Micro Processor Power Estimation Using Profile-Driven Program Synthesis

Cheng-Ta Hsieh and Massoud Pedram

CENG 98-19

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213-740-4458
August 1998

Micro-Processor Power Estimation Using Profile-Driven Program Synthesis

Cheng-Ta Hsieh
chengtah@zugros.usc.edu

Massoud Pedram
massoud@zugros.usc.edu

Department of Electrical Engineering – System
University of Southern California
Los Angeles, California

Abstract *– This paper presents a new approach for estimating power dissipation in a high performance microprocessor chip. A characteristic profile (including parameters such as the cache miss rate, branch-prediction miss rate, pipeline stalls, instruction mix, and so on) is first extracted from the application programs. Mixed integer linear programming and heuristic rules are then used to gradually transform a generic program template into a fully functional program. The synthesized program exhibits the same characteristics (and hence the same performance and power dissipation behavior), yet it has an instruction trace which is 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 planning and design of such systems, power dissipation of the microprocessors is an important design concern. An efficient and accurate power estimation tool can be very useful during this early design stage.

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

The previous works are 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 model is proposed as shown below:

$$E_p = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k$$

where E_p is the total energy dissipation of the program which is divided into three parts: The first part is the summation of base energy cost of each instruction, (B_i is the base energy cost and N_i is the number of times instruction i is executed). The second part takes the circuit state into account ($O_{i,j}$ is the energy cost when instruction i is followed by j during the program execution), while the third part accounts for energy contribution of other instruction effects such as pipeline stalls and cache misses during the program execution.

Profile Driven Program Synthesis, presented in this paper, is a new approach for performing RT-level power estimation of high performance CPUs. Instead of using a macro-modeling equation to model the energy dissipation of a microprocessor, a synthesized program is used to exercise the microprocessor in such a way that the resulting instruction trace behaves in the same way (in terms of performance and power dissipation) as the original trace. The new instruction trace is, however, much shorter than the original trace (see Figure 1). The advantages of the proposed technique are twofold:

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. A compression program, then, 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 in the range of a 5% error.

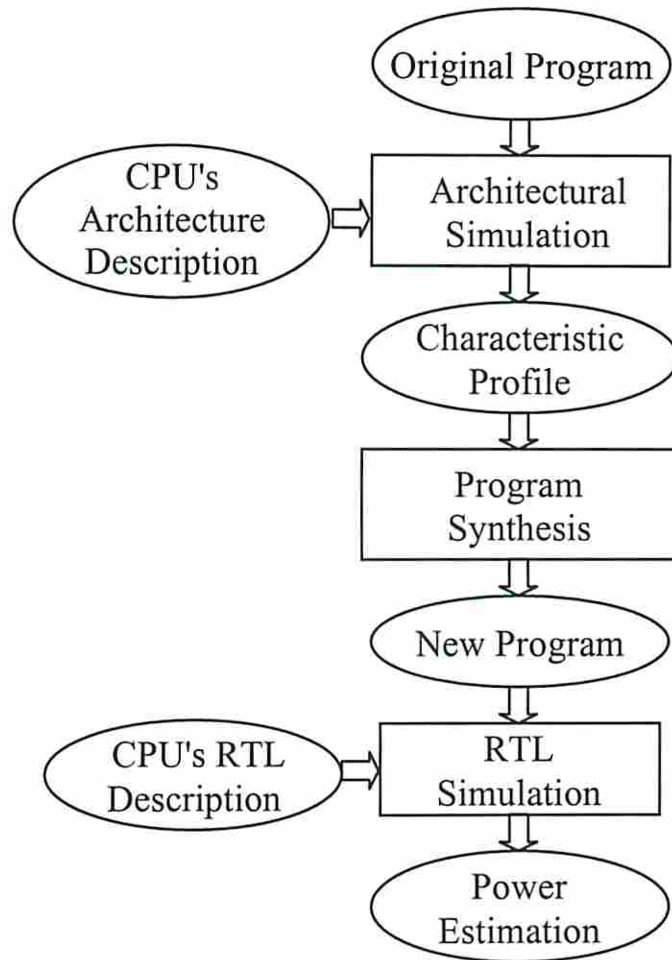


Figure 1. Power estimation procedure

We consider microprocessors with super-scalar pipelined architecture in this paper. For more advanced architectures (such as VLIW and speculative execution), a greater number of constraints needs to be incorporated into the synthesis process, although the program synthesis approach is still applicable.

