

# PBExplore: A Framework for Compiler-in-the-Loop Exploration of Partial Bypassing in Embedded Processors

Aviral Shrivastava<sup>†</sup>

Nikil Dutt<sup>†</sup>

Alex Nicolau<sup>‡</sup>

Eugene Earlie<sup>‡</sup>

aviral@ics.uci.edu

dutt@ics.uci.edu

nicolau@ics.uci.edu

eugene.earlie@intel.com

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

<sup>‡</sup>Mass. Microprocessor Design Center  
Intel Corporation  
75 Reed Road, Hudson, MA, 01749

## Abstract

*Varying partial bypassing in pipelined processors is an effective way to make performance, area and energy trade-offs in embedded processors. However, performance evaluation of partial bypassing in processors has been inaccurate, largely due to the absence of bypass-sensitive retargetable compilation techniques. Furthermore no existing partial bypass exploration framework estimates the power and cost overhead of partial bypassing. In this paper we present PBExplore: A framework for Compiler-in-the-Loop exploration of partial bypassing in processors. PBExplore accurately evaluates the performance of a partially bypassed processor using a generic bypass-sensitive compilation technique. It synthesizes the bypass control logic and estimates the area and energy overhead of each bypass configuration. PBExplore is thus able to effectively perform multi-dimensional exploration of the partial bypass design space. We present experimental results on the Intel XScale architecture on MiBench benchmarks and demonstrate the need, utility and exploration capabilities of PBExplore.*

## 1 Introduction

Register bypasses or forwarding paths improve the performance of a processor by eliminating certain data hazards in pipelined processors [9, 12]. 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 to the register file. However, extensive bypassing implies that very wide multiplexors or buses with several drivers may be needed. Paths including the bypasses often are timing critical and cause pressure on cycle time (especially the single cycle paths). The delay of the bypass logic can be significant for wide issue machines[2]. Thus the performance benefits of bypassing may be accompanied by significant increase

in the cycle time, chip area, energy consumption, wiring congestion, and overall design complexity. Recent research thus advocates the use of partial bypassing in processors[1]. It has been shown that in a design with extensive bypasses, several bypasses have low utilization, and thus can be removed, thereby reducing the power/cost/area of the design without significantly affecting the performance[6]. Moreover, the bypassing in a processor can be changed in most cases without affecting the instruction set of the processor. Exploring bypasses is therefore a valuable technique for designing application specific embedded processors.

Traditionally the decision of which bypasses to add/remove is based on the designer's intuition and/or simulation-only exploration. The traditional method of exploring partial bypasses i.e. *simulation-only* exploration is performed by measuring the performance of the same compiled code (binary) on processor models with different bypass configurations. The configuration with the best performance is chosen. However, once a bypass configuration is chosen, a "production compiler" is developed for the chosen bypass configuration. Although it takes a lot of time and effort to develop the production compiler, finally it is able to exploit the bypasses present in the processor. It has been shown that tuning the compiler for the bypass configuration has significant impact on the performance of a partially bypassed processor [13]. This implies that the performance estimation done by the simulation-only exploration incurs significant errors. Furthermore in a simulation-only exploration, since the code that executes on the processor may not be the correct representative of the code that will be finally executed on the processor, it leads to inaccuracies in other estimates e.g. power. There is thus a crucial need of a bypass-sensitive compiler-in-the-loop exploration of partial bypassing in embedded processors.

Embedded systems, which are characterized by multi-dimensional design constraints including power, perfor-

mance and cost, critically require an exploration framework which is able to accurately evaluate the performance, area and energy of each design alternative and thus perform meaningful multi-dimensional trade-offs.

In this paper we present PBExplore: A Compiler-in-the-Loop Framework to explore Partial Bypassing in processors. PBExplore evaluates the performance of a bypass configuration by generating code for the processor with given bypass configuration and simulates the generated code on cycle accurate simulator of the processor with the same bypass configuration. PBExplore also synthesizes the bypass control logic and evaluates the area and energy overhead of the bypass configuration. Thus PBExplore is able to effectively perform meaningful multi-dimensional (performance-area-power) trade-offs among bypass configurations. This makes PBExplore a valuable assist for designers of programmable embedded systems.

