Energy Efficient Code Generation Exploiting Reduced Bit-width Instruction Set Architectures (rISA)*

Aviral Shrivastava
Center for Embedded Computer Systems
School of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA
aviral@ics.uci.edu

Nikil Dutt
Center for Embedded Computer Systems
School of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA
dutt@ics.uci.edu

1. ABSTRACT

Energy consumption is emerging as a critical design concern for programmable embedded systems. Many Reduced Bit-width Instruction Set Architectures (rISA) (e.g., ARM Thumb) are being increasingly used to decrease code size. Previous work has explored energy savings in non-cached rISA architectures as a byproduct of code size reduction. In this paper we present an energy efficient code generation technique for rISA architectures, and furthermore explore energy savings for both cached and non-cached architectures. Our code generation technique uses profile information to find the most frequently executed parts of the program. By aggressively reducing code size on frequently executed parts, fewer fetches to instruction memory are incurred, thus reducing the power consumption of the instruction memory. We achieve an average 30% reduction in instruction memory energy consumption in cached systems, on a variety of benchmarks, as compared to non-rISA architectures.

2. INTRODUCTION

With the decreasing cost of memory and stringent energy requirements of emerging processors, much research has been done towards applying/modifying code compression techniques to lower the energy consumption for low cost, battery operated embedded applications[8, 2, 6]. However all these code compression techniques employ complex translation schemes. Such complex translation scheme has two distinct disadvantages: first, the hardware needed for such complex translation schemes is either slow or costly, and second, the translation scheme is too complex for a compiler to make good use of it. This has been recognized as a problem and solutions have been attempted.[1, 5].

Reduced bit-width Instruction Set Architecture (rISA) is a popular architectural feature that is used to reduce instruction memory size of programs. rISA processors can execute instructions from two different Instruction Sets (IS): the "normal" IS, and the "reduced bit-width" IS. The reduced bit-width instruction set encodes the most frequently occurring instructions in fewer bits, resulting in code size reduction. A common example of this feature is ARM7TDMI processor, which has a 32-bit normal IS, and 16-bit wide "Thumb" IS. Other processors that include the rISA feature are MIPS32/16 bit TinyRISC, STMicro's ST100, and the ARC Tangent processor.

If the whole program can be expressed in terms of rISA instructions, then upto 50% code size reduction may be achieved, which may in turn result in upto 50% reduction in instruction memory energy consumption. Thus the main advantage of rISA lies in achieving high code compression and energy savings with minimal changes in hardware. The simplicity of rISA design implies a simple "translation unit". Simplicity of rISA also makes it amenable to aggressive exploitation using compiler techniques[4, 3].

Although [7] shows a 20% decrease in power consumption of ARM7TDMI (rISA architecture) core over ARM7DMI (non-rISA architecture), they have not shown energy consumption results on cached systems, and their objective for compilation is code compression, rather than energy savings.

The contributions of this paper are two fold: first we propose a novel profile guided energy efficient code generation technique for processors that have rISA, and second we explore energy savings in rISA architectures on cached as well as non-cached systems.

In Section 3, we refine the problem of energy efficient code generation, in Section 4, we present our solution, and demonstrate the efficacy of our approach in Section 5.

3. PROBLEM DEFINITION

Code compression techniques (including rISA) owe the decrease in energy consumption (and even performance improvement, if any) to code size reduction; this in turn decreases the number of fetches to the memory subsystem and thus reduced power consumption. However it is noticeable that the goals of code compression and energy reduction are not the same. Code compression techniques aim to minimize *Static Code Size* i.e. $\sum_{i=1}^{i=n} c_i$, while energy efficient techniques aim to minimize the *Dynamic Code Size* i.e. $\sum_{i=1}^{i=n} c_i x_i$. Where c_i is the number of instructions in the i^{th} Basic Block, and x_i is the number of times it is executed.

As rISA processors can operate in either the rISA mode or in the normal, our rISA Archietcural model assumes the presence of mode change instruction in both the Instruction Sets (*mx* in normal IS, and *rISA_mx* in rISA IS). This provides us the ability to switch execution mode at instruction level granularity.

