

Adaptive Program Execution for Low Power in Superscalar Processors

Diana Marculescu, Steve Haga

Department of Electrical and Computer Engineering

University of Maryland

Phone: (301) 405-6703

E-mail: dianam@eng.umd.edu

October 1999

Abstract - *This paper proposes a novel technique for code optimization for low power based on a profile-driven methodology. Specifically, we show analytically that there is an optimal level of parallelism for energy consumption (which may not be necessarily the same as for performance) and we propose a compiler-assisted technique for code annotation that adaptively selects at run-time the optimal number of instructions to be executed in parallel as far as energy is concerned. As it is shown by the experimental results, significant savings can be obtained in both energy (up to 28%) and power consumption (up to 69%), at the expense of a decrease in performance. The techniques proposed in this paper can be used by an ACPI-compliant power manager either for prolonging system battery life or as passive cooling features in an efficient thermally managed system.*

1 Introduction

Power dissipation has become a critical design concern in recent years, driven by the increased levels of complexity and emergence of mobile applications. Also, the drive towards increasing levels of performance has pushed operating clock frequencies higher, further increasing the level of power consumption. As a consequence, power consumption may even set the limit on the integration level on a chip and how fast it can be clocked. Power has especially become important for designers of high-performance microprocessors, multi-media and digital signal processors, high-speed networking and battery powered or wireless electronics. While it is generally agreed that tools for power estimation and optimization do exist for hardware specifications at different levels (circuit, gate, register-transfer or architectural) [1-3], more work is needed in the area of software power analysis or optimization. This is especially important since, in the case of general purpose or application specific processors, what actually determines their energy consumption is the software that runs on that particular processor. Having tools that are able to quantify the effect of different performance or power optimization schemes for a given piece of code is of extreme importance for computer architects and compiler engineers who can characterize different architecture styles not only in terms of their performance, but also in terms of the corresponding energy efficiency. This will also enable the fine tuning of any existing energy/performance trade-offs. While some software level power optimization techniques have already been proposed, the previously proposed approaches cannot capture mechanisms that are now used virtually in every modern processor such as out-of-order [4,5] or speculative execution [6], superscalar architectures [7] or Very Long Instruction Word (VLIW) architectures [8].

Some areas addressed by relevant previous work are software power analysis for general purpose, embedded or Digital Signal Processing (DSP) applications and instruction scheduling for power optimized software. In the area of power modeling for embedded software, [9] proposes a *per-instruction base power model* that can be easily used to find an aggregate power estimate for a sequence of instructions. This base cost is extracted by measuring the current drawn from the power supply for every type of instruction. The *inter-instruction overhead* due to changes in circuit state are also taken into account, but their effect is found to be negligible. Other effects such as pipeline stalls due to resource constraints or cache misses are also taken into consideration and extracted again by measuring the current drawn from the power supply for each of these scenarios. Two types of processors were taken into consideration: a general purpose CISC processor (Intel 486DX2) and a RISC processor targeted for embedded applications (Fujitsu SPARClite 934). The basic model, enhanced by the inter-instruction effects, remains valid for both processors. One of the main advantages of this approach is the simple power model that can be used as part of any energy-aware compiler or energy-driven code optimization back-end tool. The approach presented in [10] targets instruction scheduling to reduce circuit state overhead. The proposed *cold scheduling* technique reduces the switching activity in the control path by reordering the instructions based on the inter-instruction energy overhead. More recently, for the special case of real-time systems, an approach for reducing energy via scheduling while still meeting the deadlines has been presented [11].

In [12], the case of DSP applications is addressed. There, while the same type of models as in [9] have been developed, the inter-instruction effects turn out to be much more prevalent, thus making it possible to develop instruction scheduling techniques that target power minimization. The model assumes an energy cost for each instruction or pair of instructions, increasing the complexity of the model from linear to quadratic in terms of the instruction set size. Savings of up to 73% in energy consumption can be achieved when a modified version of the *list scheduling* algorithm [13] is used in conjunction with other techniques such as memory bank assignment, instruction packing and operand swapping. However, the reduction in energy consumption from instruction scheduling alone does not exceed 14%. More recently, in [14] a more efficient model with space complexity linear in the instruction set size has been presented. The base power cost and the inter-instruction overhead is extracted using a power simulator on a standard-cell implementation of the Motorola DSP56000 processor. However, both approaches mentioned above do not

consider the effect of data dependency and it is pointed out in [14] that this effect can be very important (up to 28% spread over different data values). More recently, [15] presents an architectural enhancement to reduce the extra work or energy in the pipeline of a superscalar processor due to mispredicted branches. Up to 33% savings in extra work have been reported without significant loss in performance by using confidence estimators for branch prediction schemes.

