Automatic Generation of Operation Tables for Fast Exploration of Bypasses in Embedded Processors

Sanghyun Park§
shparkid@compiler.snu.ac.kr

Aviral Shrivastava†
aviral@ics.uci.edu

Nikil Dutt†
dutt@ics.uci.edu

Eugene Earlie‡
eugene.earlie@intel.com

Alex Nicolau†
nicolau@ics.uci.edu

Yunheung Paek§
ypaek@ee.snu.ac.kr

ACES Lab, CECS†
School of ICS,
UC Irvine, CA 92697

SO&R Labs§
Department of EE,
SNU Seoul, South Korea

Strategic CAD Labs‡
Intel Corporation,
Hudson, MA, 01749

ABSTRACT

Customizing the bypasses in an embedded processor uncovers valuable trade-offs between the power, performance and the cost of the processor. Meaningful exploration of bypasses requires bypass-sensitive compiler. Operation Tables (OTs) have been proposed to perform bypass-sensitive compilation. However, due to lack of automated methods to generate OTs, OTs are currently manually specified by the designer. Manual specification of OTs is not only an extremely time consuming task, but is also highly error-prone. In this paper, we present **AutoOT***, an algorithm to automatically generate OTs from a high-level processor description. Our experiments on the Intel XScale processor model running MiBench benchmarks demonstrate that AutoOT greatly reduces the time and effort of specification. Automatic generation of OTs makes it feasible to perform full bypass exploration on the Intel XScale and thus discover interesting alternate bypass configurations in a reasonable time. To further reduce the compile-time overhead of OT generation, we propose another novel algorithm,* **AutoOTDB***. AutoOTDB is able to cut the compile-time overhead of OT generation by half.*

1. INTRODUCTION

Modern embedded processors are deeply pipelined and employ extensive bypassing to improve performance. Bypassing improves the performance of a pipelined processor by eliminating certain data hazards. However, extensive bypassing may have significant impact on the cycle time, wiring congestion, power consumption, area and the overall chip complexity [1]. Embedded processor systems have strict multi-dimensional constraints, like power, performance, cost etc. In order to be able to meet all the design constraints *in-chorus*, embedded systems need to *customize* bypassing.

Customization of bypassing implies keeping only the "most benficial" bypasses and removing the "less needed" ones. Owing to the lack of bypass-sensitive compilation techniques, this decision is primarily based on designers intuition and/or a *simulation-only* exploration. In a simulation-only exploration, the same binary is executed (simulated) on several processor models, and the best performing processor model is chosen. However, the presence/absence of bypasses effects the pipeline hazards that occur on a given schedule of instructions. The significance of this effect can lead to incorrect performance and power estimations and result in sub-optimal design decisions [11]. To perform meaningful exploration of bypasses, a *bypass-sensitive* compilation technique is required. Recently a bypass-sensitive instruction scheduling technique has been presented by Shrivastava et al.[12]. The key idea of the proposed technique is to use *Operation Tables* (OTs) to detect all pipeline hazards in a given schedule. The Operation Table, OT of an operation specifies the processor resources and registers the operation will use during its execution. Meaningful bypass exploration can be performed by generating bypass-sensitive code for the set of bypasses present in the processor, and then executing (simulating) it on the processor model (with the same set of bypasses). Such an exploration is called Compiler-in-the-Loop (CIL) exploration. In a CIL exploration, architecture-sensitive compilation has to be performed in order to evaluate each design point. OT-based Compiler-in-the-Loop (CIL) exploration is therefore *a must* for meaningful Design Space Exploration (DSE) of bypasses in pipelined embedded processors.

