Instruction Scheduling for TriMedia

Jan Hoogerbrugge HOOGERB@NATLAB.RESEARCH.PHILIPS.COM
Lex Augusteijn LEX@NATLAB.RESEARCH.PHILIPS.COM
Philips Research Laboratories, Prof. Holstlaan 4,
5656 AA Eindhoven, The Netherlands

Abstract

Instruction scheduling is a crucial phase in a compiler for very long instruction word (VLIW) processors. This paper describes the instruction scheduler of the second generation compiler for the TriMedia VLIW mediaprocessor family as well as related compiler issues to increase the size of a scheduling unit. The paper discusses the guarded decision tree scheduling unit, how guarded decision trees are scheduled, register allocation and its interaction with instruction scheduling, issue slot assignment, and scheduling of jump operations. Furthermore, the paper presents several experiments that quantify various aspects of scheduling.
Keywords: Instruction scheduling, register allocation, VLIW, TriMedia

1. Introduction

TM1000 is the first member of Philips Semiconductors' TriMedia embedded mediaprocessor family [1] originating from the LIFE research project during the late eighties [2, 3]. The heart of TM1000 is a five-issue very long instruction word (VLIW) processor that contains 28 functional units and a central 128×32-bit register file. Its operation set contains a large number of operations to accelerate computationally intensive parts of multimedia applications. Examples are operations for MPEG motion estimation and frame reconstruction, and DSP operations such as saturating addition and multiply-add. Around TM1000's VLIW core are a large number of on-chip peripherals and co-processors to reduce system cost. TM1000's peripherals include video in/out, audio in/out, PCI, a modem front-end, and a variable length decoder for MPEG decoding.

An essential element for VLIW performance is the *instruction scheduler* of the compiler. The instruction scheduler is responsible for translating the sequential code produced by the *core compiler* into VLIW instructions each containing independent operations that are issued in parallel by the VLIW. This paper describes TriMedia's second generation instruction scheduler, a machine description based instruction scheduler that is capable of generating code for a large variety of TriMedia family members, including TM1000. The paper introduces *guarded decision trees*, an extension of the decision tree scheduling unit proposed by Hsu and Davidson [4]. Furthermore, it describes how issue slot allocation takes places as well as how register allocation is implemented and how it interacts with instruction scheduling.

This paper is organized as follows. Section 2 describes the aspects of TM1000 that are necessary to understand the remainder of the paper. Section 3 describes instruction scheduling for TM1000 based on guarded decision trees. Section 4 describes how register allocation is performed. Section 5 describes several measurements that quantify various aspects of the scheduler. Section 6 concludes the paper.

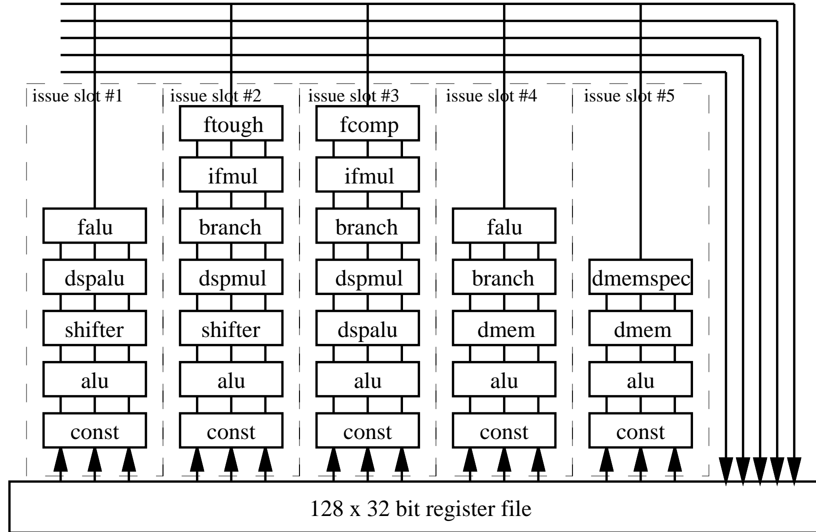

Figure 1: Organization of the VLIW core of TM1000

Name	Latency	Issue slots					Operations
const	1	1	2	3	4	5	iimm, uimm
alu	1	1	2	3	4	5	iadd, isub, igtr, igeq, bitand, bitor, ...
dmem	3				4	5	ild8d, uld8d, ld32d, st8d, st16d, st32d, ...
dmemspec	3					5	dcb, dinvalid, prefd, prefr, allocd, ...
shifter	1	1	2				asli, roli, asri, lsri, asl, rol, ...
dspalu	2	1		3			ume8ii, dspiadd, dspisub, dspidualadd, ...
dspmul	3		2	3			ifir16, ufir16, ifir8ii, dspidualmul, ...
branch	4		2	3	4		jmpf, jmpt, ijmpf, ijmpt, ...
falu	3	1			4		fadd, fsub, fabsval, ifloat, ifixrz, ...
ifmul	3		2	3			fmul, imul, umul, imulm, dspimul, dspumul, ...
fcomp	1			3			fgtr, fgeq, feql, fneq, fsign, ...
ftough	17		2				fdiv, fsqrt, fdivflags, fsqrtflags

Table 1: Functional units of TM1000. All FUs except ftough are fully pipelined.

2. TM1000 from a Scheduling Point of View

TM1000 contains 28 functional units (FUs). All multi-cycle latency FUs, except the FU that performs floating point divisions and square roots, are fully pipelined. Each FU is accessible from only one of the 5 issue slots. Up to 5 results per cycle can be written to the central register file of 128×32-bit registers. There are no restrictions on combinations of FUs that may produce results simultaneously. Figure 1 shows the relation between FUs, issue slots, the central register file, and write-back busses. Table 1 lists the FUs together with their properties.

TM1000 supports *guarded* or *predicated* execution to facilitate scheduling and to reduce branches. The least significant bit of a guard operand controls whether the operation will be issued or suppressed. All 128 registers of the cental register file can be used for guarding.

Two load/store FUs are present to access memory. Instead of a truly dual-ported data cache, which would make it twice as expensive as a single-ported cache, the data cache is banked into 8 banks [5]. Low order interleaving is used to map words on banks. Two memory operations can access the data cache in parallel as long as they access different banks, i.e., bits 2–4 of their addresses are different. When two memory operations access the same bank simultaneously the processor stalls for one cycle to serialize the accesses. The compiler can attempt to reduce bank conflicts by not scheduling memory operations in the same instruction that are likely or guaranteed to access the same bank.

In order to reduce code size and to improve instruction cache performance, instructions are stored compressed in the instruction cache and main memory. Compression is achieved by compact encoding of unused issue slots (nops), compact encoding of always true guards, compact encoding of the most frequently used opcodes, and a few other techniques. Decompression is part of the instruction pipeline which results in a jump latency of four cycles, i.e., three jump delay instructions.

TM1000 has no interlocking. The scheduler has to guarantee that values are not used before the operation that computes them has been completed. Only cache misses and data cache bank conflicts stall the processor.