In this paper we address the problem of energy optimization in *superscalar processors* by using compiler-assisted code annotation for *variable issue* or *execution rate*. In general, superscalar processor may issue and execute several instructions in parallel for the purpose of increasing performance. However, energy, as opposed to performance, is a much more data-dependent parameter. As it will be shown subsequently, it is indeed possible to use less than the maximum number of functional units available, and achieve less energy consumption. We study this effect only for the *execution stage* of a superscalar processor and for a single type of functional unit with multiple instances, but the analysis can be extended to any number of different type functional units. For the first time to our knowledge, our study shows that there exists an inherent trade-off between *performance* and *energy consumption* per instruction, mainly due to the data-dependency effect, but also due to some other side-effects, such as *speculative execution*. This paper improves the state-of-the-art by proposing a novel technique for code optimization for low power based on a *profile-driven methodology*. Specifically, we show analytically that there is an optimal level of parallelism for energy consumption (which may not be necessarily the same as for performance) and we propose a *compiler-assisted* technique for code annotation that adaptively selects at run-time the optimal number of instructions to be executed in parallel as far as energy is concerned. As a measure for energy consumption, we use the *switched capacitance* (which is proportional to the number of transitions on the inputs of the functional unit) [1]. As the experimental results show, significant savings can be obtained in both energy and power consumption, at the expense of a decrease in performance.

The techniques described in this paper can thus be used as a means for prolonging the *battery life* (by decreasing the energy consumption), but most importantly, for *thermal management* [16] by achieving significant average power reductions in the execution stage. Such an approach could be used in the context of power management schemes using the Advanced Configuration and Power Interface (ACPI). Thermal management in ACPI is achieved via average power reduction through “clock throttling” in the case of *passive cooling*, or by turning on the on-chip fan in the case of *active cooling*. However, both techniques are likely to actually decrease the battery life. Our proposed technique reduces not only the power consumption in the execution stage (thus achieving a lower operating temperature), but also the total energy consumed, thus prolonging battery life as a by-product.

The paper is organized as follows: in Section 2 we present the rationale behind profile-driven multiple instruction execution. In Section 3, we present our proposed methodology for code-annotation for low energy instruction execution. Section 4 shows our experimental results for the proposed technique applied on a subset of SpecInt95 benchmarks. We conclude in Section 5 with some final remarks.

2 Profile-Driven Multiple Instruction Execution for Energy Optimization

In a typical superscalar processor, the hardware may issue from one to eight instructions per cycle. Usually, these instructions must be independent and satisfy some constraints. If some instruction is dependent or doesn't meet the constraints, only the instructions preceding it in the sequence are issued, hence the variability in issue rate. If, in addition, out-of-order execution of instructions is permitted, any of the succeeding instructions may be executed if there are no data dependencies present. Let us consider the simple case of an in-order execution of a computation-intensive piece of code. In what follows, only the case of a single type of functional unit is considered. Via a simple example, we show how, by trading-off performance for power, the total energy per operation is actually decreased.

Example 1. Consider the computation of the product of two input streams and the availability of up to 4

16-bit multipliers able to perform the multiplication. We consider 4 possible scenarios corresponding to using one, two, three or all of the functional units. The four scenarios, labeled with the corresponding *total energy per operation*, are depicted in Fig.1. (It is assumed that the input stream x_i includes both operands.)