The remaining sections are organized as follows: Section 2 gives an overview of the proposed methodology and some terminology and definitions. Section 3 summarizes characteristics of interest for power estimation in microprocessor chips. Section 4 describes the synthesis procedure in details. Experimental results and the conclusion are given in Sections 5 and 6.

2. Overview and Background

2.1 Overview

Instruction trace refers to the sequence of instructions that are executed when a program runs on a processor. The instruction trace has become the major means for performance evaluation of a processor. For instance, the instruction trace can be fed to a trace-driven simulator to collect performance characteristics such as instruction-cache miss rate, data-cache miss rate, and pipeline-stall rate. The length of the instruction trace (*trace length*) is the number of instructions executed during the program run.

In this paper, *input trace* refers to the instruction trace of the input program that is profiled. The *output trace* refers to the instruction trace of the synthesized program. The *compression ratio* equals the ratio of input trace length to the output trace length. The *synthesis quality* is a measure of how well the performance characteristics of the input trace are preserved in the output trace. The profile-driven program synthesis is then defined as follows: *synthesize a new program for a given compression ratio with the highest synthesis quality, or synthesize a new program with maximum compression ratio for a specified synthesis quality*. The synthesized program should be a valid program that runs properly on the target microprocessor.

This problem is solved in two steps:

1) **Profile collection**: Extract characteristics of the input program that determine its performance and power dissipation behavior. The set of relevant characteristics includes instruction mix, branch-prediction miss rate, pipeline usage, instruction/data cache miss rate, etc. This set is referred to as the *characteristic set*.

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

Step (1) is accomplished by simulating the input instruction trace at the micro-architectural level using an architectural simulator[1]. Step (2) is solved by partitioning the characteristic set into four sub-profiles. These sub-profiles are disjoint, yet collectively form the whole set. Characteristics within each subset

[1] Note that an architectural simulator is 10^4-10^6 times faster than RT-level simulator for a complex microprocessor.

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.

2.2 Background

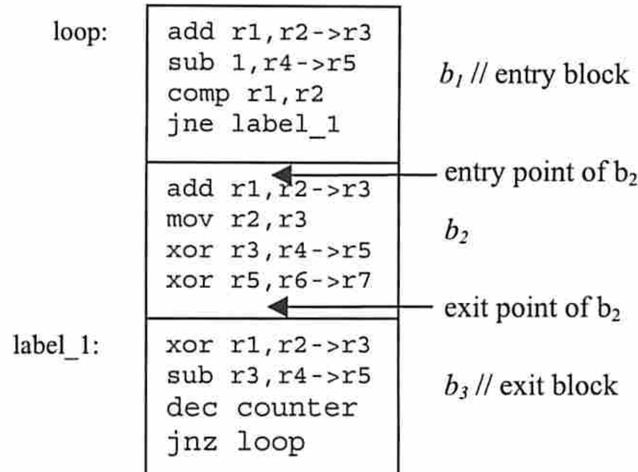
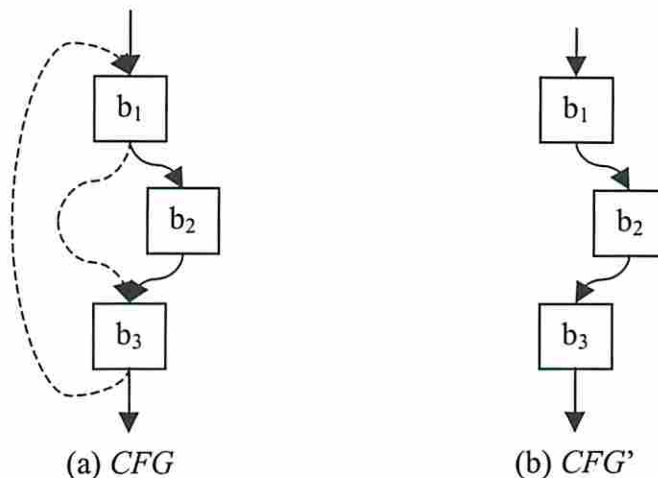


