Operation Tables for Scheduling in the Presence of Incomplete Bypassing*

Aviral Shrivastava† Eugene Earlie‡ Nikil Dutt† Alex Nicolau†

aviral@ics.uci.edu eugene.earlie@intel.com dutt@ics.uci.edu nicolau@ics.uci.edu

Center for Embedded Computer Systems†
School of Information and Computer Science
University of California, Irvine, CA 92697

Intel Labs‡
77 Reed Road
Hudson, MA, 01749

ABSTRACT

Register bypassing is a powerful and widely used feature in modern processors to eliminate certain data hazards. Although complete bypassing is ideal for performance, bypassing has significant impact on cycle time, area, and power consumption of the processor. Due to the strict constraints on performance, cost and power consumption in embedded processors, architects need to evaluate and implement incomplete register bypassing mechanisms. However traditional data hazard detection and/or avoidance techniques used in retargetable schedulers break down in the presence of incomplete bypassing. In this paper, we present the concept of Operation Tables, which can be used to detect data hazards, even in the presence of incomplete bypassing. Furthermore our technique integrates the detection of both data, as well as resource hazards, and can be easily employed in a compiler to generate better schedules. Our experimental results on the popular Intel XScale embedded processor platform show that even with a simple intra-basic block scheduling technique, we achieve upto 20% performance improvement over fully optimized GCC generated code on embedded applications from the MiBench suite.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*Code Generation, Compilers, Optimization, Retargetable Compilers*

General Terms

algorithms, measurement, performance, experimentation

Keywords

Operation Table, Reservation Table, Bypass, Scheduling, Retargetable Compilers, Hazard Detection

*This work was partially funded by grants from Intel Corporation, UC Micro(03-028), and SRC contract 2003-HJ-1111

CODES+ISSS'04, September 8–10, 2004, Stockholm, Sweden.

1. INTRODUCTION

Bypasses or forwarding paths are simple yet powerful and widely used feature in modern processors to eliminate some data hazards[11]. With Bypasses, additional datapaths and control logic are added to the processor so that the result of an operation is available for subsequent dependent operations even before it is written in the register file. This benefit of bypassing comes with significant impact on the wiring area on the chip, possibly widening the pitch of the execution-unit datapaths. Paths including the bypasses often are timing critical and cause pressure on cycle time, especially the single cycle paths. The delay of bypass logic can be significant for wide issue machines. Due to extensive bypassing very wide multiplexors or buses with several drivers may be needed. Apart from the delay, bypass paths increase the power consumption of the processor. Thus bypasses have a significant impact, in terms of area, cycle time, and power consumption of the processor[3]. The situation is aggravated owing to the trend of long pipelines and high degrees of parallelism in modern processors.

Initial exploration experiments have shown that some bypasses may be rarely used in an applications, and a few of them may not be used at all. Thus some bypasses may be removed with a very minimal impact on performance[7]. With incomplete bypassing becoming popular in modern processors, developing retargetable compilers for such processors is important not only to easily adapt to minor changes in the design, but also for rapid and automated design space exploration of processors with such features.

Traditionally retargetable compilers use constant operation latencies to avoid data hazards[13]. Operation latency is defined as the delay (in cycles) between the cycle when the operation is issued to the cycle when dependent operations can use its results. Complete bypassing enables the subsequent dependent operations to read the value of the result as soon as the result is generated, irrespective of when it is written back in the register file. Complete bypassing reduces the operation latency. In the absence of bypassing or in the presence of complete bypassing, the operation latency can be modeled by a single value, and can be used in a DFG to detect data hazards. In processors with incomplete bypassing however, the result of an operation is only sometimes bypassed and thus available for dependent operations intermittently. Due to incomplete bypassing, the operation latency cannot be accurately described by a single value.