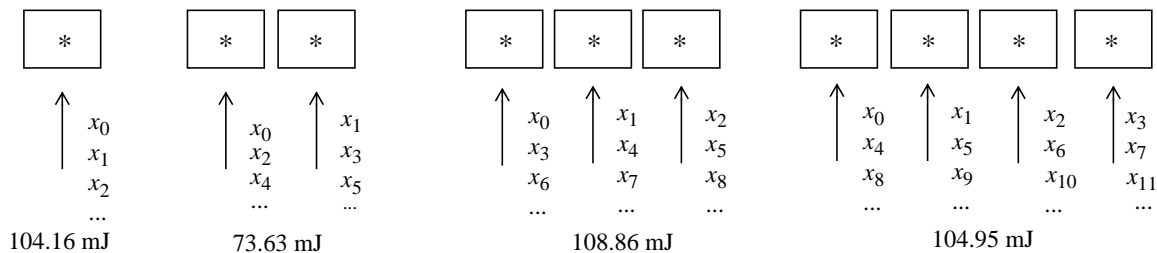


Fig.1 Four possible scenarios for computing a series of multiplications

As it can be seen, the energy consumption per operation varies a lot among the 4 scenarios. This is mainly due to the very different profile of the data that is sent to each of the available functional units. Also, when comparing the performance, we can see that 4 multipliers give us the largest performance (4 computations per clock cycle), but using 2 multipliers reduces the total energy by more than 32% (for 3 units) and 29% (for 4 units). This type of behavior is actually found very frequently in practice, especially in computation intensive applications. These effects become even more important in the case of DSP processors where most of the power consumption is due to the datapath performing additions or multiplications [14]. Another scenario where this effect may become prevalent is the one of a mix of load/store and arithmetic instructions that alternatively use the same functional units for computing effective addresses and output values. Typically, memory addresses look much more “different” than the operands of arithmetic instructions and thus, the above behavior will arise. As it will be shown later, one way to exploit this effect is to *selectively* use the number of instructions to be executed in parallel according to the behavior of the data. Using this approach, one can envision an *adaptive scheme* that can be used in a run-time environment to dynamically trade-off the power consumption for a decrease in performance. Our approach proposes the use of a *variable issue rate* and *partial usage* of the physically available functional units via *profile-driven code-annotation*. This scheme can be used for thermal management of power-aware computing systems: whenever the operating temperature increases over a given limit, the “low-power” annotated code can be activated and, at the expense of a decrease in performance, the operating temperature will be decreased. As a by-product, if in addition the total energy is also decreased, the battery life will also be prolonged.

2.1 The analytic model

In what follows, a simple technique which detects the behavior described above by just analyzing the profile data, is presented. As it is well known, power consumption is a function of the switched capacitance of a module. Also, there is a close relationship between the power consumption of a functional unit and the number of transitions on the primary inputs. Thus, a simple and effective way to characterize the effect of an input sequence on the power consumption of the circuit is via its average Hamming distance. The following result provides a characterization of the average Hamming distance between vectors that are separated by an arbitrary (but fixed) number of time steps. The input stream is assumed to be modelled by a lag-one Markov chain [17].

Proposition 1. If an input sequence $\{x_n\}_{n \geq 0}$ is modeled by a lag-one Markov chain, $p(x)$ is the probability of occurrence of vector x and $p_k(x|y)$ is the conditional probability of making a transition from y to x in exactly k steps, then the *average Hamming k -distance* denoted by:

$$d_k = \sum_{x, y} p(y) \cdot p_k(x|y) \cdot d(x, y) \quad (1)$$

converges when $k \rightarrow \infty$ and the limit is $D = \lim_{k \rightarrow \infty} d_k = \sum_{x, y} p(x) \cdot p(y) \cdot d(x, y)$.

Proof: For an irreducible and aperiodic lag-one Markov chain, the k -step conditional probability $p_k(x|y)$ converges as $k \rightarrow \infty$ to the limiting probability $p(x)$, that is, the state probability of x . Thus, the above result holds. \square

d_k is nothing but the average Hamming distance between input vectors that are exactly k steps apart. Thus, d_k is a measure of the average switching activity or total energy per operation when exactly k units are used to process the input sequence in parallel. Although the above result is valid only for finite-order Markov chains, we should point out that in practice, data traces or memory references do exhibit a finite time interval in which dependencies are present (also known as spatial and temporal locality [4]). Thus, for all practical purposes, the finite order assumption can be safely considered when characterizing data and/or memory traces.

This result can be extended to any finite-order Markov chain and to other measures that may be relevant to the actual power dissipation, such as signal probabilities or number of ones (especially for Booth multipliers) in the input sequence. In addition, since most macro-models for power consumption are based on switching activity values and other inputs statistics that satisfy similar properties as the ones in Proposition 1, the average energy per operation will show a similar trend. More precisely, the following result holds for the energy per operation of a given type of functional unit:

Proposition 2. If a data trace $\{x_n\}_{n \geq 0}$ is modeled by a lag-one Markov chain, and $E_k = E_{charact}(d_k)$ is the energy consumed per operation when k identical functional units are available, then $E = \lim_{k \rightarrow \infty} E_k = E_{charact}(D)$ where $E_{charact}$ is the energy macromodel for the functional unit under consideration and D is as in Proposition 1.

Proof: According to Proposition 1, $D = \lim_{k \rightarrow \infty} d_k$ and thus,

$$E = \lim_{k \rightarrow \infty} E_k = \lim_{k \rightarrow \infty} E_{charact}(d_k) = E_{charact}(D) \text{ for any characterization macromodel } E_{charact}. \square$$

The results presented above offers an answer to the question:

“Given a set of identical functional units and the behavior of the input data, what is the optimal number of instructions to be executed in parallel?”

The energy cost per operation for k instructions (using the same type of resource) issued in parallel is proportional to d_k and from the above result, it can be seen that no gain can be achieved by increasing the parallelism over some limit. Moreover, the optimal solution (that is, the k for which d_k is minimized) is found for small values of k . After k is large enough, d_k reaches its limit d . Typically, d_k has the behavior depicted in Fig.2. The minimum energy can be obtained for any value of k , including 1

