

EAC: A Compiler Framework for High-Level Energy Estimation and Optimization

I. Kadayif, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam
Microsystems Design Lab
Pennsylvania State University
University Park, PA, 16802, USA

Abstract

This paper presents a novel Energy-Aware Compilation (EAC) framework that can estimate and optimize energy consumption of a given code taking as input the architectural and technological parameters, energy models, and energy/performance constraints. The framework has been validated using a cycle-accurate architectural-level energy simulator and found to be within 6% error margin while providing significant estimation speedup. The estimation speed of EAC is the key to the number of optimization alternatives that can be explored within a reasonable compilation time.

1. Introduction

Compiler is a critical component in determining the types, order (to some extent), and number of instructions executed for a given application. Thus, it wields a significant influence on the power consumed by the system. However, most compiler optimizations focus on metrics such as performance and code size, and do not account for power considerations during optimizations or code generation. There have been a few studies that have focused on specific low-level optimizations for reducing power such as instruction scheduling [9] and register allocation [3]. High-level (source-level) optimizations can complement these techniques, and more importantly, as shown in a few recent simulation-based studies (e.g., [8, 5]), can have a larger impact on the system power consumption. In order to develop and evaluate new energy-conscious compiler optimizations, we need mechanisms to estimate the energy consumption in a quick and accurate manner.

In this paper, a novel *Energy-Aware Compilation* (EAC) framework that can estimate and optimize energy consumption of a given code is presented. This framework has the ability to estimate the energy consumption of a high-

level (source) code given the architectural and technological parameters, energy models, and energy/performance constraints. This capability also allows us to apply high-level code and data transformations (both at loop-level and procedure-level) to optimize energy. In other words, the proposed compilation framework (Figure 1) can be used for either quick energy estimation (without performing any energy-oriented optimization) or energy optimization under several constraints. This paper makes the following contributions:

- It presents a high-level energy estimation model that can be incorporated into an optimizing compiler. Further, it discusses the necessary compiler analyses for extracting the required parameters for this energy model from the code.
- It presents a validation of the compiler-directed energy estimation using a cycle-accurate architectural-level energy simulator for a simple architectural model.

This work proposes, to the best of our knowledge, the first energy-aware compilation framework that can estimate and optimize the energy consumption of a given application at the *high-level (source-level)* under different constraints. Note that estimating energy at high-level is crucial because only then can the energy/performance tradeoffs involving high-level optimizations such as tiling and other loop and data optimizations be made and high-level optimizations that target energy be performed. In contrast, if we estimate the energy consumption at assembly level, we can only evaluate a small number of alternative optimization strategies, the energy estimation process would be slow, and it would be difficult to integrate high-level energy optimization strategies with commonly-used source-level performance optimizations.

The first version of the EAC framework has been implemented using the SUIF [11] compiler and evaluated using several array-dominated benchmark codes. Note that array-dominated codes are very common in digital signal and media processing applications that consist of multiple nested loops. It is anticipated that, in many array-dominated codes, the highest energy gains will come from high-level

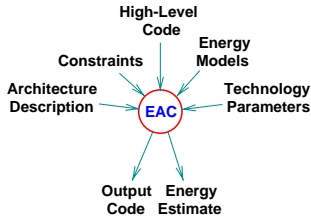


Figure 1. Energy-Aware Compilation (EAC) framework.

code optimizations [1, 10]. This framework can be used by compiler designers and system architects in the following ways:

- Given a fixed architecture and technology parameters, various performance-oriented compiler optimization techniques can be evaluated from an energy viewpoint. For example, the energy impact of loop tiling, loop distribution, unroll-and-jam, and other loop and data transformations can be investigated using EAC.
- For specific compiler optimization techniques preferred for performance reasons, the potential energy savings from architectural enhancements/modifications can be evaluated. Conversely, compiler optimizations designed to exploit energy-efficient architectural features can be studied.
- When designing new energy-aware compiler optimization techniques, the impact of imminent changes in technology on the effectiveness of the proposed techniques can be explored. Further, the impact of optimizations on the energy consumption of different components of a system (e.g., on-chip or off-chip) can be evaluated.
- Given a specific set of operating constraints (energy and performance) and a set of high-level optimizations, the tool can be used to generate code meeting these constraints, whenever possible. Since the EAC-directed estimation is moderately accurate, it can also be used (if desired) just as a fast, high-level energy estimation tool.

