the jump can only be scheduled after all the other operations have already been scheduled. For a single-issue processor, this will result in the schedule as shown in Figure 4b. The schedule has a length of 6 cycles because the two delay slots that follow the branch before are empty.

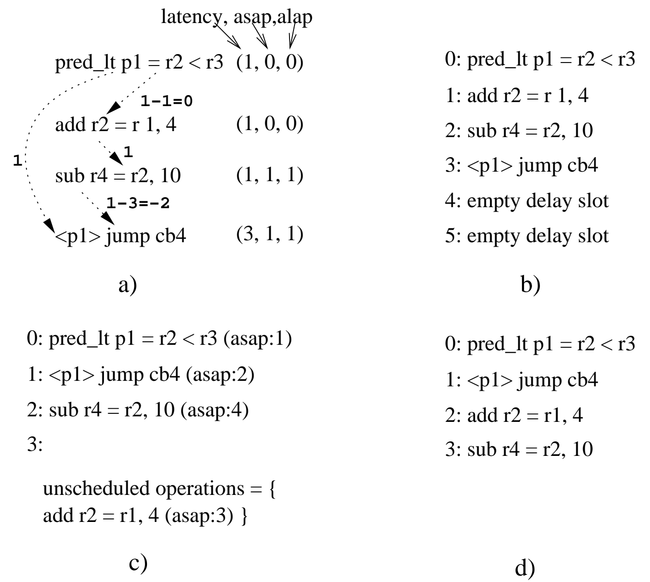

Figure 4: Example of dealing with branch delay slots in the instruction scheduler. Unscheduled code (a). Code scheduled with a list scheduler without backtracking (b). Scheduling state of the proposed backtracking scheduler after unscheduling the add operation (c). Code scheduled with our proposed backtracking list scheduler (d).

3.2 Static and Dynamic ASAP and ALAP Times

Before continuing, it is necessary to explain the concept of *As Soon As Possible* (ASAP) and *As Late As Possible* (ALAP) scheduling times. Two version are distinguished. Static ASAP and ALAP times are computed without scheduling info. Dynamic ASAP/ALAP times use scheduling info for the already scheduled operations.

The *static* ASAP scheduling times are calculated using dependence constraints of operations. If an operation has no incoming dependencies its ASAP time is 0. For other operation, the ASAP time is maximum over all incoming dependencies of the ASAP time of the incoming operation, plus the latency of the incoming edge. For ALAP times, first the ASAP time of every operation needs to be known. If an operation has no outgoing dependencies its ALAP time is the maximum ASAP time of all operations. For other operation, the ALAP time is minimum over all outgoing dependencies of the ALAP time of the destination of the edge, minus the latency of the edge. Once the static ASAP and ALAP times are known, each operation's *slack* can be computed. Slack(op) is computed as static ALAP(op) - static ASAP(op)

The difference between static and *dynamic* ASAP/ALAP times is that scheduling information is used to calculate dynamic ASAP/ALAP times. If an operation is scheduled its ASAP is equal to its ALAP time, is equal to the scheduling time of the operation. If no scheduling info is available, ASAP/ALAP times are computed as above. This means the dynamic ASAP/ALAP times need to be updated during scheduling. Each time an operation is scheduled or unscheduled, the depending and dependent operation's ASAP and ALAP times change.

ASAP and ALAP times are for the example are annotated to Figure 4a.

3.3 Backtracking schedulers with standard priority function

Existing backtracking schedulers recognize the above situation where the jump cannot be placed because the add is blocking the slot. They will backtrack in the sense that they will unschedule the add to place the jump in cycle 2. Figure 4c shows the scheduling state after unscheduling the add.

