Compiler-in-the-Loop ADL-driven Early Architectural Exploration*

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

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

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

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

ABSTRACT

Processor architects today critically need software tools that accurately track architectural changes made during exploration, and provide fast and quantitative feedback for each design point. Indeed, Design Space Exploration (DSE) without the compiler-in-the-loop (CIL) can be meaningless. To effectively explore the processor-memory-coprocessor design space, a system architect critically needs a compiler that can exploit the advantages of micro architectural features, hide memory latency and effectively use the coprocessor. This paper presents a CIL framework for processor architecture DSE. The framework is developed around EXPRESSION, an Architecture Description Language (ADL) that captures the functionality and structure of the processor at a high level. A software toolkit comprising an optimizing compiler, an instruction-set simulator and a cycle-accurate simulator are parameterized from the ADL, allowing for early estimation of performance, power and code size. System designers can modify the ADL to reflect architectural changes; for each change, the applications are re-evaluated using the architecture-sensitive compiler and the cycle-accurate simulator and feedback on performance as well as power is provided. Furthermore, the ADL can be used as a golden reference model for the ensuing phases of design and analysis. This paper demonstrates the need and usefulness of CIL DSE methodology for the exploration of register bypasses in the Intel XScale architecture.

1. INTRODUCTION

Modern Embedded Systems have strict multi-dimensional constraints, including power, performance and cost. As a result of ever-changing demands, microprocessors with increasing programmable content are gaining market share. Designing such processors require customizing the processor to meet all the constraints in chorus, which in turn necessiates exploring several architectural modifications and studying their impact on the design parameters including design-time. Design Space Exploration (DSE) is therefore a very important phase in embedded processor design methodologies. Furtherm, increasing design complexity and decreasing time-to-market are pushing this exploration phase early in the design process. Indeed, there is an undeniable need for automated and early DSE methodologies.

Previously software for such programmable embedded systems was hand-coded in assembly languages. However increasing software content in newer designs and shrinking time-to-market has resulted in migration to high-level language based software development environments. Thus a compiler - that can exercise novel microarchitectural features - becomes an essential part of the design flow. Indeed, DSE without the compiler-in-the-loop (CIL) can be meaningless. For example, a modification to the pipeline or adding a coprocessor is of no use unless the compiler exercises code to exploit such features. Furthermore, a smart compiler can obviate the need of some costly architectural features. e.g, a compiler can aggressively hide memory latencies based on the organization and types of memories employed by the programmable architecture, and thus render unnecessary, costly prefetching techniques. To effectively explore the processor-memory-coprocessor design space, a system architect therefore critically needs a compiler that can exploit the advantages of micro architectural features, hide memory latency and effectively use the coprocessor.

A key requirement for CIL DSE is that the compiler should be able adapt itself to the architectural variations. Traditionally such compilers are labeled as "retargetable compilers", which can be adapted to generate code for different target processors with significant reuse of the compiler source code. Many years of research on the topic of retargetable compilers has led to new approaches that employ Architecture Description Languages (ADL) to define the machine model, including the processor, memory and coprocessing engines. The ADL can then be used to generate/parameterize entire software toolkits (typically comprising a compiler, simulator, debugger and associated software development utilities). Furthermore, such ADLs can be used to perform effective DSE of novel architectural and microarchitectural features. Although a number of ADL based DSE methodologies have been proposed, most of them do not have a CIL, which we believe is critical for developing modern programmable embedded systems. Furthermore, there is a lack of compiler technology that can generate good quality code which exploits novel microarchitectural features present in many emerging embedded processors.

In this paper we demonstrate the need and usefulness of

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

ADL-driven early CIL DSE on modern embedded processors. We explore the performance and power impact of varying the register bypasses in the Intel XScale processor. However in the case, when only some of the bypasses are present (partial bypassing), traditional compilers cannot generate optimal code for them. Therefore first we developed a novel scheduling algorithm for a partially bypassed processor. We then show that traditional exploration, that does not use CIL results in significantly inaccurate evaluation of bypass configurations, which can eventually lead to sub-optimal design decisions.

2. EXPRESSION BASED CIL DSE

EXPRESSION is an Architecture Description Language (ADL) that captures the functionality and structure of the architecture at a higher level, focused primarily to facilitate the development of a architecture-sensitive retargetable compiler, that will be used for CIL DSE.