While, architectural-level energy simulators (e.g., [10]) can be used to study some of these issues, the proposed tool is considerably faster as it is not simulation-driven. Estimation speed is the key to the number of optimization alternatives that can be explored within a reasonable amount of time.

The rest of this paper is organized as follows. The next section explains how software can influence energy consumption for a given architecture and presents our energy model embedded in EAC. Section 3 describes the necessary compiler analyses to extract the parameters that are needed by the energy model. Section 4 presents a validation of the compiler-based energy estimation using a cycle-accurate energy simulator. Section 5 concludes the paper

Component	Compiler-Supplied Parameters	Architecture/Circuit Parameters
datapath	number of activations for each component type	structure, bit-width, and switched capacitances for components
cache	number of hits/misses, number of reads/writes	bitline, wordline & decoder capacitance, tag size, voltage swing during transitions
memory	number of accesses, intervals between accesses	off-chip capacitance, refresh rate, number and types of low-power modes
buses	number of transactions	wire capacitance, bus width
clock network	number of execution cycles, number of memory stall cycles	clock generation circuitry, size and capacitance of distribution buffers, load capacitances of clocked components

Figure 2. A summary of the major parameters for estimating the energy consumed by different hardware components.

with a summary and an outline of the future research on this topic.

2. Modeling Energy Consumption of Software

Energy consumption is dependent on how different components of a system are exercised by the software. In this work, we focus on dynamic energy consumption. The dynamic energy consumed in a system can be expressed as the sum of the energies consumed in the different components such as the *datapath*, *caches*, *clock network*, *buses*, and *main memory*. The activity, and consequently the energy consumed in these components, is determined by the software being executed on the system. The software can modify the number of transitions in the nodes (note that this affects switching power) by altering the input patterns, reduce effective capacitance by reducing the absolute number of accesses to high-capacitance components (e.g., large off-chip memories), or scale voltage/clock frequency to adapt the energy behavior of the code to the needs of the application.

This paper focuses on a on a single-issue, five-stage (instruction fetch (IF), instruction decode/operand fetch (ID), execution (EXE), memory access (MEM), and write-back (WB) stages) pipelined datapath of typical embedded processors with a single-level on-chip cache. Currently, this is the only architectural model for which our compiler estimates energy. We selected this model mainly because we have access to an accurate energy simulator that generates the detailed energy behavior of a given application running on this architecture, and this allows us to evaluate the accuracy of the compiler-directed energy estimation. In the following, we explain the energy consumption in the individual components of the system and the impact of software/compiler on these components with the help of our target architecture. The third column in Figure 2 gives a summary of the major architecture/circuit parameters whose

values are used to estimate the energy consumption within the compiler.

- **Datapath:** The energy consumed in a datapath is dependent on the number, types, and sequence of instructions executed. The type of an instruction determines the components in the datapath that are exercised while the number of instructions determines the duration of the activity. The energy of each instruction was obtained for our target architecture by accounting for the energy consumed in each component exercised by the instruction. Whenever a component is exercised, there is a switching activity in that particular component contributing to dynamic energy consumption. For example, when an integer addition instruction is executed, energy is consumed in the instruction fetch logic in the first stage of the pipeline, in the register file when accessing the source operands in the decode stage, in the ALU when executing the operation, and, again, in the register file during write-back. It must be noted that energy is also consumed in the pipeline registers of the datapath, and that the components in the memory stage of the pipeline are not exercised by this instruction. Using a cycle-accurate energy simulator, we are able to capture the activity of each individual component of the architecture that is exercised by a given instruction. This information is fed to the compiler and used along with the estimated number of times each instruction executes to obtain the energy consumed in the datapath.

