

Profile-Driven Program Synthesis for Evaluation of System Power Dissipation*

Cheng-Ta Hsieh

Massoud Pedram

Gaurav Mehta

Fred Rastgar

Department of Electrical Engineering - System
University of Southern California, Los Angeles, CA

Mobile & Handheld Product Group
Intel Corporation, Santa Clara, CA

Abstract – This paper presents a new approach for estimating power dissipation in a high performance microprocessor chip. First, characteristic profile (including parameters such as the cache miss rate, branch prediction miss rate, pipeline stalls, instruction mix, memory references, etc.) is extracted from application programs. Then, mixed integer linear programming and heuristic rules are used to gradually transform a generic program template to into a fully functional program. The synthesized program exhibits the same performance and power dissipation behavior (as characterized by the extracted profile), yet it has an instruction trace orders of magnitude smaller than the initial trace. The synthesized program is subsequently simulated on a register-transfer level description of the target microprocessor to provide the power dissipation value. Results obtained for the Intel's Pentium processor executing standard benchmark programs show a simulation time reduction by 3-5 orders of magnitude.

1. Introduction

The market demand for high-performance mobile computer systems is increasing rapidly. During the system design and planning of such systems, power dissipation is an important design concern. An efficient and accurate power estimation tool can be thus very useful during this early design stage.

The first task in the estimation of typical power consumption of computer systems is to identify the typical application programs that will be executed on these computer systems. A non-trivial application program consumes millions of machine cycles, making it almost infeasible to perform power estimation using the complete program at, say, the RT-level.

The previous works are mainly based on power macro-modeling approaches. In [1], the power cost of a CPU module is characterized by estimating the average capacitance that would switch when the given CPU module is activated. In [2], the switching activities on (address, instruction, and data) buses are used to estimate the power consumption of the microprocessor. In [3], an instruction level energy dissipation model is proposed, which consists of three terms: The first term is the summation of base energy cost of each instruction times the number of times each instruction is executed. The second term accounts for the circuit state (temporal correlation between instructions) whereas the third term accounts for energy contribution of effects such as stalls and cache misses during the program execution.

In this paper, we present a new approach, i.e. *profile-driven program synthesis*, to enable the RT-level power estimation for

*This research was supported in part by DARPA under contract F33615-C1627, SRC under contract 97-DJ-559, and a grant from Intel Corporation.

Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and /or a fee.

DAC 97, Anaheim, California
(c) 1997 ACM 0-89791-920-3/97/06 ..\$3.50

high performance CPUs. Instead of using a macro-modeling equation to model the energy dissipation of a microprocessor, we use synthesized program to exercise the microprocessor in such a way that the resulting instruction trace behaves similarly (in terms of performance and power dissipation) to the original trace. The new instruction trace is much shorter than that of the original one. The advantages of the proposed approach are two folds:

- 1) It is more accurate than the macro modeling approaches since it does not rely on simple macro-model equations for power prediction. In addition, it does not require macro-model calibration, which is a difficult and error-prone process.
- 2) It is more efficient than direct RT-level simulation since the synthesized program has a much shorter instruction trace.

A similar idea has been applied to circuit level power estimation. Input sequence compression approaches are proposed in [4][5]. The input vector sequence is first analyzed and its bit-level statistics are extracted. Then, a compression program synthesizes a new vector sequence that is shorter than the original sequence by orders of magnitude and exhibits the same statistics as the original sequence. Reported results indicate that the compressed input sequence predicts power dissipation of the original sequence within 5% error.

In this paper, we consider microprocessors with super-scalar pipelined architecture. For more advanced architectures (such as VLIW), more constraints need to be incorporated into the synthesis process.

The remainder of this paper is organized as follows. Section 2 gives an overview of the proposed methodology and some terminology and definitions. Section 3 enumerates the list of characteristics of interest for power estimation in microprocessor chips. Section 4 describes the synthesis procedure in details. Experimental results and conclusion are given in Section 5 and 6.

2. Overview and Background

Instruction trace is defined as the sequence of instructions in a program as that they are executed by an equivalent scalar processor. The instruction trace has become the major tool for performance simulation. For instance, the instruction trace can be fed to a trace-driven simulator to collect the instruction cache miss rate, data cache miss rate, etc. The length of the instruction trace (*trace length*) is the number of instructions executed in the program. The *input trace* refers to the instruction trace of input program that must be profiled. The *output trace* refers to the instruction trace of the synthesized program. The *compression ratio* equals to the ratio of output trace length (L_1) over the input trace length (L_0). The *synthesis quality* is a measure how well the power and performance properties are preserved in the output trace with respect to the input trace. The profile-driven trace synthesis problem is thereby defined as follows: *synthesize a new program by a given compression ratio and with highest synthesis quality*. The synthesized program should be a valid program that runs correctly on the target microprocessor.