3. Guarded Decision Tree Scheduling

Global instruction schedulers differ mainly in the scheduling unit on which they operate. Well-known scheduling units from literature are *traces* [6], *superblocks* [7], *regions* [8], and *decision trees* [4, 9, 10]. All these scheduling units have in common that they are acyclic control flow graphs of basic blocks and that they have a single header basic block from which all other blocks within the scheduling unit are reachable. The two main differences between the above mentioned scheduling units are:

1. whether they contain a single control flow path (traces and superblocks) or multiple control flow paths (regions and decision trees) and

2. whether they contain join points, i.e., basic blocks other than the header basic block with multiple predecessors (traces and regions), or not (superblocks and decision trees).

When choosing a scheduling unit one has to make a trade-off between performance and engineering complexity. For the best performance one does not want to restrict to single control flow path scheduling units without join points. However, from an engineering complexity point of view, join points are hard to handle since complex bookkeeping is required when operations are scheduled above a join point [11, 12]. It is not clear whether including join points is worth the effort that could otherwise be spent on other aspects of the scheduler.

For the TriMedia scheduler we have chosen decision trees as the scheduling unit mainly because it is a good trade-off between performance and engineering complexity. To make a better match with TriMedia's hardware capabilities, we extended decision trees with

```
tree          :  tree ( execution_count ) tree_body endtree
tree_body     :  oper* jump_oper
oper          :  value guard opcode modifier value* constraints ;
guard         :  if value          /* guarded operation */
              |  empty             /* unguarded operation */
modifier      :  ( integer )       /* modifier (= literal) */
              |  empty             /* no modifier */
constraints   :  after value+      /* reorder constraints due to mem. dep. */
              |  empty             /* no reorder constraints */
jump_oper     :  gotree label      /* absolute jump */
              |  cgoto value       /* computed jump */
              |  if value ( probability ) then tree_body else tree_body end
```

Figure 2: Simplified syntax of scheduler input

guarded execution. Arbitrary acyclic control flow graphs without side-entries can be transformed into guarded decision trees prior to scheduling by the core compiler by means of if-conversion [13]. This makes it possible to have scheduling units that originally had join points. This extension is similar to hyperblocks [14], which are superblocks containing guarded operations. The difference between hyperblocks and guarded decision trees is that for hyperblocks the result of if-conversion should be a linear sequence of basic blocks without side-entries while for guarded decision trees the result should be a tree of basic blocks without side-entries. The latter is more general.

Earlier work on decision tree scheduling is described by Hsu and Davidson [4] and Banerjia, Havanki, and Conte [9] . Hsu and Davidson introduced decision tree scheduling for scalar deeply-pipelined machines and describe the basic concepts, the scheduling priority function (Section 3.2.2), and the grafting code replication technique to increase ILP (Section 3.1). Banerjia, Havanki, and Conte describe decision tree scheduling for a VLIW and compares it with basic block scheduling and superblock scheduling. Our contributions to decision tree scheduling research include: integration with register allocation (Section 4), optimistic jump scheduling (Section 3.5), and controlling register pressure (Section 4.4).

3.1 Guarded Decision Trees Format

Guarded decision trees are maximally tree-shaped control flow graphs of basic blocks, where each basic block consists of a sequence of possibly guarded operations followed by a jump operation. Figure 2 shows a syntax description of the input of the scheduler. Besides the actual sequential code, the input contains execution counts obtained from profiling, execution probabilities, and reordering constraints between memory references resulting from alias analysis performed by the core compiler.

Values in Figure 2 should be interpreted as pseudo registers that have to be assigned by the scheduler to physical registers. Guarded decision trees have to be in static single assignment [15] form, meaning that each value should be defined by precisely one operation.

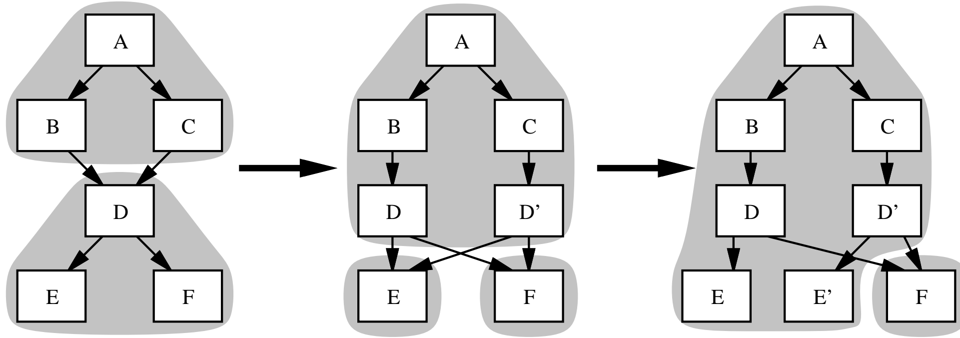

Figure 3: Grafting to improve ILP along path $A \rightarrow B \rightarrow D \rightarrow E$

Pseudo *phi* operations are provided to merge values. For example, the following code fragment merges values 10 and 14 and stores the result.

```
10 if 11 iadd 12 13;     /* if value 11 is true then store */
14 if 15 isub 12 13;     /* the result of the iadd operation */
16 phi 10 14;            /* otherwise store the result of */
17 st32 18 16;           /* the isub operation. */
```

Values that are merged by phi operations should be defined by operations that have mutually exclusive guards. In the example above, values 11 and 15 should not both be *true* during a tree invocation.

A phi operations merges two values. A tree of phi operations is used to merge more than two values (merging n values requires $n-1$ phi operations). Again, all guards of the operations that define the values being merged should be mutual exclusive.

In practice, decision trees are often too small to contain sufficient ILP, especially in control intensive applications. A technique called *grafting* attempts to improve this. Grafting removes join basic blocks (that lead to decision tree boundaries) by duplicating them. Figure 3 illustrates grafting. Assume that control flow path $A \rightarrow B \rightarrow D \rightarrow E$ is the most likely path. Initially this path is part of two decision trees, which prevents ILP exploitation between $A\&B$ and $D\&E$. Applying grafting twice by first duplicating D followed by duplicating E leads to code in which the path is contained within one decision tree. Grafting is performed by the core compiler after global optimization and is steered by profiling information and code expansion consequences.

Grafting is the counterpart of tail duplication of superblock scheduling, which transforms traces into superblocks by duplicating code below join points. Decision tree determination is profile independent and grafting is controlled by profiling, while in the case of superblock scheduling, trace selection is profile dependent and tail duplication is profile independent. Making the code expanding transformation profile dependent makes code expansion better controllable, which is very important for embedded processors like the TM1000.

Currently both if-conversion and guarding have to be enabled by the user by means of command-line options. Automatically deciding what is most optimal in a certain situation is not yet completely understood.