- **Cache:** The energy consumed in a cache is dependent on the number of cache accesses, number of misses, the cache configuration (e.g., associativity, capacity, line size), and the extent of utilizing energy-efficient implementation techniques (e.g., subbanking, bitline isolation [4]). There are two important components of the cache, namely, the tag and the data arrays. The major components (of a tag/data array) that consume energy are the row and column decoders, wordlines, bitlines, and sense amplifiers. The energy consumed during reads and writes vary since the voltage swings in the bit lines are different for these two (i.e., full swing for writes and limited swing for reads). Also, the sense amplifiers are not used during the write cycle. Consequently, it is important to estimate the number of reads and writes individually. The energy consumed in the caches is largely independent of the actual data accessed from the caches, and the prior work has shown that the number of cache accesses is sufficient to model energy accurately [4]. Our compilation framework estimates the number of cache misses and hits given the high-level code as explained in the next section. This information along with the cache configuration that determines the length of the bitlines and wordlines, and the size of decoders and sense amplifiers can be used to evaluate the energy consumption. Further, we parameterize the energy model based on the extent of energy-efficient implementation techniques used to reduce the capacitance on

the bit and wordlines.

- **Main Memory:** The organization of the main memory arrays is similar to that of the caches, but is different in two ways. First, the memory arrays have no tag comparison portion. Second, the basic cell for implementing memory storage (DRAM cells) is different from that used in on-chip caches (SRAM cells). Consequently, there is a difference in the energy consumed during read/write accesses. The energy consumed in the memory can be modeled fairly accurately by capturing the number of accesses and the interval between the accesses within our compiler.

- **Buses:** The buses are used to communicate the addresses from the datapath to the caches and memories and to transfer the data between these units. The energy consumed on a given bus is dependent on the number of transactions on the bus, the bus capacitance, the bus width, and the switching activity on the bus. While the width of the bus and the bus capacitance are readily available once the design is finalized, the other two factors are a function of the software. Our compiler estimates the number of transactions, and assumes a 50% switching activity since data values are not known at the source level.

- **Clock Network:** The components of the clock network that contribute to the energy consumption are the clock generation circuit (PLL), the clock distribution buffers and wires, and the clock-load on the clock network presented by the clocked components. The energy consumed in a single cycle depends on the parts of the clock network that are active. The PLL and the main clock distribution circuitry are normally active every clock cycle during execution. Therefore, our compiler captures the energy consumption due to those two components by estimating the number of cycles that the code would take. However, the participation of the clock-load varies based on the active components of the circuit as determined by the software executing on the system. For example, the clock to the caches are gated (disabled) when a cache miss is being serviced. The EAC exploits the estimation techniques for the datapath and caches explained above to effectively account for this varying clock-load in a given cycle.

3. Extracting Parameters for Energy Models

In order to compute the energy expended in different hardware units, the compiler should analyze the program and extract the application-dependent parameters required by the energy models. The second column in Figure 2 gives a list of these compiler-supplied parameters. In this section, we explain the techniques to extract these parameters from the nested loop-based codes used in this work.

The first step in developing the automated process involves identifying the high-level constructs used in these codes and correlating them with the actual machine instruc-

tions. The constructs that are vital to the studied codes include a typical loop, a nested loop, assignment statements, array references, and scalar variable references within and outside loops.

To compute datapath energy, we need to estimate the number of instructions of each type associated with the actual execution of these constructs. To achieve this, the assembly equivalents of several codes were obtained using our back-end compiler (a variant of `gcc`) with the `O2`-level optimization. Next, the portions of the assembly code were correlated with corresponding high-level constructs to extract the number and type of each instruction associated with the construct. In order to simplify the correlation process and to partially isolate the impact of instruction choice due to low-level optimizations, instructions with similar functionality and energy consumption are grouped together. For example, both branch-if-not-equal (`bne`) and branch-if-equal (`beq`) are grouped as a generic branch instruction under the name `bne`.