After unscheduling the add, the sub operation, that is still scheduled in cycle 3, will also need to be unscheduled. The reason for this is can be derived from the dynamic ASAP and ALAP times: the sub can be placed the earliest one cycle after the add, which on its term can be scheduled the earliest one cycle after the jump. Since the jump has just been scheduled in cycle 2, the earliest schedule time of sub is cycle 4. This conflict with the fact that the sub is scheduled in cycle 3. The final schedule after the add and sub have been *rescheduled* is shown in Figure 4d. It is two cycles shorter than the schedule obtained not using backtracking.

From the above, we see two reasons to unschedule operations:

1. if a operation with a higher priority needs to be scheduled there. The priority function can be chosen freely. For example, later in this paper the amount of slack will be used as priority.

2. if the operation is scheduled outside its ASAP-ALAP range.

In [1] two backtracking schedulers are presented. One, ListBT, uses the ready-list of the traditional list scheduler. It schedules operations in dependence order and only allows operations to be unscheduled if they have an negative latency dependency with a branch. This precondition limits the number of scheduling steps, which speeds up scheduling time, but might not produce the optimal schedule. The other scheduler, OperBt, allows operations to be scheduled before all predecessors are scheduled. This will increase the chance of finding a better schedule, but will also increase the number of scheduling steps.

In the next section, an algorithm will be presented that tries to combine the best of these two algorithms.

3.4 Backtracking scheduler with ASAP priority

By carefully selecting the order in which operations are scheduled, we can limit the number of scheduling steps, even if we allow operations to be scheduled before all their predecessors have been scheduled, and even if we do not limit the type of operations that can be unscheduled.

The solution is to select the operation that currently has the lowest ASAP time (*urgency*), or if two operation have the same lowest ASAP time, the operation with the least slack. This is the backtracking scheduler proposed in this paper, which we call AsapBt.

The main scheduling loop is shown in Figure 5. All the operations that need to be scheduled are in `UnscheduledOps`. First static and dynamic ASAP and ALAP times for all of these operations are calculated. Since no operations are scheduled yet, the static and dynamic values are identical. Later the dynamic times will be updated as operations become scheduled. Static times will be used to calculate operation slack. To guarantee that the algorithm finishes in a finite amount of steps, the number of times an operation can be unscheduled is limited (`MaximumUnplaceCount`). `UnscheduleCount` keeps track on how many times an operation has already been unscheduled.

The scheduling loop selects the most urgent operation, i.e., the operation with the earliest dynamic ASAP time. This operation (`CurrentOp`) can be scheduled in a valid way in its dynamic ASAP-ALAP range. If we cannot find a schedule in this range, meaning that `Cycle` > ALAP, all depending operations need to be unscheduled. This type of unscheduling is not taken into account in the `UnscheduleCount`.

For the current `Cycle` and for each `Slot` in the VLIW, the algorithm checks if there are some `BlockingOps` that prevent scheduling in this `Cycle` and `Slot`. If those `BlockingOps` all have more slack than the `CurrentOp` and if none of the `BlockingOps` are already unscheduled more times than `MaximumUnplaceCount`, the `BlockingOps` are unscheduled and their `UnschedulCount`s are updated. Also the dynamic ASAP and ALAP times are recalculated. Unscheduled operations go back to the `UnscheduledOps` list, scheduled operations are removed. The algorithm finishes when the list is empty.

4. RELATED WORK

Basic list scheduling only schedules the operations within a basic block. First, it constructs a data-ready-set, a set of operations whose predecessors have all be scheduled. It selects an operation from the set based on a certain priority function. When an operations is scheduled, other operations might become ready and are put in the data-ready-set. The priority function has been the topic of much research, as it is highly related with the resulting schedule quality. See [2] for an overview of priority functions. As already indicated above, scheduling using the data-ready-set does not work well for an architecture with delayed branching.

Normally operations are scheduled top-down (first the instructions at the beginning of the basic block). In [10] a bottom-up approach is presented where operations are scheduled from the end of the basic block to the beginning. In the bottom-up approach operations whose *successors* (as opposed to predecessors) have all been scheduled become ready. The authors of that paper claim their approach works well with delayed branching. However, we were unable to produce good results using bottom-up scheduling.

