Revision of Manuscript 1934
Analysis of the Influence of Register File Size on Energy Consumption, Code Size and Execution Time

L. Wehmeyer, M.K. Jain, S. Steinke, P. Marwedel, M. Balakrishnan

Abstract— **Interest in low power embedded systems has increased considerably in the past few years. To produce low power code and to allow an estimation of power consumption of software running on embedded systems, a power model was developed based on physical measurement using an evaluation board and integrated into a compiler and profiler. The compiler uses the power information to choose instruction sequences consuming less power, whereas the profiler gives information about the total power consumed during execution of the generated program.**

The used compiler is parameterized such that e.g. the register file size may be changed. The resulting code is evaluated with respect to code size, performance and power consumption for different register file sizes. The extracted information is especially useful during application analysis and architecture space exploration in ASIP design. Our analysis gives the designer the ability to estimate the desirable register file size for an ASIP design.

The size of the register file should be considered as a design parameter since it has a strong impact on the energy consumption of embedded systems.

Keywords— **low power, compiler, power model, application analysis, register file size**

I. Introduction

Recently, energy consumption in embedded systems has become more and more important, mainly due to the fact that many systems are now being designed as mobile devices, i.e. they have to operate on battery power instead of using abundant power from wall sockets. One important aspect of such devices is their running time. How much energy saving is possible by modifying software alone was recently demonstrated by Siemens [1]: the standby time of one of their mobile phones was increased from 100 to 160 hours without changes in the hardware. Using our energy aware compiler framework *encc* we are able to generate low power code, perform specific optimizations aimed at energy saving and evaluate the resulting program's energy dissipation. The basis for the incorporated power model was given by physical measurement using an evaluation board.

The fact that the *encc* compiler is parameterizable enables further studies: By changing the configuration of the proposed target processor within the compiler configuration file, it is possible to evaluate the effect of architectural changes on code quality. This is particularly interesting for architectural design exploration targeting ASIPs which have a limited number of variable parameters. ASIC development, despite resulting in highly efficient designs, is often too expensive whereas general purpose processors do not provide the required performance. ASIPs are often targeted today, since they offer high flexibility and performance at a relatively low cost. Usually, ASIPs are designed for a small set of applications which enables the designer to exploit knowledge about application characteristics so as to meet the performance, cost and power requirements. Using the *encc* compiler in the design process, we show how to explore the design space provided by the register file size in ASIP design.

The rest of this work is organized as follows: Chapter II gives an overview of previous work concerning low power compiler optimizations, energy models and ASIP design methodologies. Chapter III introduces the *encc* compiler framework along with the integrated instruction level power model. Chapter IV gives a short overview of the ARM7TDMI processor that was used for our experiments. The following chapter shows the used application benchmarks, whereas chapter VI explains the experimental setup and the obtained results. Finally, the last chapter gives a summary and provides some areas for future work.

II. Related Work

Compiler optimizations like power aware instruction selection, i.e. choosing the instruction sequence that will cause the least energy dissipation when the program is executed, will not require changes in the hardware to reduce energy consumption. Often, optimizing for performance will result in a program that also yields good energy results. There are cases, however, where these two goals will result in different instruction sequences. The compiler optimization technique 'Register Pipelining' is an example for this effect. Usually aimed at performance as well as energy optimization, this optimization tries to reduce the number of power hungry off-chip memory accesses within a loop by holding array values that are needed in later loop iterations within the processor registers. This leads to additional instructions being inserted into the code, thus prolonging execution time. Since memory accesses consume up to an order of magnitude more power than instructions operating merely on the register file, the resulting program may consume less overall energy despite its longer running time. This particular example can be found in [2].

Another compiler based optimization for power is instruction scheduling. Subsequent instructions are ordered

This work has been supported by a DST-DAAD project and by Agilent Technologies, USA.

Lars Wehmeyer, Stefan Steinke and Peter Marwedel are with the Department of Computer Science XII of the University of Dortmund, Germany. E-mail: {wehmeyer,steinke,marwedel}@ls12.cs.uni-dortmund.de

Manoj Kumar Jain and M. Balakrishnan are with the Department of Computer Science and Engineering, Indian Institute of Technology, Delhi, India. E-mail: {manoj,mbala}@cse.iitd.ernet.in

so as to incur as little change in circuit state as possible. Some techniques, aimed especially at VLIW architectures, are described in [3].

Beside these compiler oriented optimizations, there is a further group of optimizations that combines architectural changes with modifications to the compiler. If e.g. a special bus encoding used in an architecture is not transparent to the programming interface, the compiler may need to take the coding into account to generate correct code. Using a scratch pad memory within the system's memory hierarchy may require compiler support to efficiently take advantage of this feature, since in contrast to caches, it is not transparent to software. Therefore, either the programmer or the compiler has to decide which data or instructions are to be placed within the scratch-pad memory [4], [5].

One architectural parameter that also requires compiler support is the register file size. There have been several approaches to allow compilers to adapt to different target architectures by supplying them with architectural information about the target processor. In the Trimaran compiler [6], the machine description language MDES is used to model the underlying hardware. A different approach was taken by [7], where a compiler can be retargeted to different processors of the DSP domain described in MIMOLA [8], a VHDL-like hardware description language. In this approach it is possible to specify the architecture in the form of an executable specification and at the same time supply the necessary information for the compiler.