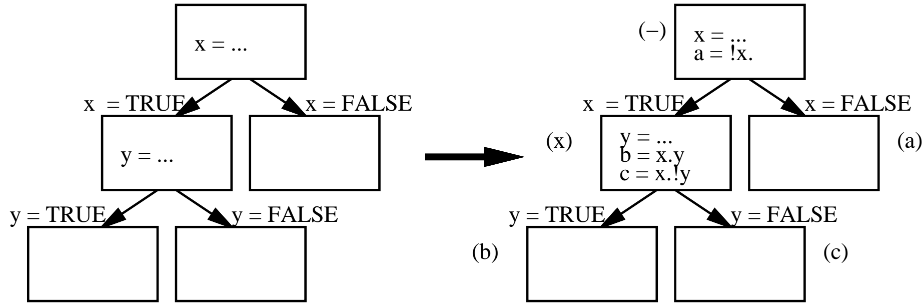

Figure 4: Assigning guards to basic blocks. Assigned guards are shown between parentheses.

3.2 Preprocessing Steps

Before scheduling, a number of preprocessing steps are performed. The two most important ones are discussed in the following two sections: assigning guards to basic blocks and assigning priorities to operations. Other preprocessing steps, which are not further discussed, include the lay-out of the basic blocks (the most likely control flow path will become the fall-through path to improve instruction cache locality) and the elimination of `rdreg` and `wrreg` pseudo operations (see Section 4.1).

3.2.1 Assigning guards to basic blocks

Parallelism studies have shown that speculative execution is one of the most successful effective techniques for exploiting ILP [16]. In the case of decision trees, speculative execution means moving operations up in the tree in the direction of the tree entry. For most operations this can be done without any problems, but for certain operations this is only possible if the operations are guarded such that their guard evaluates *true* if and only if the basic block from which they were moved will be executed. Among these operations are stores, jumps, operations that define off-live values [17], and operations that produce exceptions, such as divide and floating point operations on the TM1000[1]. Load operations do not generate exceptions on TM1000 and can therefore be executed speculatively without guarding.

To achieve speculative execution of the above mentioned operations, the scheduler assigns a guard to each basic block in the decision tree except the entry basic block. This guard operation produces value *true* if and only if the basic block it guards will be executed. Figure 4 illustrates this process. The decision tree is traversed in pre-order. When a basic block B is visited, it assigns to each successor S of B a guard which computes the conjunction of the guard of B and the condition under which S will be executed after B. TriMedia's `bitand` and `bitandinv` operations are used for this purpose. The entry basic block of the decision tree is handled differently since no guard will be assigned to it.

1. Normally these exceptions are disabled; whenever an application requires them they should be enabled by a compiler switch that prohibits speculation of divides and floating point operations without proper guarding.

3.2.2 Priority calculation

Like other schedulers based on list scheduling, decision tree scheduling prioritizes operations. The priority of an operation reflects the urgency that it should be scheduled whenever it becomes ready for scheduling. The TriMedia scheduler uses the priority function proposed by Hsu and Davidson [4]. For each path p from the decision tree entry to an exit point, the minimal completion time $mincompl(p)$ for an infinite resource machine is determined. This figure is based on dependences and operation latencies. Furthermore, we determine for each operation o on path p the latest cycle $latest(o,p)$ in which it can be placed in order to achieve $mincompl(p)$. Scheduling priority is computed from *latest* and *mincompl* by:

$$priority(o) = \sum_{p \in paths(o)} probability(p) * (1 - \frac{latest(o,p)}{mincompl(p)}),$$

where $paths(o)$ is the set of control flow paths through o and $probability(p)$ is the expected probability that p is executed whenever the decision tree is invoked. This probability is based on profiling information if available; otherwise it is estimated.

3.3 Scheduling a Decision Tree

A decision tree is scheduled by scheduling all its basic blocks in pre-order order. A basic block B is scheduled using a top-down[2], operation-based[3] list scheduling algorithm that considers both operations from B as well as operations from descendants of B. When all operations of B are scheduled, B's *fixed* flag is set to indicate that it is not allowed to add empty instructions to B. When operations from descendants of B are selected for scheduling in B and its fixed flag is set and it is not possible to schedule them in B, then these operations are marked as failed and they are not selected for scheduling in B again. Basic block B is scheduled when no more operations can be added to it. In that case the failed flags are cleared and the scheduler proceeds with scheduling the successors of B.

Jump operations are treated very similar to non-jump operations. They can be scheduled into ancestor basic blocks. However, for simplicity, it is not allowed to reorder jump operations. Thus they can be scheduled next to or in delay instructions of jump operations in ancestor basic blocks, but not above them. To simplify interrupt handling in the presence of multi-cycle latency operations and pipelined execution of jump operations, the TriMedia operation set distinguishes interruptible jumps and non-interruptible jumps [18]. Interrupts are only handled when interruptible jumps are taken. At that moment FU pipelines should be empty and there should be no outstanding jumps. Therefore the TriMedia scheduler uses non-interruptible jumps for decision tree internal jumps and interruptible for jumps between decision trees.

2. A top-down scheduler schedules an operation when all its predecessors have been scheduled; a bottom-up scheduler schedules an operation when all its successors have been scheduled. A top-down approach is more 'natural' in combination with global instruction scheduling.
3. An operation-based scheduler repeatedly selects an operation and places it in the earliest possible instruction in which resource and dependence constraints are satisfied. A cycle-based scheduler repeatedly fills an instruction with operations and moves on to the next instruction when no more operations can be added to it. An operation-based scheduler is likely to give better results for machines like the TM1000 in which operations need resources for multiple cycles.

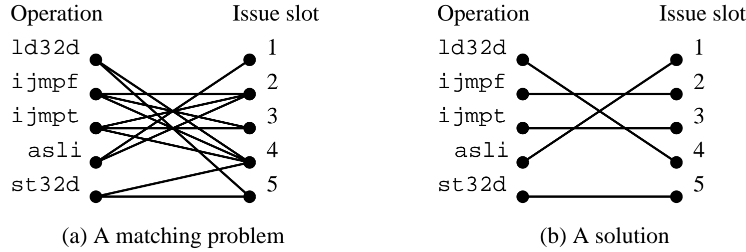

Figure 5: Issue slot assignment via bipartite graph matching

3.4 Scheduling an Operation

Scheduling an operation consists of finding the earliest possible instruction where free resources are available for it and incoming dependence constraints are satisfied. For TriMedia the scheduler has to check whether issue slot assignment is possible, whether sufficient write-back busses are available, and whether so-called crossovers exist (see Section 3.4.3). Furthermore, the scheduler could try to avoid data cache bank conflict as much as possible by not scheduling in one instruction two memory references that are guaranteed to access the same bank.

3.4.1 Issue slot assignment

The problem of whether and how an issue slot assignment is possible when an operation is placed in a certain instruction in which several operations have already been scheduled is solved by translating it into a bipartite graph matching problem for which efficient algorithms are available [19]. This problem has to be solved for every attempt the scheduler makes to schedule an operation in a certain instruction in which already a number of operations could have been scheduled. Let $o_1 \ldots o_n$ be the operations for which an issue slot assignment has to be found and $i_1 \ldots i_m$ the available issue slots. A bipartite graph (V, E) is constructed where $V = \{o_1 \ldots o_n, i_1 \ldots i_m\}$ and $(o, i) \in E$ if and only if o can be issued from issue slot i. A matching M is a subset of E such that no two edges of M are adjacent. Clearly, a matching of cardinality n corresponds to an issue slot assignment.