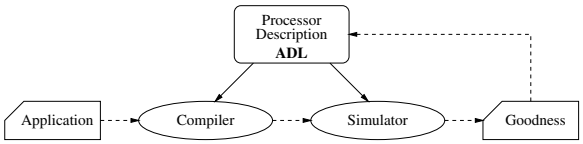

Figure 1: ADL based CIL DSE

Another important characteristic of embedded systems is the *short time-to-market*. Short time-to-market makes it imperative to employ automated DSE methodologies in embedded processor development. Architecture Description Language (ADL) based techniques are a popular means of performing top-down and automated DSE of processors. As shown in Figure 1, in the ADL-based CIL DSE framework, the processor architecture is described in an ADL, and a cycle-accurate simulator, and an optimizing compiler are parameterized over the processor description in ADL. An architectural modification can be evaluated by compiling the application using the architecture-sensitive compiler and executing the resulting binary on the generated cycle-accurate simulator.

3-9810801-0-6/DATE06 © 2006 EDAA

Currently OT-based CIL DSE is performed by manually specifying the OTs of the operations in a processor. However, even moderately complex embedded processor cores like the Intel XScale may have a large number of OTs. Manually specifying all the OTs is not only tedious and time consuming, but is an extremely error-prone task. Furthermore, during DSE, OTs may need to change as a result of a change in the processor architecture, e.g., adding/removing pipeline. In the absence of automated methods of generating OTs, such situations need to be manually identified and the appropriate modifications be done to the OTs. Manually specifying OTs is therefore a major bottleneck in automated DSE of bypasses.

In this paper, we present **AutoOT**, an algorithm to **Auto**matically generate **O**peration **T**ables from a high-level processor description (in an ADL). Instead of creating OTs by hand, designer now only has to specify a high-level processor description. Our experiments on the Intel XScale processor demonstrate that AutoOT can reduce the specification effort by 3X. Additionally, the high-level processor specification may be used for several other purposes, like verification, simulation etc. In addition to the reduction of the first time specification effort, AutoOT reduces the specification effort in each step of architectural exploration by approximately 300X. Thus for long-runs of DSE, AutoOT is indispensable. AutoOT generates OTs on-demand and therefore has high compile-time overhead. To reduce this overhead, we propose another novel technique **AutoOTDB**, which pre-generates partial OTs and stores them in a database. At compile-time, minimal data-dependent effort is required to create the OTs, cutting the compile-time overhead by half.

2. RELATED WORK

Several approaches have been proposed for Architecture Description Language (ADL) based Design Space Exploration (DSE) [15, 4, 6, 7, 13, 10, 8]. Several of the newer ADLs have focused on Compiler-in-the-Loop (CIL) exploration of processors.

Many ADLs and the corresponding "retargetable compilers" of CIL ADLs use Reservation Tables (RTs) to detect and avoid resource conflicts while scheduling. The concept of RTs to represent the resources used by individual instructions in each stage of pipeline was developed in [2, 3]. Resource conflicts between instructions are identified by adding their RTs. ADLs capture resource hazards between instructions by either specifying explicit RTs on a per instruction basis [6], or as an attributed grammar that allowing only legal combinations [4], or only non-legal combinations [7]. However it was realized that manual specification of RTs introduces redundancy in the processor description. In addition, during DSE, structural changes to the processor architecture may propagate through the description, requiring the user to manually update the changes in the RT section also, especially if these changes are not obvious.

Manual specification of the conflicts was recognized as a tedious and error-prone task, especially for the complex and long-pipeline processors. RTGen [5] was the first attempt to automatically generate Reservation Tables from high-level structural processor description in EXPRESSION [8]. PIPEGEN [9] is another approach to automatically generate RTs from an even lower-level description of the processor.

However, owing to the recent advances in process technology, silicon resources becoming cheap, and the rising processor clock, the share of resource conflicts in modern processors is reducing. The balance is shifting more towards data hazards. Data hazards (e.g. Read After Write hazard) are a result of linear execution semantics of most existing functional programming languages on pipelined execution of instructions. This imposes a partial order in which the operands of instructions can be read or written. This order is enforced in most processors by stalling in the presence of data hazards.