One recent publication by Brooks et al. [9] presents a framework enabling power analysis and investigation of the effect of hardware modifications as well as compiler optimizations. Their power analysis is based on parameterizable power models of common structures found in modern microprocessors.

For VLIW architectures, Zalamea et al. [10] provide results concerning cycle time, area and power consumption for register files of different sizes.

A number of approaches to ASIP design where the design space consists of a number of architectural parameters like number and kind of functional units, issue width and the size of caches have been reported [11]. Most of these search the design space for an area-time tradeoff. If power is also evaluated then it is on the basis of circuit power models which are primarily non-application specific [12], [13]. To our knowledge, this is the first work which evaluates energy consumption by changing an ASIP architectural feature i.e. register file size, in an application specific evaluation.

III. The *encc* compiler framework

In this work, the *encc* compiler framework was chosen for code generation and evaluation. *encc* is an energy aware compiler developed for the RISC class of architectures, capable of generating code with reduced energy consumption. *encc* uses a cost function that not only considers the 'traditional' optimization options like code size and execution time, but also the energy consumption. To be able to make decisions during code generation, the compiler requires a database containing information about the power consumption of single instructions. This information can also be used to determine the energy consumption of an instruction sequence by adding the individual contributions of each instruction and accounting for inter-instruction effects (c.f. section III-A).

The input for the *encc* compiler consists of an application program written in the programming language C. Some standard optimizations are performed on this file using the LANCE frontend [14]. The backend of the compiler then generates assembly instructions using the tree pattern matcher *olive* [15], which employs the energy model as a cost function during code selection. Register allocation is performed using the graph coloring based algorithm described in [16]. Some backend optimizations like instruction scheduling, register pipelining, memory layout, jump optimization or peep hole optimizations are also available.

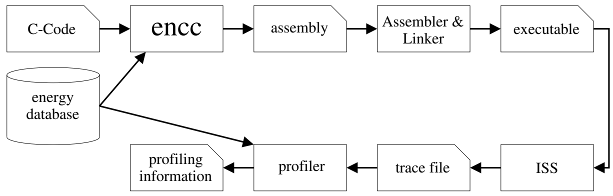

Fig. 1. Workflow within the *encc* framework

The use of *encc* is simplified by a graphical user interface, which gives the user full control over the compilation process. The available optimization options are made up of 'execution time', 'energy' and 'code size'. Supplemental optimizations like instruction scheduling, register pipelining or support for different kinds of memory are also provided. Using the GUI, the user can e.g. generate and execute assembly code and run the profiler to generate a report on execution time, memory accesses and energy consumption.

The profiler that was used to generate the results presented in this work will be presented in greater detail in section III-C.

Several architectural parameters of the targeted processor can easily be changed in the compiler. After recompilation, the compiler is able to generate code for a modified target architecture. Examples for these parameters include the register file size, number of register used for subroutine parameter passing as well as the available addressing modes. This feature of *encc* was used to retarget the compiler from the ARM7 to the LEON processor, which is a SPARC V8 based architecture. The VHDL model of the LEON is available under the GNU public license [17].

For this work, the compiler configuration was initially set up to describe the *ARM7TDMI* processor [18]. The number of registers was then successively changed to generate a compiler for processors differing in the number of registers. The performance of the generated programs was evaluated in a simulation run using the energy database: The output generated by the instruction set simulator is passed to the profiler which calculates the number of accesses to different kinds of memory, the number of bytes occupied in instruction and data memory as well as overall energy consumption. In this way, all the executed instructions along with their individual contribution to energy dissipation are considered.

A. Instruction Level Energy Model

In the literature, power and energy are often used synonymously. “Low Power” and “Energy Aware” have developed into standard phrases. Generally, the important factor to consider in embedded system design is energy, since battery lifetime is crucial.

The power model used in the *encc* compiler is based on the model developed by Tiwari et al. [19], which distinguishes between basic costs and inter-instruction effects. The basic costs consist of the measured current during execution of a single instruction. The additional power consumption due to the change of internal circuit states caused by differing subsequent instructions has to be accounted for. This contribution is summed up in the so-called inter instruction effect.

Our enhancement of Tiwari’s power model consists of the consideration of power consumption within the memory system [20]. This contribution to overall power should not be ignored since the energy consumed by off- chip memory can be significantly higher than the amount dissipated by the processor itself.

For our target processor, the ARM7TDMI, neither a VHDL model that could be simulated to determine power nor an instruction level power model is available. Therefore, the instruction level power values had to determined by experimental physical measurement using an evaluation board.

B. Physical Power Measurement

To perform the measurements, we used the evaluation board AT91EB01 by Atmel [21]. The processor used on this board is an AT91M40400, consisting of an ARM7TDMI core with 4 KB on-chip scratch-pad memory. Additionally, the board offers 512 KB off-chip memory. To perform the power measurements, we used the connections on the board that allow direct monitoring of the processor current. To measure memory power consumption, we had to cut all supply power connections of one of the memory chips and insert an amperemeter into the electric circuit. The evaluation board with all the connections for power measurement in place is shown in figure 2.