In order to illustrate the extraction process in more detail, we focus on some specifics of the following example constructs. First, we focus on a loop construct. Each loop construct is modeled to have a one-time overhead to load the loop index variable into a register and initialize it. Each loop also has an index comparison and an index increment (or decrement) overhead whose costs are proportional to the number of loop iterations (called trip count, or *trip*). From correlating the high-level loop construct to the corresponding assembly code, each loop initialization code is estimated to execute one load (`lw`) and one add (`add`) instruction (in general). Similarly, an estimate of $trip+1$ load (`lw`), store-if-less-than (`stl`), and branch (`bne`) instructions is associated with the index variable comparison. For index variable increment (resp. decrement), $2 \times trip$ addition (resp. subtraction) and $trip$ load, store, and jump instructions are estimated to be performed. Next, we consider extracting the number of instructions associated with array accesses. First, the number of instructions required to compute the address of the element is identified. This requires the evaluation of the base address of the array and the offset provided by the subscript(s). Our current implementation considers the dimensionality of the array in question, and computes the necessary instructions for obtaining each subscript value. Computation of the subscript operations is modeled using multiple shift and addition/subtraction instructions (instead of multiplications). Finally, an additional load/store instruction was associated to read/write the corresponding array element. Note that these correlations between high-level constructs and low-level assembly instructions are a first-level approximation for our simple architecture and array-dominated codes with the `O2`-level optimization and obtained through extensive analysis of a large number of codes fragments. Our current calculation method

does not take into account the energy spent in datapath during stalls (due to branches or cache misses) as we assume the existence of clock gating which reduces stall energy significantly (i.e., it does not affect any trend observed in this study).

In this study, we focus only on data cache as our high-level optimizations influence data cache energy and performance behavior more dramatically as compared to the instruction cache. To compute the number of hits and misses, our current implementation uses the miss estimation technique proposed by McKinley et al [6]. This approach first groups the array references according to the potential group-reuse (i.e., the type of reuse that originates from multiple references to the same array) between them. Then, for each representative reference, it calculates a *reference cost* (i.e., the estimated number of misses during a complete execution of the innermost loop). Basically, the reference cost of a given array reference with respect to a loop order is 1 if the reference has temporal reuse in the innermost loop; that is, the subscript functions of the reference are independent of the innermost loop index. The reference cost is $trip/(cls/stride)$ if the reference has spatial reuse in the innermost loop. In this expression, *trip* is the number of iterations of the innermost loop (trip count), *cls* is the cache line size in data items (array elements), and *stride* is the step size of the innermost loop multiplied by the coefficient of the loop index variable. Finally, if the reference in question exhibits neither temporal nor spatial reuse in the innermost loop, its reference cost is assumed to be equal to *trip*; that is, a cache miss per loop iteration is anticipated. After all the reference costs are calculated, the technique computes the *loop cost* (i.e., the total number of estimated misses due to nest execution) considering each reference in the nest. The overall loop cost of the nest is the sum of the contributions of each reference it contains. The contribution of a reference is its reference cost multiplied by the number of iterations of all the loops (that enclose the reference) except the innermost one. Note that this miss calculation process is a good first degree approximation if one does not consider inter-nest data reuse. As an extension to McKinley et al.'s algorithm, our compiler also distinguishes reads and writes, and computes the read and write misses separately. Note that the sum of read and write misses also gives us the number of accesses to the main memory, a parameter necessary to compute the main memory energy.

Estimating the number of execution cycles (which is necessary to compute the clock energy and performance) is not very difficult in our architecture as it is a single-issue machine. Since each instruction (omitting stalls) requires one cycle to be initiated, the number of instructions is a lower bound for the number of cycles. To this lower bound, we add the number of estimated *stall cycles* (a fixed number of cycles for each estimated cache miss) to reach the final

Parameter	Value
Supply Voltage	3.3 V
Data Cache Configuration	4 KB, 2-way, 32 bytes line size
Data Cache Hit Latency	1 cycle
Memory Access Latency	100 cycles
Per Access Read Energy for Data Cache	0.20 nJ
Per Access Write Energy for Data Cache	0.21 nJ
Per Access Energy for Memory	4.95 nJ
On-Chip Bus Transaction Energy	0.04 nJ
Off-Chip Bus Transaction Energy	3.48 nJ
Per Cycle Clock Energy	0.18 nJ
Technology	0.35 micron

Figure 3. Base configuration parameters used in the experiments. The energy numbers are obtained by providing the (hardware) configuration parameters to our energy models.

compile-time estimate of the number of clock cycles. Note that the two parameters required here, namely, the number of instructions and the number of misses, are estimated by the compiler as explained above.