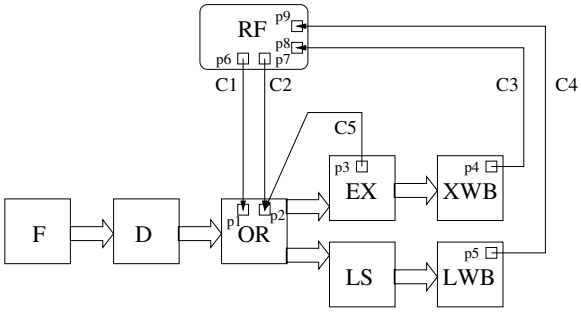

Figure 2: Example Pipeline

Traditional data hazard detection techniques employed in the retargetable compilers break down in the presence of partial bypassing. Operation Tables (OTs) can detect and avoid data hazards in a retargetable compiler. In addition OTs integrate the detection and avoidance of both data and resource hazards in such processors. However, to perform automated design space exploration, there is a need to generate OTs automatically from the processor description in ADL.

In this paper we present techniques to automatically generate OTs from high-level processor structure description in the EXPRESSION ADL.

3. PROCESSOR MODEL

In this section, we define the processor model. We will then define operations on this processor model, and then describe the automatic generation of Operation Tables for an operation on the processor model.

3.1 Pipeline Model

A pipelined processor can be divided into pipeline units by the pipeline registers. The processor pipeline can be represented as a Directed Acyclic Graph (DAG) of the pipeline units, $u_i \in U$ which represent the nodes of the DAG, and a directed edge (u_i, u_j) represents that operations may flow from unit u_i to unit u_j. There is a unique "source node", u_0, to which there are no incoming edges. This unit generates operations. Further, some nodes are "sink nodes", which do not have any outgoing edges. These nodes represent writeback units. In the pipeline shown in Figure 2, F is the source unit and XWB and LWB are the writeback units. The operations flow along the block arrows.

3.2 Operation Model

Each operation $o_i \in O$ supported by the processor is defined using an *opcode* $o_i.opcode$ and a list of source and destination operands, $o_i.sourceOperands$ and $o_i.destOperands$. The *opcode* defines the path of the operation in the processor pipeline. Each source or destination operand, *operand*

is defined by a 3-tuple, <*arg, rf, rn*>, where arg is the argument of the operand, rf is the register file it belongs to, (or IMM for immediate operands), and rn is the register number (or immediate value for immediate operands). The operand argument describes how to read/write the operand. Thus the operation, *ADD R1 R2 5*, has opcode *ADD*, and has one destination operand and two source operands. The destination operand is represented by <*D_1, RF, 1*>. The first source operand is represented as <*S_1, RF, 2*>, and the third as <*S_2, IMM, 5*>.

3.3 Pipeline Path of Operation

The pipeline path of an operation o_i is the ordered list of units that an operation flows through, starting from the unique source unit u_0, to at least one of the writeback units. Each unit $u_i \in U$ contains a list of operations that it supports, $u_i.opcodes$. The add operation, *ADD R1 R2 5* has opcode *ADD*, and the pipeline units F, D, OR, EX and XWB have the *ADD* operation in the list of opcodes they support.

3.4 Register File

We define a register file as a group of registers that share the read/write circuitry. A processor may have multiple register files. The processor in Figure 2 has a register file named *RF*.

3.5 Ports in Register File

A register file contains read ports and write ports to enable reading and writing of registers from and to the register file. Register operands can be read from a register file rf via read ports, $rf.readPorts$, and can be written in rf via write ports, $rf.writePorts$. Register operands can be transferred via ports through register connections. The register file *RF* in the processor in Figure 2, has two read ports ($p6$ and $p7$) and two write ports ($p8$ and $p9$).

3.6 Ports in Pipeline Units

A pipeline unit, u_i can read register source operands via its read ports, $u_i.readPorts$, write result operands via its write ports, $u_i.writePorts$, and bypass results via its bypass ports, $u_i.bypassPorts$. Each port in a unit is associated with an argument arg, which defines the operands that it can transfer. For example a $readPort$ of a unit with argument S_1 can only read operands of argument S_1. In the processor in Figure 2, pipeline unit *OR* has 2 read ports, $p1$ and $p2$ with arguments, S_1 and S_2 respectively. The units, *XWB* and *LWB* have write ports $p4$ and $p5$ respectively with arguments D_1, and D_2 respectively while *EX* has a bypass port $p3$ with argument D_1.