The digital multimeter used for the measurements is an “ESCORT 95”. It is characterized by a basic precision of 0.06% and a measuring frequency of 20 measurements per second. Using this multimeter, one instruction at a time was measured and its power consumption determined. In order to do this, the instruction was repeated many times within a loop. We found 100 instruction instances to suffice so as to get a stable reading on the digital multimeter and to be sure we are actually measuring the instruction within the loop, not the loop’s branch instructions. The average processor and memory currents that can be read from the multimeter are measured during execution of this loop, and the result is transferred to the power database

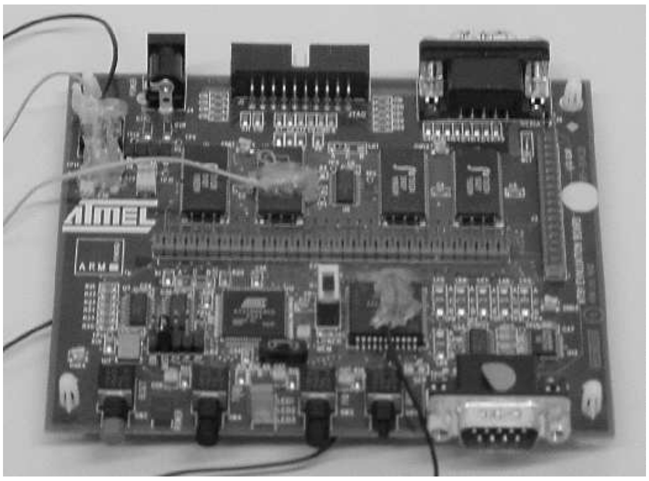

Fig. 2. The evaluation board with measuring connections. Top left: processor current. Top middle: memory current. Bottom: address bus monitoring

used by the compiler and by the profiler. The results of these measurements can be found in [22].

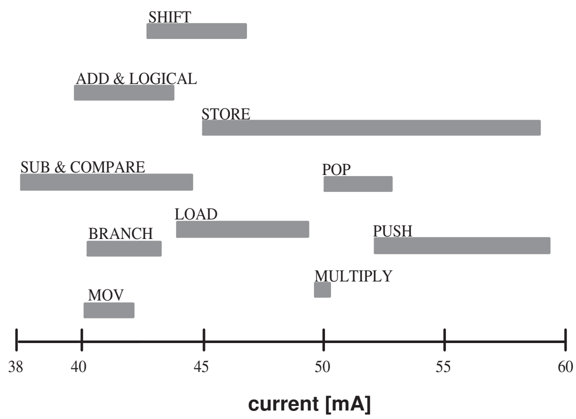

Fig. 3. Measured power consumption of individual instructions

Figure 3 gives an idea of the possible improvements that can be achieved by using the power model in the code selection phase of the compiler. Note that the power required to access external memory is not included in this figure! Even without the memory current, it becomes clear that the instructions that by far consume the most power are memory access instructions. Consequently, there is a high potential for optimizations especially in those instructions involving memory accesses.

Another aspect that has to be considered in the power model is the inter instruction effect that describes the power consumed by changes in the system state when different instructions are executed in sequence. The influence of this effect was also measured using the described methodology using tuples of instructions instead of single instructions. The exact results of these measurements can be found in [22]. The overall average inter instruction effect contributes about 2mA to power consumption, which is below 5% for nearly all instructions. Therefore, the inter instruction effect was taken into account in all experiments by adding these average 2mA to the base costs of the instructions. To show that precision does not suffer from this simplification, we conducted some validation experiments to determine the precision of our power model. Results show that the average deviation of our power model predictions from the measured results was only 1.7% [22]. We thus claim that handling the inter instruction effect in the way described above is appropriate for our purpose. Our power model is sufficiently precise to allow it to be used within a power aware compiler.

C. Integration of Power Model into the encc *compiler*

The values measured as described in the last section were transferred into a power database that is used by the compiler during code selection and within the profiler. The code selector uses the database to choose optimal instruction sequences with respect to power, whereas the profiler evaluates the overall power consumption of a program that is being executed.

The *encc* compiler uses the LANCE-frontend [14] to transform the initial C-code to an intermediate representation in 3-address-format. From this form, the compiler generates data flow trees that can be passed to the code selector, which performs the actual choice of the corresponding assembler instructions. As a code selector, we chose *olive* [15], the successor of the well-known *iburg*. *olive* requires a grammar to specify which assembly instructions cover the nodes in the data flow tree. According to a supplied cost function, it will decide on an optimal cover for each tree and generate the corresponding assembly code structure. Since *olive* allows for arbitrary cost functions, the corresponding values are retrieved from the power database and fed to *olive*'s cost function. The code selector will then guarantee that an optimal cover is generated for each data flow tree.

The power information from the database is not only used during code selection, but also for profiling during simulation runs. Since we are investigating overall energy consumption of benchmark programs in our work, the profiler takes the output of the instruction set simulator, which contains information on executed instructions and memory accesses, and annotates this information with the corresponding power values from the database. The individual values are summed up and a statistical output for the complete program is generated.

IV. The *ARM7TDMI* Processor

The *ARM7TDMI* 32-bit RISC processor [18] was chosen for our work since it is widely used in embedded systems today, and is particularly recommended for ultra low energy applications, making it a suitable target processor for the *encc* compiler. It offers a 32 bit RISC instruction set, 16 general purpose registers with 32 bits each as well as an ALU, a multiplier and a dedicated barrel shifter, but no data or instruction cache (c.f. figure 4).

