Proc. Estonian Acad. Sci. Eng., 2005, **11**, 4, 347–357

Towards compiler backend optimization for low energy consumption at instruction level

Kimmo Surakka[a], Tommi Mikkonen[a], Hannu-Matti Järvinen[a], Timo Vuorela[b] and Jukka Vanhala[b]

[a] Institute of Software Systems, Tampere University of Technology, Korkeakoulunkatu 1, FIN-33720 Tampere, Finland; {kimmo.surakka, tommi.mikkonen, hannu-matti.jarvinen}@tut.fi
[b] Institute of Electronics, Tampere University of Technology, Korkeakoulunkatu 3, FIN-33720 Tampere, Finland; {timo.vuorela, jukka.vanhala}@tut.fi

Received 3 October 2005, in revised form 3 October 2005

Abstract. Compiler backend is the part of a compiler that is responsible for generating compiled code. By optimizing the backend, one can easily create a tool chain for a new environment, where some restrictions are to be taken into account. One such restriction is energy consumption, which can be affected by code generation. In this paper, we discuss the different candidate optimizations that we have identified, and whether or not they can be implemented within the scope of the compiler backend.

Key words: energy consumption, compilers.

1. INTRODUCTION

Code generation for a certain hardware platform has several phases. First, a compiler reads and analyses the source code, producing an intermediate representation of it. The part of a compiler, executing this phase, is commonly called compiler frontend. The internal representation is then analysed and a number of optimizations is applied to it. This part of the compiler is sometimes referred to as the middle end. Then, the compiler generates a low-level code that is specific to the hardware platform. The part of a compiler that does this is referred to as backend. Finally, the generated code is linked and loaded to the hardware using different tools.

As a result of the above scheme, once a new programming language is used or some extensions are included in an already existing language, one has to write a compiler frontend that reflects the particularities of the language, but after that already existing backends can be used for code generation. For example, GCC [1] offers frontends for C, C++, Objective-C, Fortran, Java, and Ada, together with a remark that further frontends are available. In contrast, when a new piece of hardware is introduced, one can rely on already existing frontends, but must implement a backend, specific to this particular hardware platform. In addition, before a completed program can be run, a linker and a loader are needed to install the program to the target device. These may be vendor specific, as certain types of special actions may be needed, or, alternatively, standardized tools. In any case, as long as the output of the backend is at the level of assembly instructions, the same tool chain can be used.

As backends are always specific to a certain hardware platform, they are the natural location for hardware-dependent optimizations when instruction level optimizations are aimed at. The motivation is that, as long as we do not need to modify compiler frontend or associated tools that are needed for installing compiled software to a hardware platform, the results of optimization are reusable, disregarding programming languages and auxiliaries needed for platform-specific installation.

In this paper, we study the types of optimizations that are applicable in compiler backend when aiming at low energy consumption at instruction level. The way we have carried out the study is the following. First, we measured energy consumption of some real-life executions in our target platform. Based on the measured values, we then went on to create candidate optimizations that could be implemented in order to conserve energy. Provided with the candidates, we then analysed them to see if they could be handled at compiler backend.

The paper is organized as follows. Section 2 introduces the environment, in which our case study has been conducted, including the target hardware and the compiler we used as the reference. Section 3 then goes on to introduce the measurements, we conducted in order to find candidates for energy-aware optimization. This part of the paper has already been discussed in [2]. Section 4 forms the core of this paper by introducing the candidates as well as the analysis on their applicability at the compiler backend level. Section 5 discusses related work, and Section 6 finally concludes the paper.

2. ENVIRONMENT

The measurements were made on an AT90S8515-based custom-built measurement board. The AT90S8515 microcontroller is a 8-bit RISC-type microcontroller, manufactured by Atmel Corporation [3]. It was selected as the target platform because of its relative simplicity, the availability of an existing compiler backend, and because the research team had previous experience with the same microcontroller.