## 2 Related Work

Bypassing has been popular since it was first described in the IBM Stretch[3]. The impact of bypassing on the area, cost, power, and overall chip complexity has been examined in detail, consequently suggesting the use of partial bypassing[5, 1, 7]. Code generation techniques for partially bypassed processors and framework to customize the bypasses therein, have been proposed[4, 6]. However, the compilation techniques and the frameworks are for VLIW processors only, furthermore, they employ crude cost estimates, and no energy estimates at all. Thus existing frameworks are unable to perform a multi-dimensional exploration of processors with partial bypassing, which is very important for embedded processors.

PBExplore uses a generic retargetable bypass-sensitive compilation technique proposed in [13] to accurately evaluate the performance of each bypass configuration. PBExplore automatically synthesizes the bypass control logic to estimate the area and energy overhead of the bypass configurations. PBExplore is thus able to effectively and meaningfully perform multi-dimensional exploration of processors with partial bypassing.

The rest of the paper is organized as follows: In Section 3 we describe our partial bypass exploration framework. Section 3.1 describes bypass-sensitive compilation using Operation Tables. Section 3.2 discusses the issues in the design of a bypass-sensitive cycle accurate simulator. Section 3.3 describes the design of a generic bypass control logic and briefs on our area and energy overhead estimations. Section 4 we show that simulation-only exploration and our compiler-in-the-Loop exploration, differ significantly and that they may lead to different design decisions. We then perform experiments to show that PBExplore can effectively perform a multi-dimensional exploration of partial bypassing. We conclude in Section 5.

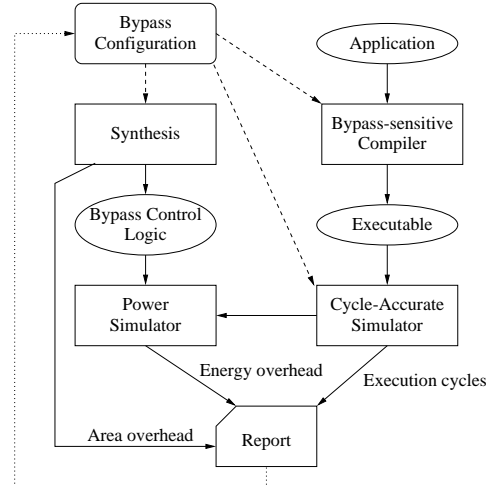


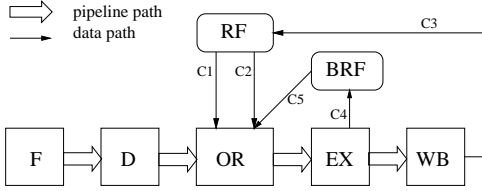
Figure 1. PBExplore: A Compiler-in-the-Loop Framework for Partial Bypass Exploration

## 3 PBExplore

PBExplore is driven by bypass configuration as shown in Figure 1. All the bypasses present in the processor are described in the bypass configuration. A bypass is defined in terms of the pipeline stage where it is generated, and the operand that can use it. The application is compiled using a bypass-sensitive compiler that is parameterized on the bypass configuration. The generated executable is then simulated on a cycle accurate simulator that is parametrized on the same bypass configuration. The cycle accurate simulator reports the runtime (in cycles) for the application. The bypass configuration is used to synthesize the bypass control logic and estimate the area overhead of bypasses. A Power simulator uses the synthesized bypass control logic, and the input stimuli in each cycle (generated by the cycle accurate simulator) to estimate the energy consumed by the bypass control logic for the execution of the application. Thus PBExplore is able to make an accurate estimation of performance (cycles of execution), area and energy consumption overhead for each bypass configuration. We now describe the different components of PBExplore.