The above problem is solved in two steps:

- 1) **Profile collection:** Extract characteristics of the initial trace that determine its performance and power dissipation behavior.

The set of relevant characteristics includes instruction mix, branch prediction miss rate, pipeline usage, data dependencies, and memory reference, etc. This set is referred to as the *characteristic set*.

2) **Program Synthesis:** Synthesize a new program that matches the extracted characteristics profile while satisfy the compression ratio constraint.

Step (1) is accomplished by simulating the whole instruction trace at the micro-architectural level using an architectural simulator.

Step (2) is solved by partitioning the characteristic set into four sub-profiles. These sub-profiles are disjoint, yet collectively constitute the whole set. Characteristics within each subset are then captured by gradual refinement of a generic program template during four phases: block allocation, instruction allocation, memory allocation, operand assignment and instruction scheduling.

A program consists of a *code session* and a *data session*. The code session can be decomposed into a set of basic blocks. A *basic block* is a sequence of consecutive instructions in which control flow of the program enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The *entry point* of a basic block is the position immediately before the first instruction in the block; the *exit point* of a basic block is then defined as the position immediately after the last instruction in the block. The *block length* refers to the number of instructions in the basic block. We build our synthesis procedure upon the basic block structure.

Definition: A program *control flow graph (CFG)* is defined as a directed graph $G(B,E)$ where B is set of basic blocks: $\{b_1, b_2, b_3, \dots, b_n\}$; b_1 is the *entry block* and b_n is the *exit block*; and E is set of directed control flow edges, $\{(u,v) \mid \text{if there is possible control flow from } u \text{ to } v\}$. The program execution starts from the entry point of the b_1 and ends at the exit point of b_n .

Not every *CFG* can be mapped to a code session. This is because the code session must reside in the memory before it can be executed. The process of mapping *CFG* into the linear memory space is referred to as *embedding*. After embedding, each basic block in the *CFG* is labeled with its beginning address; the new *CFG* is called an *embedded CFG*. Let *taken(b)* be the next block to be executed if the branch in b is taken; *not-taken(b)* is defined similarly. Given a *CFG* $G(B,E)$, for any block b in B , the *not-taken(b)* must be immediately placed after b in the memory space if *not-taken(b)* exists. Any *CFG* that satisfies the above constraint is said to be *feasible*.

Proposition 2.1: Let $G'(B,E')$ be a *CFG* derived from *CFG* $G(B,E)$, and E' is derived from E by removing all the edges except $(b, \text{not-taken}(b))$ edges from E . If G' is an *acyclic* graph, G is *feasible*.

Control sequence of a basic block is defined as the trace of the branch evaluation result at the exit point of the basic block when the *CFG* is simulated. *Concatenation* of two *CFG*'s, $G_1(B_1, E_1)$ and $G_2(B_2, E_2)$, is a new *CFG* $G(B,E)$ where $B = B_1 \cup B_2$, and $E = E_1 \cup E_2 \cup (\text{exitblock of } B_1, \text{entryblock of } B_2)$. Concatenation of multiple *CFG*'s is obtained by repeated concatenation of two *CFG*'s until only one *CFG* is left.

3. Characteristic Set

This paper focuses on the super-scalar pipelined architecture, which represents a wide range of today's high performance microprocessors. We will assume that the microprocessor has the following features: separate data and instruction set-associated

caches, a branch target buffer (branch prediction), and multiple instruction issues in one cycle.

The following characteristic set is used as the characteristic profile: *branch prediction miss, instruction cache miss rate, data cache read miss rate, data cache write miss rate, pipeline stall rate, clock per instruction (CPI), instruction mix, and average instruction size* (if applicable).

Most of the above characteristics impact both performance and power dissipation. Performance metrics are important because each time the microprocessor encounters a situation that leads to a performance penalty, the CPU has to work harder to recover the lost performance, which means it will have to consume more energy.