In [1] the backtracking scheduler is introduced that is used as a reference case in this paper. Our approach is better because it does not limit the type of operations that can be unscheduled.

The scope of *global* schedulers is not limited to basic blocks. To increase scheduling freedom, and like this also ILP, these schedulers schedule superblocks [5], hyperblocks [11] or code traces [3]. We have chosen to use the superblock as a scheduling scope, since it fits well with the propose code hoisting technique, but there is no reason why the proposed scheduling technique could not be applied to hyperblocks or traces.

Code motion techniques have been extensively studied in the past [8], but not in the context of delayed branching. The authors of [4] propose code hoisting techniques that are very similar to our techniques, but without the benefit of using predication. Hyperblock creation [11] is another technique that uses predication to avoid branch hazards. It

```
Populate UnscheduledOps list
Initialize ASAP and ALAP times of all operations
Initialize UnscheduleCounts for all operations
while UnscheduledOps not empty do
  CurrentOp = Operation in UnscheduledOps with lowest ASAP
  for all Cycle = ASAP(CurrentOp) to +∞ do
    /* If outside ASAP-ALAP range, the schedule of the depending operations becomes invalid. */
    if Cycle > ALAP(CurrentOp) then
      /* Forcibly unschedule all depending ops */
      Recursively unschedule all operations depending on CurrentOp
      Update ASAP and ALAP times because some operations were unscheduled.
    end if
    for all Slot = FirstSlot to LastSlot do
      BlockingOps = Operations that need to be unscheduled to be able to schedule CurrentOp
      if BlockingOps is empty OR
      (slack of all BlockingOps > slack of CurrentOp AND
      UnscheduleCount of all BlockingOps < MaximumUnplaceCount) then
        Unschedule BlockingOps /* If empty, there is nothing to do here */
        Increment UnscheduleCount of all BlockingOps /* If empty, there is nothing to do here */
        Schedule CurrentOp in current Cycle and Slot
        Update ASAP and ALAP times because some operations were (un)scheduled.
        Remove CurrentOp from UnscheduledOps
        Goto next operation (next iteration of while loop)
      end if
    end for
  end for
end while
```

Figure 5: Backtracking list scheduler using ASAP/ALAP times as priority (AsapBt).

is complementary to this work because it predicates and hoists complete basic blocks, while we only predicate and hoist single operations.

The front-end compiler used (IMPACT) already contains hyperblock creation [11] and traditional code hoisting. Since these optimizations are always used, this means the results presented here are improvements on top of those.

5. RESULTS AND DISCUSSION

This section evaluates the proposed techniques. It studies the performance improvements for several multimedia and general purpose benchmarks using the two techniques (code hoisting and backtracking). It compares the AsapBt scheduler with the OperBt and ListBt scheduler on which AsapBt is based. Also, it looks at the compilation overhead due to the backtracking.

5.1 Experimental Setup

The target processor, called ADRES [12], is a tightly integrated combination of a 2D coarse-grain array used only for the innermost loops and a VLIW processor for the sequential code. It is a processor template, meaning it supports a wide variety of similar processors. Thus, the designer can freely choose the number of functional units (FUs) and registers and the interconnection network between the FUs in the coarse-grain array. Furthermore, he can configure the operations supported on the different FUs, and their latencies. Figure 6 shows an example instantiation of the template where the VLIW has 4 FUs and the 2D array 16 FUs. The two parts of the processor communicate through the shared VLIW register file. In the 2D array part most of the FUs do not have access to the global register file. To save

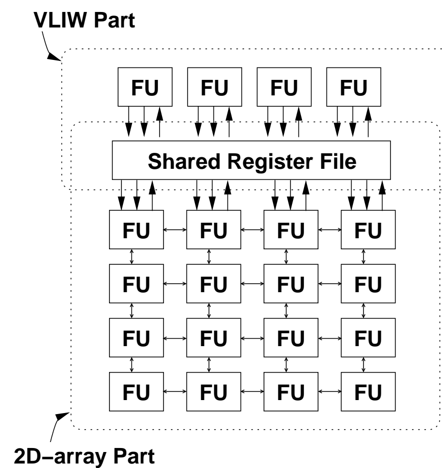

Figure 6: ADRES architecture template

on power, they communicate using a local interconnection network. Conceptually, the array can be seen as a very wide VLIW machine with special loop control, that operates in as a tightly coupled co-processor/accelerator for the main CPU, which is the VLIW processor.