### 3.1 Bypass Sensitive Compiler

The goal of a bypass-sensitive compiler is to generate code such that operations use the bypasses present to exchange values and do not suffer from hazards due to missing bypasses. Consider the simple 5-stage processor pipeline as shown in Figure 2. The *Operand Read* (OR) pipeline stage is shown in detail to describe incomplete bypassing. There is a bypass from the *Execute* (EX) pipeline stage to the second operand in the OR. For modeling purposes we cluster all the bypasses to an operand into a virtual *Bypass Register File* (BRF). All the bypasses to the operand write into its



**Figure 2. Example Processor Pipeline**

bypass register file, and it is read while reading the operand. The bypass from EX to the first operand is modeled using connections C4, C5 and BRF. In the processor pipeline in Figure 2, the first operand can be read from the *Register File* (RF) only, while the second operand can also be the result of the operation in EX. In this pipeline, if an operation in OR needs the result of the operation in EX as the first operand, there will be a pipeline hazard, while if it needs to read it as the second operand, there will be no hazard. Thus a bypass-sensitive retargetable compiler needs to be cognizant of the processor pipeline and the bypasses present/absent.

Operation Tables (OTs) succinctly capture the processor pipeline and the bypasses present and enable the compiler to generate code sympathetic to the bypass configuration. Like RTs (reservation Tables), OTs describe the resources that an operation may use in each cycle of its execution. In addition OTs describe when an operation reads/writes/bypasses its operands, and operand itself (register identifier). OTs also describe the resources that are available to read/write/bypass the operands.

The OT for a simple instruction, *ADD R1 R2 R3* of a the processor pipeline in Figure 2 is shown in Table 1. The ADD operation utilizes resource *Fetch* (F) pipeline stage in the first cycle and *Decode* (D) pipeline stage in the second cycle. In the third cycle, it reads two operands R2 and R3. The first operand R2, may be read only from the *Register File* RF via connection C1, while the second operand, R2 can be read from the RF via connection C2, or from the BRF via connection C5. In the fourth cycle, add operation is executed and the result R1 is bypassed to BRF via connection C4. In the fifth and final cycle R1 is written back to RF via connection C3. OTs of operations in a given schedule can be combined to discover all the pipeline hazards. Table 2 shows how OTs can be used to detect data hazard between two simple dependent instructions:

ADD R1 R2 R3  
SUB R5 R1 R4

on the pipeline shown in Figure 2. The OT-based compiler has to maintain the state of the machine in terms of busy resources and available registers in each register file for each cycle. This bookkeeping allows the detection of data hazards (e.g. when a required register is not present in a reachable register file), and resource hazards (e.g. a required resource is busy). Table 2 shows that after scheduling the first operation (ADD), register R1 is not available in RF in cy-

Operation Table of ADD R1 R2 R3	
1	F
2	D
3	OR
	ReadOperands
	R2
	C1, RF
	R3
	C2, RF
	C5, BRF
	DestOperands
	R1, RF
4	EX
	WriteOperands
	R1
	C4, BRF
5	WB
	WriteOperands
	R1
	C3, RF

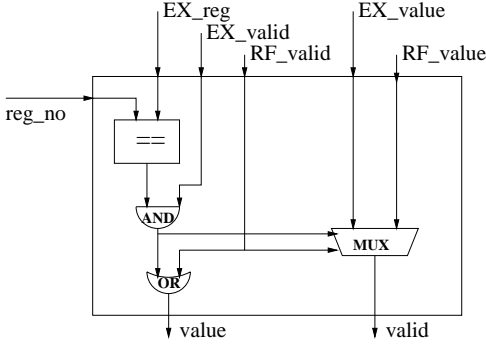
**Table 1. Operation Table of ADD R1 R2 R3**

cle 4. Register scoreboarding makes destination of a issued instruction unavialbe in the register file until it is written back to to avoid a WAW hazards. The next operation (SUB) can read R1, its first operand only from RF (due to absence of a bypass from EX pipeline stage to the first operand of RF). Thus there is a data hazard in cycle 4. The data hazard is cleared in the next cycle (cycle 5), when R1 becomes available via RF.