3.7 Register Connection

A register connection rc facilitates register transfer from a source port $rc.srcPort$ to destination port $rc.destPort$. In the processor diagram in Figure 2, the pipeline unit *OR* can read two register source operands, first from the register file *RF* (via connection *C1*), and second from *RF* (via connection *C2*) as well as from *EX* (via connection *C5*). The register connection C5 denotes a bypass.

3.8 Register Transfer Path

Register transfers can happen from a register file to a unit (register read), from a unit to a register file (a writeback operation), and even between units (register bypass). The register transfers in our processor are modeled explicitly via ports. A register transfer path is the list of all the resources used in a register transfer, i.e., the source port, the register connection, the destination port, and the destination register file or unit.

Operation Table Definition		
OperationTable	:=	{ otCycle }
otCycle	:=	unit ros wos bos dos
ros	:=	ReadOperands { operand }
wos	:=	WriteOperands { operand }
bos	:=	BypassOperands { operand }
dos	:=	DestOperands { regNo }
operand	:=	regNo { path }
path	:=	port regConn port regFile

Table 1: Operation Table Definition

4. OPERATION TABLE

An Operation Table (OT) describes the execution of an operation in the processor. OT is a DAG of $OTCycles$; each $OTcycle$ describes what happens in each execution cycle, while the directed edges between $OTCycles$ represent the time-order of $OTCycles$. Each OTCycle describes the unit in which the operation is, and the operands it is reading ros, writing wos and bypassing bos in the execution cycle. The destination operands dos are used to indicate the destination registers, and are required to model the dynamic scheduling algorithms in the processor. Each *operand* that is transferred (i.e., read, written, or bypassed) is defined in terms of the register number, $regNo$, and all the possible $paths$ to transfer it. A $path$ is descibed in terms of the ports, register connections and the register file involved in the transfer of the operand.

Table 2 shows the OT of the add operation, *ADD R1 R2 5*. In the absence of any hazards, the add operation executes in *5* cycles, therefore the OT of the add operation 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, the add operation occupies *OR* pipeline stage and needs to read its source operands *R2* and *5*. All the paths to read each *readOperand* are listed. The first *readOperand, R2* can be read only from the *RF* via

Operation Table of ADD R1 R2 5	
1	F
2	D
3	OR
	ReadOperands
	R2
	p1, C1, p6, RF
	DestOperands
	R1, RF
4	EX
	BypassOperands
	R1
	p3, C5, p2, OR
5	WB
	WriteOperands
	R1
	p4, C3, p8, RF

Table 2: Operation Table of ADD R1 R2 5

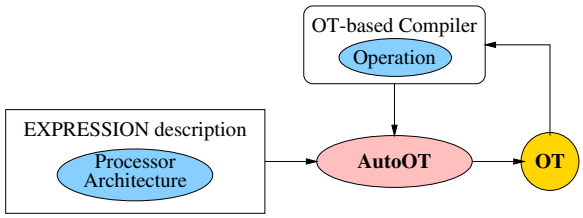

Figure 3: AutoOT Flow

connection *C1*. The second operand is immediate and no resources are required to read it. Since the sources are read in this cycle, the *destOperands* are listed. In the fourth cycle the add operation is executed and needs *EX* pipeline stage. The result of the operation *R1* is bypassed via connection *C5*. It can be read as the second operand of the operation occupying the *OR* unit. *WB* pipeline stage is needed in the fifth cycle. In the otCycle the result of the add operation *R1* is written back to *RF* via connection *C3*.