4. Synthesis Procedure

The following is the a summary description of the four synthesis phases:

1. **Block allocation:** An embedded *CFG* is generated to match the characteristic sub-profile: branch prediction miss rate, and instruction cache miss rate.
2. **Instruction allocation:** Set of instructions is allocated for each basic block in the *CFG*. Only the opcode of these instructions are fixed in this phase; the operands and the order of instructions will be determined in later phases. The characteristic sub-profile of this phase is: instruction mix.
3. **Memory allocation:** Memory operands of instructions are determined to match the characteristic sub-profile: data cache read miss rate, and data cache write miss rate.
4. **Operand assignment and instruction scheduling:** Registers, immediate operands, and the order of instructions within each basic block are assigned to match the metrics: pipeline stalls, CPI, and average instruction length(if applicable).

We will use the following notation in the remainder of this paper.

BL	Average block length of the input trace
LS	Line size of the instruction cache
$iCache_{miss_rate}$	Cache miss rate of the input trace
BP_{miss_rate}	Branch prediction miss rate of the input
$dCache_read_{miss_rate}$	Data cache read miss rate of the input
$dCache_write_{miss_rate}$	Data cache write miss rate of the input

Variable names with primes refer to the corresponding statistics for the output trace. For example, $iCache'_{miss_rate}$ denotes the cache miss rate of the output trace. The average block length, BL , is defined as $1/(\text{fraction of branch instructions in input trace})$.

Control Types	Control Sequences
type 1: alternating taken/not-taken	taken, not-taken, taken, not-taken, ...
type 2: always taken	taken, taken, taken, ..., taken
type 3: never taken	not-taken, not-taken, not-taken, ..., not-taken
type 4: loop back with loop count k	taken repeated k times, followed by not-taken

Table 1 Regular control sequences

4.1 Assumptions and Constraints

In this section, we state our assumptions and constraints, which have enabled us to solve the synthesis problem.

We limit the number of target addresses of a branch instruction to one. This excludes indirect branch with possible multiple targets,

but it does not restrict our ability to match the given characteristic profile.

The control sequences that are used during in the program synthesis are listed in table 1. They are regular and periodic, and easy to analyze and synthesize. In addition, experimental results show that the synthesis quality is not compromised.

A basic block is said to be *type i* if it has a control sequence of type *i*. To make equations more readable, we assume that loop count in type 4 is always an *even* number.

A CFG $G(B,E)$ is a *loop-back CFG* if all the blocks in the CFG are type 1 to type 3 and the exit block is type 4 with branch targeting the entry point of the entry block. The *period* of the *loop-back CFG* is defined as the minimum number of loop iterations such that the block sequence will repeat itself. For example, the block sequence of the CFG in figure 2(a) is A,C (iteration 1), A,B,C (iteration 2), A,C, (iteration 3), A,B,C, Its period is two. Only loop-back CFG's with a period of at most two are used in our synthesis procedure.

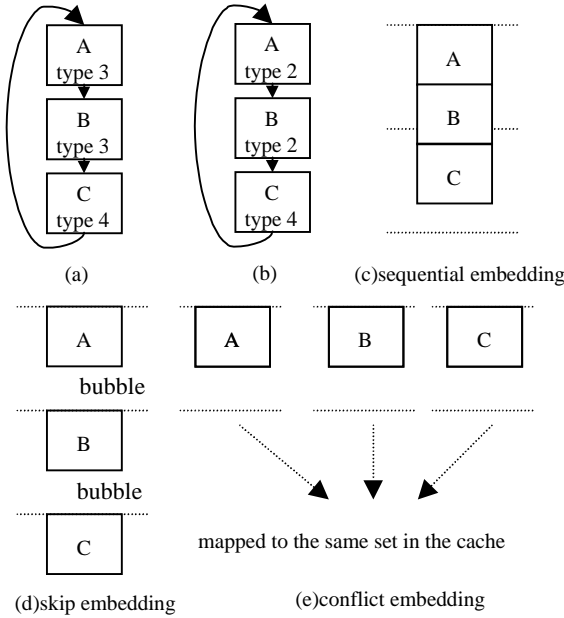


Figure 1. CFG examples

4.2 Block Allocation

The block allocation is done in two steps: design set of CFG templates (macro block templates) as building blocks, then formulate the block allocation problem as an ILP problem.

4.2.1 Macro Block Templates

The number of instruction cache misses of the synthesized program is given by:

$$\frac{\text{output trace size} + \text{bubble}}{LS} + \text{conflict} \quad (1)$$

where *bubble* is the number of bytes fetched by instruction cache, but never executed; *conflict* is the number of cache lines replaced and revisited (thrashing).