Thus OTs can be used to detect all the pipeline hazards in a given schedule, even in the presence of partial bypassing. This ability of accurate hazard detection can be used to re-order operations and generate bypass sensitive code. Since PBExplore is an exploration framework, and we are interested in estimating the performance potential of a bypass configuration, we look at all possible orderings of operations in a basic block, and pick up the one with best performance. Since this scheme is exponential, we impose a limit on the number of operation orderings (10, 000) that we try. This bound leaves the compile-time (few mins) negligible as compared to cycle accurate simualtion time (few hours).

Cycle	Busy Resources		!RF	BRF
	ADD R1 R2 R3	SUB R5 R1 R4		
1	F		-	-
2	D	F	-	-
3	OR C1 C2	D	-	-
4	EX C4	<b>Data Hazard</b>	R1	R1
5	WB C3	OR C1 C2	-	-
6		EX C4	R4	R4
7		WB C3	-	-

**Table 2. Hazard detection using OTs**



**Figure 3. Bypass control logic for the second operand**

### 3.2 Cycle Accurate Simulator

The code generated for a particular bypass configuration has to be executed on a cycle accurate processor simulator with the same bypass configuration to estimate the execution cycles required for the application. Structural cycle accurate simulators that are written in sequential languages like C, model the processor pipeline by modeling pipeline stages and the pipeline registers explicitly. Modeling pipeline registers eliminates the dependencies between the pipeline stages, making it possible to execute the pipeline stages in any order. Each pipeline stage reads the pipeline register just before it, executes, and then writes the result into the pipeline register after it. Updating the contents of the pipeline register results in an increment in the simulation cycles of execution.

However, bypasses impose a dependency between the pipeline stage generating the bypass and the one using it. The bypass value generating stage should be executed before the one that uses it. To solve this problem we represent the processor pipeline using a directed graph, which has an edge from the pipeline stage that generates the bypass to the one that uses it. Any breadth first ordering of this graph (topological sort) produces a legal execution order of the pipeline stages. The cycle accurate simulator also generates the inputs and outputs of the bypass control logic every cycle. This is used by the power simulator to estimate the energy overhead for the bypass configuration.

### 3.3 Area and Energy Overhead Estimation

We quantify the area and energy consumption overhead of bypassing by synthesizing the bypass control logic for each bypass configuration. Figure 3 shows the bypass logic for the second operand in the OR pipeline stage in the pipeline in Figure 3, which receives only one bypass (from the EX pipeline stage). Each operand can potentially receive bypass from each pipeline stage. Of course for real processors that have large number of such bypasses for each operand, the bypass control logic scales and result in significant area and energy consumption overhead. Each

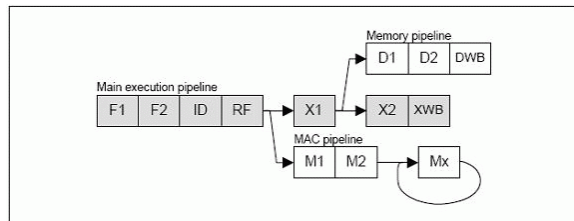
pipeline stage that is a source of a bypass, generates a bypass value, a bypass valid and a bypass register number. If the operand to be read matches any of the incoming bypass register numbers, then the corresponding bypass value is chosen, otherwise the value from the register file is chosen. We synthesize the bypass logic using the Synopsys Design Compiler[11] and estimate the area overhead of the bypass control logic. Synopsys Power Estimator[11] is then used to simulate this bypass control logic with the input stimuli generated by the cycle accurate simulator to estimate the energy consumption of the bypass control logic.