To reduce energy consumption, the *ARM7TDMI* features an additional 16 bit instruction set called THUMB

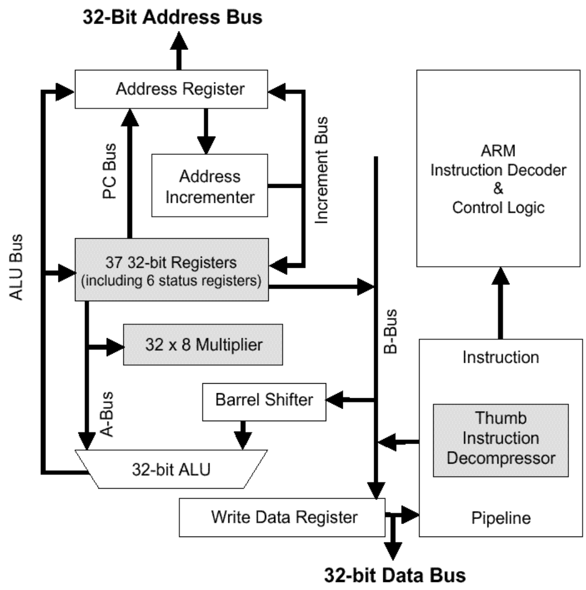

Fig. 4. Block Diagram of the ARM7TTDMI processor

with less functionality compared to the 32 bit instruction set. One major restriction is that only eight of the 16 general purpose registers are available in THUMB mode. Some of the 32-bit instructions using e.g. conditional execution or the multiply-accumulate operation are also not supported. The energy savings compared to the 32-bit instruction set is due to the fact that code density is on average 30% higher in THUMB mode [23] as only 16 bits of instruction data have to be fetched for each instruction instead of 32. Since instruction fetches consume a large amount of energy, this technique generally results in lower energy dissipation during program execution. Since execution times are not significantly longer when the THUMB instruction set is used, it is the first choice for energy aware embedded systems.

V. Benchmark Suite

To investigate the effect of changing the register file size, a number of benchmark applications have to be selected. The areas that are covered by ARM processors include automotive equipment, consumer entertainment, digital imaging, industrial applications, networking, security, storage and wireless applications [24].

The chosen benchmarks were taken from the domains of digital signal processing and multimedia, along with standard sorting algorithms.

The *biquad_N_sections* program, part of the DSP-kernel benchmark suite [25], performs the filtering of input values through N biquad IIR sections. *lattice_init* calculates the output of a lattice filter, whereas *matrix_mult* implements the multiplication of two 2D matrices. *me_ivlin* is a demonstration for a multimedia application, mainly consisting of integer arithmetic operations. The standard sorting algorithms all sort a given array of integers using different methods.

Note that the ARM7TDMI does not feature a floating

TABLE I

Benchmarks used for the experiments

• biquad_N_sections	DSP
• lattice_init	DSP
• matrix-mult	mult. of two m x n matrices
• me_ivlin	media application
• bubble_sort	standard sorting algorithm
• heap_sort	standard sorting algorithm
• insertion_sort	standard sorting algorithm
• selection_sort	standard sorting algorithm

point unit. Since the use of the data types `float` or `double` would result in inefficient library calls, only data of type `integer` is generally used in real-world applications. In order to provide realistic benchmark applications, our chosen programs only consider data of type integer as well.

VI. Experimental Methodology & Results

In this section we first present the methodology used to gather the results presented in this work. Along with the plots for the obtained values, some application specific explanations for the results are given. This analysis allows an experienced programmer to decide on a minimum number of registers by analyzing the C source code of an application.

Using the compiler *encc*, the number of registers available for the application is varied by changing the corresponding value in the compiler configuration file. This modified compiler is then used to compile the benchmarks. The resulting assembly code is analyzed and executed using the instruction set simulator for the ARM7TDMI. The chosen register file size is limited to the range from three to eight. Three is the minimum number of registers necessary to execute our benchmarks in THUMB instruction set mode, and eight is the number of registers in the 'real' ARM processor. The instruction set simulator does not provide a means of reconfiguring register file size, hence simulation is limited to applications using a maximum of eight registers. This limitation is not a great drawback, however, since most of the used benchmarks do not benefit significantly from more than eight registers. For a static analysis of the generated assembly code not requiring simulation, up to 20 registers were considered.

It should be mentioned that we have not taken into account some of the potential positive effects that arise from using less registers. A smaller register file size will cause less chip area to be used and will also bring down power consumption in the circuitry itself. Another effect that was not considered is the fact that using less registers generally results in shorter instruction words, since the operands in registers can be addressed with fewer bits.

'Live values' are values that will be read at a later point in time during program execution (c.f. [26]). In order to execute a program that requires more live values than available registers, additional memory is needed to hold the live values. Such values are usually stored in external memory and retrieved just before they are needed. (Another solution for this problem is data-regeneration, in which values are not always stored to memory but may also be recalculated [27]). If the register file size is very small, the compiler thus needs to insert code to store values to memory once they have been computed and to retrieve them from memory when they are needed. This extra code is called *spill code*, as the values are 'spilled' to memory. It is clear that this will increase the size of the generated program. During execution of the program, the spill code will require time to execute, thus prolonging execution time. Finally, since (off-chip) memory accesses are usually very power-hungry, one can expect the power consumption for a program to go up significantly if a lot of spill code is generated. Consequently, a strong decrease in application performance is to be expected when the register file size is too small for an application.