Figure 5 illustrates issue slot assignment via bipartite graph matching. An issue slot assignment for five operations, namely a load, two jumps, a shift, and a store operation, on TM1000 has to be found. Figure 5a shows the bipartite graph for this problem. Figure 5b shows a matching of cardinality 5 which corresponds to an issue slot assignment.

Multi-cycle latency non-pipelined FUs lead to complications since issue slot assignment cannot be performed for each instruction in isolation of other instructions. The TriMedia scheduler handles this situation by inspecting L_{max} instructions preceding and following the instruction i for which the issue assignment problem has to be solved, where L_{max} is the maximal FU latency. If these instructions contain operations scheduled on non-pipelined FUs then these FUs are not available for issue slot assignment in instruction i. Furthermore, for reasons of simplicity, the issue slot assignment of an operation scheduled on a non-pipelined FU is no longer changed after it has been scheduled. Due to the infrequent occurrence of operations executed on non-pipelined FUs, this limitation has no significant performance impact.

3.4.2 Checking write-back busses

To check whether sufficient write-back busses are available for scheduling an operation o in instruction i, the scheduler counts the number of values that are produced in the cycle that o completes. This is implemented by inspecting i and a number of instructions preceding and following i and counting how many result producing operations complete in the same cycle as o. The number of instructions that have to be inspected is bounded by L_{max}. Currently the scheduler does not take guarding into account while checking write-back bus constraints. Experiments have shown that write-back busses are currently not a serious performance constraint.

3.4.3 Checking for crossovers

A *crossover* is a situation where a misspeculated operation is in flight during a transition from one decision tree A to another decision tree B and produces a result after the first instruction executed in B. It is misspeculated in the sense that it was originally not on the executed path leading to B. From decision tree B's point of view this results in an unexpected write-back bus usage and a register write which could lead to incorrect code. Therefore, crossovers are not allowed and the scheduler has to check for them when scheduling an operation. When a crossover situation is detected the scheduler attempts to handle it by proper guarding. If this is not possible then the operation will be delayed until an instruction where proper guarding is possible or past the last delay instruction of the jump targeting B.

Hardware support that suppresses operations in flight between decision tree transitions could make the task of the scheduler easier. This support is not available in TM1000. However, in practice, the performance impact of the crossover scheduling constraint is very small, less than 0.5% for control intensive applications.

3.4.4 Avoiding bank conflicts

As mentioned in Section 2, the data cache of TM1000 is banked into 8 banks and low order interleaving is used to determine the bank. Successive words (four bytes) are mapped in successive banks (modulo 8). When two memory accesses scheduled in the same cycle access the same bank the machine stalls for one cycle to serialize the accesses. This is called a bank conflict. The scheduler employs a simple technique to reduce bank conflicts. When a memory access is scheduled in an instruction in which already a memory access has been scheduled, and both memory operations have a common base pointer and a constant offset, the scheduler is able to estimate the probability that a bank conflict will occur during execution of the instruction. Possible probabilities are: 0%, 25%, 50%, 75%, and 100%. In the case of a probability of 25% or more, the two memory accesses are not scheduled in the same instruction. A few examples of combinations of memory accesses and bank conflict probabilities are given below:

```
ld32d(0) r4 -> ...   }
ld32d(8) r4 -> ...   } 0% bank conflict probability

ld32d(32) r4 -> ...  }
ld16d(64) r4 -> ...  } 100% bank conflict probability
```

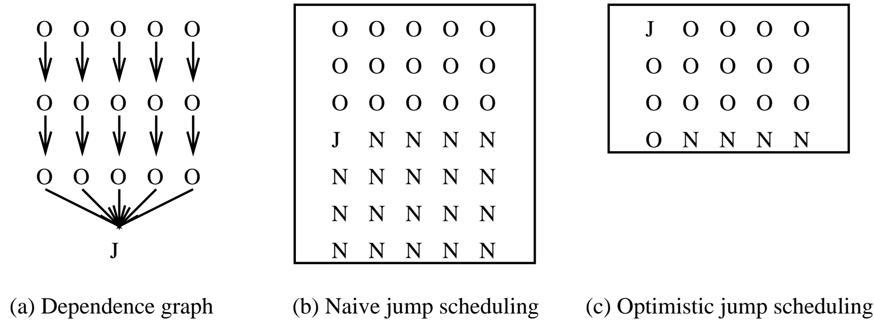

Figure 6: Scheduling jump operations. O: a non-jump, J: a jump, N: a nop.

```
ld8d(0) r4 -> ...
ld8d(3) r4 -> ...
```
} 25% bank conflict probability

Notice that the operation name indicates the data size in bits and offsets in bytes are specified between parentheses. Furthermore, n byte data should be n byte aligned in memory.

3.5 Scheduling Jump Operations

A non-backtracking list scheduler can schedule jumps when all operations that should be completed before the jump takes place are scheduled. This leads to dependences between non-jump operations O and jump operations J with delay $L_O - L_J$, where L_O is the latency of O and L_J is the latency of J. For TM1000 this often leads to negative dependence delays, which are not handled well by list scheduling. This is illustrated in Figure 6. Figure 6a shows a relatively parallel basic block of 15 operations O followed by a jump J. Assume that J has four cycles latency (like TM1000) and the Os have single cycle latency. The dependences between the Os and J with delay -3 result in a schedule as shown in Figure 6b when a five issue uniform machine is assumed. Clearly, the delay slots of J are not filled. To obtain a schedule as shown in Figure 6c, the TriMedia scheduler does not make J dependent on the Os and schedules J *optimistically*. Whenever it turns out that J was scheduled too early, the scheduler unschedules J as well as all operations scheduled after J. To do this, the scheduler time-stamps each operation and instruction when it is scheduled or created respectively. To unschedule a jump operation J, the scheduler unschedules all operations with a time stamp newer than J's time stamp and deletes all instructions with a time stamp newer than J's. To prevent J from being scheduled in the same cycle again, the scheduler records that J should be scheduled in a later cycle.

To reduce the amount of rescheduling and therefore compilation time, a lower bound on the cycle in which a jump operation will be scheduled is computed. No jump will be scheduled before this bound since this will lead inevitable to rescheduling. The lower bound is based on resources and dependences of code that should complete before the jump takes place. With this bound the average number of times an operation is unscheduled is 0.36 times. This is measured with the benchmarks used in Section 5 with grafting enabled.