## 4 Experiments

To demonstrate the need, usefulness and capabilities of PBExplore, we perform several experiments on the Intel XScale[10] architecture. XScale is a popular embedded processor for wireless and handheld devices. We perform experiments on benchmarks from MiBench[8] suite, which represent applications in the same domain. Due to space constraints we present only a few results, however the results and conclusions are similar for other benchmarks. Figure 5 shows the 7-stage out-of-order super-pipeline of XScale. XScale has three execution pipelines, the X pipeline (units X1, X2, and XWB), the D pipeline (units D1, D2 and DWB), and M pipeline (units M1, M2 and Mx(referred to as MWB in this paper)). For our experiments we assume that 7 pipeline stages, X1, X2, XWB, M2, MWB, D2 and DWB can bypass to all the 3 operands in RF. Thus there are  $7 \times 3 = 21$  different bypasses in XScale. No computation finishes before or in the pipeline units M1 and D1, thus there are no bypass connection from these units.

We synthesized the bypass control logic for each bypass configuration using *design compiler* of Synopsys-2001.10 and  $0.8\mu$  library *lsi\_10k*. To estimate the area overhead we synthesized the bypass control logic for minimum delay. We used *Synopsys power\_estimate* and the input stimuli from cycle accurate simulator to estimate the energy consumed by each bypass configuration for the given application.

In the first part of experimental section, we show that performance evaluation by the simulation-only exploration and our bypass-sensitive compiler-in-the-loop exploration differ significantly, and may lead to different design decisions. In the second part we perform a multi-dimensional



**Figure 5. 7-stage super-pipeline of XScale**

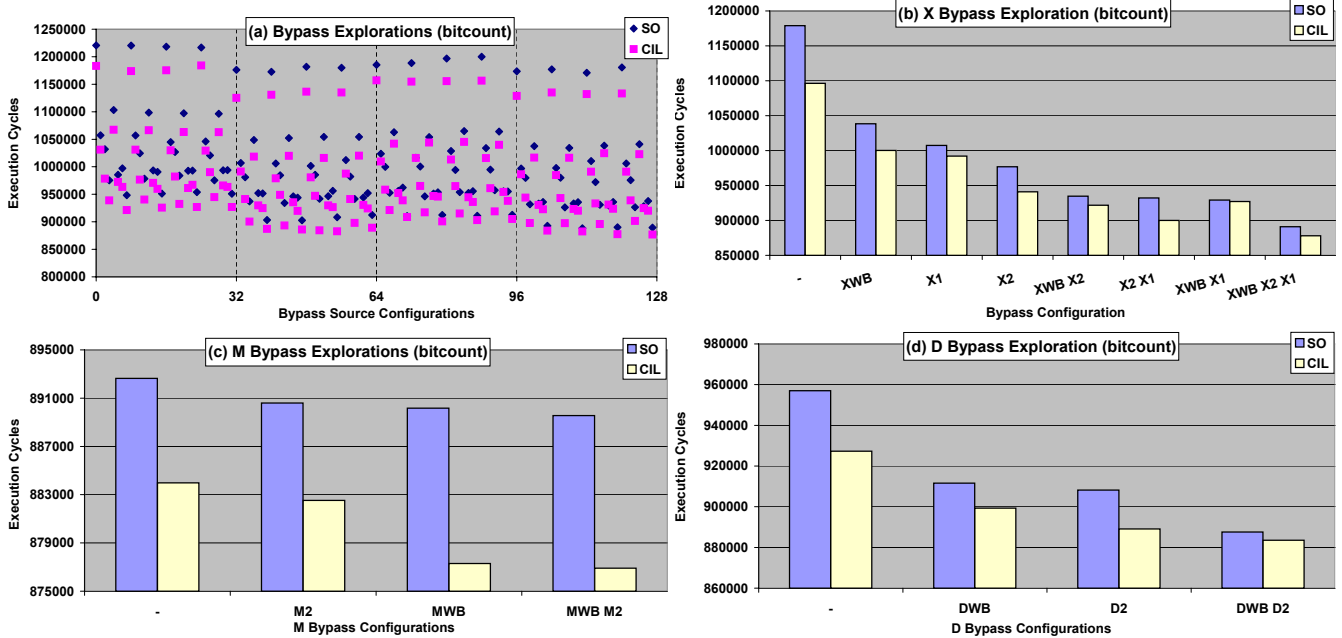


Figure 4. Simulation-only vs. Compiler-in-the-Loop Exploration

(performance-area-energy) exploration to demonstrate the usefulness and capabilities of PBExplore.