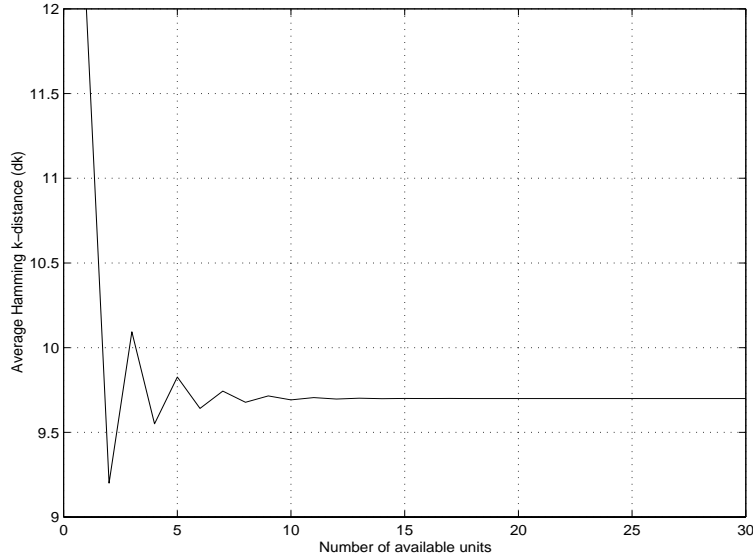


Fig.2 Typical behavior of the average Hamming k -distance for different values of k

In practice, since the level of parallelism in most user programs is limited (usually, the execution rate is below 6 instructions per cycle), to find the value of k for which the total energy per operation (or d_k) is minimized, we only need to consider a finite number of configurations. We shall see in Section 3 how we can achieve that via a profile-driven methodology.

2.2 The Effect of Speculative Execution

The results presented above did not assume anything about the behavior around branches. More precisely, modern processors use a fair amount of branch prediction techniques and speculate the execution of some of the succeeding instructions to increase performance [6]. However, this does not come without a price in the amount of energy consumed, especially when a branch is mispredicted. Thus, as the execution width is increased from 1 to 2, 3, or more functional units, the amount of unnecessary energy consumed due to mispredictions is likely to increase.

Example 2. Let us consider a loop computing the product of 2 vectors with length stored in $R6$; it is assumed that all instructions have a latency of 1 additional cycle in the case of a *Read-After-Write* (RAW) dependency [4]. In addition, the code is assumed to run on a superscalar architecture which can perform 4 additions, 2 multiplications and 2 load/store operations in one clock cycle. In Fig.3(a), we show the original loop, while in Fig.3(b) we give the unrolled loop (4 times) executed without speculation. In Fig.3(c), we show the effect of speculative execution when the branch is being predicted as taken. As we can see, when no speculation is used, the configuration of the pipeline during run-time execution is the same, while in the case of speculation, it changes according to the actual instructions that are speculated and being executed. If, however, the branch prediction scheme has a high rate of misprediction, then the amount of unnecessary energy consumed by the speculated LW instructions (denoted in bold in a shaded area in Fig.3(c)) will increase. Thus, it will also change the behavior of the actual switched capacitance seen in the processor pipeline.