If a basic block *b* has only one immediate predecessor *b'* and $\text{taken}(b')$ equals *b*, then *b* is defined as *free*.

The (memory) embedding methods are classified as follows:

- *Sequential embedding*: The basic block *b* is put in memory immediately before the basic block *not-taken(b)*.

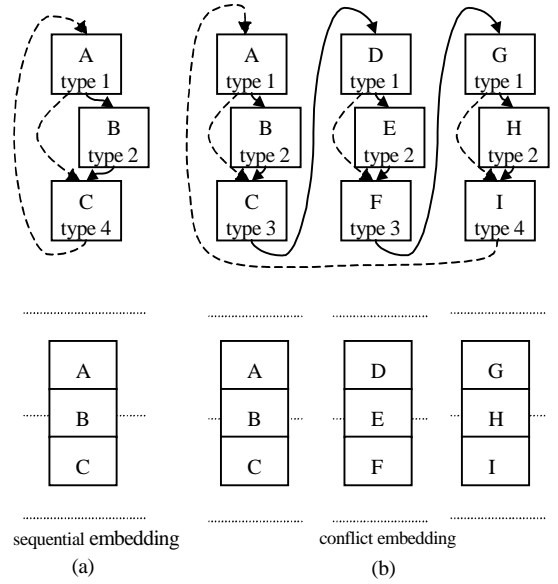


Figure 2. Macro block templates (dash edge denotes taken branch)

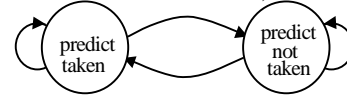


Figure 3 Branch prediction

- *Skip embedding*: Free basic blocks are assigned to memory locations that are not immediately adjacent to other basic blocks. The purpose is to increase *bubble* in Equation (1).
- *Conflict embedding*: Several *free* basic blocks are mapped to the same cache block. The purpose is to increase *conflict* in Equation (1).

For instance, the loop-back CFG in figure 1(a) can only be embedded sequentially as shown in figure 1(c) because of feasibility constraint. In contrast, the loop-back CFG in figure 1(b), can be embedded using skip embedding as in figure 1(d), or conflict embedding as in figure 1(e). Dash lines denote the cache line boundaries.

A *macro block* is a *feasible loop-back CFG* with annotated embedding method and with period at most two.

Proposition 4.1: Concatenation of macro blocks is feasible.

A *macro block template* is a predefined structure for building macro blocks by providing the following:

- a template for a feasible CFG with period at most two
- embedding methods for all the basic blocks in CFG
- average number of blocks *NB* being executed per loop
- a function returning average number of instruction cache lines fetched per loop iteration $Lines(lp_cnt) = C_2 \cdot lp_cnt + C_1$
- a function returning average number of branch prediction miss per loop iteration $BP_{miss}(lp_cnt) = C_4 \cdot lp_cnt + C_3$

lp_cnt (loop count) is a parameter of the macro block template. $BP_{miss}(lp_cnt)$ and $Lines(lp_cnt)$ are linear functions of lp_cnt because of periodic behavior of the CFG. Constants: C_1 , C_2 , C_3 , and C_4 are derived by the specified embedding methods and by the control types of the basic blocks in the CFG.

The following macro block template examples assume that instruction cache line size is 32 bytes, instruction size is 4 bytes, every basic block has four instructions, and the branch prediction scheme is as shown in figure 3.

For figure 2(a), sequential embedding is used:

$$NB = 2.5, Lines(lp_cnt) = 1.5, BP_{miss}(lp_cnt) = lp_cnt + 2$$

For figure 2(b), conflict embedding is used:

$$NB = 7.5, Lines(lp_cnt) = 6 \cdot lp_cnt, \\ BP_{miss}(lp_cnt) = 3 \cdot lp_cnt + 2$$

4.2.2 MILP Formulation

The block allocation problem is solved by concatenating macro blocks from the set of predefined macro block templates to minimize the following objective function subject to the trace length constraint.

$$\left| BP_{miss_rate} - BP'_{miss_rate} \right| + \left| iCache_{miss_rate} - iCache'_{miss_rate} \right|$$

We cast the problem as an integer linear programming problem and solve it using a MILP solver.