Estimating the number of bus transactions is also relatively easy as it is proportional to the number of cache and memory accesses, both of which are captured by the compiler during cache miss analysis.

Our compiler also calculates the static code size (in the number of assembly instructions). This process is very similar to that of execution cycle estimation; the difference is that in static code estimation the compiler does not multiply the static estimates (for individual constructs) by the trip counts of the enclosing loops.

This paper is a first step towards compiler-based energy estimation and optimization. Consequently, it focuses on rather a simple architecture and tries to measure the effectiveness of compiler-based analysis. The accuracy of the compiler-based estimates discussed above (and, of course, that of overall energy estimate) can be improved by employing more sophisticated techniques. More refinements are essential to capture the influence of other compiler and architectural aspects. Note that once an estimation model is selected, the rest of our technique is independent from instruction count, cache hit/miss and bus transaction estimates. In other words, our framework is general enough to accommodate different estimation strategies where available. In our current implementation, in cases where the loop bounds and array sizes are not known at compile-time, we exploit the available profile information.

4. Validation

Estimating the energy consumption at high level (source code level) is not very useful unless the estimation is *accurate enough* to guide programmer-directed and compiler-directed high-level optimizations. Therefore, validating the

Type	Cost	Type	Cost	Type	Cost
li	166.39	bne	200.04	mult	366.01
sll	157.90	mflr	152.67	lw	533.50
sw	168.79	add	156.81	sll	152.67

Figure 4. Datapath energy costs of the most frequently-used instructions. The main sources of datapath energy consumption are the register files, pipeline registers, functional units and the datapath multiplexors and contribute to 20%, 40%, 20%, and 8% of the overall energy consumption, respectively, when averaged over different applications.

compiler-directed energy estimation is of critical importance. In this section, we compare the compiler-estimated energy consumption to that obtained through a cycle-accurate energy simulator that uses *transition-sensitive* energy models for the datapath and analytical energy models for other components. Transition-sensitive models quantify the energy consumption based on the current and previous data inputs to a circuit; hence, they are very accurate. However, they are also time-consuming and difficult to develop. While transition-sensitive models are essential for modeling datapaths accurately [10], analytical models based on *activity-based* approaches are more than sufficient for modeling caches, memories and clock [4]. Activity-based energy models assume a fixed (component-specific) energy consumption when the component is accessed independent of the specific data input values. Our compiler-based estimation approach uses such activity-based energy models for all components including the datapath. The difficulty in predicting data input sequences from the high-level source code mandates the use of the activity-based approach in EAC. In this section, we first perform this validation for different benchmarks to observe the error margin in the estimates across different components of the system. Next, the validation is performed for one of the benchmarks when the type of high-level compiler optimizations is varied. This is done to ensure that the trends due to optimizations are correctly predicted by EAC (that is one of the main goals of the compiler-directed energy estimation).

In this work, we use an enhanced version of SimplePower, an architectural-level, cycle-accurate energy simulator. SimplePower [10] is an execution-driven power estimation tool and is publicly available. It is based on the architecture of a five-stage pipelined datapath. The energy models utilized in the simulator are within 8% error margin of measurement from real systems [10]. The energy simulator can work under the assumption of different supply voltages and micron technologies. All simulator-based results reported in this paper are obtained using the param-

Benchmark	Source	Input Size (KB)	Description
fir	DSPstone	80	Filtering
conv	DSPstone	80	Convolution
lms	DSPstone	80	LMS FIR filtering
real	DSPstone	160	N real updates
biquad	DSPstone	200	IIR biquad
complex	DSPstone	320	N complex updates
mxm	-	120	Matrix multiply
vpentai	Spec	625	Pentadiagonal inversion
tsfi	Perfect Club	204	Transonic flow analysis
tomcatvi	Spec	280	Mesh generation
adi	-	246	ADI computation

Figure 5. Benchmark codes used in the experiments.

eters given in Figure 3.