5. AUTOMATIC GENERATION OF OT

Figure 3 shows the interface of our automatic OT generation algorithm *AutoOT* with the compiler. AutoOT takes the high-level description of the processor as an input and generates OTs *on-demand* for the compiler. Figure 4 outlines the algorithm to automatically generate the OT of an operation from a high-level processor description (e.g. an ADL). First the pipeline path of the operation is discovered recursively starting from the unique source unit u_0. An *otCycle* is generated for each execution cycle of the operation, and the OT is formed by appending them as per the flow of the operation in the processor pipeline DAG.

The function *CreateOTCycle* described in Figure 5 creates *otCycle* for each cycle of execution of an operation. A *readOperand* of an operation can be read only if there are ports in the unit, which can read the operand (line 04). A port can read an operand, if they share the same argument. When an operand can be read, all the possible paths to read the operand are discovered (lines 08-15), and listed in the OT. An operand can be read from all the register connections to the port, that have the same source register file as that of the operand (line 10). It can also be read from register connections that have a unit as a source (that means it is a bypass) (line 13). Similarly all the paths for writing (lines 21-28) and bypassing (lines 29-36) operands are discovered and listed in the OT. In the cycles when the operands are read, the destination operands are also listed (lines 37-40).

```
GenerateOT(Operation op)
01: processorPipeline = (U, E)
02: u0 = processorPipeline.root()
03: otCycle = GenerateOTCycle(op, u0)
04: return otCycle

GenerateOTCycle(Operation op, Unit u)
01: otCycle = createOTCycle(op, u)
02: foreach (c ∈ U : (u, c) ∈ E)
03:   if (op.opcode ∈ c.opcodes())
04:     childOT = generateOTCycle(op, c)
05:     otCycle.addChildOT(childOT)
06: return otCycle
```

Figure 4: GenerateOT

```
CreateOTCycle(Operation op, Unit unit)
01: otCycle = new OTCycle(unit)
02: foreach (opnd ∈ op.sourceOperands)
03:   < arg, rf, rn > = opnd
04:   if (unit.reads(arg))
05:     if (rf ≠ IMM)
06:       ro = new OperandInfo(rn)
07:       rp = unit.getReadPort(arg)
08:       foreach (rc ∈ rp.registerConnections())
09:         if (rc.srcPort.inRF())
10:           ro.addPath(rp, rc, rc.srcPort, rc.srcPort.RF())
11:         else // (rc.srcPort.inUnit())
12:         endIf
13:         ro.addPath(rp, rc, rc.srcPort, rc.srcPort.Unit())
14:         otCycle.addReadOperand(ro)
15:       endFor
16:     endIf
17:   endIf
18: endFor

19: foreach (opnd ∈ op.writeOperands)
20:   < arg, rf, rn > = opnd
21:   if (unit.writes(arg))
22:     wo = new OperandInfo(rn)
23:     wp = unit.getWritePort(arg)
24:     foreach (rc ∈ wp.registerConnections())
25:       wo.addPath(wp, rc, rc.destPort, rc.destPort.RF())
26:       otCycle.addWriteOperand(wo)
27:     endFor
28:   endIf
29:   if (unit.bypasses(arg))
30:     bo = new OperandInfo(rn)
31:     bp = unit.getBypassPort(arg)
32:     foreach (rc ∈ bp.registerConnections())
33:       wo.addPath(bp, rc.conn, rc.destPort, rc.destPort.unit)
34:       otCycle.addBypassOperand(bo)
35:     endFor
36:   endIf
37:   if (unit.isReadUnit())
38:     do = new OperandInfo(rn, rf)
39:       otCycle.addDestOperand(do)
40:   endIf
41: endFor
42: return otCycle
```

Figure 5: Create otCycle

6. OT GENERATION WITH DATABASE

The algorithm *AutoOT* generates OTs *on-demand*. This may cause an increase in compile-time. To reduce this impact we note that large parts of OTs can be generated statically. In fact, all data independent portions of the OT can be generated statically, if all the static information is provided in the high-level processor description file. At compile-time, just the data-dependent modifications need to be made to create the OT.