To our knowledge there is no existing technique that models such multi-valued operation latency, and uses it to detect data hazards. However conservative scheduling can be performed by assuming the operation latency to be equal to the latency in absence of any bypassing. Although such scheduling produces legitimate schedules (even for statically scheduled processors), it does not allow the compiler to effectively exploit available bypasses. The other option is to perform optimistic scheduling by assuming the operation latency to be equal to the latency in presence of complete bypassing. It can be shown that any scheduling technique that uses constant operation latency will produce sub-optimal results, and that a better schedule may be generated using precise latency of operations. Thus there is a growing need for a schema to detect data hazards in a retargetable compiler framework in the presence of incomplete bypassing.

We solve this problem using Operation Tables (OTs). An OT models the resources and registers used by each operation supported by the processor. OTs can then be used to detect both the resource and data hazards more accurately in an integrated manner. We also show that the OT based hazard detection mechanism can be used by most existing scheduling algorithms.

2. MOTIVATING EXAMPLE

Consider the three flavors of bypassing in a simple 5-stage pipeline shown in Figure 1. In all these pipelines we assume that the write in the register file takes place at the end of the cycle. Thus if the same register is read and written in a cycle, the old value is read.

The pipeline in Figure 1 (a) does not have any bypasses. Figure 1 (b) contains bypasses from both *EX* pipeline stage and *WB* pipeline stage to both the operands of *OR* pipeline stage. This is an example of "complete bypassing". The pipeline in Figure 1 (c) contains bypass only from *EX* pipeline stage to both the operands of *OR* pipeline stage. There is no bypass from *WB* pipeline stage. This is an example of incomplete bypassing.

Now consider the execution of an ADD operation in these pipelines. In absence of any hazards, if the ADD operation is in *F* pipeline stage in cycle i, then it will be in *OR* pipeline stage in cycle $i+2$. At this time it needs to read the two source registers. The ADD operation will then write back the destination register in cycle $i+4$, when it reaches *WB* pipeline stage. The result of the ADD operation can be read from the register file in and after cycle $i+5$.

In Figure 1 (a), there is only one way to read operands, i.e. from *RF*. Thus the operation latency of ADD is 3 cycles. Any dependent operation should be scheduled 3 cycles after ADD to avoid any data hazard. In the completely bypassed pipeline in Figure 1 (b), the operation latency of ADD is 1 cycle. A data dependent operation scheduled 1 or 2 cycles after ADD can read the result of ADD from the bypass, while a data dependent operation scheduled 3 or more cycles after ADD can read the result from *RF*. The effect of having complete bypassing is to reduce the operation latency. But in either case (no bypassing, or complete bypassing), the operation latency can be accurately described by a single value. Now consider the pipeline in Figure 1 (c). Scheduling a data dependent operation 1 cycle after ADD will not result in a data hazard, because the result of ADD can be read from *EX* pipeline stage via the bypass. However if the data dependent operation is scheduled 2 cycles after scheduling ADD, there is no way to read the result of ADD. There is a data hazard. But again, if the data dependent operation is scheduled 3 or more cycles after ADD, then the result of ADD can be read from the *RF*. Thus the data hazard can be avoided by scheduling a data dependent operation of ADD 1 cycle or 3 or more cycles after scheduling the ADD operation. The operation latency of ADD in the incompletely bypassed pipeline in Figure 1 (c) is denoted by 1, 3, which means that scheduling a data dependent operation 1 or 3 or more cycles after the schedule cycle of ADD will not cause a data hazard.