The compilation process for rISA code generation consists of three main steps. In the first the compiler marks instructions that can be converted into rISA instructions. Contiguously marked instructions constitute a *rISABlock*. In the second step, profitable rISABlocks are converted to rISA Instructions. And finally mode change instructions are added at the begining and end of *rISABlock*. The addition of mode change instructions however results in increased code size (even though it may be estimated and accounted for in the profitability analysis). The problem then translates to trying

*This work was partially supported by SRC contract 2003-HJ-1111 and NSF grants CCR-0203813 and CCR-0205712

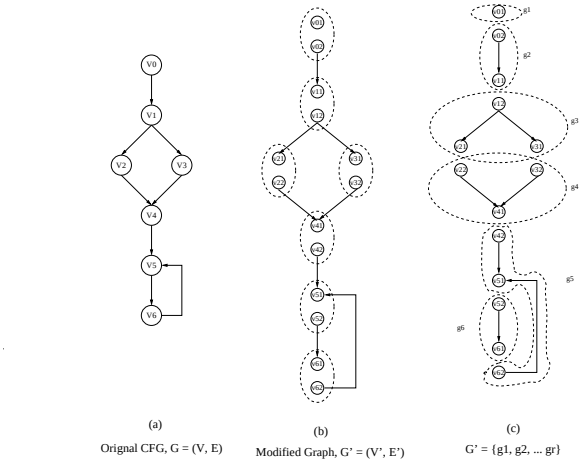

Figure 1: Mode Change Instruction Insertion

to minimize the addition of mode change instructions, and to add these mode change instructions so as to minimize the *Dynamic Code Size*.

4. APPROACH

The insertion of mode change instructions is done in two steps. If mode change occurs inside a basic block, corresponding mode change instruction is inserted at the boundary of rISA Block in the first step.

After the first step, the CFG (Control Flow Graph) can be visualized as a directed graph $G = (V, E)$, where V represents the basic blocks, and E represent the Control Flow edges as shown in Figure 1(a). G has two distinguished vertices, the start vertex v_0 and the end vertex v_n. Three functions are thus defined on V,

- $ExecFrequency : V- > N$ gives the execution frequency for each vertex.
- $EntryMode : V \rightarrow \{Normal, rISA\}$ gives entry mode of the basic block represented by the vertex is Normal or rISA. $EntryMode(V_i)$ is *rISA* if the first instruction of the basic block is a rISA instruction. Otherwise it is *Normal*.
- $ExitMode : V \rightarrow \{Normal, rISA\}$. $ExitMode(V_i)$ is *rISA* if the last instruction of the basic block is a rISA instruction. Otherwise it is *Normal*.

We get $ExecFrequency$ for each basic block from the profile information. The functions $EntryMode$ and $ExitMode$ are computed for each basic block.

We can switch the $EntryMode$, or $ExitMode$ of a vertex by inserting a mode change instruction at the start of the basic block, or at the end of the basic block respectively. However switching the $EntryMode$ or $ExitMode$ of the vertex v_i costs $ExecFrequency(v_i)$.

The problem of mode change instruction insertion is to find $EntryMode$ and $ExitMode$ for each vertex so that,
for each edge $(v_i, v_j) \in E$,
$ExitMode(v_i) == EntryMode(v_j)$
such that the switching cost is minimized. The switching cost essentially represents the *Dynamic Code Size*.

To solve this problem we transform our graph G. We break each vertex v_i into two vertices, v_{i1} and v_{i2} in graph G' as shown in Figure 1(b). Vertex v_{i1} represents the entry of v_i, and v_{i2} represents the exit of v_i. All the incoming edges into v_i now come to v_{i1}, and all the outgoing edges from v_i, now emanate from v_{i2}. Two functions are defined on vertices of G',

- $ExecFrequency(v_{ij}) = ExecFrequency(v_i)$
- $Mode(v_{i1}) = EntryMode(v_i)$
- $Mode(v_{i2}) = ExitMode(v_i)$

The new Graph $G' = (V', E')$ is a forest of connected components. Our problem now reduces into finding $Mode$ for each vertex so that all the vertices in a connected component have the same mode.

We identify all the connected components of $G' = \{g_1, g_2, ...g_k\}$, as depicted in Figure 1(c). Each connected component is a subgraph $g_i = (V_i, E_i)$, containing a subset of vertices, $V_i \subset V', V_i = \{u_1, u_2, ...u_r\}$.

These vertices are partitioned into two (possibly empty) sets V_{Normal} and V_{rISA}.