The objective function can be restated as follows:

$$X_1 + X_2 + Y_1 + Y_2$$

subject to the following constraints:

$$iCache_{miss_rate} - iCache'_{miss_rate} + X_1 - X_2 = 0 \quad (2)$$

$$BP_{miss_rate} - BP'_{miss_rate} + Y_1 - Y_2 = 0 \quad (3)$$

$$X_1, X_2, Y_1, Y_2 \geq 0$$

Let m denote the number of macro block templates. Subscript i on a variable or a function shows that it is associated with the i th templates. Let n_i be the number of macro blocks formed by the i th template; $lp_cnt_{i,j}$ denotes the loop count of the j th macro block of the i th template.

The total number of blocks (TB) in the output trace is given by:

$$TB = \sum_{i=1}^m \sum_{j=1}^{n_i} lp_cnt_{i,j} \cdot NB_i$$

The output trace length must satisfy the following:

$$min_length \leq TB \cdot BL \leq max_length$$

where min_length and max_length are used to provide a tolerance range around target output trace length L_1 . The extra freedom for output trace length often leads to better results than a fixed length constraint.

The $iCache'_{miss_rate}$, and BP'_{miss_rate} are calculated by:

$$iCache'_{miss_rate} = \frac{1}{TB \cdot BL} \sum_{i=1}^m \sum_{j=1}^{n_i} Lines_i(lp_cnt_{i,j}) \\ = \frac{1}{TB \cdot BL} \sum_{i=1}^m \sum_{j=1}^{n_i} C_{2,i} \cdot lp_cnt_{i,j} + \sum_{i=1}^m C_{1,i} \cdot n_i \quad (4)$$

$$BP'_{miss_rate} = \frac{1}{TB} \sum_{i=1}^m \sum_{j=1}^{n_i} BP'_{miss,i}(lp_cnt_{i,j}) \\ = \frac{1}{TB} \sum_{i=1}^m \sum_{j=1}^{n_i} C_{4,i} \cdot lp_cnt_{i,j} + \sum_{i=1}^m C_{3,i} \cdot n_i \quad (5)$$

Let $lp_cnt_i = \sum_{j=1}^{n_i} lp_cnt_{i,j}$. Note that lp_cnt_i and n_i must satisfy

$$lp_cnt_i \geq 2 \cdot n_i \text{ to have a feasible solution.}$$

$$\text{Let } X'_1 = TB \cdot X_1, X'_2 = TB \cdot X_2, Y'_1 = TB \cdot Y_1, \text{ and } Y'_2 = TB \cdot Y_2.$$

The objective function becomes $(X'_1 + X'_2 + Y'_1 + Y'_2) / TB$

Because TB is bounded by tight trace length constraint:

$$\frac{min_length}{BL} \leq TB \leq \frac{max_length}{BL}, \text{ it is nearly constant and can}$$

be thus removed from the objective function.

Then, objective function is approximated by:

$$X'_1 + X'_2 + Y'_1 + Y'_2.$$

Substituting Equations (4) and (5) into Equations (2) and (3):

$$\sum_{i=1}^m (iCache_{miss_rate} \cdot NB_i - C_{2,i} / BL) \cdot lp_cnt_i \\ + \sum_{i=1}^m \frac{C_{1,i} \cdot n_i}{BL} + X'_1 - X'_2 = 0 \quad (6)$$

$$\sum_{i=1}^m (BP_{miss_rate} \cdot NB_i - C_{4,i}) \cdot lp_cnt_i \\ + \sum_{i=1}^m C_{3,i} \cdot n_i + Y'_1 - Y'_2 = 0 \quad (7)$$

Finally, the block allocation is formulated as integer linear programming problem as following:

$$\mathbf{min} \quad X'_1 + X'_2 + Y'_1 + Y'_2$$

subject to

Equations (6) and (7)

$$lp_cnt_i \geq 2 \cdot n_i \quad 1 \leq i \leq m$$

$$lp_cnt_i = 2 \cdot w_i \quad 1 \leq i \leq m$$

$$min_length \leq TB \cdot BL \leq max_length$$

$$n_i, w_i \in Z^+$$

$$X'_1, X'_2, Y'_1, Y'_2 \geq 0$$

Once the MILP is solved, the loop count ($lp_cnt_{i,j}$), block length, and embedding for each basic block can be decided efficiently.

4.3 Instruction Allocation

The instruction allocation phase has two goals: to allocate instructions to form the control flow specified in previous phase, to match the instruction mix of the input trace. Hence, the instruction allocation is divided into two steps: *control-related* allocation and *control-unrelated* allocation.