The ADRES instance used for our experiments has a three-issue VLIW with one branch unit and two load-store units. The branch and load-store units also support arithmetic operations and multiplications. Loads have a three-cycle delay and multiplications two cycles. All other operations are single cycle. We have done experiments with three different latencies for branches: one cycle, two cycles and three cycles. A three issue VLIW provides a good trade-off between energy and performance: since most of the ILP is inside the innermost loops that are mapped to the 2D array, not much parallelism is needed in the VLIW. More slots would not increase performance significantly, but would increase power consumption.

In our compiler tool chain, IMPACT [6] is used as the front-end and ILP-optimizing compiler. This compiler includes the traditional code-hoisting optimizations described in Section 4. Three-operand unscheduled assembly is passed to the ADRES-specific back-end. The back-end schedules the innermost loops on the 2D array and does acyclic scheduling and register allocation for the VLIW. We have extended the existing VLIW scheduler to support the code hoisting and backtracking-based scheduling mechanisms. The loop scheduler uses a software pipelining modulo scheduling algorithm, but since this paper focuses on acyclic scheduling, the cycles spent in innermost loops are ignored in the results.

One might ask if it is fair to ignore the cycles spent in innermost loops, since especially in multimedia applications, 90% upto 99% of the instructions are executed in innermost loops [15]. However, we've done experiments where we mapped an H.264 AVC video decoder application [16] on an ADRES processor with 16 FUs. Even though 75% of the instructions executed are in inner loops, this accounts only for 36% of the total execution time, meaning that 64% of the time is spent in sequential code. The reason behind this is that instruction level parallelism is much high in inner loops (9.0 instructions per cycle (IPC)) than in non-inner loop code (1.88 IPC).

Since the ADRES processor is mainly targeting multimedia applications, we have chosen the MediaBench suite [9] for evaluation. Next to that, we have also evaluated a set of 7 SPECINT2000 benchmarks [14] that are certified to work with the IMPACT compiler. (Tables 1 and 2)

Table 1: List of benchmarks from the MediaBench suite

Benchmark	Description
AES	Encryption
Blowfish encode	Encryption
JPEG decode	Image
JPEG encode	Image
EPIC	Image
g721 decode	Audio
g721 encode	Audio
ghostscript	Image
gsm decode	Audio
gsm encode	Audio
mesa	3D graphics
MPEG2 decode	Video
MPEG2 encode	Video
Rasta	Speech Recogn.
SHA	Encryption
avg mediabench	Average of MediaBench

Table 2: List of SPECINT2000 benchmarks

Benchmark	Description
164.gzip	Compression
175.vpr	Versatile Place and Route
181.mcf	Combinatorial Optimization
197.parser	Word Processing
255.vortex	Object-oriented Database
256.bzip2	Compression
300.twolf	Place and Route Simulator
avg spec	Average of the SPEC Applications

5.2 Dynamic Cycles

Figure 7 shows performance improvements due to code-hoisting and AsapBt backtracking. Branches for these results have two delay cycles. The base case (100%) uses a scheduler without backtracking and does not do code-hoisting. On average across all applications (global average in the figure), code-hoisting using a scheduler without backtracking results in an improvement of 18%. Enabling both code-hoisting and backtracking gives the best results: 24% of gain compared to the base case. The gain is consistent across all applications. Multimedia applications seem to have more opportunities for code-hoisting than SPECINT2000 applications (21% improvement for MediaBench compared to 16% for SPECINT2000).