Figure 2. A simple program

A program can be decomposed into a set of basic blocks (c.f. Figure 2). A *basic block* is a maximal sequence of consecutive instructions in which the control flow of the program enters at the beginning and leaves at the end without stopping or the possibility of branching. 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 the position immediately after the last instruction in the block. The *block length* refers to the number of instructions in a basic block. We form our synthesis procedure on the basic block structure. Note that the branch instruction, if present, will be the last instruction in a basic block. If the last instruction is not a branch instruction (c.f. block b_2 in Figure 2), it can be thought as a branch instruction, with the caveat that the branch is never taken. In our synthesis procedure, however, we always put a branch instruction at the end of each synthesized basic block.

A branch is *taken* if the condition tested by the branch is true and the next instruction to be executed is the target of the branch, otherwise it is *not taken*. An unconditional jump, therefore, is always taken. Let *taken(b)* denote the basic block which is executed next if the branch in basic block b is taken; *not-taken(b)* is defined as the basic block which is executed next if the branch in b is not taken. For example, in Figure 2, the branch instruction in b_1 is "`jne label_1`", *taken*(b_1) is b_3, and *not-taken*(b_1) is b_2.

Definition 2.1: The *control flow graph (CFG)* of a program is defined as a directed graph *G(B,E)* where *B* is the set of basic blocks: $\{b_1, b_2, b_3, \ldots, b_n\}$ and *E* is the set of directed control flow edges. Let b_1 denote the *entry block* where the program starts and b_n be the *exit block* where the program exits.



(a) *CFG* (b) *CFG'*

Figure 3. Control flow graph

Figure 3(a) shows the *CFG* of the program in Figure 2. The dashed arrows denote the edges: {(*b, taken(b)*) | for any *b* in the *CFG*} whereas the solid arrows denote the edges: {(*b, not-taken(b)*) | for any *b* in the *CFG*}. This convention will be used throughout the paper.

Not every *CFG* can be mapped to a valid program. This is due to the fact that the program 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 the memory address of its entry point; the new *CFG* is called an *embedded CFG*.

Given a *CFG G(B,E)*, for any block *b* in *B*, the *not-taken(b)* must be placed immediately after *b* in the memory space unless *not-taken(b)* does not exist. Any *CFG* that satisfies the above (embedding) constraint is said to be *feasible*. For example, in Figure 2, *not-taken*(b_1) is b_2, thus b_2 must be placed immediately after b_1 such that it will be executed next after b_2 if "`jne label_1`" is not taken.

Definition 2.2: *G' (B,E')* is a CFG derived from *CFG G(B,E)*, where *E'* is derived from *E* by removing all the *(b, taken(b))* edges from *E*. *G' (B,E')* thereby captures all the embedding constraints.

Figure 3(b), for example, is the derived *CFG* from the *CFG* in Figure 3(a).

Fact 2.1: *G* is *feasible* if and only if *G'* is acyclic.

It is true because the *not-taken* edges should not form a cycle in a linear memory space.