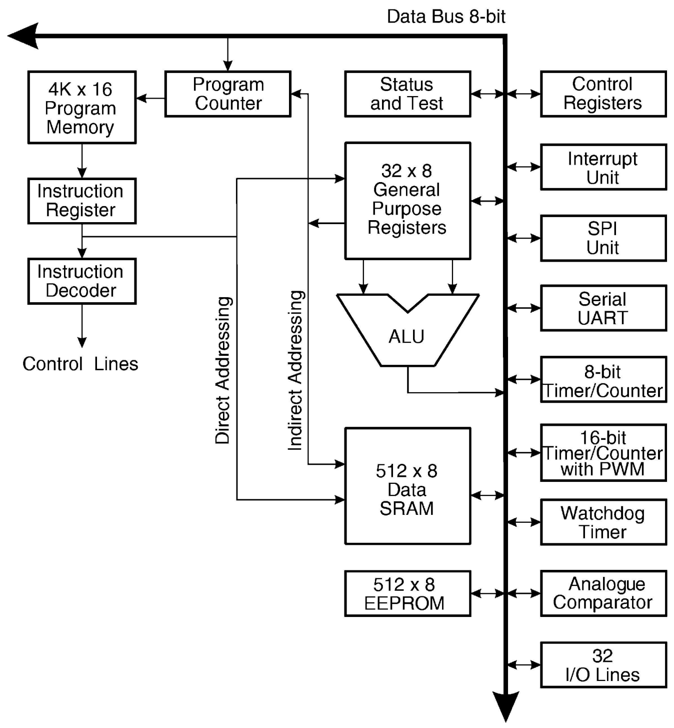

Fig. 1. The AT90S8515 architecture [3].

AT90S8515 is based on the AVR architecture (Fig. 1). It has 32 8-bit general-purpose registers, 8 KB of in-system programmable FLASH memory, 512 bytes of EEPROM memory, and 512 bytes of data SRAM. Six of the 32 registers (r26,...,r31) also serve as three 16-bit registers that can be used as indirect data access pointers (registers X, Y and Z). The processor follows the Harvard architecture, i.e. it has separate memories and buses for program and data. Programs are executed with a two-stage pipeline: while one instruction is being executed, the following one is being pre-fetched from the program memory.

The AT90S8515 instruction set consists of 118 instructions [4]. Most of the instructions have 16-bit opcodes that contain the instruction parameters embedded within the opcode. The only exceptions are the branch commands "`call`" and "`jmp`", and the SRAM access commands "`lds`" and "`sts`". The parameters of these four instructions are too large to fit inside a 16-bit opcode, so the instructions are encoded with 32-bit opcodes. The execution time of an instruction varies from one (most arithmetic-logical instructions) to four (`reti`, return from interrupt) clock cycles.

An example of the opcode encoding is the instruction "`add Rd, Rr`" which calculates the sum of two registers and stores the result in the first register. The encoding is presented in Fig. 2. The bits marked with `d` are used to specify the destination register number while the bits marked with `r` specify the other register.

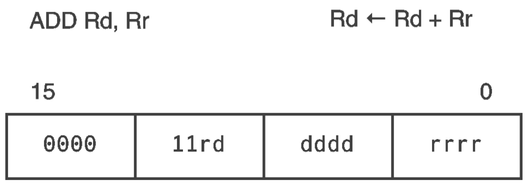

Fig. 2. The 16-bit opcode for the ADD command.

The test programs were compiled with GNU C compiler of WinAVR development environment. This compiler was selected because its sources are freely available, and therefore the implementation of possible backend optimizations is more feasible than with a closed-source tool. Furthermore, this would also support the idea of reuse, discussed above.

3. MEASUREMENTS

Two types of measurements were made. First, energy consumption of various test programs was measured at the level of instructions. The data from these measurements was then inspected and used as a basis for a series of hypothesis, which were then verified by measuring the mean current over the controller while it was running small test programs.