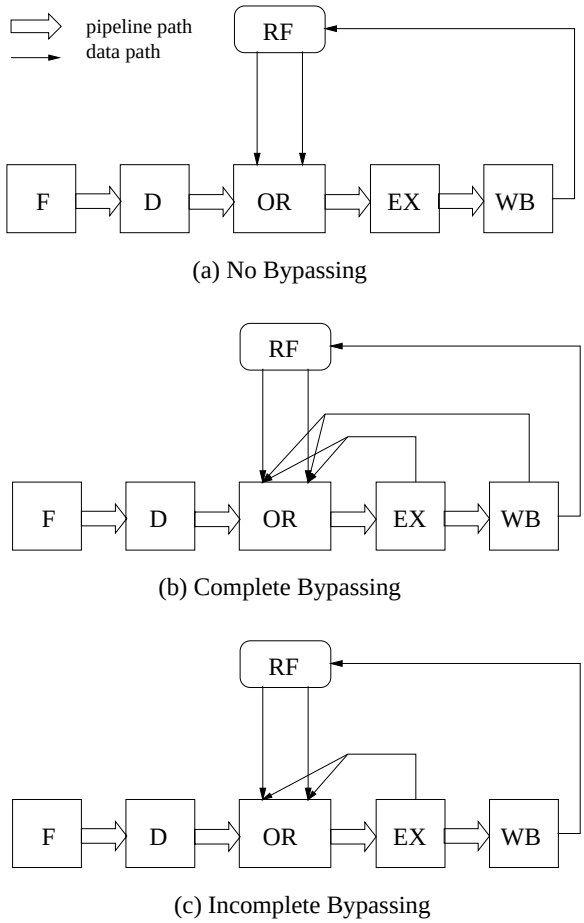

Figure 1: Pipelines with variations in bypasses

In general for similar pipelines the operation latency of an operation can be represented as, $\{l_1, l2, ...l_k\} | l_i \in N, 1 \leq i \leq k, l_i < l_{i+1}$, which implies that there will be no data hazard if any dependent operation is scheduled j cycles after scheduling the operation, such that, $j = l_i, i \leq k$, or $j > l_k$. *Thus due to incomplete bypassing, the operation latency of ADD cannot be accurately specified using just one value.* Unlike previous approaches that use a single value, in this paper we show how OTs can be used to accurately model such multi-valued latencies.

The operation latency in the presence of incomplete bypasses is very much linked to the structure of the pipeline and the presence and absence of bypasses, and the path operation takes in the pipeline. Operation Tables define a binding between an operation and the resources it may use and the registers it will read/write in each cycle of its execution. Using a resource and register model of a processor,

OTs can be used to model both the data and resource hazards.

3. RELATED WORK

Most retargetable scheduling algorithms use constant operation latencies are used to detect data hazards[13]. Most scheduling algorithms use a DFG based data hazard avoidance.

Reservation Tables (RTs)[6] or Finite State Automata (generated from Reservation Tables) are used to detect resource hazards in retargetable compiler frameworks[12, 14, 9]. Reservation tables model the structure of the processor including the pipeline of a processor and the flow of operations in pipeline. OTs, on the other hand, add register information of operands, so that *both data and resource hazards* can be effectively modeled.

Bypassing was first implemented in IBM Stretch[4]. Since then it has been used extensively to eliminate certain data hazards[11]. There is a definite trade-off between the amount of bypassing and the performance, cost and power of processor. Initial experimental results show that some bypasses are rarely used and can be removed with minimal performance penalty[3, 7, 2, 5]. But all these works either do not include the compiler in the design space exploration, or perform optimistic scheduling.

Traditional DFG based data hazard detection techniques break down in the presence of incomplete bypassing. Our approach using OTs alleviates this problem and provides a retargetable mechanism to detect (and possibly avoid) data hazards, and furthermore allows its use with many scheduling approaches.

The rest of the paper is as follows: In Section 4 we define the concept of Operation Tables (OTs), and in Section 5 we develop a data and resource hazards detection algorithm using OTs. Section 6 illustrates the working of the algorithm on a given schedule of operations on a simple processor pipeline. In Section 7 we demonstrate that OTs can be easily integrated into popular scheduling algorithms, using list scheduling as an example. In Section 8, we compare the result of a simple basic block scheduling technique using OTs with the best schedule generated by GCC. And finally in Section 9, we present a summary of this contribution.

4. OPERATION TABLE

An Operation Table (OT) is a binding between an operation and the processor resources and registers. An OT lists resources that an operation uses in each cycle of its execution. It also contains information about the registers the operation reads and writes. The operation table is a hierarchical structure defined in Table 1.