We specify all the possible operation formats in the high-level processor description. An operation format *of* of an operation *o* is just like the operation, but it does not have the opcode and the register number fields set. It should be noted that the operation format of different *add* operations which differ in register numbers is the same. Furthermore, even different operations may share the same operation formats. For example *add* and *sub* instructions share the same formats. In fact there are a very small number of operation formats for a given architecture. Thus only a few instruction formats need to be specified. For each operation format, *AutoOTDB1* pre-generates the OT format and stores them in a database. OT format similarly is an OT with the opcode and register number fields not set. At compile-time,

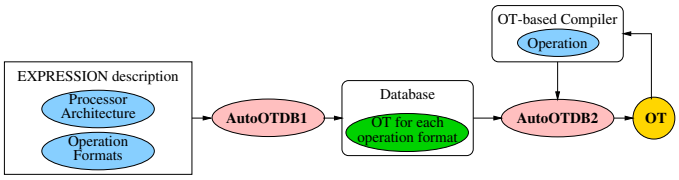

Figure 6: AutoOTDB Flow

when *AutoOTDB2* gets the operation, it finds its format, and looks up for the OT format in the database. The OT format is then decorated with the opcode and register number and returned to the compiler.Thus only minimal amount of dynamic work is done at compile-time to generate OTs to reduce the impact on the compile-time.

7. MICRO-OPERATIONS

In many processors, complex operation break down into simpler micro-operations before execution. The break-down of the complex operation may be statically determinable or data dependent. We specify the break down of a complex operation into micro-operations in the pipeline unit in which the operation breaks down. The OT for the operations is only from the start to the unit in which it breaks down. Then the OTs for the micro-operations are generated. If the break-down of the operation is statically determinable, the OT of the operation can be completely generated by adding the OTs of the micro-operations as child of the main OT. However, in the case when the operation break-down is data-dependent, the complete OT cannot be fully generated. The OTs for the micro-operations are generated and are stored separately in the database. At compile-time, they are stitched together according to the data-dependent breaking mechanism, decorated with the register numbers and returned to the compiler.

8. EXPERIMENTS

We have modeled the popular Intel XScale embedded processor using Operation Tables. We perform several experiments to demonstrate the need and usefulness of automatic OT generation.

8.1 OTs vs. RTs

Reservation Tables (RTs) have long been used in retargetable compilers to detect resources hazards in a given schedule. Although RTs are not defined to handle the complexities of modern processors, like register bypassing, and micro-operations, we extend their definition for comparison purposes. For the Intel XScale processor pipeline there will be 15,592 RTs. In contrast, the number of Operation Tables (OTs) is only 59. Thus even simple architectures with not-so-long pipelines (7-stages) and not-so-many bypasses (up to 21) can have thousands of RTs. Furthermore, RTs can detect only resource hazards while OTs can detect both resource and data hazards. Therefore using OTs is a superior choice than using RTs.

8.2 Reduction in Specification

Although there are only 59 OTs for the Intel XScale pipeline, they comprise about 2000 lines of specification. Their specification has a lot of redundancy, which makes it highly error prone to manually specify all the OTs. On the other hand, AutoOT requires us to specify the high-level processor description, which is only 500 lines; reducing the time and effort required in manual specification. Note that while OTs can be used for scheduling only, the high-level processor description has been shown to be useful in simulation, verification of the processor.

However the true reduction in specification is achieved during design space exploration. Consider a common architectural modification that designers might be interested in; the impact of adding/removing a pipeline unit. If we remove the a pipeline unit in the integer pipeline of the Intel XScale, 21 (36%) OTs (300 lines) need to be modified, and it will take approximately 2 days to do it manually. In contrast, only (18 lines) need to be modified in the processor description, and it takes only 5 minutes. Thus there is a huge time and effort savings in each exploration step by using automatic generation of OTs.

