Understanding Energy Consumption on the C62x

Keith D. Cooper and Todd Waterman

Department of Computer Science
Rice University
Houston, Texas, USA

Abstract

The emergence of portable computing devices and high-power processors has fueled recent interest in energy consumption. This paper examines how a processor consumes energy and a compiler can assist in energy conservation. Successfully transforming programs to reduce energy consumption is heavily dependent on the target machine and type of program. We examine a specific processor, the TMS320C6200 from Texas Instruments, and demonstrate how a compiler can reduce energy consumption for certain programs. We hope that our methodology can be used as a blueprint for future work on reducing energy consumption in other processors.

1 Introduction

Energy consumption has recently become an important concern for various computer systems. Interest in energy consumption has been fueled by the proliferation of portable computing devices that have a limited supply of energy, and high-power processors that dissipate large amounts of heat. These devices have motivated architects to design new processors with lower energy needs and power saving features [4, 7, 11]. It has also caused systems and compiler researchers to inspect how they can design software which consumes less energy [5, 10].

Most optimizations that decrease a program's running time also decrease its energy consumption. Dead code elimination decreases the number of instruction fetches. Removing memory accesses helps with both speed and power. Transforming programs solely to conserve energy, however, has produced mixed results. The profitability of an energy reduction technique depends heavily on the target machine and the type of program being transformed. While individual papers show improvement on specific processor models, those results are often heavily dependent on the implementation of a specific processor. Thus, it is unlikely that a general method for a compiler to reduce energy consumption on all processors and all programs will ever be found. This paper shows how program transformation can save energy for some programs on a particular processor. The techniques presented in this paper will not apply to all, or even most, other processors. However, the goal of this paper is to serve as a blueprint for other researchers hoping to reduce energy consumption on different machines.

The machine investigated in this paper is the TMS320C6200 series DSP from Texas Instruments. All the experiments described in this paper use a power simulator provided to us by Texas Instruments. The simulator calculates power consumption assuming a 100MHz C6201 CPU with 512K on-chip program memory and 512K on-chip data memory [3]. Analyzing the energy consumption of the C6200 series shows that a large portion of the energy dissipation on the C62x occurs during instruction fetch. We examine how instruction fetch occurs on the C62x, and how this allows for the creation of programs that produce equivalent results, but consume varying amounts of energy. This paper presents multiple optimization techniques that can sub-

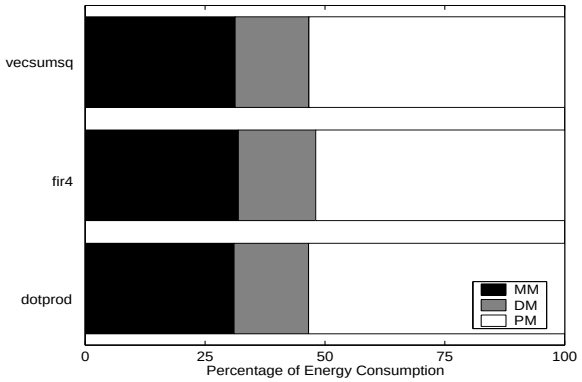

Figure 1: Distribution of Energy Consumption

stantially reduce the energy consumption of certain types of programs on the C62x.

2 The C6200 Architecture

The C6200 series of DSPs produced by Texas Instruments are VLIW processors with fully pipelined functional units and predicated operations [8]. The C6200 has two similar data paths that each contain four functional units and 16 registers. Three of the functional units on each path are ALUs with varying capabilities while the fourth is a dedicated multiplier.

Figure 1 shows the distribution of energy consumption on the C6200 for three different benchmarks. These benchmarks are hand-optimized procedures created by Texas Instruments for the C6200. Energy consumption is divided into three different sources: the functional units and registers (MM), data memory (DM), and program memory (PM). The graph clearly shows that more than half of the energy dissipation of these codes on the C6200 occurs in the program memory, and more than thirty percent occurs in the functional units. Data memory does not play as important a role in energy consumption for these codes.

We began by examining the energy consumption of the functional units with a simple experiment. We analyzed the energy consumption of two programs that executed for the same number of cycles. The first program used all eight functional units each cycle, while the second program performed no actual work. In the second program each operation that was removed was replaced with a NOP to prevent changes in the instruction fetch behavior. Both programs consisted of a 105-cycle loop that executes for 1000 iterations. This produced programs that ran long enough to exhibit stable-state behavior and were not dominated by branch instructions. The second program achieved a 44% energy savings in the functional units over the first program and a 13% energy saving overall. Though these savings are large, it indicates that over half of the energy consumed in the functional units is a fixed overhead that depends solely on running time.