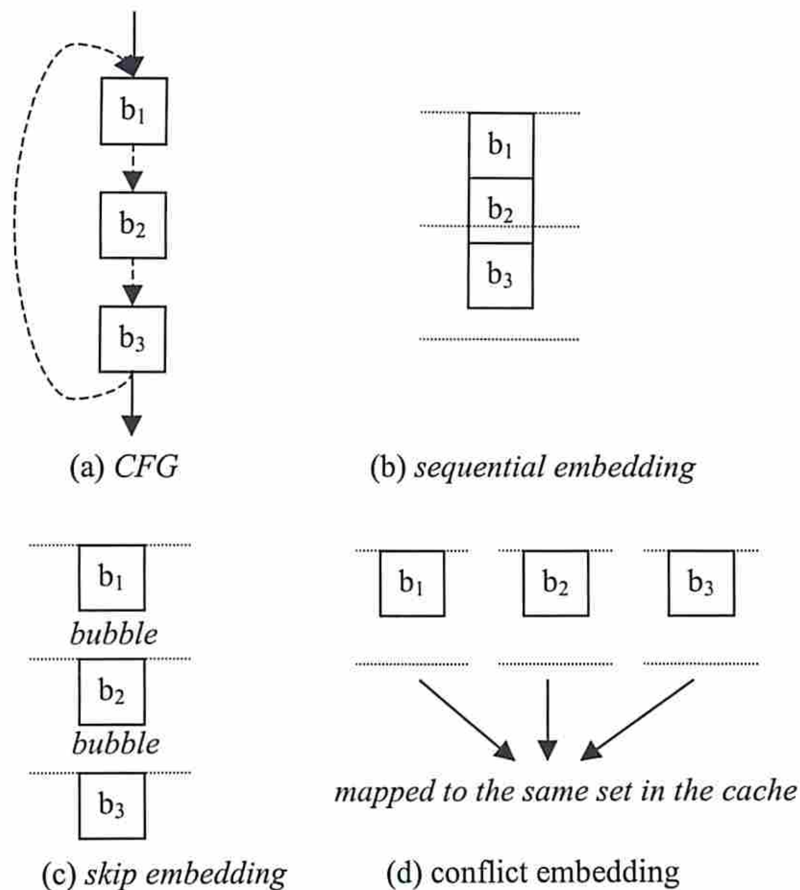


Figure 4. Embedding methods (dotted lines represent cache line boundaries)

If a basic block b has only one immediate predecessor b' and *taken(b')* equals b, then b is said to be a *free basic block*. Free basic blocks can be assigned to any location without violating the feasibility of the *CFG*. The instruction-cache miss rate of a feasible *CFG* is then determined by how the program is embedded in the main memory. Different embedding options are classified as follows:

- *Sequential embedding*: The basic block b is put in memory immediately before the basic block *not-taken(b)*.
- *Skip embedding*: Free basic blocks are assigned to memory locations that are not immediately adjacent to other basic blocks. This creates empty space (bubble) between blocks.
- *Conflict embedding*: Several free basic blocks are mapped to the same set in the cache.

For instance, the loop-back *CFG* in Figure 4(a) can be embedded by sequential embedding as shown in Figure 4(b), by skip embedding as in Figure 4(c), or by conflict embedding as in Figure 4(d).

Definition 2.3: 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* $E = E_1 \cup E_2 \cup \{(\textit{exit block of } B_1, \textit{entry block of } B_2)\}$.

Definition 2.4: Concatenation of multiple *CFG's* is obtained by repeated concatenation of two *CFG's* until only one *CFG* is left.

3. Characteristic Set

3.1 The Processor

This paper focuses on the super-scalar pipelined architecture, which represents a wide range of today's high performance microprocessors. A typical example is the Pentium Processor. We assume that the microprocessor has features that include separate data and instruction set-associative caches, a branch target buffer (branch prediction), and multiple instruction issues in one cycle [6].

3.2 The Characteristic Set

The characteristics in the characteristic set are chosen according to the following test:

1. The characteristic has a relatively large impact on performance or power dissipation on the micro-architecture of the CPU under estimation. For example, the branch-prediction misses cause pipeline flushes, which force the CPU to consume more power in order to bring in the target instruction stream.

2. The characteristic is significant in the input instruction trace. For example, the data-cache miss occurs frequently in most programs, and hence cannot be ignored. On the other hand, the TLB (translation lookaside buffer) miss rate is usually very small and can therefore be ignored.