The processor pipeline divides the execution of an operation into cycles. An *otCycle* is defined corresponding to each execution cycle of an operation. The OT is defined as an ordered list of *otcycle*. Thus the length of the OT is equal to the lifetime of the operation in the pipeline, assuming no hazards. An *otcycle* specifies the list of resources that the operation may use in the cycle. It also contains a list of operands to read and write, i.e., *ReadOperands*, *WriteOperands*, and *DestOperands*. A *readOperand*, or *writeOperand* describes all the *Paths* to access (read or write) the operand *Register*. Each *path* describes a list of processor *Resources* (e.g. connection, port etc.) required

Operation Table Definition		
OperationTable	:=	[otCycle]
otCycle	:=	(Resources, ReadOperands, WriteOperands, DestOperands)
ReadOperands	:=	{readOperand}
WriteOperands	:=	{writeOperand}
DestOperands	:=	{destOperand}
readOperand	:=	(Register, Paths)
writeOperand	:=	(Register, Paths)
destOperand	:=	(Register, RegisterFile)
Paths	:=	path
path	:=	(Resources, RegisterFile)

Table 1: Operation Table Definition

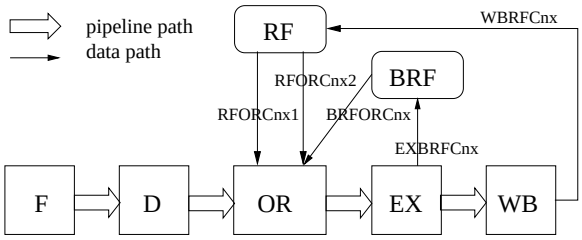

Figure 2: A simple 5-stage pipeline

to access the *Register*, from *RegisterFile*. *DestOperands* specify the *Register* that should be present in *RegisterFile* to avoid Write After Write (WAW) hazard while issuing an operation.

A register bypass can be modeled as a read and write into a bypass register. The read and write in a bypass register happen in the same cycle and the bypass value is not available in the next cycle. Although each bypass can be modeled separately in the OT, doing so results in a verbose description. Bypasses to the same operand destination can be aggregated to form a bypass register file.

Consider the pipeline shown in Figure 2 with incomplete bypassing. This pipeline has a bypass from the EX pipeline stage to OR pipeline stage. The bypass is modeled as a read and write in an imaginary register file BRF. Note that the bypass can be used to read the second operand only.

Consider the execution of an ADD operation (ADD R1 R2 R3), or ($R1 \leftarrow R2 + R3$) in the pipeline in Figure 2. Table 2 describes the Operation Table of the ADD operation. In the absence of any hazards, the ADD operation executes in 5 cycles, thus the OT of ADD contains 5 *otCycles*. In the first cycle of its execution, the ADD operation needs the F pipeline stage, and in the second cycle it needs D pipeline stage. In the third cycle, ADD occupies OR pipeline stage and needs to read its source operands R2 and R3. There is only one *path* to read the first *readOperand* R2, while there are two *paths* to read the second *readOperand* R3. The first *readOperand* R2, must be read from the register file RF via the connection $RFORCnx1$. The second *readOperand* R3, can be read either from the register file RF via the connection $RFORCnx2$, or from the register file BRF via the connection $BRFORCnx$. In the fourth cycle the ADD operation is executed and needs EX pipeline stage. The result of the operation is written to the bypass register file BRF via connection $EXBRFCnx$. WB pipeline stage is needed in the fifth cycle, and the result of the ADD operation is written back to RF via connection $WBRFCnx$.

Operation Table thus models all the resources that the

Operation Table of ADD R1 R2 R3		
1	F	
2	D	
3	OR	
		ReadOperands
		R2
		RFORCnx1, RF
		R3
		RFORCnx2, RF
		BRFORCnx, BRF
		DestOperands
		R1, RF
4	EX	
		WriteOperands
		R1
		EXBRFCnx, BRF
5	WB	
		WriteOperands
		R1
		WBRFCnx, RF

Table 2: Operation Table of ADD R1 R2 R3

operation may use during each cycle of its execution. It also models when and which registers are read and written. By combining the OTs of operations, both the data and resource hazards can be detected.