The instruction-level energy consumption was measured with a high-speed digital oscilloscope that continuously measured the voltage drop over a set of resistors, which were connected serially to the microcontroller and the total voltage over both the microcontroller and the resistors. From these figures it was possible to calculate the current and the voltage over the microcontroller, and thus the total energy consumption in the controller. To synchronize this calculated data with the actual instructions that the controller was executing, two more signals were monitored: the microcontroller clock signal and an I/O pin in the controller. The test programs were written so that they ran in a loop, whose body consisted of first signalling the monitored I/O pin, then running the instructions to be measured. Figure 3 shows the measurement arrangement.

The data, collected by the oscilloscope, was transferred to a desktop PC via a serial cable. There it was loaded into a spreadsheet program that calculated the energy consumption at different points of time. To minimize the effect of noise, all test programs were measured twenty times, and the spreadsheet calculations were performed on the average values of these measurements.

From the first results it became evident that the energy consumption of consecutive instructions varied even when the instruction codes were exactly identical. Therefore it was assumed that the energy consumption of the microcontroller at a given time is not a simple function of the current instruction code, but rather a more complicated function, involving multiple variables. These variables included the data being processed, the instruction address in the memory and even

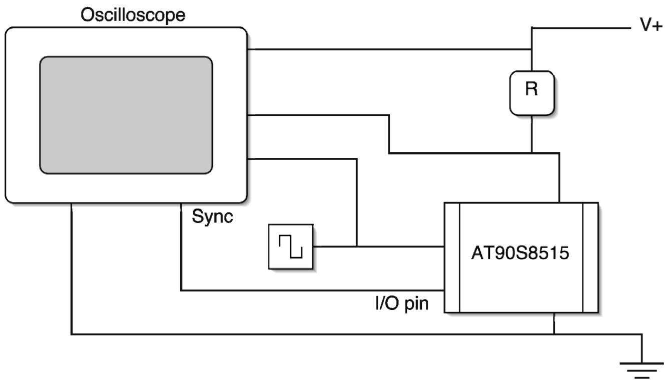

Fig. 3. The measurement arrangement.

the instruction code of the previously executed instruction, whose results were still being written back to registers or memory. Thus, instead of trying to create a simple mapping from instruction codes to their energy consumption, we started experimenting with the effects of the aforementioned variables. By varying the memory location of the test programs, the data values that were processed and the instructions that were executed immediately before the instructions were measured, we were able to find candidates for energy-aware optimizations.

4. CANDIDATE OPTIMIZATIONS

In the following, we discuss the different alternatives that we considered applicable for energy conservation.

4.1. Favouring set bits in the code

The instruction-level energy consumption measurements suggested that an increase in the number of set bits in the instruction code decreases the energy consumption of the instruction. A possible explanation for this is a decrease in the static energy consumption (i.e. the leakage current) in the processor logic circuitry. This correlation was preliminarily verified by measuring the overall current of two test programs; one that used registers with few set bits, and another that was otherwise identical to the first one, but used registers with many bits set. The measurements showed that the latter program had a smaller current consumption. Thus, it would seem beneficial to favor instructions with many set bits over those that have less bits set. To accomplish this, we examined several different alternatives.

4.1.1. *Selecting the right registers*

When an instruction accesses a register, the number of the register to access is encoded into the instruction code. For example, the instruction code for the “move” instruction “`mov rd,rs`” in AVR architecture is the binary number

`001011sdddddssss`, where the bits marked with `d` tell the number of the destination register, and the bits marked with `s` tell the source register. This can be used to increase the number of set bits in program instructions by using primarily those register numbers that have a high number of bits set, for example, preferring the register r31 over r16.

However, a compiler backend has a very limited freedom in selecting which register numbers to use. For example, the hardware we used only supports register indirect addressing with the 16-bit registers X, Y and Z (i.e., the 8-bit registers r26,...,r31); no other registers can be used. Also, many instructions can only use the registers between r16 and r31, and the compiler parameter passing model limits the available user registers even more.

