Power Aware Techniques: Extensions to ISAs

Eva Oliveira

Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal
evaoliveira@di.uminho.pt

Abstract. Mobile computing is heavily dependent on battery life. Although circuit designs already take advantage of microelectronics and microarchitecture-level optimization techniques, device longevity can be further extended through energy aware compilation techniques. This communication gives an overview of software-based power aware techniques, namely re-starters to make temporary processor state visible to software without clogging hardware exception management, exposed bypass latches to manage register file traffic and tag-unchecked loads and stores to improve cache access

1 Introduction

Currently, the major concern in terms of high-performance systems is power-aware design techniques to maximize performance under power dissipation and power consumption constraints. On the other hand, low-power design techniques attempt to reduce power or energy consumption in potable equipments to meet a desired performance or a target throughput. As a result, power dissipation has become a critical design concern in recent years, driven by the increased levels of complexity and emergence of mobile applications.

The number of portable devices is largely increasing, and systems with more capabilities are required due to the more complex applications for these processors. The power reduction has become a main concern to software especialists and the interface between soft/hadrware is more important. Until now we did not have the need to comprehend how processor circuits are made and how much energy did they spend doing any operation, but now it is extremely important that hardware arquitecture designers expose to software compilers designers how everything works.

This works aims to fall into software design techniques, more precisely in compilers design and how they can improve the power reduction mostly by information given by hardware arquitecture designers, about spending energy in internal circuits.

In the last years, the research has been mainly focused on the hardware part. Now, the software component is receiving more attention to obtain a lower power system [1] [2]. The development of a software optimizer requires the construction of a profiler of power consumption of each processor instruction and the adequate selection of compiling techniques that can reduce the energy consumed by a program. This paper presents a set of techniques reported on this subject. Section 2 presents methods used to determine the power consumption of the processor instruction set. Section 3 presents the software restart markers technique and how it exposes the internal details of a processor to the compiler as a temporary state between restart points. Section 3 refers to bypass latches technique and how it reduces compilation times. Section 4 introduces the tag-unchecked loads and stores with direct addressing and how it allows software to access cache data without the hardware performing a cache tag check. The baseline processor adopted to perform some of the tests was an energy-efficient five-stage pipelined MIPS RISC microprocessor, which evaluated the three energy-exposed instruction set techniques [3].

2 Energy Exposed Instruction Sets

Energy-exposed instructions sets are the first software technique, which requires significant knowledge on the hardware. As stated by Bunda et.al. [4], since the number of executed instruction as well as the density of the compiled code have impact in power dissipation, instruction sets can be energy-efficient by using smaller program encoding.

The interface between hardware and software is the key for three techniques that will be exposed. Asanovic et.al. [5] proposed energy-exposed hardware-software interfaces to give software more fine-grain control over energy-consuming microarchitectural operations:

- *software restart markers*: a compiler annotation of the point at witch an exception must occur, reducing this way the enrgy consumption and making temporary processor state visible to software without complicating exception handling. The instruction stream is divided in restartable regions by the compiler, and after handling a trap the OS return to the beginning of the restart region for that specific instruction.
- *exposed bypass latches*: the compiler eliminates register traffic directly working with the hardware, targeting the processor bypass latches. Here we can see the hardware-software interface when software turns off the register fetch and writes back stages of the pipeline, and thereby removes the microarchitectural energy overhead.
- *tag-unchecked loads and stores*: releases the hardware, allowing software to access cache data when the compiler can guarantee that an access will be to the same line as an earlier access.

3 Software Restart Markers

Restart points are encoded by marking the last instruction in a restart region. This instruction is called the *barrier* instruction because it acts as a trap barrier that will commit and irrevocably update machine state only if it is guaranteed that it will not raise an exception and that any preceding instruction will not raise an exception. Also, the barrier instruction ensures that if an exception does occur before it commits, the effects of following instructions will not be visible[3].

The restart program counter will be updated when the barrier instruction commits, and points to the next instruction to be executed; this instruction is the beginning of the next restart region. Marking every instruction as a barrier instruction is equivalent to conventional precise exception semantics. It must be insured that the code in a region is capable by the compiler so that the operating system kernel can restart the process after an exception by simply jumping to the restart PC. Some results of this restart analysis are shown in the next figure for SPECint95 and MediaBench benchmarks.