The same configuration (shown in Figure 3) is also used in the compiler estimation experiments. An important issue in estimation within the compiler is associating an accurate, *activity-based energy cost* for each type of instruction. To achieve this, we averaged energy values by executing multiple (1000) instances of the same instruction with random data using the transition-sensitive simulator. The resulting values are stored as a table where each entry lists an instruction type and corresponding energy. Figure 4 gives the most frequently used instructions (by the back-end compiler) and corresponding energy consumptions. Note that the energy value given for an instruction (group) involves all the energy consumed in different parts of the datapath. Such energy tables could be built for other target architectures using power measurement on an actual system [9]. Alternately, one could extract these numbers by embedding energy models in architectural-level simulators that are normally available for many target platforms.

Figure 5 lists the codes in our experimental suite. The first six codes are array-based versions of the corresponding DSPstone benchmarks. The remaining codes are array-based programs that are re-written to use integer data instead of floating-point data (as our simulator currently operates only with integer data).

Figure 7 compares the compiler-directed energy estimation with that of the simulator. We see that the average difference between energy estimations is 12.21%, 6.09%, 6.16%, 6.78%, and 4.62% for datapath, cache, main memory, buses, and clock network, respectively. Overall, the compiler-estimated total energy is within 5.9% of the simulator value.

We now explore the causes for these error margins component-wise. The datapath inaccuracies stem from two major factors. First, estimating the number of instructions executed requires accurate (high-level) modeling of back-end code generation techniques. The inaccuracies in this factor were observed to contribute to an average 5% error in EAC estimates across all the bench-

Benchmark	EAC	Simulator
fir	0.31	434.44
conv	0.27	260.51
lms	0.28	598.89
real	0.34	563.24
biquad	0.35	1,366.60
complex	0.41	2,762.11
mxm(100)	0.28	36,602.23
vpentai	0.53	2,265.45
tsfi	0.30	547.12
tomcatvi	0.46	5,614.99
adi	0.33	2,662.31
mxm(200)	0.28	278,447.03
mxm(400)	0.28	2,177,698.46
mxm(800)	0.28	17,209,663.22
mxm(1600)	0.28	137,344,710.07

Figure 6. Comparison of estimation times (in seconds) for EAC and the simulator. In the first column, the numbers within parentheses give the input size. The last four entries for the transition-sensitive simulator are extrapolated values based on smaller input sizes.

marks. As an example, depending on the absolute magnitudes of some variables that are involved in address calculations, we noted that the back-end compiler generates different code sequences (which our current implementation does not capture). Second, the impact of input transitions (i.e., transition-sensitivity) in the datapath are not accurately captured by the compiler estimates. It is very difficult to predict the exact instruction sequence and associated data (both impact the transition activity of the components) at the high level. Hence, the resulting inaccuracies contribute to 7.2% of the overall estimation error in the datapath.

The main source of error for cache and main memory energies is the inaccurate estimation of the number of cache hits and misses. These inaccuracies are a result of not taking conflict misses and misses due to scalar accesses into account. This can lead to an underestimation on the compiler’s part as exemplified in `vpentai` and `biquad`. The underestimation of cache energy for the `fir` benchmark, however, occurs due to mispredictions in the number of loads (which is also reflected in datapath energy estimates). Further, our current miss estimation framework does not model data locality across separate nested loops (i.e., inter-nest locality). That is, it operates under the assumption that the cache is empty at the beginning of each nest execution. In contrast to the previous case, this can lead to an overestimation of misses as in the `adi` code.

The inaccuracies in the clock is a factor of the inaccuracies in estimating the number of execution cycles which, in turn, is dependent on predicting the number of instructions executed and predicting the number of cache misses. Additional inaccuracies also accrue from not accounting for