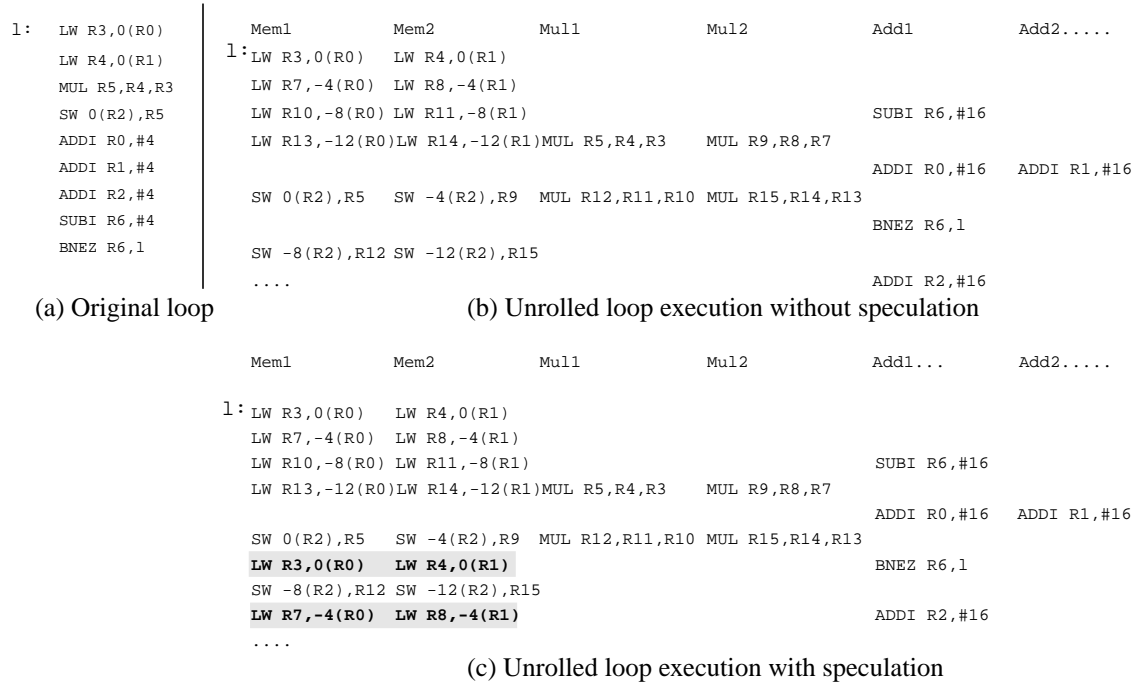


Fig.3 The effect of speculative execution

When speculation is used, the overall effect in terms of the actual switched capacitance may be different than the one presented in the previous section due to its overlapping effect. However, this overhead will reach a saturation point when the number of functional units used for parallel execution is about the same with the existing level of parallelism in the user program. In most programs, using an execution width of more than 4 functional units will not produce a significant difference in the switched capacitance, mainly because of the *limited parallelism* existing in the code generated by most compilers.

3 Profile-Driven Code-Annotation for Low-Energy Instruction Execution

As we have seen previously, based on the data streams that feed the different functional units of a superscalar processor and based on the run-time conditions, a particular sequence of instructions will exhibit a different switched capacitance depending on the issue rate and on the actual number of functional units used for performing the computation. Based on the results presented in Section 2, a simple profile-driven methodology can be used for finding the optimal number of instructions to be executed in parallel for a given *basic block* (a basic block is a straight line code sequence with no transfers in or out, except at the beginning or the end). By using a functional architectural simulator that captures the basic functionality of the underlying processor, we can find the optimal mix of level of parallelism to optimize energy. What we target is a *fine grain* characterization of the basic blocks encountered in the typical execution of a program as far as the optimal number of instructions to be executed in parallel is concerned. Typical basic blocks that will benefit from such a methodology are loops, unrolled to uncover more parallelism.

In practice, computing d_k using (1) is prohibitively expensive since it implies repeated matrix multiplications, given the characterization of the input stream in terms of conditional probabilities. In addition, the results in Section 2 do not capture the effect of speculative execution. Thus, we propose a *profile-driven methodology* to find the optimal number of instructions to be executed in parallel for each basic block. Our methodology is presented in Fig.4.

```

Low_Energy_Code_Annotate (Program)
  Extract all basic blocks ( $b_1, \dots, b_n$ );
  for  $k = 1, \dots, K$  {
    /*  $K$  = maximum number of instructions to be issued in parallel */
    for  $i = 1, \dots, n$ 
       $C_{i,k}$  = switched capacitance of basic block  $b_i$ 
        when  $k$  instructions are executed in parallel;
    }
  for  $i = 1, \dots, n$  {
    Find the value of  $k = 1, \dots, K$  such that  $C_{i,k}$  is minimized;
    Annotate all instructions in  $b_i$  with  $k$ ;
  }

```

Fig.4 Our methodology for low-energy code-annotation

The above methodology assumes that the input program is simulated for a typical input stream with an execution rate varying between 1 and K (the maximum available rate in the architecture under consideration). Then, after collecting the statistics for each basic block, the value of k which gives the minimum value of the switched capacitance for the execution stage is used to annotate all instructions in the basic block under consideration. This assumes that the instruction format allows for recording this information and that the architecture of the processor is slightly modified to support variable functional unit utilization.

3.1 Variable Rate Execution for Low Energy on a Superscalar Processor

To support and implement these features, we use SimpleScalar [18], a widely used tools for research in computer architecture. SimpleScalar is a suite of free, publicly available simulation tools that offer both detailed and high-performance simulation of modern microprocessors. SimpleScalar performs fast, flexible and accurate simulation of modern processors that implement a derivative of the MIPS-IV architecture [19]. The tool set takes binaries compiled for the SimpleScalar architecture and simulates their execution on one of several provided processor simulators. It also includes a derivative of the GNU C compiler which generates code for the SimpleScalar architecture. SimpleScalar offers an ideal environment for performance evaluation of different architectures. It has a fair amount of reconfigurability and flexibility and can be easily modified to support an extended instruction set. In addition, it supports both *in-order* and *out-of-order* execution, the latter based on the Register Update Unit scheme (RUU) [20]. RUU handles register synchronization and communication and unifies the *reorder buffer* and *reservation stations* [4] which are typically used in superscalar modern processors with out-of-order execution (Fig.5).