Assuming a program has already been optimized for speed, substantially reducing energy consumption in the functional units is difficult. The massive reduction of operations executed, which reduced power in our experiment, is not possible in real programs. Furthermore, any type of rescheduling that lengthens execution time works against the dominant factor in functional unit energy consumption.

Program memory is the largest component of energy consumption shown in figure 1. Traditional optimizing compilers do not consider the impact of optimization on the power consumption in program memory. Since almost all operations on program memory are instruction fetches, reducing program fetches is an area where energy consumption might successfully be reduced. To reduce program fetches, it is necessary to have a detailed understanding of how this process works on the C6200.

The C6200 uses a packed VLIW architecture which attempts to minimize code size and eliminate the fetching of superfluous operations. In contrast to a traditional VLIW machine, the C6200 only requires an operation to be specified for a functional unit when it is used in a given cycle. This leads to the existence of both fetch packets and execute packets. A fetch packet is a group of eight operations that are simultaneously loaded from memory. An execute packet consists of one to eight instructions that are executed at the same time. An execute packet

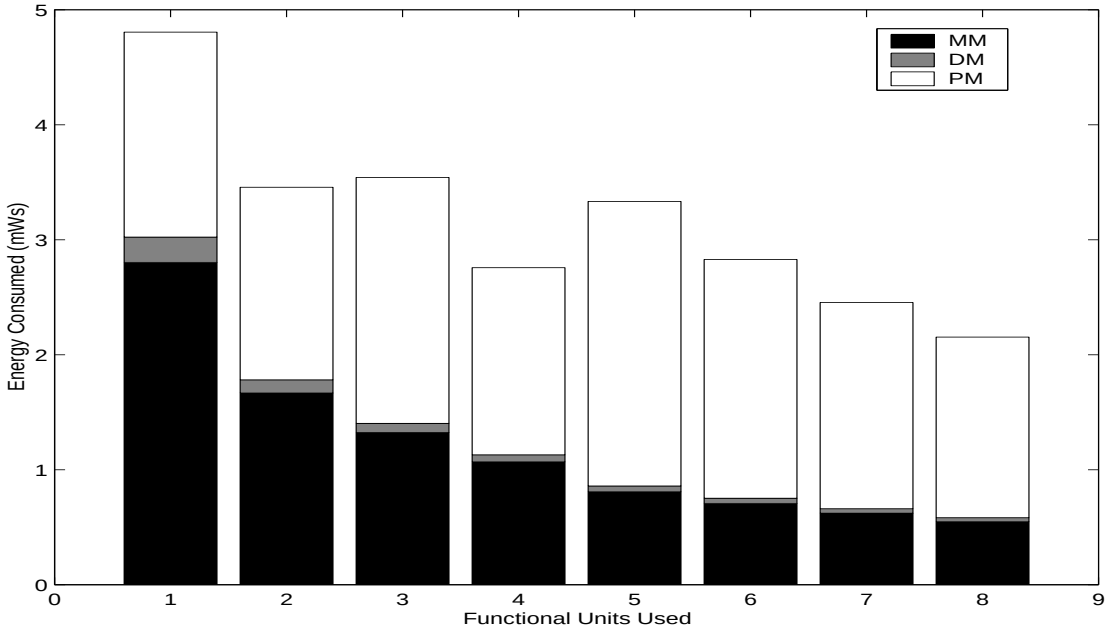

Figure 2: Impact of Functional Units on Energy Consumption

must exist within a single fetch packet. Hence, a fetch packet can contain from one to eight execute packets. When packing execute packets into fetch packets, if the next execute packet will not fit in the current fetch packet, the remainder of the fetch packet is filled with NOPs. These NOPs become a part of the previous execute packet and do not require any extra cycles to execute, but obviously take up space in the program memory.

The existence of fetch and execute packets complicates the instruction fetch and decode stages of the C6200s pipeline. The C6200 series has an 11 stage pipeline with four stages for instruction fetch and two for instruction decode. The first decode stage determines the number of execute packets in the current fetch packet. Therefore, the processor is not aware of how many execute packets are in a fetch packet until four cycles after fetching begins. The processor begins by loading a fetch packet from memory every cycle until the first fetch packet has been decoded and it knows how many execute packets are contained within. If the fetch packet contains multiple execute packets then the fetch and decode portion of the pipeline can be stalled a single cycle for each execute packet beyond the first. This gives the processor time to place each of the execute packets into the remainder of the pipeline.