Still, it is possible to optimize the order in which the GCC backend allocates the available user registers. After measuring the mean energy consumption with program snippets that used different registers, we made an initial attempt on optimizing the register allocation order. We modified the GCC backend so that it tries to allocate first those registers that consume the least amount of energy. Next, a test program was compiled with the modified compiler and loaded into the target processor. When executed, the program consumed 0.5% less energy than the same program when compiled with the original compiler. Although this difference in energy consumption was relatively small, it was clearly observable.

Furthermore, it is likely that the allocation order we used was not the optimal one, so there is a distinct possibility of achieving larger savings. The effects of different register allocation orders need to be evaluated in further experiments.

4.1.2. *Using displacement with tables*

Another way to increase the number of set bits in a program involves table lookups. In the AT90S8515, there are three addressing modes suitable for table access. If the data to access is at a fixed index in the table (for example, in C code snippet "`int t=table[2];`"), the compiler can use *data direct addressing*, where the location of the data is embedded into the instruction code. If, however, the table element to access is not fixed (e.g. "`p = table[i];`"), the data location cannot be embedded into the instruction code. In this case, the compiler needs to use *data indirect addressing*. In data indirect addressing, the actual location of the data is stored in a register, and the instruction code contains only information about which register to use. The basic form of data indirect addressing is thus "use the data in the location in register r". However, another form is also available: the *data indirect addressing with displacement*. In this addressing mode, the instruction code contains two constants: the register number to use and a small displacement to add into the register content. Thus, this form is "use the data in the memory location that you get by adding d to the value in the register r".

Indirect addressing with displacement can be used to increase the number of set bits in executed instructions. If the table to be used starts at location loc and the table element to be accessed is nth in the table, the straightforward way to access the element is to calculate $loc + (n - 1) * elem\_size$, store the result in some register and then use register indirect addressing to access the data. However, using indirect addressing with displacement makes it possible to store the value of $loc + (n-1) * elem\_size + disp$ to the register and then use $[reg - disp]$ as the location of the data. If the displacement value is selected to be 63 (the maximum possible), the number of set bits in a "`load`" instruction can be six more than with no displacement. From a total of 16 bits per instruction, this is more than 35 percent increase in the number of set bits. In the case of "`ld r31, Y`" instruction, the number of set bits increases from six to twelve – a total of 100% increase.

The effect of this technique to the total current was measured by constructing a program that calculated the sum of two hundred elements in a table. The first version loaded the base address of the table to the `Z` register and then used "`ld r29, z+`" in a loop to go through the table. This command uses indirect addressing with post-increment; after accessing the data, it automatically increments the value of the `Z` register. The program ran in an endless loop, counting the sum of the values over and over. The microcontroller took 7.12 mA current.

The program was then modified so that instead of loading the base address of the table to the `Z` register, the value $base - 63$ was loaded. The "`ld r29, z+`" instruction was also replaced with the instruction pair "`ldd r29, z+63`", "`inc r30`". Since the table to access was entirely within a 256-byte segment, there was no need to modify `r31` (the high byte of the `Z` register). This program was then loaded to the test board and the mean current was measured. The current was now 6.98 mA.

Thus, by replacing "`ld r29, z+`" with "`ldd r29, z+63`" and "`inc r30`", the mean current taken by the test board was cut down by two percent. Unfortunately, this decrease in power consumption was more than compensated by the increase in clock cycles to go through the table – the "`inc r30`" instruction added one clock cycle to the inner loop in the test program, thus increasing the total execution time by roughly 15 percent. So, even though indirect addressing with replacement may decrease the current through the controller, it may often need supplementary instructions that result in a net increase in the total energy consumption. Still, when such supplementary instructions are not needed (e.g. when the auto-incrementing form of `ld` is not used), the compiler backend should consider using indirect addressing with replacement instead of ordinary indirect addressing.

4.1.3. *Choosing the right instruction codes*

When there are more than one instruction codes that accomplish the same task, the compiler should select the instruction code with the highest number of set bits. An example of this is clearing the register `r15`: it can be achieved with instructions "`clr r15`", "`and r15,r0`", "`mov r15, r0`", or "`sub r15,r15`". The instruction code that has the largest number of bits set should be used – in this case `clr` (Table 1).