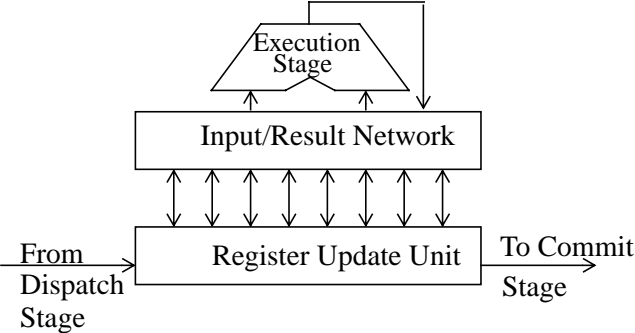


Fig.5 A simplified representation of the RUU and execution stage [18]

We give in Fig.6 the instruction format of the SimpleScalar architecture. In addition to providing a clean and simple instruction set architecture (similar to the MIPS/DLX [4,19] architecture), it also allows bi-endian instruction set definition which facilitates portability on both little and big endian machines. The 16-bit annotate space may be thus used to support additional operations or for annotating the code to inform the processor of any decisions that must be taken at run-time.

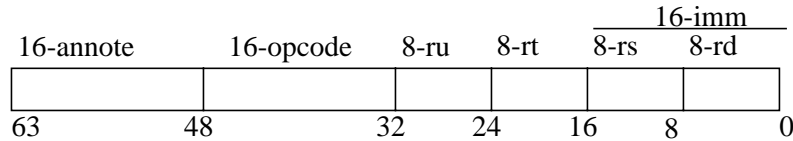


Fig.6 The instruction format in SimpleScalar architecture

In our case, the annotate field will be used to inform the processor (more precisely, the issue logic) to use only a limited number of the available functional units available to perform a given operation. Depending on the contents of the RUU, an instruction will be allowed to execute concurrently with a subsequent instruction only if they are both annotated with a number of functional units larger than 1. We should point out that the above considerations hold for a single type of functional unit. For multiple types of resources, the optimal number of functional units from each class is chosen so as to minimize the total switched capacitance or energy consumption.

3.2 Practical considerations

As explained previously, what we target is a *fine grain* characterization of the basic blocks encountered in the typical execution of a program as far as the optimal number of instructions to be executed in parallel is concerned. However, in practice, a basic block is comprised of about 6 instructions on average [4]. This actually means that the inter-block energy cost becomes comparable with the intra-block cost. Thus, in this case, it is likely that the real savings in energy will actually be less than the ideal one because of the important effect of the inter-block energy contribution. However, there is a solution to this. Since most of the time the two branches of a conditional jump are taken with unbalanced probabilities, one basic block is followed by the same succeeding block most of the time. In addition, this is always true for unconditional jumps. Thus, such blocks can be considered as a single *meta-block* for which the same value can be used for code-annotation. This will counter the inter-block effect and will smooth-out any differences in execution rates between succeeding basic blocks. In Fig.7, we illustrate how a meta-block is built from basic-blocks when information about the branch probabilities is provided by a profile-driven scheme.

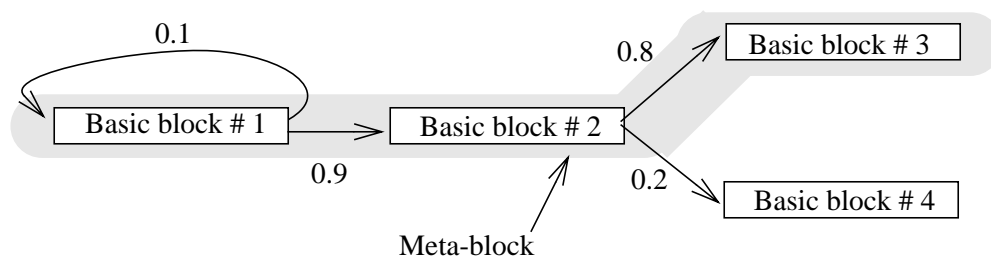


Fig.7 Meta-blocks obtained from successive basic-blocks

The choice of the maximum number of basic blocks to be put in a meta-block can affect the quality of the results. By putting few basic blocks in the same meta-block, we are closer to the single basic block scenario where the inter-block effect becomes important. On the other hand, increasing the size of the meta-block too much will reduce the energy savings since the code-annotation is no longer adaptive. In practice, a size of maximum 10 basic blocks per meta-block has turned out to be a good choice.

4 Experimental Results

We have implemented the methodology described in Section 3 by using the *sim-outorder* simulator from the SimpleScalar suite [18]. The experimental setup is shown in Fig.8. First, the input program is analyzed and all basic blocks are extracted. Then, using a typical input stream and up to 4 functional units, we derive the values of the switched capacitance of the execution stage for each of the 4 scenarios. Finally, each basic block is annotated with the level of parallelism which gives the minimum energy consumption. To validate the results, the annotated code is then run through a modified version of *sim-outorder* simulator which allows for dynamic selection of the execution width according to the information stored in the *annotate* field.