4. Integrated Register Allocation

The combination of instruction scheduling and register allocation is a well-known phase ordering problem. Performing register allocation before scheduling is likely to introduce false dependences that restrict scheduling freedom; performing scheduling before register allocation may lead to situations where the scheduler schedules too aggressively which results in a register pressure that cannot be handled without severe spilling by the register allocator. From a performance point of view the best solution is to combine scheduling and register allocation. However, from an engineering point of view it is better to handle the tasks in separate modules. In the TriMedia compiler we have made a compromise on the basis of which we get most of the performance benefit from integrated scheduling and register allocation with only modest additional engineering effort.

4.1 Work division between core compiler and scheduler

The TriMedia compiler divides live ranges into local and global live ranges. Local live ranges do not pass decision tree boundaries, while global live ranges do. Furthermore, by convention, the register file is divided into local and global registers. Global live ranges are assigned by the core compiler to global registers using a graph-coloring-based algorithm. Local live ranges are assigned to local registers by the scheduler while it is scheduling. The motivation for this work division is that the scheduler is not able to reorder global live ranges and therefore global live ranges can be allocated prior to scheduling without performance problems. Local live ranges, on the other hand, are likely to be reordered by the scheduler and are relatively easy to allocate while scheduling.

Access to global registers at the intermediate representation which is exchanged between core compiler and scheduler takes place via `rdreg` and `wrreg` pseudo operations. For example, consider the following decision tree, which corresponds to a function that returns the sum of its two arguments:

```
tree (10)
    1 rdreg(5);    /* read global r5, function arg. #1 */
    2 rdreg(6);    /* read global r6, function arg. #2 */
    3 iadd 1 2;
    4 wrreg(5), 3; /* write global r5, function result */
    5 rdreg(2);    /* read global r2, return address */
    6 cgoto 5
endtree
```

Prior to scheduling, the scheduler eliminates nearly all `rdreg` and `wrreg` pseudo operations by letting (real) operations access the global registers directly. The remaining operations are implemented by copy operations. The reason for not eliminating all `rdreg` and `wrreg` pseudo operations is that doing this might introduce false dependences. Therefore, the scheduler can trade off resource usage for scheduling freedom. Consider the following code:

```
...
1 rdreg(5);
2 iaddi(1) 1;
...
```

```
7 isubi(1) 6;
8 wrreg(5) 7 after 1;
...
```

Eliminating the `rdreg` and `wrreg` by letting operation 2 read directly from global register `r5` and letting operation 7 write directly to `r5` will create an anti-dependence between operations 2 and 7. This can limit the scheduling freedom of them as well as that of operations that are related to them, e.g., operations that use value 7. To make the trade-off between scheduling freedom and resource usage, the TriMedia scheduler makes use of the *early* value of an operation. This is the earliest cycle in which an operation can be scheduled on an infinite resource machine. The heuristic used for the trade-off is that early values may not be increased by false dependences. This heuristic tries to retain scheduling freedom as much as possible.

4.2 Integrated local register allocation

The static single assignment form of local live ranges greatly simplifies integrated local register allocation. All live ranges are tree-shaped and definitions are scheduled before usages are scheduled. The scheduler performs integrated local register allocation by assigning a local register to a value produced by an operation at the moment the operation is scheduled. To do this the scheduler maintains the set of registers that are in use during each instruction and the registers that are in use on the boundary between the scheduled part of a basic block and the not yet scheduled part. The latter is used to initialize register usage information for newly created instructions.

4.3 Register allocation in the presence of phi operations

Phi operations are used to create static single assignment in the presence of guarded operations. Single assignment simplifies the register allocator as well as other components of the scheduler. The scheduler treats phi operations as normal operations except for the fact that they do not consume resources and have zero cycle latency. Furthermore, the register allocator must arrange that both source values and result value are assigned to the same local register. The register allocator does this by checking whether a value defined by a non-phi operation is used by a phi operation and if so whether one of the values that it is merged with has already been assigned to a local register. If the same local register is assigned. In case a value is defined by a phi operation, the register allocator assigns the same register to it as the source values are assigned to.

Live ranges of values pass through phi operations. Therefore, if a register defined by operation o_1 is used by phi operation o_2 and o_2 defines a register that is used by operation o_3, then the register defined by o_1 is live (in use) between o_1 and o_3.

Currently register allocation is not guard-aware which potentially results in more efficient register allocation [20]. Register allocation is seldom a problem for TM1000 in control intensive code where if-conversion is applied. This unlike tuned multimedia kernels which are usually not control intensive. Furthermore, in order to improve scheduling freedom and reduce code size, guarding is only applied in situation where global state is affected or values are merged. Therefore, definition points of overlapping mutual exclusive live ranges do not have to be guarded which would be necessary to allocated them to the same register.

4.4 Controlling register pressure

Although TM1000 has 128 general purpose registers, which is much more than contemporary RISC architectures, registers are a precious resource in tuned key components of multimedia applications like 2 dimensional IDCTs (inverse discrete cosine transforms). To tune applications for TriMedia, application programmers use *restrict* pointers to aid alias analysis and replace small arrays (usually 8×8 in the case of video algorithms) by scalar variables which will be allocated to registers. The TriMedia scheduler has two mechanisms to deal with code where plain list scheduling is creating too much register pressure such that the register allocator has to spill heavily: dynamic scheduling priority and scheduling floaters as late as possible. In the following we will describe the two mechanisms used by the TriMedia scheduler to deal with register pressure.

4.4.1 Dynamic scheduling priority

The first mechanism is to keep track of the fraction of registers that are in use in the last instruction I of the currently scheduled basic block. When this fraction exceeds 75% (an empirically determined value), then an extra component is added to the priority function described in Section 3.2.2. This component equals the increase of registers that would be available in I if the operation where selected for scheduling. The effect of this priority component is that extra priority will be given to operations that decrease register pressure and less priority to operations that worsen it.

4.4.2 Scheduling floaters as late as possible

The second mechanism to keep register pressure under control involves the notion of *floater* operations. Floaters are operations that have either no predecessors in the data dependence graph of the decision tree or at most one predecessor that is also a floater[4]. Furthermore, results of floaters are used only once. These operations are called floaters because they tend to float to the top of the decision tree when a list scheduler schedules them as-soon-as-possible. Figure 7a shows a dependence graph in which floaters are marked. Scheduling the code without treating floaters specially leads to a schedule as shown in Figure 7b; all floaters are scheduled in the first two instructions. This schedule needs 5 registers. A schedule using 3 registers as shown in Figure 7c is possible by scheduling floaters differently. In the TriMedia scheduler this is realized as follows. First, floaters are not in the ready list and are therefore not selected for scheduling. When a non-floater is scheduled, its preceding floaters, if any, are scheduled as close as possible before it, i.e., as late as possible. Whenever this is not possible, because the floater reached the first cycle of the decision tree or no register is available for the result of the floater, the non-floater is delayed by one cycle. This is repeated until the non-floater and all its preceding floaters are scheduled. Although the reduction from 5 to 3 may seem small in this example, this technique can have significant impact on highly parallel decision trees of hundreds or thousands of operations, amounts which are not unusual in tuned multimedia applications. For example, a tuned 8×8 DCT function, including memory access functions, takes 830 operations on TM1000.

