Energy Reduction through Compiler Directed Resizing of Configurable Structures

Tejas Karkhanis

Abstract

Compiler Directed Resizing is a method for resizing configurable structures in processors with the help of compiler. Program behavior profiles are collected at the function call level and are then used to determine the correct size of the structure. A hardware structure to detect the hot functions collects the pertinent information about them. The runtime sys tem and the compiler use this information to insert the resizing instructions at appropriate locations. With this scheme we are able to save 9% of the activity in the instruction fetch part of the processor.

1 Introduction

Energy (or power) efficiency is becoming an increasingly important aspect of high performance processor design. Techniques can be applied at the device, circuit, logic design, microarchitecture, and compiler levels. Eventually, innovations in all these areas will be required. There are proposals for dynamically resizing the processor structures mainly [7,8,5,6] through hardware or through a small layer of software. But the proposed technique are "trail and error" based. That is, they do not look at the program behavior before resize the structures. Our primary interest here is to demonstrate that compiler-directed resizing is feasible and should be explored further. Our approach is: profile the program while it is running and later, feed it back to the runtime system and the dynamic compiler to recompile the program to include resource resizing instructions.

We consider resizing the structures at the function call level. First "hot" functions are located with a function behavior buffer (see section) Instructions to resize the structures, we con-

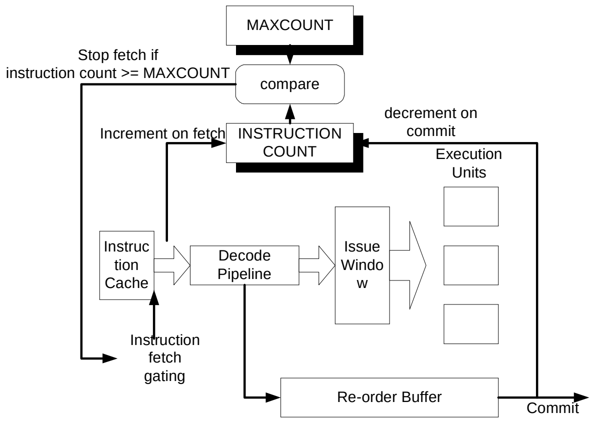

Figure 1: Pipeline control logic to dynamically limit the number of instructions in the processor.

sider, are added to the ISA. These instructions are inserted in the prolog and epilog of the functions. Since in the prolog and epilog of every function there are spills and fills, we hope that by inserting the resizing instructions here, their overhead is not very pronounced. We introduce special instructions to resize the structure(s). At the compiler level, it is unlikely that there will be a single technique for reducing energy consumption; rather there will likely be a wide variety of methods, some of which are closely coupled with the underlying microarchitecture -- as has been the case with performance-enhancing features.

Our primary interest here is to reduce power in the instruction delivery portion of the processor. In order to turn-off parts of units (such as the Issue Window slots) for a long time – on orders of 1 million committed instructions – a method to predict when to turn it off is needed. Furthermore, to prevent performance loss it is necessary to predict accurately their turn-on times. Several research papers have proposed the implementations of multi-configuration units [Alper's circuit design of the issue window, JIT] to dynamically turn-off the unnecessary portions of those units based on the application requirements.

The rest of the paper is organized as follows: Section 2 explores the architectural mechanisms that enable our research; Section 3 has the compiler support for our research;

Section 4 details our experimental environment and the results; and Section 5 summarizes our work.

2 Architecture Support

2.1 The reconfigurable processor

2.2 Profiler Design

Figure 2 is the design of the hot function detector/profiler. The purpose of the hot function detector is first to detect a hot function and second to profile the behavior of the hot function. The profiler consists of two main components: the *function behavior buffer (FBB)* and the *function ID stack (FIS)*. The function behavior buffer consists of several *info blocks*. Every *info block* has the following fields:

• *Tag*:	the tag part of the function identifier.
• *Cycles*:	a count of the number of cycles spent in the function.
• *Correct Branches*:	a count of the number of correct predicted branches in the function.
• *Mispredicted Branches*:	a count of the number of mispredicted branches in the function.
• *Data Cache Misses*:	number of data cache misses while in this function.
• *Data Cache Hits*:	number of data cache hits while in this function.
• *Instruction Cache Misses*:	number of instruction cache misses while in this function.
• *Instruction Cache Hits*:	number of instruction cache hits while in this function.
• *Num Calls*:	number of calls made to this function.

The runtime system and the dynamic compiler use the information in the info block to detemine the value of MAXcount. At every function call, the *current function ID* is pushed on the *function ID stack.* Next, the PC of the next dynamic instruction – the first instruction of the function -- is assigned to *current function ID.* Every cycle the *cycles* is incremented in the *info block* pointed to by the *current function ID.* When, a branch is committed and the prediction is correct the *correct*

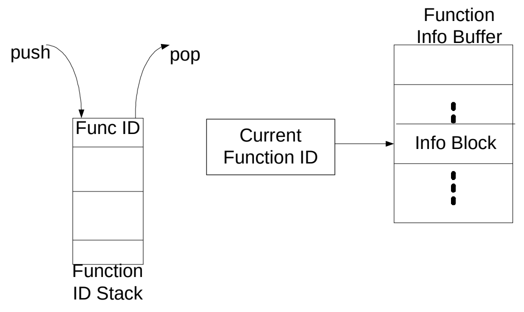

Figure 2: Hot Function Detector

branch counter is incremented. If that branch was mispredicted the *mispredicted branches* counter is incremented. Hence with these two counters the misprediction rate can be derived. Similarly, the Instruction/Data Cache Hits/Misses are incremented when there are Instruction/Data Cache hits/misses. Also there is a refresh timer, which is decremented every cycle. When the timer reaches its max the entries in the FBB are cleared. A hot function is detected when the ratio of the *cycles* field of a function to the max value of the refresh timer is greater than the *hot candidate ratio.* This is done in the hardware by monitoring the proper bit of the *cycles* field counter.