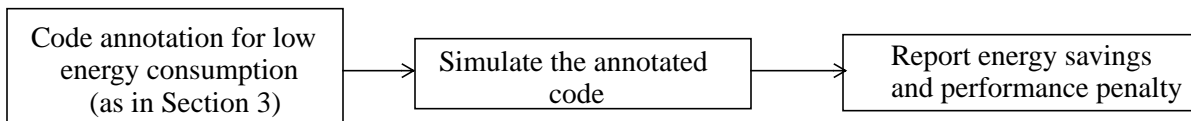


Fig.8 The experimental setup

We report in Tables 1 and 2 our results for some of the SpecInt95 benchmarks [21]. We have considered only the execution stage for a configuration with 4 integer ALUs, 1 multiplier and 1 divider. In Table 1 we present the effect of using less than 4 integer ALUs during the execution of each program.

Table 1: Fixed execution rate for some SpecInt’95 benchmarks

Benchmark	1 IntALU			2 IntALUs			3 IntALUs			4 IntALUs		
	Power	Energy	Cycles	Power	Energy	Cycles	Power	Energy	Cycles	Power	Energy	Cycles
compress	0.62	0.85	1.37	0.90	0.95	1.06	0.99	0.99	1.00	1.00	1.00	1.00
li	0.51	0.85	1.68	0.87	0.94	1.08	0.98	1.01	1.03	1.00	1.00	1.00
ijpeg	0.32	0.89	2.81	0.83	1.08	1.29	0.96	0.96	1.00	1.00	1.00	1.00
mk88sim	0.55	0.92	1.68	0.93	1.02	1.09	1.04	1.04	1.00	1.00	1.00	1.00
Average	0.50	0.87	1.88	0.88	0.99	1.13	0.99	1.00	1.01	1.00	1.00	1.00

As we can see, decreasing the level of parallelism will decrease performance in all cases when comparing to the 4 integer ALUs base case (“Cycles” column). On the other hand, the total energy (“Energy” column) does not follow the same trend. While in the case of *compress*, the total energy increases steadily when the level of parallelism is increased, for the other three benchmarks, the worst energy case is obtained either for 3 integer ALUs (*li* and *mk88sim*) or 2 integer ALUs (*ijpeg*). This is due the heavy data dependent characteristic of the energy measures, as opposed to performance. In addition, the average power (“Power” column) varies a lot among the 4 scenarios and it can be about 68% less (for *ijpeg*) in the case of one integer ALU when compared to the base case of 4 ALUs. The decrease in power comes from two sources: the decrease in total energy and the increase in the total number of cycles needed to complete the computations.

To see the effect of adaptively selecting the optimal number of ALUs to be used for each basic block, we have used the experimental setup in Fig.8 and considered two scenarios: adaptive execution rate, with and without performance constraints. In the constrained case, we selected the number of instructions to be executed in parallel which minimizes the total energy per meta-block, but doesn’t increase the number of cycles by more than twice. We have implemented the *meta-block* concept in which each basic block is assigned the same execution rate as its neighbors, when they execute in sequence most of the time. For comparison, all results are normalized with respect to the worst energy case (that is, 4 ALUs for *compress*, 3 for *li* and *mk88sim* and 2 for *ijpeg*). As we can see, in the unconstrained case we get larger energy savings than any of the fixed execution rate cases in Table 1 (21% compared to 13% on average). In addition, an

average power savings of 45% can be obtained with about 54% decrease in performance.

Table 2: Adaptive execution rate for some SpecInt'95 benchmarks

Benchmark	Adaptive without performance constraints			Adaptive with performance constraints			Worst energy case		
	Power	Energy	Cycles	Power	Energy	Cycles	Power	Energy	Cycles
compress	0.67	0.81	1.21	0.75	0.90	1.21	1.00	1.00	1.00
li	0.59	0.79	1.33	0.68	0.82	1.19	1.00	1.00	1.00
jpeg	0.31	0.72	2.33	0.42	0.79	1.85	1.00	1.00	1.00
mk88sim	0.65	0.86	1.31	0.73	0.85	1.15	1.00	1.00	1.00
Average	0.55	0.79	1.54	0.64	0.84	1.35	1.00	1.00	1.00

In the constrained case, the total energy savings is about 16% on average, while the power savings is 38% on average at the expense of a 35% decrease in performance. Thus, our approach can be efficiently used for trading-off performance for a decrease in both total energy and average power consumption. While the first will affect the battery life, the second can be used as an efficient means to reduce the operating temperature in a system equipped with a smart thermal manager. The annotated code for low-power can be activated whenever the operating temperature increases over a threshold limit as a *passive cooling* technique, similar to the “clock throttling” state of the CPU in the ACPI specification [16].

In addition, the technique described above can be also applied at run-time with support from hardware. A hardware-driven profiling scheme to support run-time optimization has been the focus of recent research [22]. The hardware cost of such a scheme does not exceed 10% of the area and thus, it is likely to be useful for run-time power-performance trade-off as well.