4. The restriction of one predecessor simplifies the implementation. The concept could be generalized.

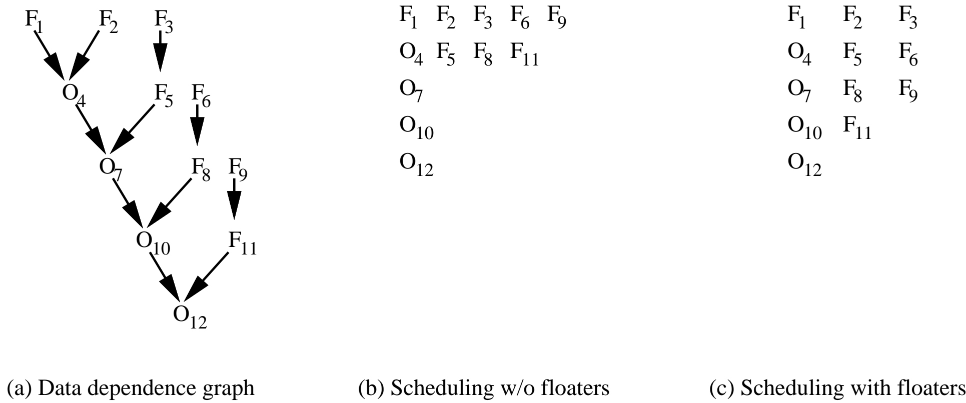

Figure 7: Data dependence graph and schedules to illustrate floaters. O_x are non-floaters, F_x are floaters.

4.5 Spilling

When it turns out that there is no free register available for an operation o scheduled in instruction i of basic block b, the scheduler has to decide whether to spill a register to memory or not to schedule o in i but in a later instruction. The following decisions are made by the scheduler. First, speculatively executed operations should not cause spilling. These operations are scheduled in an instruction after i or not in b at all. Second, if there is no register available for scheduling o in i but sufficient registers are available at the end of b, then scheduling of o will be delayed until an instruction after i in which a free register is available. If neither is the case, a register is selected as defined by an operation scheduled before i. This register should have a live range such that when spilled to memory a free register becomes available for o in an instruction after i. Selection is based on the number of reload operations that have to be generated. Currently, only local live ranges are selected for spilling to memory.

Because decision trees of the TriMedia compiler do not contain function calls, it is possible to spill registers to static memory; this simplifies spilling. To spill and reload a register, the scheduler generates a store at the definition point, and loads for each consumer of the register which has not yet been scheduled. Each load and store has an associated `uimm` operation that loads the static memory address which is used for spilling. Since a register is required between the `uimm` operation and the actual spill and reload operations, four registers are reserved for operations that are related to spilling. Furthermore, dependences with negative delays are introduced such that operations that use spilled values can be scheduled directly after associated reload operations are scheduled and are not keeping reserved registers occupied. The `uimm` operations are floater operations. All these precautions are necessary to prevent that the scheduler might get stuck during spilling.

SPECint92	SPECint95	Proprietary
008.espresso (1)	099.go (7)	h261-encoder (11)
022.li (2)	124.m88ksim (8)	line (12)
023.eqntott (3)	132.ijpeg (9)	line3d (13)
026.compress (4)	134.perl (10)	md5 (14)
072.sc (5)		pharos (15)
085.gcc (6)		mpeg-audio-dec (16)
		mpeg-video-dec (17)
		tristrip (18)

Table 2: Benchmarks used for evaluation. The numbers shown between parentheses are used to refer to them.

5. Experiments

In this section we report on experiments that measure various aspects of instruction scheduling. TM1000 as described in Section 2 and the TM1000 data book [18] is used as target. TM1000's instruction and data caches are 32K byte and 16K byte respectively. Both caches are 8-way set-associative, and have 64 byte cache lines, a 30 cycle miss penalty, and hierarchical LRU replacement policy. The data cache is copyback and allocate on write.

As benchmark set we used 6 benchmarks of SPECint92, 4 benchmarks of SPECint95, and 8 proprietary benchmarks as listed in Table 2. Of the proprietary benchmarks h261-encoder and mpeg-video-dec are compressed video decoders, line, line3d, and tristrip are graphic benchmarks, md5 is a description benchmark, pharos is a PostScript interpreter, and mpeg-audio-dec is compressed audio decoder. Most of the proprietary benchmarks are tuned for the TriMedia architecture by means of restrict pointers and usage of multimedia operations. For all experiments we used profiling and loop unrolling. Performance was measured by a cycle true simulator of TM1000. With the exception of the experiment described in Section 5.1, all measurements described in this paper are for five issue slots.

5.1 Speedup curves

First we were interested in measuring how the compiler is able to exploit the parallelism offered by the hardware. We did this by restricting the number of operations that can be issued per instruction. We varied this number from one to five[5]. In the case of one operation per instruction, the machine corresponds to a scalar RISC machine except that its latencies are somewhat larger (the dynamic average latency varies for the benchmarks between 1.21 for line and 1.91 for h261-encoder). In the case of five operations per instruction the machine corresponds to TM1000.

Figure 8 shows the results for the 18 benchmarks when grafting was enabled. There are several reasons why performance does not scale linearly with the issue-rate. (1) The

5. Currently we are unable to experiment with configurations of more than five issue slots because the instruction format is currently fixed for five issue slots.

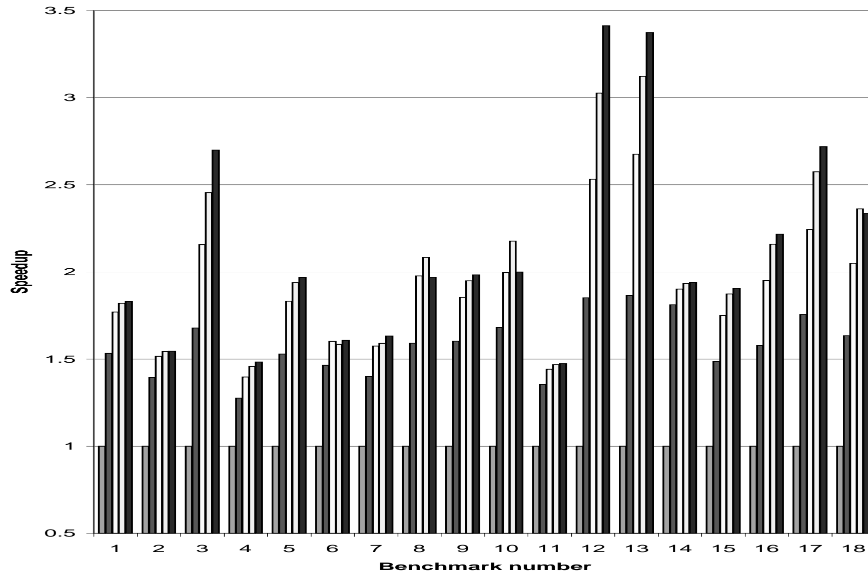

Figure 8: Speedup for different issue widths relative to single issue. The five bars per benchmark correspond to one to five issue slots. The first 6 benchmarks are from SPECint92, the next 4 are from SPECint95, and the last 8 are proprietary benchmarks. The proprietary, multimedia, benchmarks have more exploitable parallelism than the SPECint benchmarks.