As mentioned in chapter III-B, the evaluation board used for the power measurements offers a 4 KB on-chip scratch-pad memory and a 512 KB off-chip memory (c.f. figure 5). Scratch-pad memory is like an on-chip cache without any tag memory or cache organization hardware, i.e. the management of the scratch-pad memory is left to the programmer. The *encc* compiler supports the use of different memory types in such a way that the developer may specify where instructions and data are to be stored. Since on-chip memory is limited to 4 KB, program code and data do not fit into the scratch-pad memory, so we decided to keep the data off-chip while transferring the instructions to the scratch-pad memory. This makes sense for our observations, since we are interested in the effect of accesses to data in memory and less in the effect of instruction fetching. By partitioning the memory objects in the described way, instruction fetches account only for a small additional power consumption and execution time, whereas data transfers which are also required for storing or retrieving spilled variables show a high contribution to overall power consumption and execution time. The chosen memory layout also coincides with the situation often found in embedded systems, where the program is usually stored in a ROM and only the dynamic data is stored in external RAM.

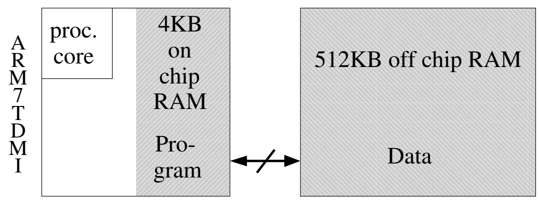

Fig. 5. Memory configuration of the Atmel AT91M40400 evaluation board

A. Results for the Ratio of Spill Instructions to Total Static Code Size

The addition of spill code for decreasing number of registers leads to a larger code size. The spilling instructions are made up of memory load and store operations as well as the necessary address calculations. Since the total number of instructions related to spilling is not very meaningful, the ratio of spill instructions to total code size is given. The obtained results are shown in figure 6.

As expected, the ratio of spill code decreases with increasing number of registers. In our observations, we found that the 'saturation' of the benchmark applications, i.e. the point where no spillcode is present in the assembly code, varies between 7 and 19 registers. This value provides an upper bound for the register file size: if an ASIP designer wants to make sure that his application performance is not impaired by spill code, he will have to determine this 'saturation' value for a particular application. The plot also provides information about the maximum register pressure, i.e. the maximum number of values that are live at the same time.

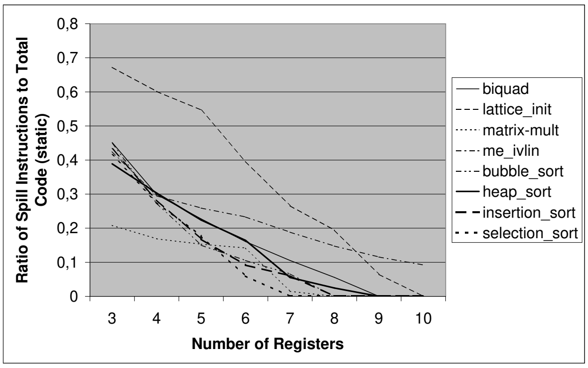

Fig. 6. Ratio of spill instructions to total number of instructions

Having looked at this first static analysis, we found that the results are useful when minimal code size (a static property of the code) is the target. However, the results are of little use for dynamic runtime analysis of applications: Spill code introduced within the innermost loop of an application is executed many times, but counted only once in this observation. To be able to estimate the influence of spill code at runtime, we have to make further experiments and observe the behavior of the applications when they are executed. A dynamic analysis is also necessary to be able to extract any kind of information on the power consumption of the application.

B. Results for the Number of Executed Instructions

For the results presented in this section, the profiler was used to count the number of instructions that were executed by the instruction set simulator. The results are shown in figure 7. The instruction count values for the different benchmark application were scaled in this figure to produce the results in a single plot. This is acceptable since the trends of the curves are preserved.

As expected, the resulting plots now show more meaningful trends since spill code inserted within an inner loop is counted each time the loop is executed.

In some curves, there is one remarkable sharp bend, e.g. in the program *insertion_sort*. When the number of registers is increased from four to five registers, the instruction count decreases substantially, whereas an increase from five to six registers hardly changes the number of executed instructions. To understand the cause for this behavior, we take a look at the innermost loop of the application which is executed more often than any other part of the code and therefore has a strong influence on the dynamic behavior of the program. The innermost loop is shown in the original C source code and in the assembly code generated assuming eight available registers (figures 8 and 9, respectively).