Benchmark	Energy Estimation														
	Datapath (1.0e-02 Joules)			Cache (1.0e-03 Joules)			Memory (1.0e-03 Joules)			Bus (1.0e-03 Joules)			Clock (1.0e-02 Joules)		
	EAC	SB	diff%	EAC	SB	diff%	EAC	SB	diff%	EAC	SB	diff%	EAC	SB	diff%
fir	0.0285	0.0357	-20.03	0.0495	0.0635	-21.99	0.0371	0.0360	3.24	0.0345	0.0361	-4.48	0.0222	0.0221	0.35
conv	0.0176	0.0206	-14.53	0.0310	0.0330	-6.04	0.0248	0.0246	0.62	0.0226	0.0229	-1.04	0.0136	0.0136	-0.12
lms	0.0414	0.0479	-13.45	0.0700	0.0760	-7.83	0.0495	0.0483	2.59	0.0467	0.0468	-0.34	0.0316	0.0304	4.17
real	0.0390	0.0421	-7.37	0.0690	0.0690	0.01	0.0743	0.0743	0.03	0.0637	0.0637	0.02	0.0346	0.0327	5.87
biquad	0.0931	0.1069	-12.91	0.1553	0.1733	-10.39	0.1300	0.1304	-0.32	0.1175	0.1209	-2.83	0.0749	0.0714	4.84
complex	0.2042	0.2062	-0.97	0.3160	0.3040	3.94	0.2971	0.2970	0.01	0.2617	0.2596	0.81	0.1744	0.1575	10.70
mxm	2.1945	2.4803	-11.52	3.2360	3.1660	2.21	5.6430	5.9400	-5.00	4.4906	4.6851	-4.15	2.0970	2.1520	-2.55
vpentai	0.1532	0.1789	-14.35	0.1911	0.2003	-4.62	0.5092	0.5450	-6.58	0.3167	0.4142	-23.54	0.1520	0.1590	-4.35
tsfi	0.0355	0.0425	-16.46	0.0515	0.0513	0.32	0.0257	0.0217	18.80	0.0269	0.0240	11.92	0.0264	0.0255	3.69
tomcatvi	0.2887	0.3377	-14.52	0.5236	0.5409	-3.19	1.168	1.229	-5.0	0.9042	0.9497	-4.80	0.3683	0.3674	0.22
adi	0.1943	0.2117	-8.21	0.1989	0.1868	6.45	0.1683	0.1341	25.54	0.1517	0.1258	20.61	0.1520	0.1334	13.97
Average:			12.21			6.09			6.16			6.78			4.62

Figure 7. Comparison of compiler-directed (EAC) and simulator-based (SB) energy estimation and percentage difference. The estimated energy consumptions are in units given below the name of each component.

pipeline stalls due to data or control hazards. Finally, the accuracy of bus and memory energy estimates are affected by the number of bus transactions and cache misses estimated.

Overall, we observe that the compiler-directed energy estimation approach is moderately accurate as compared to the simulator-based approach. The loss in accuracy is traded for the ability to perform the estimates significantly faster. Figure 6 shows the absolute times (in seconds) required for obtaining the energy estimates for different benchmarks. An important reason for the longer estimation times of the simulator is its cycle-accurate nature which causes the scaling of estimation time with the problem size. In contrast, the time taken by the compiler-based approach is independent of the problem size.

5. Conclusions and Future Work

We have presented an energy-aware compilation framework (EAC) that can estimate and optimize energy consumption of a given code for a given set of technology parameters and an architecture description. Our validation results show that the proposed framework is very accurate and within 6% error margin (on the average) of a cycle-accurate, architectural-level energy simulator. Our future research plans include developing more accurate energy estimation strategies within EAC and evaluating energy-aware versions of high-level optimizations.

References

[1] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, June 1998.

[2] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.

[3] C. H. Gebotys. Low energy memory and register allocation using network flow. In *Proc. Design Automation Conference*, Anaheim, CA, pp. 435–440, June 1997.

[4] K. Ghose and M. B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers, and bit-line segmentation. In *Proc. 1999 International Symposium Low Power Electronics and Design*, 1999, pages 70–75.

[5] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proc. the 37th Design Automation Conference*, Los Angeles, California USA, June 5-9, 2000.

[6] K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 1996.

[7] P. R. Panda and N. D. Dutt. Reducing address bus transitions for low-power memory mapping. In *Proc. European Design and Test Conference*, March 1996.

[8] T. Simunic, L. Benini, and G. De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Proc. ACM Design Automation Conference*, 1999.

[9] V. Tiwari, S. Malik, A. Wolfe, and T. C. Lee. Instruction level power analysis and optimization of software, *Journal of VLSI Signal Processing Systems*, Vol. 13, No. 2, August 1996.

[10] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. the International Symposium on Computer Architecture (ISCA)*, Vancouver, British Columbia, Canada, June 2000.

[11] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.