Assume there are p instructions, I_1, I_2, \dots, I_p in the instruction set. $Freq(I_i)$ and $Freq'(I_i)$ are the execution frequency of the instruction I_i in the input program and the synthesized program, respectively. The goal is to minimize the objective

$$\text{function: } \sum_{i=1}^p |Freq(I_i) - Freq'(I_i)|.$$

Instruction grouping is the process of grouping instructions with similar energy cost, same performance behavior, and same number of memory (register) read (write).

We will show the advantage of instruction grouping by the following example. Assume the following:

- “**add** r1,r2->r3” has similar energy cost with “**sub** r1,r2->r3”
- “**and** r1,r2->r3” has similar cost with “**or** r1,r2->r3”.

The instruction mix of input trace is as in table 2. Assume there are only four instruction slots, and they have equal execution frequencies. Then, the straightforward allocation will be:

“**mov** reg->mem”, “**mov** mem->reg”, “**add** r1,r2->r3”, and “**sub** r1,r2->r3”.

However, if we group instructions with similar energy cost together:

- “**add** r1,r2->r3” and “**sub** r1,r2->r3”
- “**and** r1,r2->r3” and “**or** r1,r2->r3”

The following four instructions will be allocated instead:

“**mov** reg,mem”, “**mov** mem,reg”, “**add** r1,r2->r3”, and “**and** r1,r2->r3”.

```

#mem1 is initialized to loop_count
mov  mem1->r1
label:
...
...
dec  r1
jnz  label

```

Figure 4(a) Type 4 control

```

# mem2 is initialize to 1
not  mem2->mem2
jnz  label
...
...
...
label:

```

Figure 4(b) Type 1 control

Instruction	Freq
add r1, r2->r3	15%
sub r1, r2->r3	15%
and r1, r2->r3	10%
or r1, r2->r3	10%
mov reg, mem	25%
mov mem, reg	25%

Table 2. Instruction mix

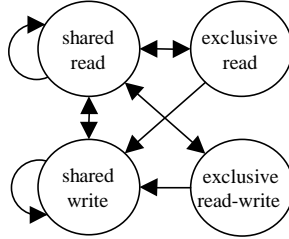


Figure 5. Compatibility graph

The second sequence is obviously more representative of the instruction profile given in the table 2. We conclude that instruction grouping is necessary and helpful.

Let the instruction set grouped into k groups, G_1, G_2, \dots, G_k . The $Freq(G_i)$ and $Freq'(G_i)$ denote the execution frequency of group G_i in the input program and the synthesized program, respectively.

The objective function becomes two levels:

1. $\min |\sum Freq(G_i) - Freq'(G_i)|$
2. $\min |\sum Freq(I_i) - Freq'(I_i)|$

The first level objective function has higher priority than the second level objective function.

Control-related allocation is based on the instruction templates for each control type. Two template examples are shown in figure 4. A force-directed scheduling algorithm [6] is used to select a proper template among the set of pre-defined instruction templates for each basic block. Then, a greedy algorithm, similar to list scheduling [7], is used to allocate *control-unrelated* instructions while minimizing the above objective function.

4.4 Memory Allocation

According to the control flow requirement, the memory accesses for instructions in a basic block are classified into one of the four types: *exclusive read*, *exclusive read-write*, *shared read*, and *shared write*. For example, the instruction "mov mem1->r1" in figure 4(a) is initialized to the loop count. The contents of mem1 must not be changed before or during the loop of the macro block where it resides. In this case, mem1 is labeled as exclusive read. On the other hand, the instruction "not mem2->mem2" in figure 4(b) is used to flip the content of mem2 such that the branch sequence is alternately taken. To avoid the possibility of changing the content of mem2 during its lifetime, it needs to be marked as exclusive read-write. All other memory accesses are marked shared read or shared write. The shared read means that the data read by the instruction is undefined and can assume any value; the shared write is defined similarly.

The purpose of memory allocation is to allocate the memory space for each memory access and decide whether it is vector access or scalar access.

The objective is to minimize the cost function:

$$|dCache_read_{miss_rate} - dCache_read'_{miss_rate}| + |dCache_write_{miss_rate} - dCache_write'_{miss_rate}|$$

An instruction with scalar access will always access the same memory location; an instruction with vector access will change its memory location every time. The exclusive read and exclusive write can only be scalar because they are responsible for generating the control sequence, whereas shared read and shared write can be either scalar or vector. The vector access is mainly used to create more cache misses. It can be implemented by taking loop counter as the offset of referenced memory address.