Next figure shows the number of dynamic instructions that are restart points for both baseline MIPS code and for code after the restart analysis. For baseline MIPS code, only branches and jumps do not have barriers and around 79–95% of all instructions have barriers. After this analysis, 25–40% of instructions are barriers with an average of around 3 instructions in each restart region.

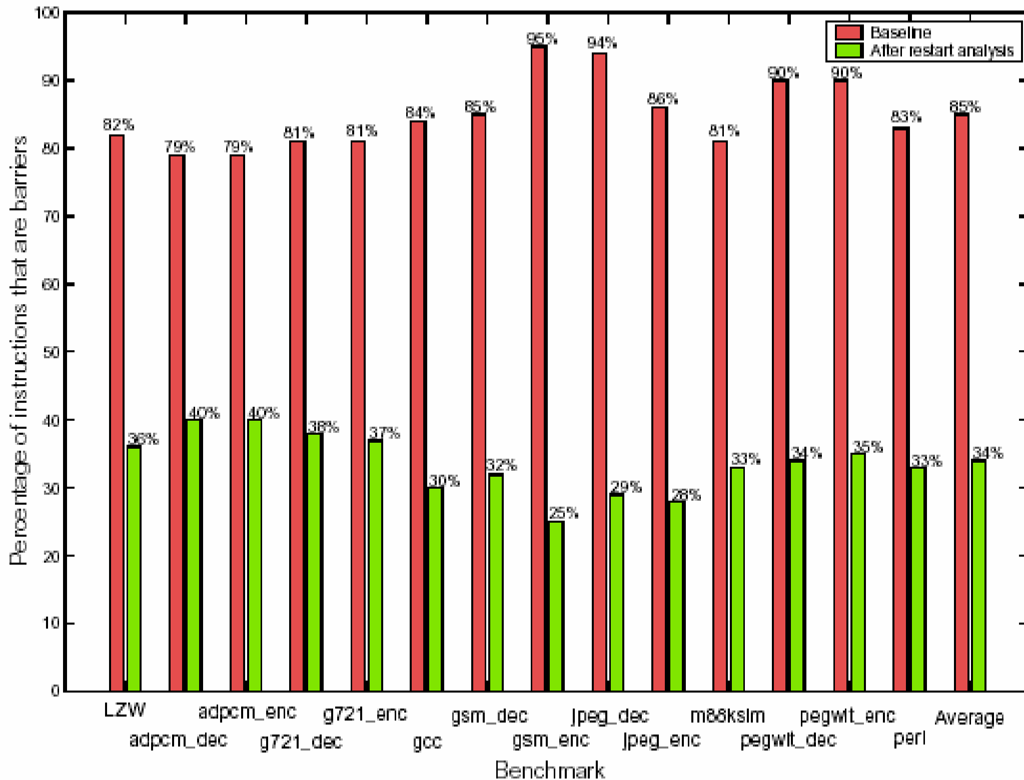

Figure 1 Percentage of dynamic barrier instructions for baseline MIPS code and code after restart analysis. For baseline MIPS, only branch and jump instructions exclude barriers.

The aggressivity of a compiler analysis should generate even larger regions, and allow entire functions to be placed into a single restart region, wich reduces energy in handling exceptions. Five-stage pipeline, restart analysis by itself only results in a minor energy saving in the exception PC pipeline. Instruction pipeline tags each instruction with its PC as it moves down the pipeline identifing the faulting instruction on an exception.

The PC is latched into the EPC register in the system coprocessor if an exception occurs. With the restart analysis only the barrier instructions cause an exception PC to shift down the pipeline, allowing the PC pipeline to be gated off in other cases. The primary advantage of software restart markers is that they make it possible to expose the internal details of a processor to the compiler as temporary state in between restart points[3].

4 Bypass Latches

The second technique is bypass latches, and to expose this issue analysis of register file traffic were considered. Half of the values written to the register file are used only once and usually after the instruction executed immediatlty after teh one producing the value, this analysis were revealed by simulations of the MediaBench and SPECint95 benchmarks. The code sequence to increment a memory variable,

```
lw r1, (r3) # Load value
add r1, r1, 1 # Increment
sw r1, (r3) # Update memory
```

the result of the load and add are only used once by the subsequent instruction and are normally read from the bypass network rather than the register file.