5. HAZARD DETECTION USING OTS

Operation Tables can be used to detect data and resource hazards in a schedule. However, hazard detection using OTs requires that the state of the machine be maintained to reflect the current schedule. The state of the machine is defined in Table 3. A *machineState* is an ordered list of *macCycle*. Each *macCycle* is a set of free resources, and the registers present in the register files (including the bypass register files).

Machine State		
machineState	:=	[macCycle]
macCycle	:=	(Resources, RF, BRF)

Table 3: Machine State

We define a function $detectHazard$, that detects both the data and resource hazards if operation *op* is scheduled at time t in a given *machineState*. Detection of hazards is a fundamental problem in scheduling. Given a more accurate $detectHazard$ function most existing scheduling algorithms should be able to leverage it to generate better code. A hazard between a cycle of OT (*otCycle*) and a cycle of *machineState* is detected by *detectCycleHazard*. There is no hazard if all the resources needed in the *otCycle* are available in *macCycle*, there is atleast one path to read each *readOperand*, one path to write each *writeOperand*, and all the *destOperand* are present in the corresponding register file.

Figure 4 defines functions $AvailRP$ and $AvailWP$ which return a path that is available for reading/writing an operand in the *macCycle*, or returns ϕ, if no path is available. A *path* is available for writing a *Register* if all the resources in the *path* are free. Reading a *Register* further requires the register to be present in the $RegisterFile$.

```
bool detectHazard(machineState, op, t)
  for (i = 0; i < op.OT.length; i++)
    if detectCycleHazard(machineState[t + i], op.OT[i])
      return TRUE;
  return FALSE;

bool detectCycleHazard(macCycle, otCycle)
  if otCycle.Resources ⊄ macCycle.Resources
    return TRUE;
  for each ro ∈ otcycle.ReadOperands
    if AvailRP(ro.Register, ro.Paths, macCycle) == φ
      return TRUE;
  for each wo ∈ otcycle.WriteOperands
    if AvailWP(wo.Register, wo.Paths, macCycle) == φ
      return TRUE;
  for each do ∈ otcycle.DestOperands
    regFile = do.RegisterFile;
    if do.Register ∉ macCycle.regFile
      return TRUE;
  return FALSE;
```

Figure 3: Detecting Hazards using Operation Tables

```
pathAvailRP(reg, paths, macCycle)
    foreach path ∈ Paths
        regFile = path.RegisterFile;
        if path.Resources ⊂ macCycle.Resources
          if reg ∈ macCycle.regFile
            return path;
    return φ;

pathAvailWP(reg, paths, macCycle)
    foreach path ∈ Paths
        regFile = path.RegisterFile;
        if path.Resources ⊂ macCycle.Resources
          return path;
    return φ;
```

Figure 4: Finding the available read and write path

The function *AddOperation* in Figure 5 updates the machine state after scheduling operation *op* in cycle t. First the earliest *macCycle* is found to schedule an *otCycle*, so as not to cause a hazard. Each *machineState* is updated by scheduling an *otCycle* by function *AddCycle*. A *macCycle* is updated by removing all the *Resources* required in *otCycle* from the *Resources* in *macCycle*. All the required resources for the operand reads and writes are also marked as busy. If there are *DestOperands*, *RemRegFromRegFile* removes the *Register* from *RF* in the later cycles. Similarly *WriteOperands* are added to *RF* in the later cycles by *AddRegToRegFile* function.

Thus Operation Tables can be combined to detect data as well as resource hazards. Moreover we defined a $detectHazard$ function which can be used in existing scheduling algorithms.