2.3 New Instructions

We add an instruction to change the value of MAXcount. To add the instruction, a new instruction was created in the PISA ISA [4]. The processor decoder was then modified to detect when this instruction enters the pipeline. Furthermore, the Strata compiler was modified to insert this instruction based on the profiler data. Where to insert this instruction in the function is explained in section 3.2.

3 Compiler Support

Here we detail the compiler support for our research. First the compiler determines the value of MAXcount (section 3.1), then it inserts the resizing instructions (section 3.2).

3.1 Determining the size of the resource (i.e MAXcount)

To find out the optimal value of the number of instructions in the processor for a function, the instruction cache, data cache miss rates and, the branch misprediction rate are examined. This is a fairly simple scheme, hence the overhead for doing this analysis in the compiler is low. The miss rates and misprediction rate are calculated from the information contained in the FBB.

3.2 Inserting the `maxcnt` instruction

Recall, the maxcnt instruction must be inserted to incur as low overhead as possible. When a function is detect as "hot", it can fall into two categories, as discussed before. If it falls in category 1 – where the program spends lot of time in the hot function just after its first call – the instruction is inserted in the prologue and epilogue of the callee. If the hot function accumulates the time in it over many calls the runtime system traverses through the calling history and inserts the instruction in the caller. We do this because we do not want the overhead of the `maxcnt` instruction to show up everytime the hot function is called.

4 Experimental Evaluation

4.1 Methodology

The Strata compiler and the underlying architecture were enhanced to support a model of the proposed architecture. The SPEC JVM 98 benchmarks were used.

4.2 Results and Analysis

Figure 3 has the Normalized Activity averaged over all benchmarks for the instruction cache accesses. The compiler directed scheme does worse among all the scheme, but it is not extremely poor; with a little bit of tuning it can be improved.

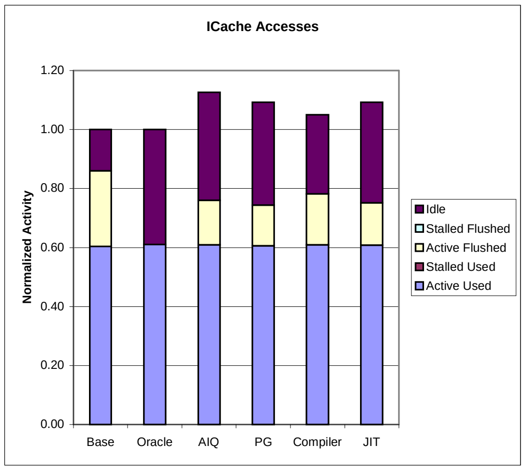

Figure 3: Normalized Activity for ICache Accesses

5 Summary

Although we have not shown the compiler directed resizing as a clear winner. Its energy savings are very close to those of the other schemes. Furthermore, it does not have the complexities like AIQ. AIQ puts its fingers in the issue queue, this can slow down the processor and degrade the overall performance. Our scheme is a non-intrusive way of reducing the energy.

6 Acknowledgement

The author would like to thank Timothy Heil and Subbu Sastry for the many enlightening discussions and their help with the Strata infrastructure.

7 Reference:

[1] Daniele Folegnani and Antonio González, "Reducing Power Consumption of the Issue Logic," *Proc. of the Workshop on Complexity-Effective Design held in conjunction with ISCA 2000*, June 10, 2000.

[2] Alper Buyuktosunoglu, et. al, "A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors," *Proc. of the on Great lakes Sym. on VLSI*, pp. 73-78, 2001.

[3] A. Klaiber, "The Technology Behind Crusoe Processors," Transmeta Technical Brief, http://www.transmeta.com/dev, Jan. 2000.

[4] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June 1997.

[5] A. Baniasadi, and A. Moshovos, "Instruction flow-based front end throttling for Power-Aware High-Performance Processors," *Proc. of International Symposium on Low Power Electronic Devices*, Aug. 2001.

[6] Personal Communication with Alper Buyuktosunoglu, Nov. 2001.

[7] S. Manne, A. Klauser, and D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction," International Symposium on Computer Architecture, Barcelona, Spain, June 1998.

[8] R. Balasubramonian, et. al, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," *33rd International Symposium on Microarchitecture*, pp. 245-257, December 2000.

[9] M. Gowan, L. Biro, and D. Jackson, "Power Considerations in the Design of the ALPHA 21264 Microprocessor," *Design Automation Conference*, pp. 726-731, 1998.

[10] E. Jacobsen, et. al, "Assigning Confidence to Conditional Branch Prediction," International Symposium on Microarchitecture, pp. 142-152, Dec. 1996

[11] Anderson et al., "Physical design of a fourth-generation POWER GHz microprocessor," ISSCC 2001 Digest of Tech. Papers, Feb. 2001, p. 232.

[12] M. Merten, et. al., ""

[13] E. Rohou, et. al, "Dynamically Managing Processor Temperature and Power",

[14] S. Savari,et. al, "Comparing and Combining Profiles",

[15] R.Barnes, et. al, "Feedback Directed Data Cache Optimizations for the X86",

[16] A. Sodani, et. al, "An Empirical Analysis of Instruction Repetition",

[17] M. Merten, et. al, "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization", International Symposium on Computer Architecture, May 1999.

[18] Personal Communication with Timothy Heil, April 2002.

[19] Personal Communication with Subbu Sastry, April 2002.