Giving software explicit control of the bypass latches, we can reduce the register file traffic considerably. Writting the above code as:

```
lw RS, (r3) # Load RS latch.
add SD, RS, 1 # Increment, put result in SD.
sw.bar SD, (r3) # Update with barrier.
```

where the `RS` operand specifies the use of the bypass latch in front of one input to the ALU and the `SD` operand specifies the use of the bypass latch that holds data being stored to memory .

This implementation of the exposed bypass latch code takes advantage of the static liveness information that is already maintained by the compiler. When the compiler determines that a value read by an instruction is being referenced for the last time—i.e. the value will be dead after the instruction executes—it appends a "`.l`" suffix to the assembly opcode with a corresponding operand number to indicate the last use of the value. The liveness information generated for each instruction is then used by the scheduler that they added to the assembler [3].

Scheduler has the job of reorders instructions within a basic block an performs several passes on the code. The first concern is to catch latencies that can cause pipeline stalls so reordering instructions attempts to maximize performance, in particular, it tries to fill load-use delay slots with independent instructions**.** It also attempts to fill the architected branch delay slot. Next, schedule determines if bypass latches can be used for general-purpose registers to statically bypass a value, by analysing the lifetime information generated by the compiler. Then the start regions are created and it looks for read caching opportunities. Finally tries to perform additional static bypassing from the memory stage of the pipeline. Additional constraints not required for bypassing from one instruction to a subsequent instruction are raised by static bypassing from the memory. Consider the following example:

```
add r1, r2, r3
sub r4, r5, r6
and r7, r1, r4
```

In this code segment, `r1` is read for the last time by the *`and`* instruction. It is a opportunity to use bypassing from the memory stage by having the first `add` instruction target the X latch. If occurs a instruction cache miss for the *`and`* instruction, the `sub` instruction will overwrite the value in the X latch as it proceeds through the pipeline. This is a problem that mast be avoided by requiring strict pipeline sequencing, making instructions go down the pipeline together with no bubbles between them, or must not permit an instruction which overwrites the X latch to be the intermediate instruction in a memory stage bypassing sequence. Second optionwas chosen, as this placed no additional constraints on the hardware implementation. Instructions which target the bypass latches are candidates for the intermediate instruction in a memory stage bypassing sequence, since they do not write back to the register file for example:

```
add X, r2, r3
sub RS, r5, r6
and r7, X, RS
```

For the simulations, it was modelled the RS, RT, SD, and X bypass latches by reserving four general-purpose registers in the compiler and using their specifiers in the scheduler when modifying an instruction to target a bypass latch. It was observed that the loss of these registers in the compiler's register allocator did not have an adverse effect on performance. Ideally, the instruction set encoding would be designed to support bypass latches directly. In the reduction in register file writes on average, 34% of all writes are

eliminated. In the reduction in register file reads on average, 28% of all reads are eliminated.[3]

5 Tag-Unchecked Loads and Stores with Direct Addressing

The third technique deals directly with hardware by releasing it of tag check. Tag check in primary data is one of the most significant source of energy consumption. If we use direct addressing we allow software to access cache data realesing hardware of this task, thus reducing energy.

These *tag-unchecked loads and stores* save the energy of performing a tag check when the compiler can guarantee an access will be to the same line as an earlier access. If the compiler cannot determine this information, or if cache lines are evicted due to interrupts or cache invalidations, direct addressing gracefully degrades back to conventional tag-checked accesses.[5]

These operations tell the hardware to remember a location of a cache line, thus when software wants to access that line again, hardware directly accesses that data without searching a tag. They augment the processor state with some *direct address registers* (DARs). These registers are set and used by software, and contain enough information to specify the exact location of a cache line in the cache data RAM as well as a valid bit. The exact width and data layout of the DARs is hidden from software to avoid exposing the implementation dependent structure of the cache. In particular, software is only made aware of the length of a cache line, but not the total cache capacity or associativity. [5]

Software places values in the DARs as an optional side effect of performing a load or store. A tag-unchecked load or store specifies a full effective virtual address in addition to a DAR number. If the DAR is valid, its contents are used to avoid a tag search; if it is invalid, hardware falls back to a full tag search using the entire virtual address. The implementation described here uses a separate DAR specifier in each instruction, which takes 3 bits from the 16-bit immediate offset. An alternative encoding is to implicitly associate a DAR with some set of base registers, which reduces ISA changes at the cost of complicating compiler register allocation. It was not considered this option further in this paper. Direct addressing is only used for data caches. Instruction caches have very regular access patterns and are only accessed via the program counter, and hence are amenable to software-invisible micro-architectural techniques to remove tag checks [6, 7, 8].

```
Old Code            New Code
sub $sp,64          sub $sp,64
sw $ra,60($sp)      swlda $ra,60($sp),$da0
sw $fp,56($sp)      swda $fp,56($sp),$da0
sw $s0,52($sp)      swda $s0,52($sp),$da0
```