6. ILLUSTRATIVE EXAMPLE

Consider scheduling the sequence of three operations in the pipeline in Figure 2.
MUL R1 R2 R3 ($R1 \leftarrow R2 \times R3$)
ADD R4 R2 R3 ($R4 \leftarrow R2 + R3$)
SUB R5 R4 R2 ($R5 \leftarrow R4 - R2$)

AddOperation(machineState, op, t)

$j = t$;
for $(i = 0; i < op.OT.length; i++)$
  while detectHazard(machineState[j], op.OT[i])
    j++;
  $AddCycle(machineState[j], op.OT[i])$;

AddCycle(macCycle, opcycle)

$macCycle.Resources- = macCycle.Resources$;
for each $ro \in otcycle.ReadOperands$
  $path = AvailRP(ro.Register, ro.Paths, macCycle)$;
  $macCycle.Resources- = path.Resources$;
for each $wo \in otcycle.WriteOperands$
  $reg = wo.Register$;
  $path = AvailWP(reg, wo.Paths, macCycle)$;
  $macCycle.Resources- = path.Resources$;
  $AddRegToRegFile(reg, path.RegisterFile, j)$;
for each $do \in otcycle.DestOperands$
  $reg = do.Register$;
  $regFile = do.RegisterFile$;
  $RemRegFromRegFile(reg, regFile, j)$;

Figure 5: Update the state of the machine

The OTs of ADD and SUB are similar, except for the register indices. The MUL operation uses the same resources but spends two cycles in the *EX* pipeline stage. An operation bypasses the results only after the execution has finished. Thus a valid bypass from *EX* pipeline stage will be generated only in the second cycle of execution of MUL.

Since MUL occupies the *EX* pipeline stage for two cycles, a resource hazard should be detected between the MUL and ADD operation. SUB requires the result of ADD operation as the first operand, for which there is no bypass, so there should be a data hazard. We illustrate the detection of hazards by scheduling these three operations in-order.

Initially we assume that all the resources are free and that all the registers are available in the *RF*. There is no hazard when MUL is scheduled in the first cycle. Figure 6 shows the *machineState* after MUL is scheduled by *AddOperation*.

Schedule after scheduling MUL R1 R2 R3 in cycle 1

Cycle	Busy Resources	! RF	BRF
	Operation 1		
1.	F	—	—
2.	D	—	—
3.	OR, RFORCnx1, RFORCnx2	—	—
4.	EX	R1	—
5.	EX, EXBRFCnx	R1	R1
6.	WB, WBRFCnx	R1	—
7.		—	—

Figure 6: Schedule after scheduling MUL R1 R2 R3

If we try to schedule ADD in the next cycle, *detectHazard* detects a resource hazard. There is a resource hazard when the fourth *otCycle* of ADD is tried in the fifth *macCycle*. The resource *EX* is not free in the *macCycle*. Figure 7 shows the *machineState* after scheduling ADD in the second cycle using *AddOperation*.

Now in the existing schedule in Figure 7, if we try to schedule SUB in the third cycle, there is a data conflict. The third *otCycle* of SUB cannot read R4 from *RF*. *AvailRP* returns ϕ because even though the connection *RFORCnx1* is free, R4 is not present in *RF*. The data hazard is resolved in the eighth cycle of *machineState*. Figure 8 shows *machineState* after SUB is scheduled in the third cycle using *AddOperation*.

Schedule after scheduling ADD R4 R2 R3 in cycle 2

Cycle	Busy Resources		! RF	BRF
	Operation 1	Operation 2		
1.	F		—	—
2.	D	F	—	—
3.	OR, RFORCnx1, RFORCnx2	D	—	—
4.	EX	OR, RFORCnx1, RFORCnx2	R1	—
5.	EX, EXBRFCnx	Resource Hazard	R1 R4	R1
6.	WB, WBRFCnx	EX, EXBRFCnx	R1 R4	R4
7.		WB, WBRFCnx	R4	—
8.			—	—

Figure 7: Schedule after scheduling ADD R4 R2 R3

Thus Operation Tables can be used to accurately detect both data and resource conflicts, even in the presence of incomplete bypassing.