#### 4.1 Simulation-only versus Compiler-in-the-Loop Exploration

In the XScale pipeline model, we vary whether a pipeline stage bypasses its result or not. If a pipeline stage bypasses, all the operands can read the result. Thus there are  $2^7 = 128$  possible bypass configurations. Figure 4(a) plots the runtime (in execution cycles) of the *bitcount* benchmark for all these configurations using simulation-only (light squares), and compiler-in-the-loop (dark diamonds) exploration. We make two important observations from this graph. The first is that *all* the light squares are below their corresponding dark diamonds, indicating that the execution cycles evaluated by compiler-in-the-loop exploration is less than the execution cycles evaluated by the simulation-only exploration. This implies that the bypass-sensitive compiler is able to effectively take advantage of the bypass configuration, and is generating good quality code for *each* bypass configuration. The second observation is that the difference in the execution cycles for a bypass configuration can be up to 10%, implying that the performance evaluation by simulation-only exploration can be up to 10% inaccurate.

A case can be made for simulation-only exploration by arguing that the error in exploration is important only if it leads to a difference in trend. To counter this claim we will now zoom into this graph and show that simulation-only exploration and compiler-in-the-Loop exploration result in different trends, and may lead to different design decisions. Figure 4(b) is a zoom-in of Figure 4(a) and shows the ex-

plorations when only the X-bypasses are varied, while the rest are present. To bring out the difference in trends, the bypass configurations in this graph are sorted in the order of execution cycles as evaluated by simulation-only exploration. Figure 4(b) shows that as per the simulation-only approach, all configurations with bypasses from two stages in the X-pipeline are similar, i.e. the execution cycles for configurations  $\langle X2 X1 \rangle$ ,  $\langle XWB X1 \rangle$  and  $\langle XWB X2 \rangle$  are similar. However, our bypass-sensitive compiler is able to exploit the configuration  $\langle X2 X1 \rangle$  better than other configurations with two X-bypasses. Figure 4(c) and Figure 4(d) focuses on varying D and M bypasses while keeping the rest in-place. Figure 4(c) shows that the simulation-only exploration evaluates the performance of the bypass configurations with one bypass as equivalent. However, our PBExplore determines that if you can have only one bypass, the bypass from the D2 pipeline stage is a superior choice. We make similar observations for the M-bypass exploration in Figure 4(d).

#### 4.2 Performance-Area-Energy Exploration

To demonstrate that PBExplore can effectively perform a multi-dimensional exploration, we vary the bypasses only for the first operand, and assume that all the bypasses reach the other two operands. Thus there are  $2^7 = 128$  bypass configurations. Figure 6 (a) and (b) show the performance-area and performance-energy trade-offs of various bypass configurations computed using PBExplore. The performance area and energy consumption are shown relative to that of a fully bypassed processor. The interesting pareto-optimal design points 1 and 2 are marked in both the graphs.