It was modified a SUIF-based C compiler and it has been implementes direct addressing. The compiler uses a simple approach to eliminate tag checks with direct addressing. First, find two references, one dominating the other, thus the paths that can cause the subordinate access to be executed, cause the dominant reference to be executed first.Then, it prove that the two references always point to the same cache line.

The second reference can then skip the tag check, by having the dominant reference write a DAR that the subordinate reference reads. Any other code between the two references, including assignments, control flow, or even function calls, can not affect correctness because hardware will invalidate DARs that point to lines that get evicted between the definition and the use of a DAR. [5]

The loop unrolling technique is the most common and efficient in alignment memory operations. This aligment optmizes performance of the code, and reduces energy,

by guarantee that each memory operation in the loop only acesses the cache with a certain aligment.

The code in the next Figure shows the original loop with unrolling. After unrolling the loop by a factor consistent with the size of the cache line, it can be guarantee that each memory operation in the loop only accesses the cache with a certain alignment. This is the case in this example assuming that A is an array of 64-bit data, and the cache line size is 32 bytes.[5]

```
for (i=0; i<N; i++) {
A[i] = 0;
}
```

a)

```
for (i=0; i<N; i += 4){
A[i + 0] = 0;
A[i + 1] = 0;
A[i + 2] = 0;
A[i + 3] = 0;
}
```

b)

```
    for (i=0; i<N; i++) {
    if (&A[i]
    % line_size == 0)
    break;
    A[i] = 0;
}
for(; i<N; i +=4) {
A[i + 0] = 0;
A[i + 1] = 0;
A[i + 2] = 0;
A[i + 3] = 0;
    }
```

c)

(a) A simple loop with a single memory reference.
(b) After loop unrolling.
(c) A pre-loop inserted to guarantee alignment in the unrolled loop body.

The tag-unchecked compiler analysis was implemented for the SUIF compiler, which was configured to output instrumented C code. This code has unrolled loops and is augmented with statistics gathering capability. Next figure shows how many tag checks were eliminated and whether the elimination was for a load or store. It is important to break these cases out once the tag check is less of the total energy of a store since the value update takes energy.

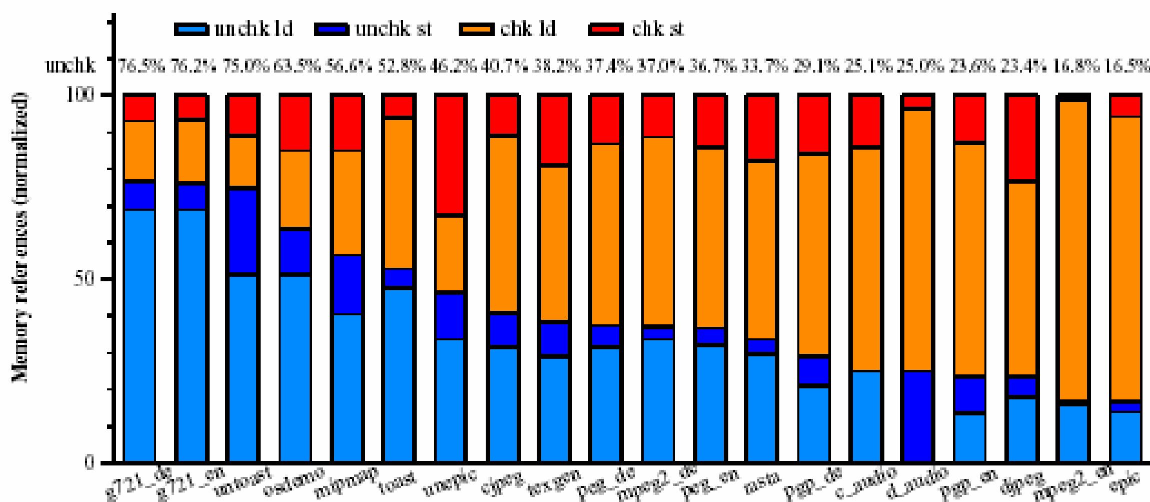

Figure 2 Tag check elimination for Mediabench programs compiled by SUIF. Eight direct address registers are used. The lowest part of the bar is tag unchecked loads, then unchecked ones. Over that are tag checked loads and stores. The number on top of each bar (unchk) is he percentage of tag checks eliminated.