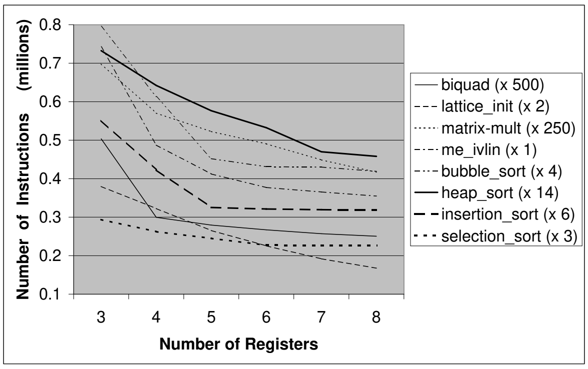

Fig. 7. Number of executed instructions over number of registers

```
/* find the insertion point */
for (indx2 = indx - 1; indx2 > 0;)
{
  int temp_val = This[indx2 - 1];
  if (CMP(temp_val, cur_val) > 0)
  {
    This[indx2--] = temp_val;
  }
  else
    break;
}
```

Fig. 8. Inner loop of *insertion_sort*: C code

```
;;     for (indx2 = indx - 1; indx2 > 0;)
       SUB r3,r6,#1
       CMP r3,#0
       BLE LL8
LL16

;;     int temp_val = This[indx2 - 1];
       SUB r2,r3,#1
       LSL r2,r2,#2
       LDR r7, [r0, r2]

;;     if (CMP(temp_val, cur_val) > 0)

       CMP r7,r4
       . . .

;;     This[indx2--] = temp_val;
       MOV r2,r3
       SUB r3,r3,#1
       LSL r2,r2,#2

       STR r7, [r0, r2]

;;     for (indx2 = indx - 1; indx2 > 0;)
       CMP r3,#0
       BGT LL16
```

Fig. 9. Inner loop of *insertion_sort*: assembly code

It is possible to determine which C variable is stored in which physical register of the ARM7TDMI processor by comparing the two code fragments. For example, `indx2` is stored in register `r3` since it is the result of the first subtraction in C as well as in the assembly code. `indx` is kept in register `r6`, but since this variable is not read again within the loop (it is only present in the initialization of the `for`-loop), it is not live within the loop and can therefore be neglected. Continuing the analysis of the assembly code, we come to the mapping shown in table II. The expression `(indx2-1)*4` is used for addressing the one-dimensional array `This[]` whose elements of type `int` require four bytes each.

This table provides an explanation of the sharp bend in the curve for the `insertion_sort` benchmark: since five registers (`r0`, `r2`, `r3`, `r4`, `r7`) are required to hold the values that are live within the innermost loop, reducing the number of registers to four will result in spilling within the innermost loop, which will inevitably lead to a severe performance degradation of the application. The knees in the other applications' curves can be explained in a similar way, i.e. by analyzing the required number of live values within the innermost loop.

TABLE II
Mapping of variables to registers

Variable	Register
indx2	r3
indx2-1, (indx2-1)*4	r2
This[]	r0
temp_val	r7
cur_val	r4

To summarize, an ASIP designer can decide on the minimum requirements for register file size by using a reconfigurable compiler and looking at the results of dynamic runtime analysis. On the other hand, given he has an understanding of the code generation process for a particular architecture, he can estimate the number of required registers by determining the number of live values within the innermost loop.

C. Results for the Number of Cycles

As the next step, the number of cycles required for execution of the benchmarks was extracted. The results obtained for number of cycles are shown in figure 10. Again, the values are scaled appropriately.

The general behavior of these curves is similar to the results shown in the previous figure concerning instruction count, since cycles and instructions are strongly interdependent. Each instruction requires a certain amount of cycles to execute (the so-called *CPI* value - cycles per instruction). For the ARM7TDMI, the CPI has a value between one and five, not taking into account the effect of pipelining. Store and Load operations require more cycles to complete than arithmetic operations: the latter require only one cycle, whereas a Store and Load to/from off-chip memory takes two and three cycles, respectively. Hence, one additional spill instruction within the application requires several more cycles every time the spill instruction is executed. That is why the sharp bends in the curves remain

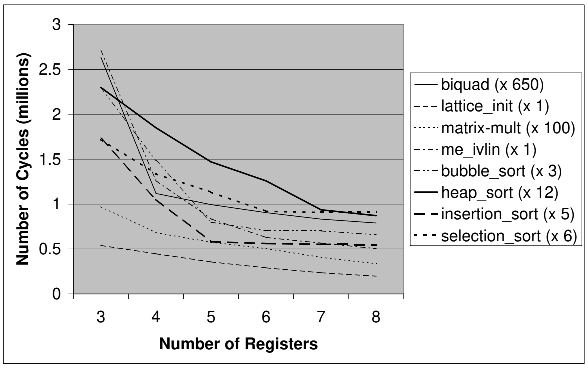

Fig. 10. Number of cycles over number of registers

in the same spots while the slope of the curves changes. Since we are observing the effects of spill code in the program, the same application characteristics (number of live values in the innermost loop) that were discussed in the previous section can be used to explain these observations.

D. Results for Power Consumption

In this section we exploit the fact that the *encc* compiler framework is energy aware: the profiler is capable of analyzing the power contributions of every executed instruction, also taking into account the inter instruction effect. These power contributions are added up to give a value for the complete program's power consumption during execution. The results obtained for the average power consumption in relation to the number of available registers are shown in figure 11.