Schedule after scheduling SUB R5 R4 R2 in cycle 3

Cycle	Busy Resources			! RF	BRF
	Operation 1	Operation 2	Operation 3		
1.	F			—	—
2.	D	F		—	—
3.	OR, RFORCnx1, RFORCnx2	D	F	—	—
4.	EX	OR, RFORCnx1, RFORCnx2	D	R1	—
5.	EX, EXBRFCnx	Resource Hazard	Data Hazard	R1 R4	R1
6.	WB, WBRFCnx	EX, EXBRFCnx	Data Hazard	R1 R4	R4
7.		WB, WBRFCnx	Data Hazard	R4	—
8.			OR, RFORCnx1, RFORCnx2	R5	—
9.			EX, EXBRFCnx	R5	R5
10.			WB, WBRFCnx	R5	—
11.				—	—

Figure 8: Schedule after scheduling SUB R5 R4 R2

7. INTEGRATING OTs IN A SCHEDULER

Detection of hazards is a fundamental problem in scheduling. Our OT-based approach generates accurate hazard detection information, allowing any traditional scheduling algorithm to perform better. For the sake of illustration we demonstrate in Figure 9, how to integrate OT-driven hazard detection mechanism into a list scheduler We believe OTs can similarly be integrated into other scheduling formulations.

List scheduling schedules a data dependence graph $G = (V, E)$, where each vertex corresponds to an operation, and there is an edge between v_1 and v_2 if v_2 uses the result of v_1. Vertex v_0 and v_n are unique start and end node. The function $parents(v)$ gives a list of all the parents of v.

8. EXPERIMENTS

To demonstrate the need and efficacy of Operation Tables, we performed scheduling experiments on the Intel XScale[1]. XScale is a popular embedded processor for wireless and handheld devices. It provides high code density, high performance and low power, all at the same time. Figure 10 shows the 7-stage out of order superscalar pipeline of XScale. XScale implements dynamic scheduling using register scoreboarding. Note that XScale uses incomplete bypassing. We present experimental results on benchmarks from MiBench[8] suite, which is representative of typical embedded applications.

ListSchedule(V)

$U = V - v_0; F = \phi; S = v_0;$
for each $v \in V$
  $schedTime[v] = 0;$
while $(U \neq \phi)$
  $F = \{v | v \in U, parents(v) \subset S\}$
  for each $(v \in F)$ /* by priority */
    $t = Max(schedTime(p)), p \in parents(v)$
    while $(detectHazard(machineState, v.op, t))$
      t++;
    $AddOperation(machineState, v.op, t);$
    $schedTime[v] = t;$

Figure 9: List Scheduling using Operation Tables

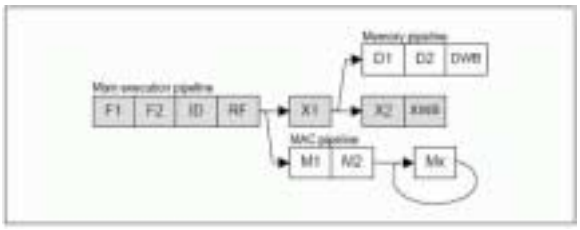

Figure 10: 7-stage pipeline of XScale

We compiled the embedded applications from MiBench suite using GCC cross compiled for XScale. The benchmarks were compiled using the -O3 option to optimize for performance. We modeled XScale in EXPRESSION[9], and generated both, a cycle accurate simulator SIMPRESS, and a bypass aware, OT based compiler EXPRESS[10]. We implemented a simple intra-basic block scheduling in EXPRESS using OTs for data and resource hazard detection. EXPRESS recompiles the output of GCC. All the optimizations in EXPRESS were turned off to isolate the effects of OT based hazard detection. The performance of the original GCC code and the EXPRESS code were measured on SIMPRESS and the performance improvement was computed and plotted in Figure 11.