8.3 Compile-time Overhead

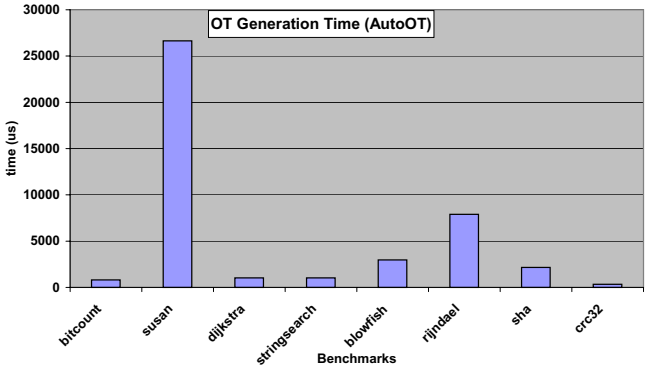

Figure 7: OT Generation Time in AutoOT

AutoOT generates OTs for the compiler to enable bypass-sensitive scheduling. Automated OT generation is the key enabler of fast Compiler-in-the-Loop exploration of processor architecture. However on-demand generation of OTs results in an increase in the compile time. Figure 7 plots the time for OT generation using the AutoOT algorithm. The OT generation time of a benchmark is the total time required to generate the OT for each instruction of the benchmark. The OT generation speed is approximately 750 OTs per second. For most applications, the OT generation takes less than 5 seconds.

AutoOTDB reduces the impact of OT generation on the compile-time by pre-generating parts of OTs and storing them in a database. At compile-time, OT is generated by stitching partial OTs together and decorating it with dynamic parameters. Figure 8 shows that AutoOTDB is able to reduce the compile-time overhead of OT generation by 50% while using only 20 KB of memory. Thus AutoOTDB is able to effectively trade-off significant compile-time overhead for a marginal memory requirement.

8.4 Bypass Exploration

Owing to the increased productivity due to automatic generation of OTs, we were able to perform full exploration of bypasses in the Intel XScale processor for the *bitcount* benchmark. Figure 9 plots the performance of the processor and the energy consumption of the bypass control logic for each bypass configuration. Since seven pipeline units generate

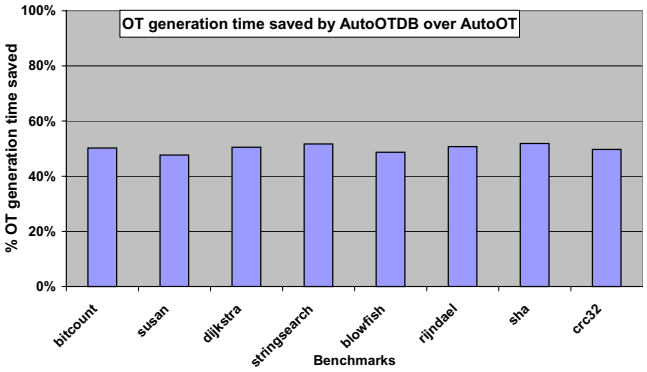

Figure 8: Percentage savings of OT generation time

bypasses, and there are three source operands, $2^{7\times3} = 128$ bypass configurations are possible. For each bypass configuration, the bypass-control logic was automatically synthesized, and the power was estimated using synopsys design compiler and synopsys power estimate [14] respectively. The configuration at (100%, 100%) represents the performance of the processor and power consumption of the bypass control logic with all the bypasses present. The performance and energy consumption of all bypass configurations are reported relative to this configuration. The performance of any other configuration is therefore less than the full configuration. The exploration discovered interesting bypass configurations. For example, configuration 1, trades of 2% performance for 14% less energy consumption. Similarly, configuration 2 has 16% less power consumption at 6% loss of performance. Using *AutoOT* we were able to perform this exploration in 2 days. We estimate that exploration via manual specification and modification of OTs could have taken several months to complete. To conclude, automatic OT generation capability empowers designers to perform full exploration *fast*, and discover interesting alternate bypass configurations.