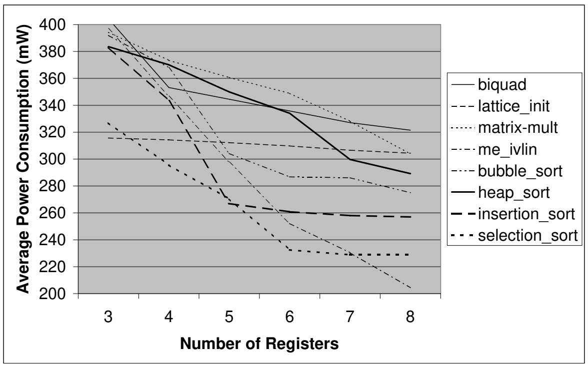

Fig. 11. Average power consumption over number of registers

Data load and store instructions introduced due to spilling increase the power consumption significantly since accesses to off-chip memory are very power consuming in RISC architectures. Due to the used memory layout, the off-chip data access instructions related to spilling consume a large percentage of the overall power of the application.

E. Results for Energy Consumption

Coming to one of the most important aspects of embedded system design today, we now take a look at the energy consumption of our benchmarks in relation to the register file size. The general behavior of the curves in figure 12 is obvious: since cycle time as well as power consumption went down with increasing register file size, energy should show a very strong decrease for growing number of registers.

The strong correlation between register file size and energy can also which show that, using the configuration we chose for our experiments (instructions in scratch-pad memory, data off-chip), additional memory accesses required for spilling consume an order of magnitude more energy than e.g. an ADD instruction. This information is given in table III.

TABLE III
ENERGY CONSUMPTION OF ADD, LDR AND STR INSTRUCTIONS

Instruction	Av. Energy Consumption @ 32.768 MHz
ADD	4.91 nWs
LOAD	49.4 nWs
STORE	48.7 nWs

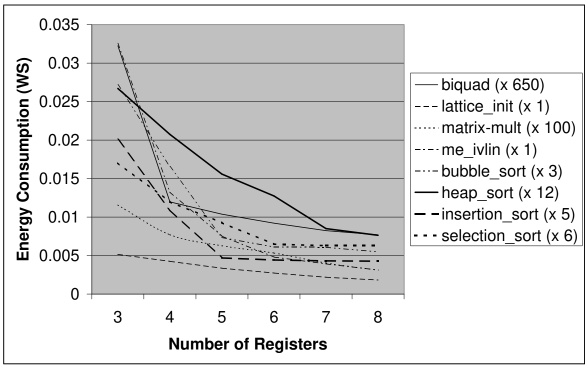

Fig. 12. Energy consumption over number of registers

Since power as well as cycle time contribute to the energy decrease observed here, it is interesting to investigate which of these two factors has the higher contribution to overall energy savings. Figure 13 provides the result of our investigation: the main contribution to energy savings comes from saving cycles, only a small amount of energy is saved due to the use of instructions that consume less power. This shows that the latency of the used memory will also have a strong effect on the achievable energy savings. For low-energy compiler systems we can deduce that optimizing source code for time is generally a good approach towards energy efficient software. The contribution of power to energy savings should not be neglected, however, since often (such as in mobile devices), improvements in power consumption by as little as 5% are regarded as interesting in the industry.

The fact that the number of registers is an important factor during the design process of an ASIP is underlined by the results given in table IV which shows the relative en-

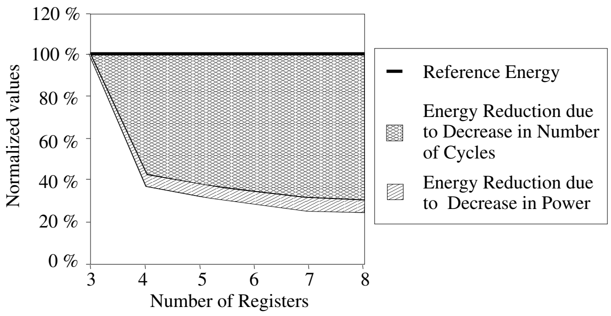

Fig. 13. Contribution of power and time to energy savings

TABLE IV
ENERGY SAVINGS BY CHANGING REGISTER FILE SIZE

benchmark	$3 \rightarrow 4$	$4 \rightarrow 5$	$5 \rightarrow 6$	$6 \rightarrow 7$	$7 \rightarrow 8$
biquad	62.89%	13.35%	11.26%	10.42%	6.71%
lattice_init	17.45%	20.97%	18.92%	19.78%	16.84%
matrix-mult	33.44%	18.57%	14.99%	24.26%	23.28%
me_ivlin	59.28%	43.16%	36.29%	18.24%	20.47%
bubble_sort	39.07%	55.63%	16.78%	0.74%	9.63%
heap_sort	22.35%	24.87%	18.27%	33.24%	10.16%
insertion_sort	45.47%	57.11%	5.60%	2.63%	0.92%
selection_sort	29.68%	22.59%	30.14%	2.94%	0.00%

ergy savings possible by increasing the number of available registers one by one. The average energy savings possible in the range from three to eight registers amount to approximately 22%. Since all curves are starting to level out even in the range we have considered here, it is clear that the positive contribution of additional registers becomes smaller and smaller the more registers are already present in the design. One should bear in mind, however, that each additional register will also consume additional energy due to added capacitances and required chip area. This energy has not been considered in our approach, but will be the subject of future work.