Using a backtracking scheduler without code-hoisting to fill the delay slots does not seem to be a good idea: improvement are marginal in most cases (2% on average) and in some cases the backtracking scheduler performs worse than the one without (e.g. epic, blowfish, adpcmdec).

The above results were for branches with two delay cycles. We have also done experiments with different numbers of delay cycles. Table 3 compares the averages for the different techniques on architectures that have a branch with two, one and no delay cycles.

Table 3: Comparison of performance improvements due to predicate-based code-hoisting (PBCH) and backtracking scheduling (BT) for branches with two, one and no delay cycles.

	no PBCH no BT	PBCH no BT	no PBCH BT	PBCH BT
2 delay cycles				
avg mediabench	100%	84%	98%	77%
avg spec	100%	79%	98%	73%
global average	*100%*	*82%*	*98%*	*76%*
1 delay cycles				
avg mediabench	100%	83%	99%	80%
avg spec	100%	81%	98%	77%
global average	*100%*	*82%*	*99%*	*79%*
0 delay cycles				
avg mediabench	100%	87%	99%	86%
avg spec	100%	87%	98%	84%
global average	*100%*	*87%*	*99%*	*85%*

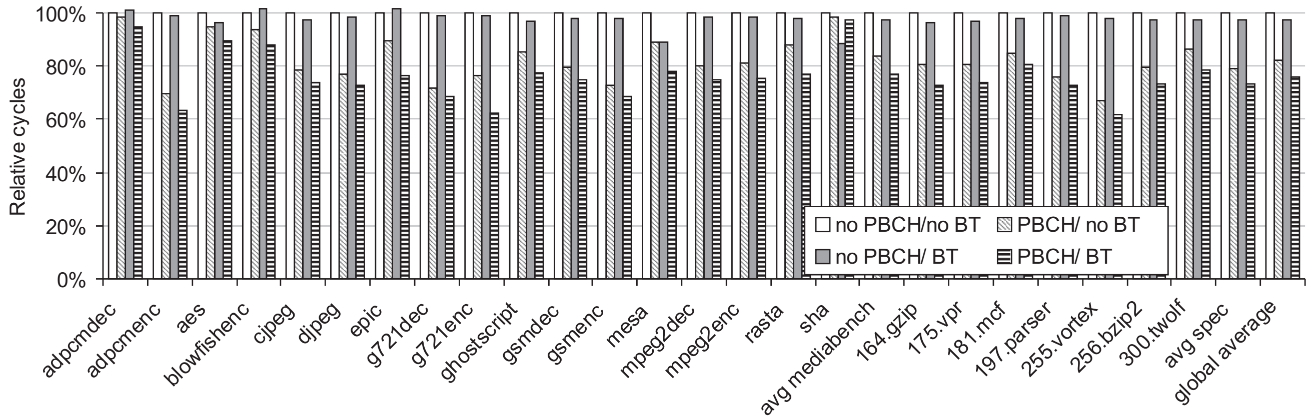

Figure 7: Performance improvements due to predicate-based code-hoisting (PBCH) and backtracking scheduling (BT). Dynamic cycles are relative to the base case: no PBCH and no BT. Branches have two delay cycles.

The results for the one cycle delay slot due to code-hoisting are surprisingly good. Although the one cycle delay instance has less delay slots where the hoisted operations can be placed, the code-hoisting algorithm still gave the same improvements on this instance as it did on the instance with two delay cycles: 18% less cycles using scheduler without backtracking. This is because the hoisted operations do not necessary need to be placed *in* the delay cycles, they can also be placed in the cycle of the branch or even *before* it. If the branch has no delay cycles, the algorithm still achieved a 13% improvement only due to code-hoisting.