7 Power Management in a Pentium M Processor : ISA Extensions

Although there is a lack of documentation about instruction level optimization on Pentium M processors, some implemented power aware features were identified. They have reduced the number of instructions per task with better branch prediction, decreasing the number of speculated instructions, thus in practice reducing the number of overall processed instructions. Another feature was the reduction of the number of micro-ops per instruction, as the out-of-order implementations of IA-32 Instruction Set Architecture break macro-instructions into a sequence of one or more simple operations (called micro-operations) and handling and executing each micro-op consumes power, eliminating micro-ops from micro-op stream or combining several micro-ops together reduces overall power.

6 Conclusions

The major conclusion of this work is that more than ever there is more and more need of an interface between hardware and software engineers to improve applications to lower power consumption, without degrading performance. Hardware complexity has to become readable to compiler designers thus they can improve their algorithms and allow that portable systems may reduce energy in complex applications. When system software designer can compile in a low power mode, programmers will have access to new program techniques in low power environments, but there is a long way even so. We are leading in this direction, where compilers concern more and more with power, tests are being made by system compiler designers to let hardware and software be more interactive with each other without disrupt security rules but turning the interoperation more flexibility, and some conclusions are being reached to make possible that one day we can for example compile our C code and tell GCC compiler that we want *–power* option enabled., thus turning our code maximized to low power.

References

[1] Tiwari, V.; Malik, S.; Wolfe, A, "Power analysis of embedded software: a first step towards software power minimization," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume: 2 Issue: 4 , Dec. 1994 Page(s): 437 –445

[2] Mehta, H.; Owens, R.M.; Irwin, M.J. "Instruction level power profiling". Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on , Volume: 6 , 1996 Page(s): 3326 -3329 vol. 6

[3] "Energy-Exposed Instruction Sets", Krste Asanovic, Mark Hampton, Ronny Krashinsky, and Emmett Witchel, Power Aware Computing, Robert Graybill and Rami Melhem (Eds.), Kluwer Academic/Plenum Publishers, June 2002.

[4] Bunda, J., Fussell, D. et.al., "Energy–Efficient Instruction Set Architecture for CMOS Microprocessors", in Proceedings of the 28th Annual Hawaii International Conference on System Sciences, 1995, pp. 298–304.

[5] Asanovic, K., Hampton, M., Krashinsky, R., Witchel, E., "Energy–Exposed Instruction Sets", in Power–Aware Computing, ed. by R. Graybill and R. Melhem, Kluwer Academic Publishing, 2002.

[6] A. Ma, M. Zhang, and K. Asanovic. Way memorization to reduce fetch energy in instruction caches. ISCA Workshop on Complexity Effective Design, July 2001.

[7] M. Muller. Power efficiency & low cost: The ARM6 family. In Hot Chips IV, August 1992.

[8] R. Panwar and D. Rennels. Reducing the frequency of tag compares for low power I-cache design. In SLPE, pages 57–62, October 1995.

[9] Tiwary, V., Malik, S., Wolfe, A., “Compilation Techniques for Low Energy: an Overview”, in Proceedings of the Internationsl Symposium on Low Power Electronics, 1994.

[10] Zhang, Y., Hu, X., Chen, D., “Global Register Allocation for Minimizing Energy Consumption”, in Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED), 1999.

[11] Gebotys, C., “Low Energy Memory and Register Allocation Using Network Flow”, in Proceedings of Design Automation Conference (DAC), 1997.

[12] Chang, J., Pedram, M., “Register Allocation and Binding for Low Power”, in Proceedings of Design Automation Conference (DAC), 1995, pp. 29–35.

[13] Bose, P., Brooks, D., Irwin, M., Kandemir, M., Martonosi, M., Vijaykrishnan, N., “Power–Efficient Design: Modeling and Optimizations”, Tutorial Notes, International Symposium on Computer Architecture (ISCA), Gothenburg, Sweden, July 2001.