5 Conclusion

In this paper, we presented a novel technique for code optimization for low power based on a *profile-driven methodology*. Specifically, we show analytically that there is an optimal level of parallelism for energy consumption (which may not be necessarily the same as for performance) and we propose a *compiler-assisted* technique for code annotation that adaptively selects at run-time the optimal number of instructions to be executed in parallel as far as total energy is concerned. As the experimental results show, significant savings can be obtained in both energy and power consumption, at the expense of a decrease in performance. The techniques described in this paper can thus be used as a means for prolonging the *battery life* (by decreasing the energy consumption), but most importantly, for *thermal management* by achieving significant average power reductions in the execution stage.

6 References

- [1] M. Pedram, ‘Power Minimization in IC Design: Principles and Applications,’ in *ACM Trans. on Design Automation of Electronic Systems*, vol.1, no.1, pp.1-54, Jan.1996.
- [2] J. Mermet and W. Nebel, ‘Low Power Design in Deep Submicron Electronics,’ Kluwer Academic, Norwell, MA, 1997.
- [3] E. Macii, M. Pedram, and F. Somenzi, ‘High-Level Power Modeling, Estimation, and Optimization,’ vol.17, no.11, pp.1061-1079, Nov. 1998.
- [4] J.L Hennessy and D.A. Patterson, ‘Computer Architecture: A Quantitative Approach,’ Morgan Kaufman, San Francisco, CA, 1996 (2nd edition).
- [5] D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo, ‘The IBM 360 Model 91: Processor Philosophy and Instruction Handling,’ in *IBM J. Research and Development*, vol.11, no.1, pp.8-24, Jan. 1967.

- [6] M.D. Smith, M. Horowitz, and M.S. Lam, 'Efficient Superscalar Performance Through Boosting,' in *Proc. IEEE/ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pp.199-204, March 1992.
- [7] T. Agerwala and J. Cocke, 'High Performance Reduced Instruction Set Processors,' *IBM Tech. Rep.*, March 1987.
- [8] J.A. Fisher, 'Very Long Instruction Word Architectures and ELI-512,' in *Proc. Tenth Symposium on Computer Architecture*, pp.140-150, June 1983.
- [9] V. Tiwari, S. Malik, and A. Wolfe, 'Power Analysis of Embedded Software: A First Step Toward Software Power Minimization,' in *IEEE Trans. on VLSI Systems*, vol.2, no.4, pp.437-445, April 1994.
- [10] C.L. Su, C.-Y. Tsui, and A.M. Despain, 'Saving Power in the Control Path of Embedded Processors,' in *IEEE Design and Test of Computers*, vol.11, no.4, Dec. 1994.
- [11] S.T. Cheng, C.M. Chen, J.W. Huang, 'Low-Power Design for Real-Time Systems,' in *Real-Time Systems*, vol.15, no.2, pp.131-148, Sept. 1998.
- [12] M.T.-C. Lee, V. Tiwari, S. Malik and M. Fujita, 'Power Analysis and Minimization Techniques for Embedded DSP Software,' in *IEEE Trans. on VLSI Systems*, vol.5, no.1, pp.123-135, Jan. 1997.
- [13] M. Johnson, 'Superscalar Microprocessor Design,' ch. 10, Prentice Hall, Englewood Cliffs, NJ, 1990.
- [14] B. Klass, D.E. Thomas, H. Schmit, D.E. Nagle, 'Modeling Inter-Instruction Energy Effects in a Digital Signal Processor,' in *Power-Driven Microarchitecture Workshop*, in conjunction with *Intl. Symposium on Computer Architecture*, June 1998.
- [15] S. Manne, A. Klauser, and D. Grunwald, 'Pipeline gating: Speculation control for energy reduction,' in *Proc. Intl. Symposium on Computer Architecture*, Barcelona, Spain, June 1998.
- [16] 'Advanced Configuration and Power Interface Specification,' Intel, Microsoft, Toshiba, Revision 1.0b, Feb. 2, 1999, at <http://www.teleport.com/~acpi/DOWNLOADS/ACPIspec10b.pdf>.
- [17] A. Papoulis, 'Probability, Random Variables, and Stochastic Processes', McGraw-Hill Co., 1984.
- [18] D. Burger, T.M. Austin, 'The SimpleScalar Tool Set, Version 2.0,' *CSD Technical Report #1342*, University of Wisconsin-Madison, June 1997.
- [19] C. Price, 'MIPS IV Instruction Set, revision 3.1.,' MIPS Technologies, Inc., Mountain View, CA, Jan. 1995.
- [20] G.R. Sohi, 'Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers,' in *IEEE Trans. on Computers*, vol.39, no.3, pp.349-359, March 1990.
- [21] SpecInt'95 Benchmark Suite, at <http://www.spec.org/osg/cpu95/CINT95>.
- [22] M.C. Merten, A.R. Trick, C.N. George, J.C. Gyllenhaal, and W.-M. W. Hwu, 'A Hardware Driven Profiling Scheme for Identifying Program Hot Spots to Support Run Time Optimization,' in *Proc. Intl. Symp. on Computer Architecture*, pp.136-147, May 1999.