$V_i = V_{Normal} \bigcup V_{rISA}$, and $V_{Normal} \bigcap V_{rISA} = \phi$.

Cost of converting all vertices to rISA Mode = $\Sigma_{i=1..|V_{Normal}|} ExecFrequency(u_i)$.

Cost of converting all vertices to Normal Mode = $\Sigma_{i=1..|V_{rISA}|} ExecFrequency(u_i)$.

We pick the lower cost conversion, thus deciding upon $Mode$ of each vertex in G', and hence $EntryMode$ and $ExitMode$ of each vertex in G. Finally we insert the appropriate mode change instructions. To change the $EntryMode$ of a basic block from Normal to rISA, we add a *mx* instruction as the first instruction of the basic block. To change the $EntryMode$ of a basic block from rISA to Normal, we add a *rISA_mx* instruction as the first instruction of the basic block. To change the $ExitMode$ of a basic block from Normal to rISA, we add a *mx* instruction as the last or second last instruction of the basic block. To change the $ExitMode$ of a basic block from rISA to Normal, we add a *rISA_mx* instruction as the last or second last instruction of the basic block.

Note that if the last instruction of a basic block is a branch operation, and the machine does not have a delay slot, then the mode change instruction has to be added as the second last instruction in the basic block.

5. EXPERIMENTS

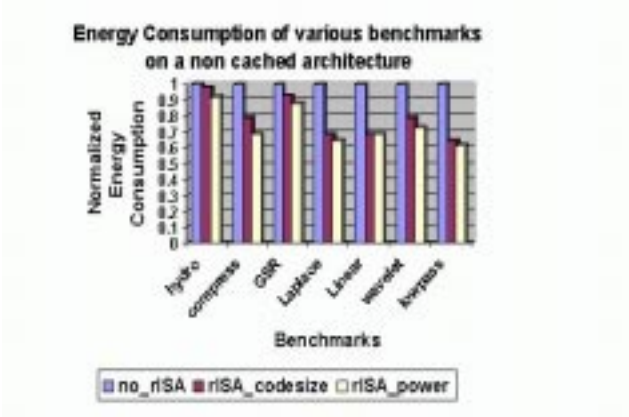

Figure 2: Energy Savings in non cached architecture

Figure 2 shows the normalized energy consumption for a non-cached architecture. The machine is MIPS R4K-like, with a 32-bit wide processor to memory bus. For each benchmark, the leftmost bar shows the energy consumption by instruction memory subsystem, assuming the processor does not have the rISA feature (base case). The middle bar is the energy consumption when code is compiled for minimum code size. The rightmost bar for each benchmark shows the energy consumption when the benchmark is compiled for minimum energy. As can be seen from Figure 2, our technique achieves about 26% instruction memory energy savings over a non-rISA architecture. Our energy saving compilation technique is responsible for about 5-10% more energy savings, over minimum code size compilation.

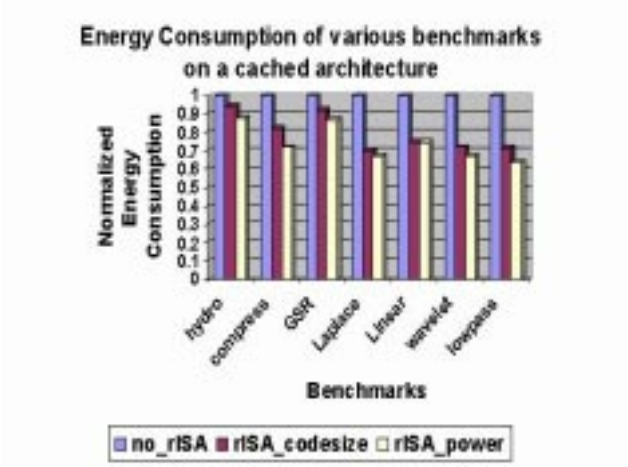

Figure 3: Energy Savings over all the benchmarks in a cached architecture

Figure 3 shows the normalized energy consumption of instruction memory subsystem of rISA architectures. The machine is assumed to have an L1 instruction cache, connected to the processor and Main memory using separate 32-bit buses. We assume that L1 cache hit latency is 1 cycle, and miss latency is 10 cycles. For each benchmark, the leftmost bar shows the energy consumption by instruction memory subsystem, assuming the processor does not have the rISA feature (base case). The middle bar is the energy consumption when code is compiled for minimum code size. The rightmost bar for each benchmark shows the energy consumption when the benchmark is compiled for minimum energy. As can be seen from Figure 2, our technique achieves about 33% instruction memory energy savings over a non-rISA architecture. Energy efficient compilation technique is responsible for about 10% more energy savings, over minimum code size compilation.