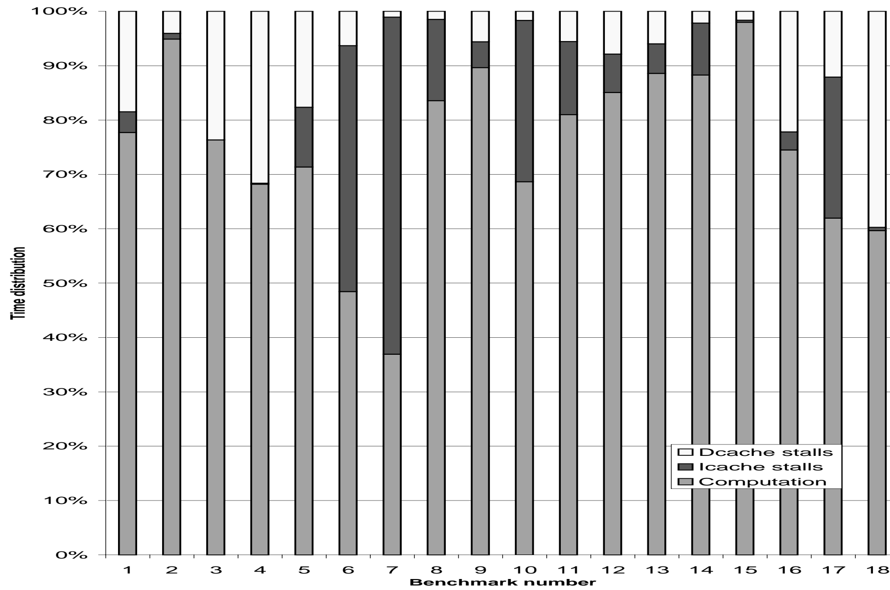

Figure 9: Fraction of time spent on cache stalls. Especially gcc (6) and go (7) spent a lot of their execution time on instruction cache stalls.

amount of ILP in non-scientific applications is limited [16], and even more so when it has to be scheduled statically. (2) A five issue machine is not five times as powerful as a single issue machine since many resources are not duplicated five times, e.g., data cache ports. (3) The time spent on cache misses remains the same and can easily become longer when code is executed speculatively. To illustrate the time spent on cache misses, Figure 9 shows how the execution time is decomposed in computation time and time needed for instruction and data cache misses. Benchmarks such as go and gcc experience many instruction cache misses and are therefore improving little from ILP exploitation.

5.2 Multi-path parallelization

In order to measure the gain of multi-path parallelization, we scheduled the benchmarks such that operations are only speculated across the most likely branch directions. This was intended to give an indication of how (guarded) decision trees compare to superblocks and hyperblocks. The experiment was performed with grafting enabled, which corresponds to some extent to tail duplication as explained in Section 3.1. Figure 10 shows the results. The average benefit on performance of multi-path parallelization is 13.4%. Most of the improvement comes from eqntott. Its well known cmppt function, which is responsible for most of the computation time, contains a loop with a few conditionals that are not clearly biased to one direction. This kind of code benefits greatly from multi-path parallelization.

There are two cases in the benchmark set where multi-path parallelization results in slightly lower performance, namely mpeg-audio-dec and tristrip. In both cases the instruction count is very similar but the miss rates for multi-path parallelization of both the instruction and data cache are slightly higher. This is a well-known phenomenon; more aggressive compilation can lead to more executed operations and data cache accesses, which can result in more instruction and data cache stall cycles than saved computation cycles.

5.3 Grafting and if-conversion

Grafting and if-conversion both aim at the same goal: creating larger decision trees. Grafting has the disadvantage that it increases code size and it puts more pressure on the instruction cache. If-conversion, on the other hand, does not have this problem, though additional operations are generated to make if-conversion possible, but it has its own limitations. Applying if-conversion too aggressively leads to the situation where frequently executed code competes for resources with less frequently executed code, or the situation where less frequently executed code determines the critical path length [21].

Figures 11 and 12 show the results of grafting and if-conversion respectively. Clearly, grafting has more effect on performance than if-conversion. Grafting improves performance by 24.4% on average and if-conversion improves performance by 9.0%. Both grafting and if-conversion have cases where performance deteriorates. In the case of grafting this occurs when the instruction cache cannot handle the increased code size, e.g., go. If-conversion shows some performance degradation when early in the compilation process the heuristics that determine what to if-convert make wrong decisions.

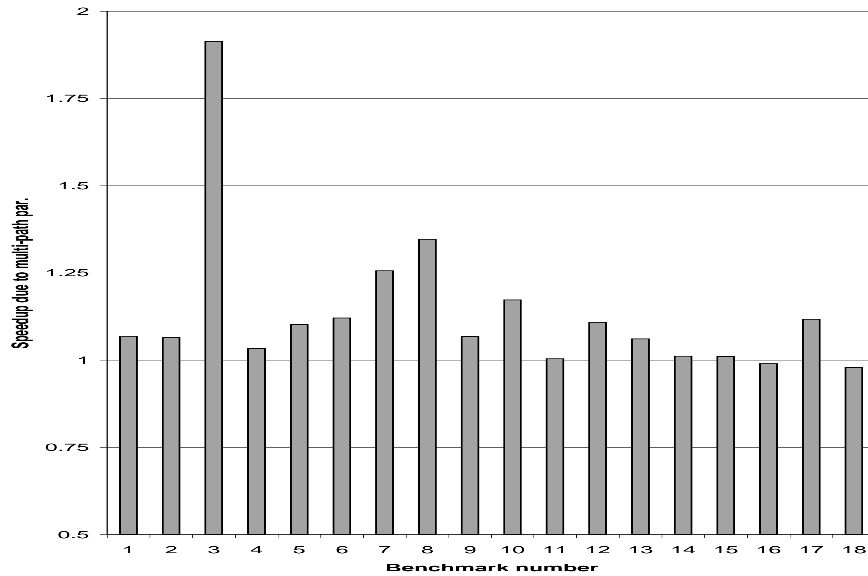

Figure 10: Speedup of multi-path parallelization (decision tree) over single-path parallelization (superblock). Eqntott (3), go (7), and m88ksim (8) show most improvement. The average improvement is 13.4%.