VII. CONCLUSION AND FUTURE WORK

In this work, we describe the methodology used to design and implement an instruction level power model for the *encc* compiler and show its integration into the compiler framework. Physical measurements using an evaluation board were performed to gather data concerning instruction level power consumption. In our experiments, the number of registers for the *ARM7TDMI* processor was changed within the *encc* compiler. This design parameter is of importance in ASIP design since it strongly influences energy consumption in those embedded systems that are sensitive to memory accesses. Several benchmark applications were compiled using the customized compiler. Some application characteristics responsible for the observed results concerning number of spill instructions, number of executed instructions, cycle time required for execution as well as power consumption and energy dissipation were identified. The results obtained are useful during the design of ASIPs, especially at an early stage of design space exploration where the designer can decide how many registers an application will need to work within the required time and/or energy constraints for the final processor. Future work concerning the register file size will consist of taking additional observations for different architectures, including the LEON architecture which in contrast to the ARM7TDMI features both instruction and data cache. Considering the architectural consequences of modifying the register file size as well as changing instruction coding will also be part of our future work. To further broaden the considered design space, we plan to develop a retargetable instruction set simulator which will enable us to make measurements for an arbitrary number of registers instead of being limited to the maximum number the processor provides. In the course of the experiments, we hope to identify other parameters that may have a strong effect on energy consumption of embedded systems. Also, more experiments will have to be performed using a compiler that supports specific features of the targeted ASIP. We plan to work on designing and implementing algorithms for the automatic extraction of application characteristics to help in an early estimation for number of required registers in the ASIP design flow.

References

[1] K. Groebmair. Bestselling C25 mobile phone now performs even better: The "C25 power" features longer standby time and improved sound quality. http://www.siemens.de/ic/products/cd/english/index/news/pressreleases/index.html or http://194.121.239.192/cgi-bin/db4web_c.exe/db/pressedb/icp_user/en/pm_einzeln.d4w?pm_id=759, August 1999.

[2] S. Steinke, R. Schwarz, L. Wehmeyer, and P. Marwedel. Low power code generation for a RISC processor by Register Pipelining. Technical Report 754, University of Dortmund, Dept. of Computer Science XII, 2001.

[3] C. Lee, J. Lee, T. Hwang, and S. Tsai. Compiler Optimization on Instruction Scheduling for Low Power. In *13th International Symposium on System Synthesis*. ACM, Septermber 2000.

[4] P.R. Panda, N.D. Dutt, and A.Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, 1999.

[5] S. Steinke, C. Zobiegala, L. Wehmeyer, and P. Marwedel. Moving Program Objects to Scratch-Pad Memory for Energy Reduction. Technical Report 756, University of Dortmund, Dept. of Computer Science XII, 2001.

[6] Trimaran. Trimaran Homepage. "http://www.trimaran.org/index.html", 1999.

[7] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997.

[8] R. Jöhnk and P. Marwedel. MIMOLA Reference Manual Version 3.45. Technical Report 470, University of Dortmund, Dept. of Computer Science XII, March 1993.

[9] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the ACM ISCA 2000*, June 2000.

[10] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Software and Hardware Techniques to Optimize Register File Utilization in VLIW Architectures. In *Proc. of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Systems (IWACT)*, July 2001.

[11] M.K. Jain, M. Balakrishnan, and A. Kumar. ASIP Design Methodologies : Survey and Issues. In *Proceedings of the Fourteenth International Conference on VLSI Design*, pages 76–81, January 2001.

[12] J. Kin, C. Lee, W.H. Mangione-Smith, and M. Potkonjak. Power Efficient Mediaprocessors: Design Space Exploration. In *Proceedings of the Design Automation Conference (DAC)*, June 1999.

[13] D. Singh, J. Rabaey, M. Pedram, F. Catthoor, S. Rajgopal, N. Sehgal, and T. Mozden. Power conscious CAD Tools and Methodologies: A Perspective. In *Proceedings of the IEEE*, April 1995.

[14] R. Leupers. LANCE V2.0. "http://ls12-www.cs.uni-dortmund.de/~leupers/lanceV2/lanceV2.html", 2000.

[15] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a Simple, Efficient Code Generator Generator. Technical report, AT & T Bell Laboratories, 1992.

[16] A.W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, Cambridge, New York u.a., 1998.

[17] J. Gaisler. Gaisler Research. "http://www.gaisler.com", 2001.

[18] ARM Ltd. ARM Ltd. Homepage. "http://www.arm.com", 2000.

[19] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis Of Embedded Software: A First Step Towards Software Power Minimization. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design ICCAD*, pages 384–390, November 1994.

[20] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model supporting Software Optimizations. In *International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, September 2001.

[21] Atmel. Atmel Corporation Homepage. "http://www.atmel.com", 2000.

[22] M. Theokharidis. Measuring Energy consumption of ARM7TDMI Processor Instructions (in german), available from "http://ls12-www.cs.uni-dortmund.de/publications/theses". Master's thesis, University of Dortmund, Department of Computer Science XII, 2000.

[23] Advanced RISC Machines Ltd (ARM). An Introduction to Thumb. available from: "http://www.arm.com", March 1995.

[24] Advanced RISC Machines Ltd (ARM). ARM Powered[tm] products. "http://www.arm.com/arm/arm_powered?OpenDocument", 2001.

[25] V. Zivojnovic, J. Martinez, C. Schlaeger, and H. Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology, ICSPAT'94, Dallas, Texas*, October 1994.

[26] St. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1. edition, 1997.

[27] C.H. Gebotys. Network Flow Approach to Data Regeneration for Low Energy Embedded System Synthesis. In *Integrated Computer-Aided Engineering, J. Wiley Publishers*, 1998.