Let mem_1 and mem_2 be memory accesses. If mem_1 is accessed before mem_2 in the execution trace of its CFG, then we say that mem_1 leads mem_2 . Otherwise, mem_1 trails mem_2 . The order within the same macro block is not important, because all the memory accesses in the same macro block are both leading and trailing each other (due to loop-back).

Let $type(mem_i)$ be the memory access types of mem_i . Two memory accesses, mem_1 and mem_2 are compatible with each other if they satisfy both of the following rules:

1. If mem_1 leads mem_2 , edge $(type(mem_1), type(mem_2))$ exists in the compatibility graph (c.f. figure 5).
2. If mem_1 trails mem_2 , edge $(type(mem_2), type(mem_1))$ exists in the compatibility graph.

Compatibility criterion: The memory spaces of mem_1 and mem_2 can overlap only if they are compatible.

The compatibility criterion must be satisfied during the memory allocation phase. Our greedy algorithm first finds an initial allocation with minimum cache miss rates by using the following rules:

- Scalar reference: Assign the memory accesses as scalar accesses.
- Word level packing: Pack all the compatible memory accesses into the same memory location.
- Block level packing: Pack as many memory accesses as possible into the same memory block (cache line).

Next, it minimizes the objective function by gradually increasing the cache miss rates. The process continues until no further improvement in the objective function is possible. The following rules are used to increase the cache miss rates:

- Word level expansion: Unpack the compatible memory accesses into different memory locations.
- Block level expansion: Scatter the memory accesses within one memory block to different memory blocks.
- Array reference: Change the scalar memory accesses to vector accesses.
- Conflict mapping: Map the memory blocks into the same cache set to cause thrashing.

4.5 Operand Allocation / Instruction Scheduling

In this phase, we try to match the pipeline stall rate and CPI in the input instruction trace. First, we assign the instruction operands and schedule the instruction sequence in such a way that the pipeline stall rate and CPI are minimal. Instruction pipeline simulator is used to calculate both pipeline stall rate and CPI during the whole synthesis phase for accuracy. Then, architecture dependent rules (data dependency, resource conflict, etc.) are used to increase the pipeline stalls and CPI gradually until both of them are matched. If the target microprocessor is a CISC CPU with variable instruction length, the average instruction size is matched by a similar technique.

5. Experimental Results

To verify the effectiveness of the proposed approach, we must use benchmark programs to perform two sets of experiments:

- 1) Power evaluation by RT-level simulation of the program generated by profile-driven synthesis.
- 2) Power evaluation by direct RT-level simulation of the input program.

Two sets of power dissipation values are then compared to verify the synthesis quality. In our experiments, we use Intel Pentium™ processor as our target microprocessor. There are two advantages:

1. The Pentium™ processor falls in our target microprocessor class (super-scalar pipelined CPU). It has 8KB 2-way set-associative data and instruction caches, branch prediction, and dual instruction pipeline [8].
2. Instead of using RT-level simulation, we are able to directly run benchmarks and measure the current on the chip, which is much faster and more accurate. This enables us to test on larger benchmark instead of small test program.

Eight integer SPEC95 programs are used as our benchmark. Instruction traces of these programs are first captured. Next, an architectural simulator analyzes these traces and extracts the characteristic profiles. The collected profiles are then fed into our profile-driven synthesis engine to synthesize programs by using seven macro block templates. The traces of these synthesized programs are smaller than the input traces by 3-5 order of magnitude as shown in chart 1.

Experimental data is collected on the Pentium processors running at 90MHZ with 2.9V power supply. Table 3 shows the synthesis qualities. The entries in table 3 correspond to percentage error between the parameter (for example, *iCache hit rate*) in the original program and the synthesized program. The last row of table gives the average percentage error over all benchmarks. Chart 2 shows the energy per instruction of the original program (*EPI*) and the energy per instruction of the synthesized program (*EPI'*). The average error of *EPI'* is 2.9%, which shows the accuracy of proposed profile-driven program synthesis.