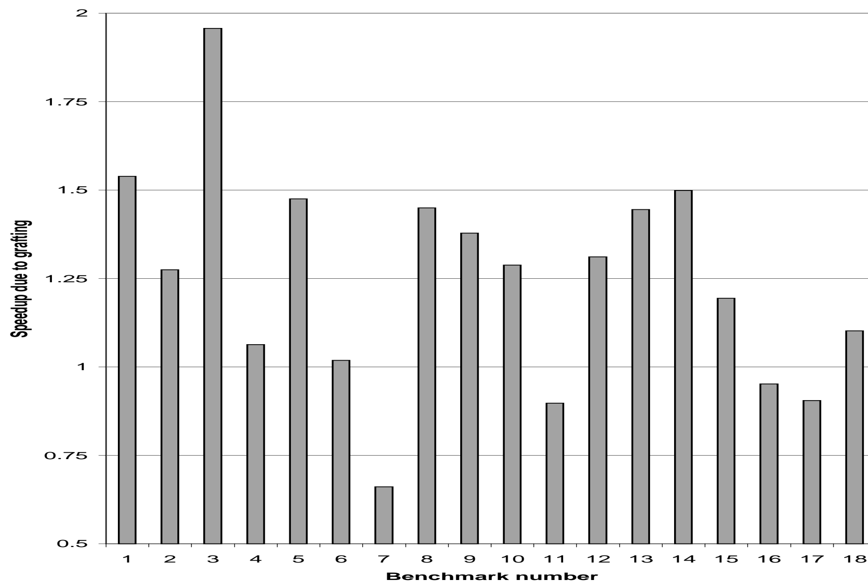

Figure 11: Speedup due to grafting. In general grafting is very effective unless the extra code leads to too much instruction cache pressure. This is happening in case of go (7). The average improvement is 24.4%.

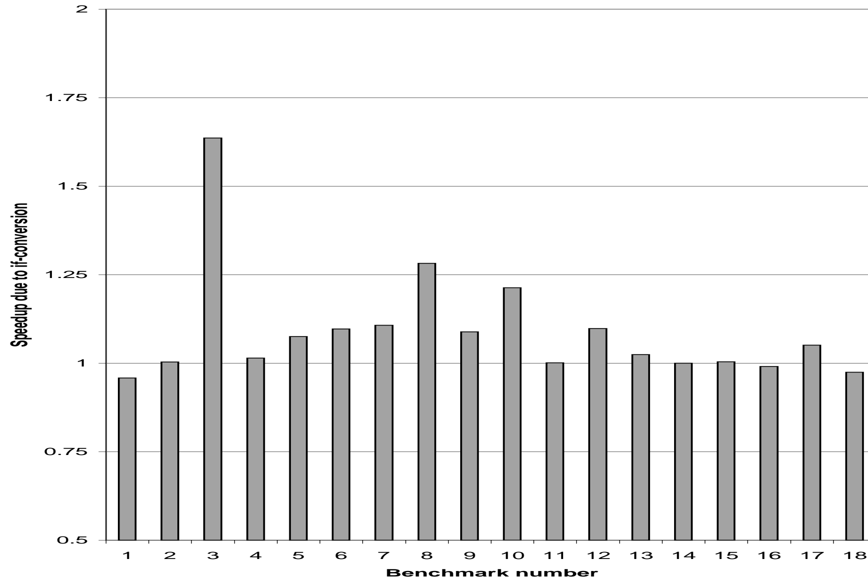

Figure 12: Speedup due to if-conversion. If-conversion is clearly less effective than grafting, but because substantially less code expansion is required it is a valuable technique. The average improvement is 9.0%.

6. Conclusions

Developing a production quality instruction scheduler is in some sense a knapsack problem. The developer has a number of design options at his disposal, each with certain engineering cost and performance impact estimations. He has to maximize performance given certain engineering costs.

For TriMedia's second generation instruction scheduler we made the following design choices: guarded decision tree scheduling unit, integrated register allocation and instruction scheduling with two mechanisms to deal with code where plain greedy list scheduling would cause severe spilling, optimistic scheduling of jump operations, and issue slot assignment by means of bipartite graph matching.

The described instruction scheduler will be shipped together with a new core compiler for C and C++ in beta release to TriMedia customers in 1998.

References

[1] G. A. Slavenburg, S. Rathnam, and H. Dijkstra, "The TriMedia TM-1 PCI VLIW Mediaprocessor," in *Hot Chips 8*, (Stanford, California), Aug. 1996.

[2] J. Labrousse and G. A. Slavenburg, "CREATE-LIFE: A Modular Design Approach for High Performances ASIC's," in *Proceedings of COMPCON '90*, 1990.

[3] J. Labrousse and G. A. Slavenburg, "A 50MHz Microprocessor with a Very Long Instruction Word Architecture," in *Proceedings of ISSCC '90*, Feb. 1990.

[4] P. Y. T. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 386–395, June 1986.

[5] G. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, California), pp. 53–62, 1991.

[6] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.

[7] W. 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," *The Journal of Supercomputing*, vol. 7, pp. 229–249, May 1993.

[8] D. Bernstein and M. Rodey, "Global Instruction Scheduling for Superscalar Machines," in *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 241–255, June 1991.

[9] S. Banerjia, W. A. Havanki, and T. M. Conte, "Treegion Scheduling for Highly Parallel Processors," in *Proceedings of the 3rd International Euro-Par Conference (Euro-Par '97)*, (Passau, Germany), pp. 1074–1078, Aug. 1997.

[10] W. A. Havanki, S. Banerjia, and T. M. Conte, "Treegion Scheduling for Wide Issue Processors," in *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA-4)*, (Las Vegas), Feb. 1998.

[11] J. A. Fisher, "Global Code Generation for Instruction-Level Parallelism: Trace Scheduling-2," Tech. Rep. HPL-93-43, Hewlett Packard Computer Systems Laboratory, Palo Alto, CA, June 1993.

[12] 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*, vol. 7, pp. 51–142, May 1993.

[13] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren, "Conversion of Control Dependence to Data Dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, Jan. 1983.

[14] 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 *Proceedings of the 25th Annual International Workshop on Microprogramming*, (Portland, Oregon), pp. 45–54, Dec. 1992.

[15] R. Cryton, J. Ferrante, B. K. Rosen, and M. N. Wegman, "An Efficient Method of Computing Static Single Assignment Form," in *Proceedings of the 16th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, (Austin, Texas), pp. 23–25, Jan. 1989.

[16] D. W. Wall, "Limits of Instruction-Level Parallelism," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, Apr. 1991.

[17] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures.* ACM Doctoral Dissertation Awards, Cambridge, Massachusetts: MIT Press, 1986.

[18] G. A. Slavenburg, *TM1000 Databook.* TriMedia Division, Philips Semiconductors, TriMedia Product Group, 811 E. Arques Avenue, Sunnyvale, CA 94088, `www.trimedia.philips.com`, 1997.

[19] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[20] D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker, "Global Predicate Analysis and its Application to Register Allocation," in *Proceedings of the 29th Annual International Workshop on Microprogramming*, (Paris, France), pp. 114–125, Nov. 1996.

[21] D. I. August, W. W. Hwu, and S. A. Mahlke, "A Framework for Balancing Control Flow and Predication," in *Proceedings of the 30th Annual International Symposium on Microprogramming*, (Research Triangle Park, North Carolina), pp. 92–103, Nov. 1997.