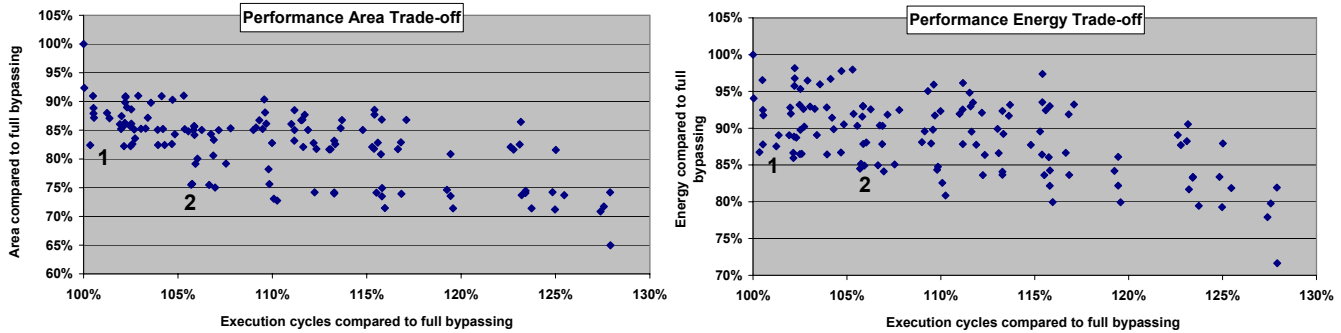


Figure 6. Power-Area-Energy trade-offs using PBExplore

Design point 1 represents the bypass configuration when MWB and XWB do not bypass to the first operand. This bypass configuration, uses 18% less area than full bypassing and consumes 14% less energy than full bypassing, while suffering only 2% performance penalty. Similarly design point 2 represents the bypass configuration when only D2 and X2 bypass to the first operand. This configuration uses 25% less area and consumes 16% less power than fully bypassed processor, while losing only 6% on performance. These configurations represent cheaper (in area and energy consumption) design alternatives, at the cost of minimal performance degradation. These are exactly the kind of trade-offs that an embedded processor designers would need to evaluate when customizing bypasses.

## 5 Summary

Effective performance, area and energy trade-offs can be achieved by varying partial bypassing in a processor. Given that a bypass-sensitive compiler significantly affects the performance of a partially bypassed processor, simulation-only exploration of partial bypass configurations is inaccurate. In this paper we present PBExplore which not only accurately evaluates the performance of a partially bypassed processor, it also accurately estimates the area and energy overhead of a partial bypass configurations. We have shown that traditional exploration leads to sub-optimal design decisions and that PBExplore is able to effectively perform multi-dimensional performance-area-energy trade-offs in embedded processor design. However the partial bypass design space is huge, and cannot be explored by evaluating each point. Our future work includes developing partial bypass design space walker.

## 6 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). We would also like to thank members of ACES lab, especially Partha Biswas for helping crystallize our thoughts in the brainstorming sessions.

## References

- [1] A. Abnous and N. Bagerzadeh. Pipelining and bypassing in a vliw processor. In *IEEE trans. on PDS*, 1995.
- [2] P. Ahuja, D. W. Clark, and A. Rogers. The performance impact of incomplete bypassing in processor pipelines. In *Proc. of MICRO-28*, 1995.
- [3] E. Bloch. The engineering design of the stretch computer. In *Proc. of EJCC*, pages 48–59, 1959.
- [4] 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.
- [5] R. Cohn, T. Gross, M. Lam, and P. Tseng. Architecture and compiler tradeoffs for a long instruction word microprocessor. In *Proc. of ASPLOS*, 1989.
- [6] 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 ASSAP*, 2003.
- [7] J. Gray, A. Naylor, A. Abnous, and N. Bagherzadheh. Viper: A vliw integer microprocessor. In *IEEE Journal of Solid State Circuits*, pages 1377–1383, 1993.
- [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] P. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.
- [10] <http://www.intel.com/design/intelxscale/273436.htm>. *Intel XScale Microarchitecture Programmers Reference*, 2001.
- [11] [http://www.synopsys.com/products/logic/design\\_compiler.html](http://www.synopsys.com/products/logic/design_compiler.html). *Synopsys Design Compiler*, 2001.
- [12] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1998.
- [13] A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau. Operation tables for scheduling in the presence of incomplete bypassing. In *Proc. of CODES+ISSS 2004*, 2004.