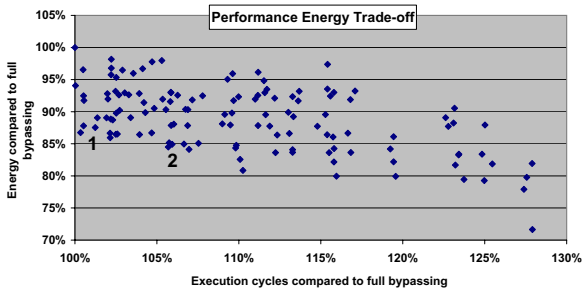

Figure 9: Performance Energy Exploration

9. SUMMARY

Customizing bypasses in processors is an effective way to perform performance-energy-complexity trade-offs. This is especially important for embedded processors, which have strict multi-dimensional constraints. However to perform meaningful exploration a bypass-sensitive compiler is required. Operation Tables are used to perform bypass-sensitive compilation. Owing to the lack of automated methods to generate OTs, currently thet are specified by hand. However, manual specification of OTs is not only a time consuming process, but is also highly error-prone. In this paper we presented *AutoOT*, an algorithm to automatically generate OTs from a high-level processor description. Our experiments on the Intel XScale processor and benchmarks from MiBench demonstrate that AutoOT can greatly reduce the time and effort required for specification. Further, to reduce the compile-time overhead of OT generation, we presented another novel algorithm, *AutoOTDB*. AutoOTDB can reduce the compile-time overhead of OT-generation by 50%. Our experimental results show that automatic OT generation makes it possible to perform full bypass exploration on the Intel XScale in reasonable time and discover interesting alternate bypass configurations.

10. ACKNOWLEDGEMENTS

This work was partially funded by Intel Corporation, UC Micro (03-028), SRC (Contract 2003-HJ-1111), and NSF (Grants CCR-0203813 and CCR-0205712), MIC (Ministry of Information and Communication), Korea, under the ITRC (Inofrmation Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031), KRF contract D00191, Korea Ministry of Information and Communication under Grant A1100-0501-0004.

11. REFERENCES

[1] 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.

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

[3] E. S. Davidson. Effective control for pipelined processors. *IEEE COMPCON*, pages 181–184, 1975.

[4] M. Freericks. *The nML machine description formalism*. T.U. Berlin, Fachbereich Informatik, Berlin, 1991.

[5] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. Rtgen: an algorithm for automatic generation of reservation tables from architectural descriptions. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(4):731–737, 2003.

[6] J. C. Gyllenhaal, W. mei W. Hwu, and B. R. Rau. Optimization of machine descriptions for efficient use. In *MICRO 29*, pages 349–358, Washington, DC, USA, 1996. IEEE Computer Society.

[7] G. Hadjiyiannis, S. Hanono, and S. Devadas. Isdl: an instruction set description language for retargetability. In *DAC '97*, pages 299–302, New York, NY, USA, 1997.

[8] 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.

[9] C. W. Milner and J. W. Davidson. Quick piping: a fast, high-level model for describing processor pipelines. In *LCTES/SCOPES '02*, pages 175–184, New York, NY, USA, 2002. ACM Press.

[10] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. Lisa - machine description language for cycle-accurate models of programmable dsp architectures. In *DAC '99*, pages 933–938, New York, NY, USA, 1999. ACM Press.

[11] A. Shrivastava, N. D. Dutt, A. Nicolau, and E. Earlie. PBExplore: A framework for compiler-in-the-loop exploration of partial bypassing in embedded processors. In *DATE*, pages 1264–1269, 2005.

[12] A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau. Operation tables for scheduling in the presence of incomplete bypassing. In *CODES+ISSS '04*, pages 194–199, New York, NY, USA, 2004. ACM Press.

[13] C. Siska. A processor desription language supporting retargetable multi-pipeline dsp program development tools. In *ISSS '98*, pages 31–36, Washington, DC, USA, 1998. IEEE Computer Society.

[14] Synopsys Inc., http://www.synopsys.com/products/logic/design_compiler.html. *Synopsys Design Compiler*, 2001.

[15] G. Zimmermann. The mimola design system a computer aided digital processor design method. In *DAC '79*, pages 53–58, Piscataway, NJ, USA, 1979. IEEE Press.