6. Conclusion

In this paper, we presented a new approach for evaluating the power dissipation of application programs running on a high performance microprocessor. The first step of this approach was to identify a characteristic set, which is sufficient to capture the microprocessor power and performance behavior. Next, a trace-driven architecture simulation was run through the instruction trace to collect the characteristic profile. MILP and heuristic rules were used to transform predefined macro block templates into full functional program after four synthesis phases. Experimental results showed the effectiveness of the proposed approach on Intel Pentium processor running SPEC int 95 benchmarks.

References

- [1] Toshinoriato, Yukio Otaguro, Masato Nagamatsu, and Haruyuki Tago. "Evaluation of architecture-level power estimation for CMOS RISC processors". In *Proceedings of the Symposium on Low Power Electronics*, pp. 44-45, 1995
- [2] Chih-Log Su, Chih-Ying Tsui, and Alvin M. Despain. "Low power architectures design and compilation techniques for high performance processors." In *Proceedings of IEEE COMPCON*, pp. 489-498, 1994
- [3] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. "Instruction level power analysis and optimization of software". In *Journal of VLSI Signal Processing*, Aug/Sept, 1996.
- [4] Chi-ying Tsui, Radu Marculescu, Diana Marculescu, Massoud Pedram, "Improving the efficiency of power simulators by input vector compaction". In *Proceedings of Design Automation Conference*, pp.165-168, 1996
- [5] Diana Marculescu, Radu Marculescu, Massoud Pedram, "Stochastic Sequential Machine Synthesis Targeting Constrained Sequence Generation", In *Proceedings of Design Automation Conference*, pp.696-701, 1996
- [6] Giovanni De Micheli, "Synthesis and Optimization of Digital Circuits", *Mc-Graw Hill*, pp.211-216, 1994
- [7] Giovanni De Micheli, "Synthesis and Optimization of Digital Circuits", *Mc-Graw Hill*, pp.207-211, 1994
- [8] Intel Corporation, "Pentium Processor Family Developer's Manual", *Volume 1: Pentium Processors*, 1996

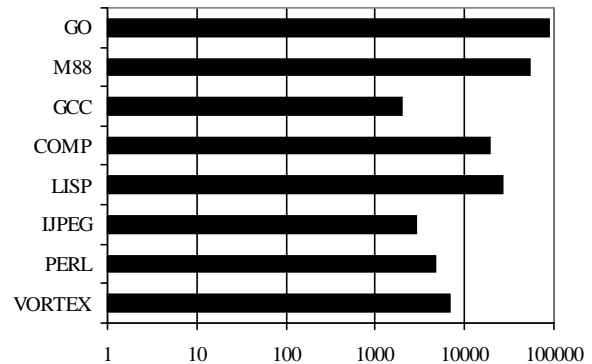


Chart 1. Compression ratio

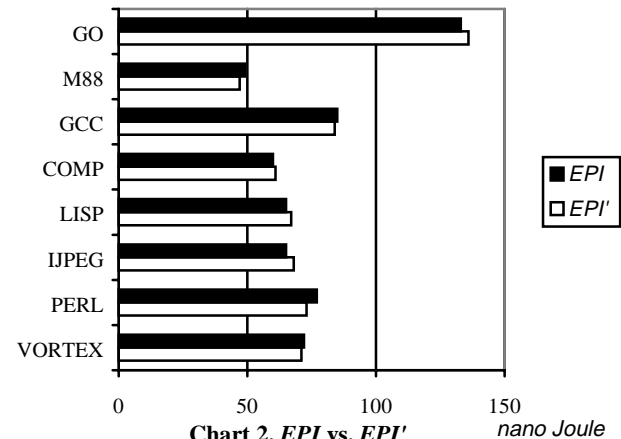


Chart 2. EPI vs. EPI'

	<i>iCache hit rate</i> err %	<i>BP hit rate</i> err %	<i>dCache read hit rate</i> err %	<i>dCache write hit rate</i> err %	<i>CPI err %</i>
GO	0.9	2.4	0.2	0.4	1.0
M88K	0.2	0.8	0.6	1.0	0.9
GCC	1.2	0.5	0.05	0.7	0
COMP	0.1	0.2	2.9	0.4	2.9
LISP	0.2	0.6	0.1	1.2	4.5
IJPEG	0.01	0.2	0.1	0.1	2.9
PERL	1.2	1.1	0.04	0.3	1.7
VORTEX	0.5	0.1	0.1	3.9	3.8
Ave Err %	0.54%	0.74%	0.51%	1.0%	2.2%

Table 3. Experimental Result

* Pentium is the trademark of Intel Corporation.