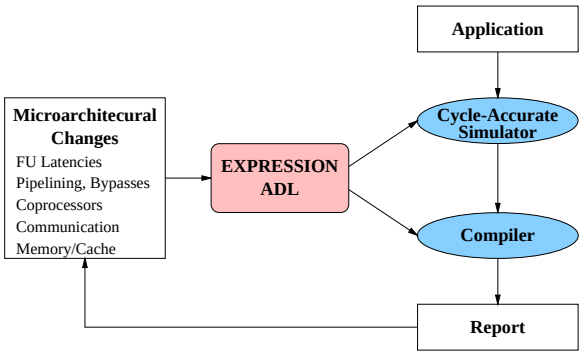

Figure 1: EXPRESSION based CIL DSE

This project uses the EXPRESSION ADL[5] to describe the processor architecture and the memory subsystem of a programmable processor system. The EXPRESS, architecture-sensitive compiler [6, 9] and the SIMPRESS instruction set simulator and cycle-accurate simulator [8] are parameterized from the system description in EXPRESSION. The application is compiled using the optimizing compiler EXPRESS, executed on the cycle-accurate, word-true simulator SIMPRESS to get the power, performance, utilization, static and dynamic code size, and other statistics in a report. Figure 1 shows our EXPRESSION-driven CIL DSE framework.

3. PARTIAL BYPASSING

Register bypasses or forwarding paths improve the performance of a processor by eliminating certain data hazards, but are accompanied with a significant increase in the cycle time, chip area, energy consumption, wiring congestion, and overall design complexity [2]. Recent research therefore 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[3]. Exploring the bypasses is therefore a valuable technique for designing application specific embedded processors. Moreover, the bypassing in a processor can be changed in most cases without affecting the instruction set of the processor.

Traditionally the decision of which bypasses to add/remove is based on the designer's intuition and/or Simulation-Only (SO) exploration. The traditional method of exploring partial bypasses i.e. SO 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 [9]. This implies that the performance estimation done by the SO exploration incurrs significant errors. Furthermore in a SO 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 inaccurarcies in other estimates e.g. power. Thus there is a crucial need of a bypass-sensitive Compiler-In-the-Loop exploration of partial bypassing in embedded processors.

4. 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 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.

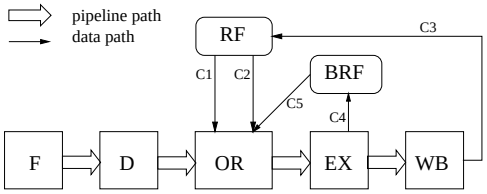

Figure 2: Example Processor Pipeline

Operation Tables (OTs) succintly 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 cycle 4. Register scoreboarding makes destination of a issued instruction unavialble 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.

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

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 reorder operations and generate bypass sensitive code. In an exploration framework we are interested in estimating the performance potential of a bypass configuration. Therefore, 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 simulation 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	**Data Hazard**	R1	R1
5	WB C3	OR C1 C2	-	-
6		EX C4	R5	R5
7		WB C3	-	-

Table 2: Hazard detection using OTs

5. EXPERIMENTS

To demonstrate the need, usefulness and capabilities of PBExplore, we perform several experiments on the Intel XScale[7] architecture on benchmarks from MiBench[4] suite.

5.1 Simulation-Only versus CIL 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. There are $2^7 = 128$ possible bypass configurations. Figure 3(a) plots the runtime (in execution cycles) of the *bitcount* benchmark for all these configurations using simulation-only (SO) (dark diamonds), and CIL (light squares) exploration. We make two important observations from this graph. The first is that *all* the execution cycles evaluated by the CIL exploration is less than the execution cycles evaluated by the SO exploration. This implies that the bypass-sensitive compiler is able to take advantage of each bypass configuration, and is generating good quality code. 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 SO exploration can be up to 10% inaccurate.

We now zoom into this graph and show that SO exploration and CIL exploration result in different trends, and may lead to different design decisions. Figure 3(b) is a zoom-in of Figure 3(a) and shows the explorations when only the bypasses in the integer 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 SO exploration. Figure 3(b) shows that as per the SO approach, all configurations with bypasses from two stages in the X-pipeline are similar, i.e. the execution cycles for configurations <X2 X1>, <XWB X1> and <XWB X2> are similar. However, our bypass-sensitive compiler is able to exploit the configuration <X2 X1> better than other configurations with two X-bypasses.