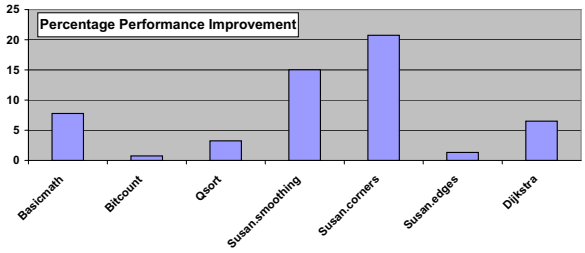

Figure 11: Experimental Results

We observe upto 20% performance improvement over the best scheduled code by GCC in the case of *susan.corners*. *susan.corners* is an image processing algorithm, which finds and marks the corners in an image. Using OTs, the scheduler was able to detect a data conflict in the innermost loop of the corner detection algorithm, and was able to find a schedule that avoided the conflict. A profiling of the results gave us some insight into the results. In the bitcount benchmark, the scheduling could not find a better schedule than GCC in the frequently executed loops. In the susan.edges benchmark, cache effects disturbed our OT based fine grain scheduling technique. Frequent cache misses disrupt the hazard predictions of the compiler. Thus although the benefits achieved by a scheduler using OTs may be diminished by coarse grain effects like cache misses, *it is always beneficial*. In fact on an average it performs 8% better than the best schedules generated by GCC on the set of benchmarks.

9. SUMMARY

Although register bypassing increases performance by eliminating certain data hazards, it has significant impacts on wiring area, cost, power and complexity of the processor. Incomplete bypassing however results in complicated effective latency values of operations, which cannot be used by traditional hazard detection mechanism employed in the compilers today. We present a novel approach to detect data as well as resource hazards even in presence of incomplete bypassing. We have shown that our hazard detection technique is generic and can be used in most existing scheduling algorithms. Our experiments show that accurate modeling and detection of data hazards due to incomplete bypassing results in upto 20% performance improvements over the best performing code generated by GCC on benchmarks from MiBench applications for Intel XScale. Future work will investigate the use of OTs in modeling other complex dynamically scheduled processors and explore the retargetability of our approach.

10. REFERENCES

[1] Intel xscale microarchitecture programmers reference manual.

[2] A. Abnous and N. Bagerzadeh. Pipelining and bypassing in a vliw processor. In *IEEE trans. on Parallel and Distributed Systems*, 1995.

[3] P. Ahuja, D. W. Clark, and A. Rogers. The performance impact of incomplete bypassing in processor pipelines. In *Proc. of Symposium on Microarchitecture MICRO-28*, 1995.

[4] E. Bloch. The engineering design of the stretch computer. In *Proc. of Eastern Joint Computer Conference*, pages 48–59, 1959.

[5] M. Buss, R. Azavedo, P. Centoducatte, and G. Araujo. Tailoring pipeline bypassing and functional unit mapping for application in clustered vliw architectures. In *Proc. of CASES*, 2001.

[6] E. S. Davidson. The design and control of pipelined function generators. *Int. IEEE Conf. on Systems Networks and Computers*, pages 19–21, 1971.

[7] K. Fan, N. Clark, M. Chu, K. V. Manjunath R. Ravindran, M. Smelyanskiy, and S. Mahlke. Systematic register bypass customization for application-specific processors. In *Proc. of IEEE Intl. Conf. on ASSAP*, 2003.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE Workshop in workload characterization*, 2001.

[9] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of Design Automation and Test in Europe*, 1999.

[10] A. Halambi, A. Shrivastava, N. Dutt, and A. Nicolau. A customizable compiler framework for embedded systems. In *SCOPES*, 2001.

[11] P. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 1990.

[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"*, 7(1-2):51–142, 1993.

[13] S. Muchnick. *Advanced Compiler Design and Implementation*. 1998.

[14] The Trimaran Consortium. *The Trimaran Compiler Infrastructure for Instruction Level Parallelism*.