Some C6200 operations, such as addition, are implemented on most or all of the functional units. The energy cost of such an operation varies from functional unit to functional unit. Our measurements, however, showed that this effect was, in general, much smaller than the memory effects described in this paper.

3 Reducing NOPs Fetched

The packed VLIW design of the C62x reduces the number of NOPs in the code when compared to a traditional VLIW design. However, since execute packets cannot cross fetch packet boundaries, code for the C6200 series still contains some NOPs. For example, consider a program that has enough low-level parallelism to constantly keep five of the functional units busy. Every execute packet would contain five operations and each fetch packet would only contain a single execute packet. Consequently, every fetch packet would contain three NOPs and instruc-

tion fetch would perform almost twice as much work as necessary.

The NOPs required for filling fetch packets cannot be eliminated without reorganizing execute packets. If the program with enough parallelism to fill five functional units was instead written to use only four functional units then no NOPs would be necessary for packing; two execute packets would fit perfectly into each fetch packet. The total number of fetch packets would be halved and the program would require roughly twenty percent more time to run. Less energy would be consumed fetching instructions, but the code would run slower incurring different energy penalties.

Figure 2 presents a more general examination of the tradeoff between functional units used and energy consumed. We created a test program with enough parallelism to constantly use all eight functional units. The program had a single loop of 840 operations that iterated 1000 times. Eight different versions of the program were then created using from one to eight functional units. The overall trend of the graph is towards less energy consumption as more functional units are used. This can be attributed to the reduction of energy consumption in the functional units, which is heavily related to the running time (see figure 3). However, there are also some anomalies in the graph. The four functional-unit program consumes considerably less energy than the program that uses five functional units, and slightly less than the one that uses six. This is due to the variation of energy consumed in program memory. The amount of energy consumed by program memory depends almost directly on the number of fetch packets loaded. It does not appear to depend heavily on the running time of the program.

The results of figure 2 demonstrate the ability of careful fetch packet construction to reduce power. If the amount of instruction level parallelism available in a program is over six operations per cycle then scheduling the code to run as quickly as possible is probably the most energy conserving choice. If the available parallelism is less, but still above four operations per cycle, then energy can be saved by trimming the

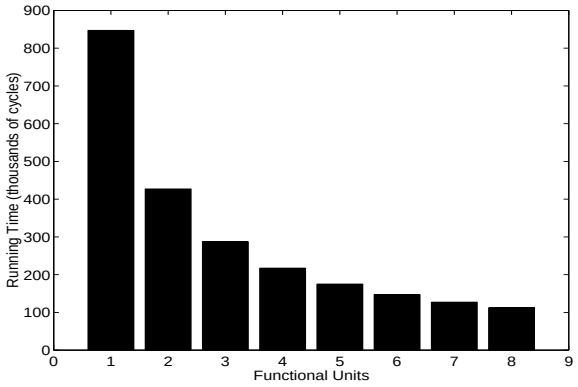

Figure 3: Functional Units vs. Execution Time

schedule to only issue four operations per cycle and minimizing the number of fetch packets necessary to contain the code.

These experiments suggest a simple heuristic: avoid fetching operations that do not accomplish productive work. To implement this, the compiler writer could alter the instruction scheduler to reduce the number of NOPs placed in fetch packets. Once the first execute packet in a fetch packet has been constructed the scheduler could guarantee that the next execute packet constructed would not be larger than the remaining space in the fetch packet. A scheduler using such a heuristic would produce a slower schedule than normal since it has an additional restriction on the amount of parallelism available at each cycle, but would construct a smaller program which would reduce energy consumption in the program memory.

4 Improving Loops

Eliminating the fetching of NOPs is one way to conserve energy on the C62x processor. However, even when there are no NOPs in the code the C6200's fetch unit can still waste energy within loops. In the C6200, by the time the processor has determined the contents of a fetch packet it has already begun fetching the next five packets. If the original fetch packet contains a branch then the processor will begin fetching future packets from the location of the branch.

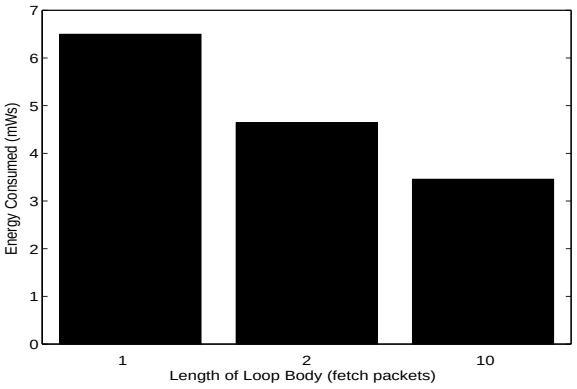

Figure 4: Energy Effects of Unrolling

When the processor is running at full utilization, each fetch packet consists of a single execute packet, then the packets already in the pipeline are needed for the branch's delay slots. If there are multiple execute packets in some of the fetch packets following the branch then not all of the fetch packets already in the pipeline will be needed to fill the delay slots. Some number of fetch packets will have been unnecessarily fetched.