A similar analysis should be done for all the operations that the compiler backend implements, and the results should then be used in the code generation.

Table 1. Clearing the register r15

Instruction	Instruction format	Instruction code	Bits set
clr r15 (eor r15,15)	0010 01sd dddd ssss	0010 0100 1111 1111	10
and r15,r0	0010 00sd dddd ssss	0010 0000 1111 0000	5
mov r15,r0	0010 11sd dddd ssss	0010 1100 1111 0000	7
sub r15,r15	0001 10sd dddd ssss	0001 1000 1111 1111	10

It should be noted that the best instruction to use depends on the instruction parameters – the cheapest way to clear register `r0` is to use the `mov` instruction.

4.2. Favouring cleared bits in the data

The instruction-level measurements also showed a decrease in the energy consumption when the instructions were accessing data that consisted mainly of zeros. Thus, it would be beneficial to prefer small data values over large ones. This is, however, something that the compiler backend cannot influence. Instead, suitable data transformations in the middle-end of the compiler should be researched.

4.3. Aligning loops within page boundaries

The measurements also showed a peak in the energy consumption at regular intervals. This is probably due to the internal organization of the microcontroller flash memory. It was observed that every 16th instruction was clearly more expensive than the average. Thus it was assumed that the flash memory was divided into pages of 32 bytes each, and that the activation of a new page lead to an increase in energy consumption. A small test program was written to test this assumption. It consisted of a small endless loop that was located in the memory so that it fitted completely within a 32-bit page. Another version of the same program was also constructed that was otherwise similar to the first one, but had the loop located so that it went over a page boundary. The mean current was then measured for both programs. The first one had a clearly lower mean current consumption, which supported the assumption. Thus, it would seem beneficial to align loops in a program so that they cross as few page boundaries as possible. However, since the actual memory addresses are assigned by the linker, the compiler backend cannot do this alignment without some help from the linker.

4.4. Choosing the right optimizations

The previous work on different architectures [5,6] has shown that many existing performance optimizations are such that applying them to a program also leads to a reduction in the program energy consumption. This is mainly due to the reduced execution time of the program. It may even be possible that all known optimizations that reduce the program energy consumption, also reduce the execution time; in other words, all energy optimizations may be performance optimizations.

However, it is possible that the reverse does not hold: all performance optimizations are not necessarily good from the energy consumption point of view. On the contrary, it is likely that some performance optimizations increase the processor power consumption more than they reduce the execution time, leading to an increase in the total energy consumption. Therefore, when crafting an energy-aware compiler, care should be taken to enable only those performance optimizations that also decrease the energy consumption. As the result, the effect of different existing optimizations on the energy consumption need to be measured.

4.5. Minimizing the switching activity

An important factor in the processor energy consumption is the switching activity in various parts of the processor. By minimizing the switching activity, it is possible to reduce the amount of consumed energy.

The compiler backend can reduce switching activity in the processor instruction bus in two ways: by instruction selection and scheduling. Reduction by instruction selection is possible when a task, such as clearing a register, can be achieved by many alternative instructions; the compiler can then choose the instruction that is closest to the instructions immediately preceding and following it (measured in the Hamming distance). The compiler can also schedule the instructions so that the total switching activity in the code is minimized. An example of such scheduling is presented in [7].

However, these optimizations are only possible if the compiler backend can obtain the actual instruction codes for different instructions. In GCC, this knowledge is contained in the separate assembler that creates the actual machine code. Thus, in order to implement any kind of switching activity optimizations, the backend must be augmented with assembler-like capabilities. This is currently outside the scope of our research. Still, optimization of the switching activity is a promising target for future work.

5. RELATED WORK

The energy impacts of various compiler optimizations have been studied in [6,8,9]. These studies have, however, used a simulator-based approach to calculate energy consumption. Therefore, the correctness of their results depends on the correctness of the used simulator and its energy model.