Clearly, the backtracking scheduler is not needed when there are no delay slots to be filled: the improvement of the backtracking scheduler is only 2%. Compare the average of "code hoisting/no backtracking" (87%) to "code hoisting/backtracking" (85%).

When comparing the AsapBt backtracking method proposed in this paper with the ListBt and OperBt schedulers (Figure 8), we see that AsapBt is either as good as the OperBt scheduler or a little bit better (1% on average for all applications). However, we will see in the next section that the overhead due to backtracking in AsapBt is significantly less than in OperBt.

5.3 Backtracking Overhead

Backtracking scheduling is slower because operations can be replaced multiple times. Figure 9 shows the overhead in number of extra scheduling steps per operation for the method proposed in this paper (*AsapBt*) and for the OperBt and ListBt scheduler described in [1].

For our method and an architecture with two branch delay cycles, each operation was placed 1.23 times on average, meaning only 23% of the operations were unplaced. For larger applications, the overhead was even smaller (e.g. 10% on 197.parser). For some kernels that were difficult to schedule because of a high inherent parallelism, the overhead raised to 50% (e.g. sha and aes).

For less delay cycles, the overhead reduces to 20% and 18% for one or no delay cycles respectively.

For the same VLIW configuration, our method has an overhead that holds the middle between the OperBt scheduler (which schedules each operation 1.36 times on average) and the ListBt scheduler (which schedules each operation 1.20 times on average), but slightly outperforms both of them in performance of the resulting schedule (as already discussed in the previous section).

6. CONCLUSIONS

Traditional list schedulers are not good at filling delay cycles that occur when delayed branching is used. In this paper we have proposed a combination of a code-hoisting transformation and a novel backtracking instruction scheduler that improve the schedules of a set of relevant benchmarks with 24% on architectures that apply delayed branching (branches have two delay cycles).

The main contribution of this paper is not either of the two optimizations, but their symbiosis. Results show that the gain of combining the two is larger that each individual gain. Predicate-based code-hoisting is needed as an enabling transformation for the backtracking instruction scheduler.

The extra compilation time due to backtracking stays limited: 23% extra scheduling steps.

Acknowledgments

This research has been carried out in the context of IMEC's multimode multimedia program which is partly sponsored by Samsung and Freescale.

7. REFERENCES

[1] S. G. Abraham, W. Meleis, and I. D. Baev. Efficient backtracking instruction schedulers. In *IEEE PACT*, pages 301–308, 2000.

[2] B. De Sutter. General-purpose architecture instruction scheduling techniques. Technical report, ELIS, Universiteit Gent, Belgium, November 1998.

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

[4] T. R. Gross and J. L. Hennessy. Optimizing delayed branches. In *MICRO 15: Proceedings of the 15th annual workshop on Microprogramming*, pages 114–120, Piscataway, NJ, USA, 1982. IEEE Press.