This problem only occurs when there is a low degree of parallelism available within the loop. If five or more operations can be executed simultaneously then the execute packets will be large and only a single packet will fit in each fetch packet. In this case, each of the fetch packets in the pipeline when the branch is detected will be necessary. When there is a lower degree of parallelism then fetch packets will be loaded unnecessarily and energy will be wasted.

Since the C62x cannot know the structure of fetch packets before they are fetched it is impossible to avoid the energy waste associated with branches in code with a low level of parallelism. However, code can be restructured to limit the number of times the situation occurs. The common loop transformation that can reduce energy waste in this case is loop unrolling. Loop unrolling reduces the number of iterations, and branches, in a loop at the expense of a longer loop body. This does not effect the running time of the program.

The effects of loop unrolling on a loop with a low level of parallelism can be seen in figure 4. Figure 4 shows the amount of energy consumed in a loop that executes only one instruction per cycle with a varying loop body length. The original loop fits inside a single fetch packet, but because of the C6200's fetching mechanism five extra packets are fetched for each that is necessary. Unrolling by a factor of two causes a great reduction in energy consumption and further unrolling continues to reduce energy consumption with diminishing returns. Loops that execute more than a single operation during each cycle will still benefit from loop unrolling provided that they do not issue at least five operations in each of the branch's delay slots.

Simply unrolling a loop with a short body and a low degree of parallelism can produce a considerable reduction in energy by reducing the wasted fetches at the end of a loop. As Figure 4 shows, the largest improvement occurs between the original loop and the loop unrolled by a factor of two. (In further unrolling, the potential savings are smaller.) On the C6200, program memory is distinct from data memory. It is often implemented with a ROM chip of fixed size. Since the program is rarely of precisely the same size as the available program memory, unrolling could be performed on the most frequently executed loops in order to fill up the extra space.

5 Bit Transitions

The reduction of bit transitions on the instruction bus is another proposed technique to reduce energy consumption in instruction fetch. This approach is motivated by the fact that the power consumption of CMOS circuits is dominated by switching costs [2]. Reordering operations within a fetch packet and renaming registers can both reduce the number of bit transitions between packets [6, 9, 12]. These techniques have the added benefit of potentially reducing energy consumption without adversely effecting running time.

Unfortunately, optimizations targeted at bit transitions have little effect on the overall energy consumption of the C6200 [12]. These techniques

Program	Trans. per cycle	Total Energy
max	113.16	2.080 mWs
min	1.72	1.891 mWs

Table 1: Energy effects of bit transitions

have two problems that prevent significant results. First, a massive reduction in the number of bit transitions is necessary to have a significant energy effect on the program. Second, renaming registers is hindered by architectural limitations that prevent a large reduction in the number of bit transitions.

Table 1 shows the energy effects of two programs with differing amounts of bit transitions. This is a contrived example with two programs that are identical besides the number of transitions on the instruction bus. A reduction of over 100 bit transitions per cycle is necessary to reduce energy consumption by nine percent. Bit transition variations of this magnitude are not realistic for programs on the C6200. Our attempt at renaming registers to reduce bit transitions rarely eliminated more than one bit transition per cycle. In addition, typical benchmarks for the C6200 have less than 30 bit transitions per cycle.

6 Other Processors

This paper's purpose is not solely to present strategies that reduce energy consumption on a single processor. Instead, the goal is to demonstrate an approach that examines an architecture and discovers methods in which the compiler can assist in saving energy. However, it is still of interest to note other architectures to which the specific techniques mentioned in this paper might apply.

The C6200 series is part of the larger C6000 set of DSPs produced by Texas Instruments. In addition to the C62x there is also a C67x which has the same properties as the C6200 with the addition of floating point capabilities. The techniques mentioned above should work equally well on programs for the C6700.

The C64x is a recent addition to the C6000 family that has several additional features. The C6400 series has double the number of registers and lacks several of the limitations of the C62x and the C67x. The feature of primary importance to this paper is the ability of execute packets to cross fetch packet boundaries. This means that codes on the C64x will not have any NOPs due to packing and nullifies the importance of our first technique (Section 3). However, the issues with energy consumption in loops remain (Section 4). The larger register set may increase opportunities for bit-transition reduction (Section 5).