In [5], the effects of compiler optimizations on the processor energy consumption are measured using an actual microprocessor. The Pentium 4 processor they use is significantly more complex that the one in our work, thereby hiding many energy effects. By focusing on a simpler microcontroller, we hope to get a better insight into the functioning of the processor.

A similar method for measuring the power, drawn by a microprocessor, was described by Tiwari et al. [^{10}]. However, their measurements were made only on the mean current over the processor, while we have developed a method to get instruction-level measurements.

6. CONCLUSIONS

In this paper, we introduced a case study on using compiler backend optimization as the means for obtaining instruction-level energy awareness. We were able to find a number of candidate optimizations that can be used to reduce energy consumption by studying the behaviour of the associated hardware platform using different reference loads. However, many of the optimizations have turned out to be such that they cannot be applied at backend level as such. Instead, they require a redefinition of the scope of the work so that some parts of the compiler frontend, linker and loader would be modified as well. This, however, was not considered an option in our work due to possible problems with compatibility with off-the-shelf tools later on.

In addition to finding candidate optimizations using hardware profiling, we intend to measure the effect of different performance optimizations, offered by the used compiler to energy consumption. Intuitively, it seems rational that when performance is improved, the processor performs less computations, which in turn results in energy savings. However, validating this assumption remains a topic of future study.

ACKNOWLEDGEMENT

This work has been supported by the Academy of Finland.

REFERENCES

1. Gnu Compiler Collection. http://gcc.gnu.org/
2. Vuorela, T., Vanhala, J., Surakka, K. and Järvinen, H.-M. Measuring and optimizing the instruction level power consumption of a simple microcontroller. Manuscript. Tampere University of Technology, Tampere, 2004.
3. Atmel Corporation. The AT90S8515 datasheet. Rev. 0841G-09/01. September, 2001.
4. Atmel Corporation. 8-bit AVR Instruction Set. Rev. 0856D-AVR-08/02. August, 2002.
5. Seng, J. S. and Tullsen, D. M. The effect of compiler optimizations on Pentium 4 power consumption. In *Proc. Seventh Workshop on Interaction between Compilers and Computer Architectures (INTERACT'03)*. Anaheim, 2003, 51–56.
6. Chakrapani, L. N., Korkmaz, P., Mooney III, V. J., Palem, K. V., Puttaswamy, K. and Wong, W. F. The emerging crisis in embedded processors: what can a poor compiler do? In *Proc. International Conference on Compiler, Architecture, and Synthesis of Embedded Systems (CASES'01)*. Atlanta, 2001, 176–181.

7. Lee, C., Lee, J. K. and Hwang, T. Compiler optimization on VLIW instruction scheduling for low power. *ACM Trans. Des. Autom. Electron. Syst.*, 2003, **8**, 252–268.
8. Valluri, M. and John, L. Is compiling for performance = compiling for power? In *5th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*. Monterrey, Mexico, 2001.
9. Kandemir, M., Vijaykrishnan, N., Irwin, M. J. and Wu Ye. Influence of compiler optimizations on system power. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2001, **9**, 801–804.
10. Tiwari, V., Malik, S. and Wolfe, A. Power analysis of embedded software: a first step towards software power minimization. In *Proc. 1994 IEEE/ACM International Conference on Computer-aided Design*. San Jose, California, 1994, 384–390.

Genereeritava masinakoodi käsutasemel optimeerimisest energiatarbe suhtes

Kimmo Surakka, Tommi Mikkonen, Hannu-Matti Järvinen, Timo Vuorela ja Jukka Vanhala

On käsitletud kompilaatori poolt genereeritava masinakoodi käsutasemel optimeerimist käsu energiatarbe kriteeriumi mõttes. Genereeritava koodi energiatarbe kriteeriumile vastavaid võimalikke optimeerivaid teisendusi on analüüsitud nii teisenduste sobivuse kui ka koodigeneraatoris realiseeritavuse seisukohalt.