5.2 Performance-Area-Energy Exploration

To demonstrate that PBExplore can effectively perform a multi-dimensional exploration, we vary the bypasses only for the first operand. We synthesize and measure the area and energy consumption of the bypass control logic. There would be $2^7 = 128$ bypass configurations. Figure 4 (a) and (b) shows the performance-area and performance-energy tradeoffs of various bypass configurations computed using PBEx-

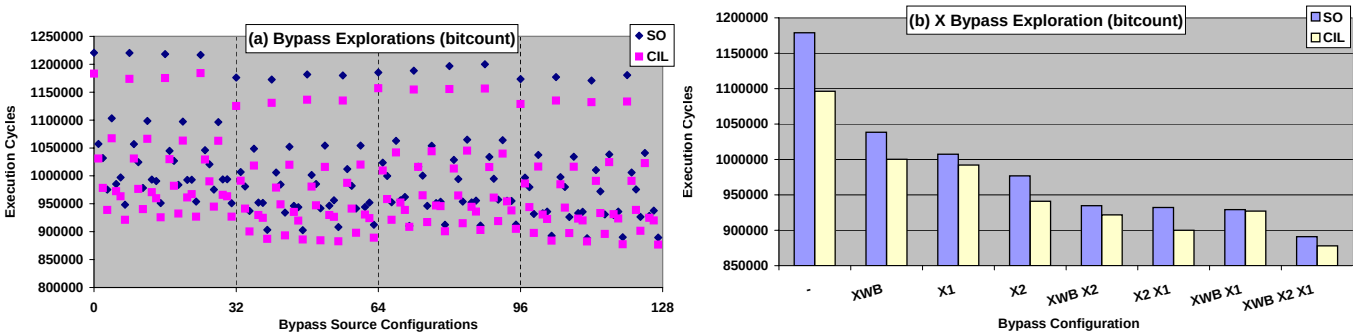

Figure 3: Simulation-only vs. Compiler-in-the-Loop Exploration

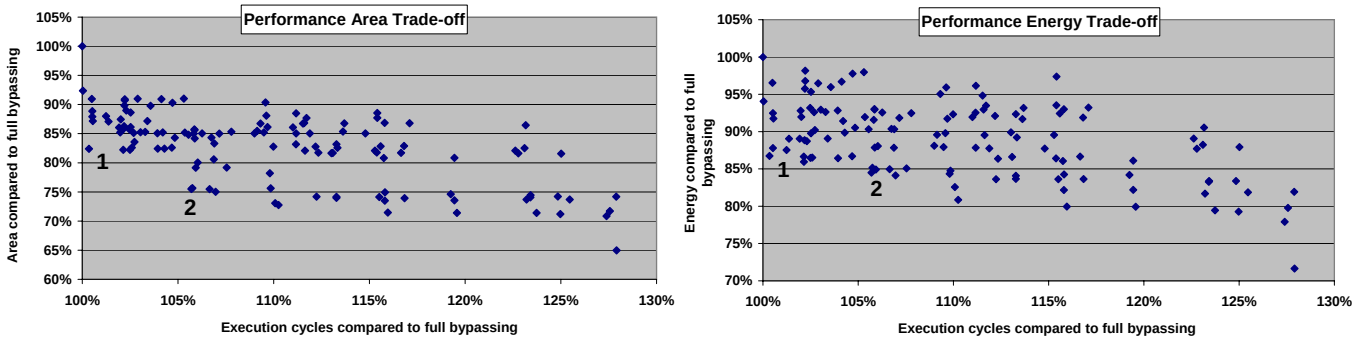

Figure 4: Power-Area-Energy trade-offs using PBExplore

plore. 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. 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.

6. SUMMARY

Programmable processors are becoming very popular to meet the ever-changing demands of embedded systems. The multi-dimensional design constraints require automated design space exploration (DSE) very early in the design phase. Effective exploration on programmable embedded processors can be performed only by using an architecture-sensitive compiler-in-the-loop (CIL). In this paper we present an EXPRESSION driven CIL DSE framework, to meaningfully perform exploration on an embedded processor by varying register bypasses. However existing code generation techniques break down in the presence of partial bypassing. Therefore, we first develop a novel compilation technique to generate code for partially bypassed processor. Our experiments on the Intel XScale processor pipeline and benchmarks from MiBench suite show that without a CIL, exploration can be erroneous and lead to sub-optimal design decisions.

7. REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] Intel Corporation, http://www.intel.com/design/intelxscale/273473.htm. *Intel XScale(R) Core Developer's Manual.*

[8] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-sat: A visual specification and analysis tool for system-on-chip exploration. In *EUROMICRO Conference*, pages 196–203, 1999.

[9] A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau. Operation tables for scheduling in the presence of incomplete bypassing. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 194–199, New York, NY, USA, 2004. ACM Press.