The Intel Itanium architecture is another VLIW processor with features similar to the C6200. The Itanium fetches instructions in three operation bundles which are the analog of the C6200's fetch packets [1]. Instead of execute packets, the Itanium uses instruction groups which are an unbounded number of instructions that can safely be executed in parallel. This allows for easy expansion of the architecture, though current versions of the Itanium can only execute six operations in parallel. Since instruction groups can consist of multiple bundles no NOPs are necessary for instruction packing on the Itanium, and eliminating NOPs is obviously unnecessary. Unrolling loops could still be beneficial when there is a low degree of parallelism since the Itanium must begin fetching two bundles for each cycle when they may not all be necessary. However, it is important to remember that the significance of energy consumption during instruction fetch could be much less on the Itanium, a general purpose processor, than on the C6200 DSP.

7 Conclusion

Traditionally, when optimizing a program there have been two important considerations: execution time and code size. Recently, energy consumption has also become an issue for compilers. However, unlike traditional considerations, it is unclear how a compiler can assist in reducing the energy consumption of a program. Energy

consumption is obviously a machine dependent property, so energy saving techniques will obviously vary between machines.

This paper examines the energy consumption characteristics of a specific processor – the Texas Instruments' C6200 DSP. After determining that a large portion of the energy consumption on the C6200 is derived from fetching instructions, we examined the instruction fetch mechanism on the C6200 and determined where energy was being expended without benefit. This lead us to develop two techniques for saving energy on the C6200.

First, we determined that the fetch packet mechanism of the C6200 could allow for a large number of NOPs to be fetched in a program. Reconstructing fetch packets to minimize the number of NOPs fetched saves energy even though it can result in a slower program. Second, we saw that loops with a low degree of parallelism consume significantly more energy than straight line code, even when there are no NOPs. This energy waste can be reduced through loop unrolling.

These energy saving techniques are specific to the C6200. Though some similar architectures may profit from these techniques as well, the majority of other processors may not. This is the nature of a problem that is as machine dependent as energy consumption. This paper hopes to demonstrate the process by which energy was saved on the C6200 and how a similar process can be conducted on other processors.

Acknowledgements

This work was supported by the State of Texas through its Advanced Technology Program. Linda Hurd built the power simulation tools that we used. Reid Tatge and Gene Frantz encouraged this work in myriad ways. Tim Harvey and Steve Reeves helped us with informed discussion throughout this work. The members of the scalar compiler group at Rice built infrastructure that we used in this project. To all these people go our heartfelt thanks.

References

[1] Volume Ia Application Architecture. Intel IA-64 architecture software developer's manual, January 2000.

[2] Anatha P. Chandrakasan, Samuel Sheng, and Robert W. Broderson. Low power CMOS digital design. *Journal of Solid State Circuits*, 27(4):473–484, April 1992.

[3] Linda Hurd. *TMS320C6201 Projected Power Dissipation on TI's TImeLine^{TM} Technology.* Texas Instruments, October 1997.

[4] Alexander Klaiber. *The Technology Behind Crusoe^{TM} Processors.* Transmeta Corporation, January 2000. Available online at http://www.transmeta.com/about/white_papers.html.

[5] Mike Tien-Chien Lee, Vivek Tiwari, Sharad Malik, and Masahiro Fujita. Power analysis and low-power scheduling techniques for embedded DSP software. *International Symposium on System Synthesis*, September 1995.

[6] Huzefa Mehta, Robert Michael Owens, Mary Jane Irwin, Rita Chen, and Debashree Ghosh. Techniques for low energy software. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 72–75, August 1997.

[7] James Montanaro *et al.* A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703–1714, November 1996.

[8] Nat Seshan. High VelociTI processing. *IEEE Signal Processing Magazine*, pages 86–101,117, March 1998.

[9] Dongkun Shin and Jihong Kim. An operation rearrangement technique for low-power VLIW instruction fetch. In *Proceedings of the Workshop on Complexity-Effective Design*, June 2000.

[10] Tajana Simunic, Luca Benini, Andrea Acquaviva, Peter Glynn, and Giovanni De Micheli. Dynamic voltage scaling and power management for portable systems. In *Proceedings of the 38th Design Automation Conference*, pages 524–529, June 2001.

[11] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of Design, Automation and Test in Europe*, March 2002.

[12] Todd Waterman. Post-compilation analysis and power reduction. Master's thesis, Rice University, May 2002.