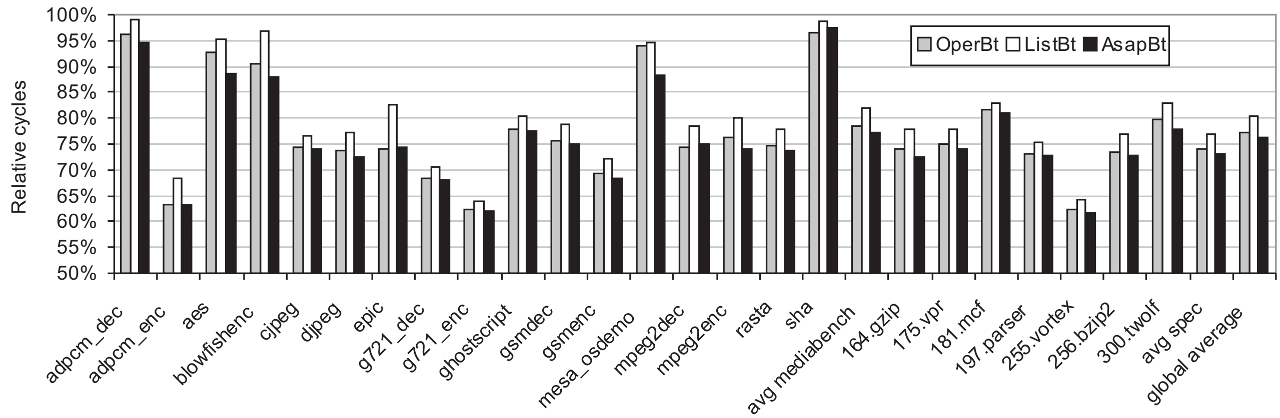

Figure 8: Comparison of performance improvements for the different backtracking strategies: for the scheduling method proposed in this paper (*AsapBt*) and for the OperBt and ListBt scheduler described in [1]. Dynamic cycles are relative to the base case: no PBCH and no BT. Branches have two delay cycles.

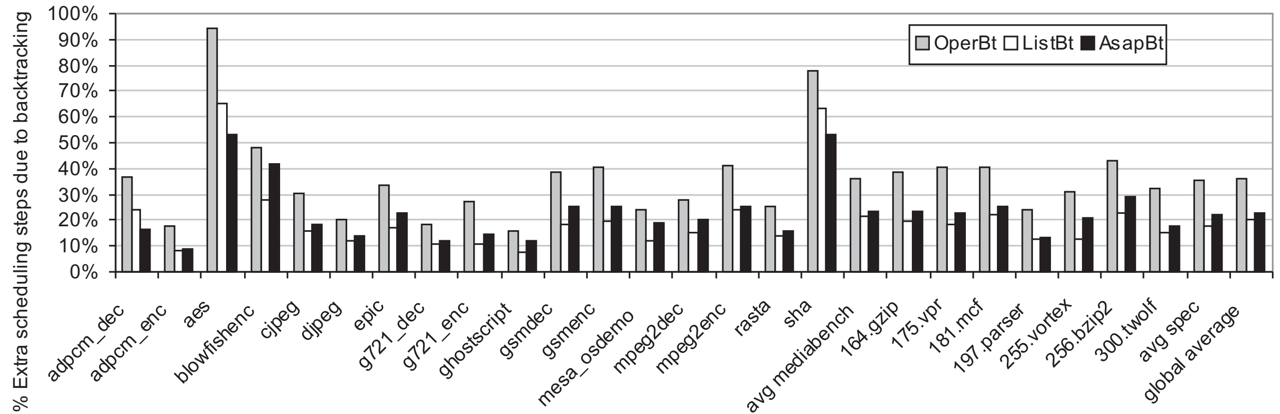

Figure 9: Scheduling overhead due to backtracking. Number of extra scheduling step for an architecture with 2 delay cycles for the scheduling method proposed in this paper (*AsapBt*) and for the OperBt and ListBt scheduler described in [1].

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

[6] The IMPACT Research Group, http://www.crhc.uiuc.edu/Impact/. *The IMPACT Research Compiler*, 1987.

[7] M. F. Jacome and G. de Veciana. Design challenges for new application-specific processors. *Special issue on Design of Embedded Systems in IEEE Design & Test of Computers*, April-June 2000.

[8] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[9] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[10] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

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

[12] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proc. of Field-Programmable Logic and Applications*, pages 61–70, 2003.

[13] M. Schlansker, B. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik, and S. Abraham. Achieving high levels of instruction-level parallelism with reduced hardware complexity. Technical Report HPL-96-120, Hewlett Packard Laboratories, February 1997.

[14] Standard Performance Evaluation Corporation, http://www.spec.org. *SPEC CPU2000*, 2000.

[15] T. Vander Aa, M. Jayapala, F. Barat, G. Deconinck, R. Lauwereins, H. Corporaal, and F. Catthoor. Instruction buffering exploration for low energy embedded processors. *Journal of Embedded Computing*, pages 341 – 351, July 2005.

[16] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.