To study the sensitivity of energy savings technique, we estimated energy consumption by instruction memory for all the benchmarks over a range of cache sizes (64 Byte, 128 Byte, 256 Byte), line sizes (8 byte, 16 byte, 32 byte), and associativities (2, 4, 8). Figure 4 shows the energy consumption of instruction memory across various cache configurations, near the *critical cache size* for the benchmark *compress*. *Critical cache size*, informally is the cache size for an application, after which increasing the cache size, does not result in a significant improvement in performance. The energy consumption of instruction memory first decreases, and then increases with the increase in cache size. Increasing the cache size reduces the cycle count, thus decreasing the energy consumption. By increasing cache size more than *critical cache size*, there is no advantage in performance, but the energy consumed per cache access increases, thus resulting in an increase in memory consumption. For the same cache size the energy consumption of the instruction memory decreases with increase in associativity. This can be attributed to the performance improvement. Finally for the same cache size and line size, the energy consumption increases with increasing associativity. Increasing associativity, implies that more cache lines tags be compared simultaneously to find the data. Thus the energy consumed per access is much more than the performance benefits. When the cache size is less than *critical cache size*, code size reduction achieves energy savings due to reduced cache and main memory accesses. However when we increase the cache size beyond *critical cache size*, reduced code size does not result in lesser main memory accesses, thus reducing energy benefits.

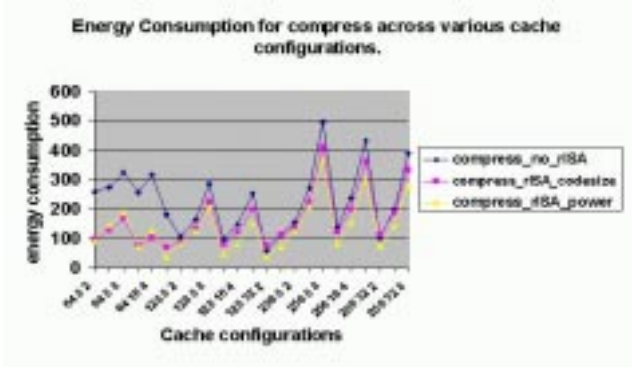

Figure 4: Energy Savings of compress over various benchmarks

Thus in conclusion, we can achieve upto 33% instruction memory energy savings as compared to a non-rISA architecture, on a SOC based system. Also our proposed energy efficient code generation technique that results in up to 10% instruction memory energy savings over code compression compilation techniques. Instruction memory energy (in this paper) comprises cache energy, memory energy, bus energy, and "translation unit" energy.

6. REFERENCES

[1] L Benini, A. Macii, and A. Nannarelli. Cached-code compression for energy minimization in embedded processors. In *ISLPED 2001, pp 322-327*, 2001.

[2] L Benini, A. Macii, and M. Poncino. Selective instruction compression for memory energy reduction in embedded systems. In *IEEE, Proceedings of Micro-30*, 1999.

[3] A. Halambi, A. Shrivastava, and el.al. An efficient compiler technique for code size reduction using reduced bit-width isas. In *DATE*, 2002.

[4] Young-Jun Kwon, Xiarong Ma, and Hyuk Jae Lee. PARE: instruction set architecture for efficient code size reduction. *Electronics Letters , pp. 2098-2099*, 1999.

[5] H. Lekatsas, Jorg Henkel, and Venkata Jakkula. Design of one-cycle decompression hardware for performance increase in embedded systems. In *DAC 2002*, 2002.

[6] H. Lekatsas, Jorg Henkel, and Wayne Wolf. Code compression for low power embdedded system design. In *DAC 2000, pp.294-299*, 2000.

[7] Simon Segars, Keith Clarke, and Liam Goudge. Embedded control problems, thumb, and the arm7tdmi. *IEEE Micro'95*, pages 22–30, 1995.

[8] Y. Yoshida, B.Y. Song, H. Okuhata, T. Onoye, and I. Shirakawa. An object code compression approach to embedded processors. In *ISLPED 1997, pp.265-268*, 1997.