To help decide the power impact of a characteristic, we identify the modules that are affected by the characteristic and then do the power estimation for each such module. The preliminary power estimation for modules is done by using standard RT-level or gate-level power estimation techniques [10][11]. Note that the accuracy of the power estimates obtained in this way may not be high, but is sufficient for identifying the power significance of the characteristics.

In case we have two or more characteristics that interfere with each other during same synthesis phase (c.f. Section 4.4.2 and Section 4.6), we can weight these characteristics by their early power estimates.

The following characteristic set is used as the characteristic profile for the target microprocessor:

- *branch-prediction miss rate*
- *instruction-cache miss rate*
- *instruction mix*
- *data-cache-read miss rate*
- *data-cache-write miss rate*
- *average instruction size (if applicable)*
- *mix of the immediate operand size (if applicable)*
- *clock cycles per instruction (CPI)*

4. Synthesis Procedure

The program synthesis is a very difficult task. Given an arbitrary computer program and an arbitrary input to that program, whether or not the program will eventually halt when it is applied to the input is undecidable [8]. This makes it impossible to verify the behavior of a random program. In this paper, we impose a set of rules and assumptions to make sure that the behavior of the synthesized program (such as whether or not it will halt) becomes easy to analyze. Specifically, we first summarize the four phases of our synthesis procedure in Section 4.1. The constraints, which are imposed on the *CFG* of the synthesized program to ensure that the control flow is deterministic, are given in Section 4.2. Note that the control flow of the synthesized program is specified in the first phase and is preserved in the remaining phases. The rationale for our 4-phase synthesis procedure is given in Section 4.3, followed by details of the four synthesis phases in Sections 4.4 to 4.7, respectively.

4.1 Synthesis Phases

The characteristic profile of the synthesized program is defined by its instruction trace. The goal of the synthesis procedure is to synthesize a new program whose instruction trace has the desired characteristic profile. It is difficult, however, and in some cases impossible, to derive a valid program corresponding to a random instruction trace. The safer approach is to synthesize the *CFG* first. In the synthesized *CFG*, we specify the exact manner in which the control flow transfers among various basic blocks. We refer to

this phase as the ***block allocation***. By simulating the synthesized *CFG*, we can then derive the instruction trace. For example, suppose there are two basic blocks: b_1 and b_2 in the synthesized *CFG*, and these two basic blocks are alternately executed three times when the *CFG* is simulated. The resulting instruction trace will be b_1,b_2,b_1,b_2,b_1,b_2. Note that both b_1 and b_2 represent sequences of instructions. In this phase, the block length for each basic block is also decided. After block allocation, the instruction-cache miss rate and the branch-prediction miss rate are fixed.

We also need to determine the instructions and their ordering in each basic block and the assignment of operands in each instruction. Both the operands to each instruction and the order of the instructions depend on the exact set of instructions allocated to each basic block, which is referred to as ***instruction allocation***. This indicates that the next phase immediately after the block allocation should be the instruction allocation. After instruction allocation, the instruction mix of the synthesized program is fixed.

There are three types of operands: memory, register, and immediate value. ***Memory assignment***, which refers to the assignment of the memory address for each memory operand, determines the data-cache-read/write miss rates. ***Operand assignment*** refers to assigning a physical register to each register operand and assigning a constant value to each immediate operand. ***Instruction scheduling*** refers to ordering the instructions within each basic block. As a result of operand assignment and instruction scheduling, the CPI, mix of immediate operand size, and the average instruction length are determined.

To obtain the relative ordering of the memory assignment, operand assignment, and instruction scheduling steps, consider the following dependencies:

- CPI and data-cache miss rate depend on the memory assignment.
- CPI and average instruction length depend on the operand assignment.
- CPI depends on the instruction scheduling.

Form these dependencies, we conclude that the instruction scheduling should be done last because it offers an opportunity for fine tuning the CPI once the memory assignment and operand assignment are completed. During memory assignment, registers can be used as memory indices to access elements of an array. Memory assignment should, therefore, be done before register assignment, which is part of the operand assignment. After the memory assignment, the operand assignment and instruction scheduling are performed next.

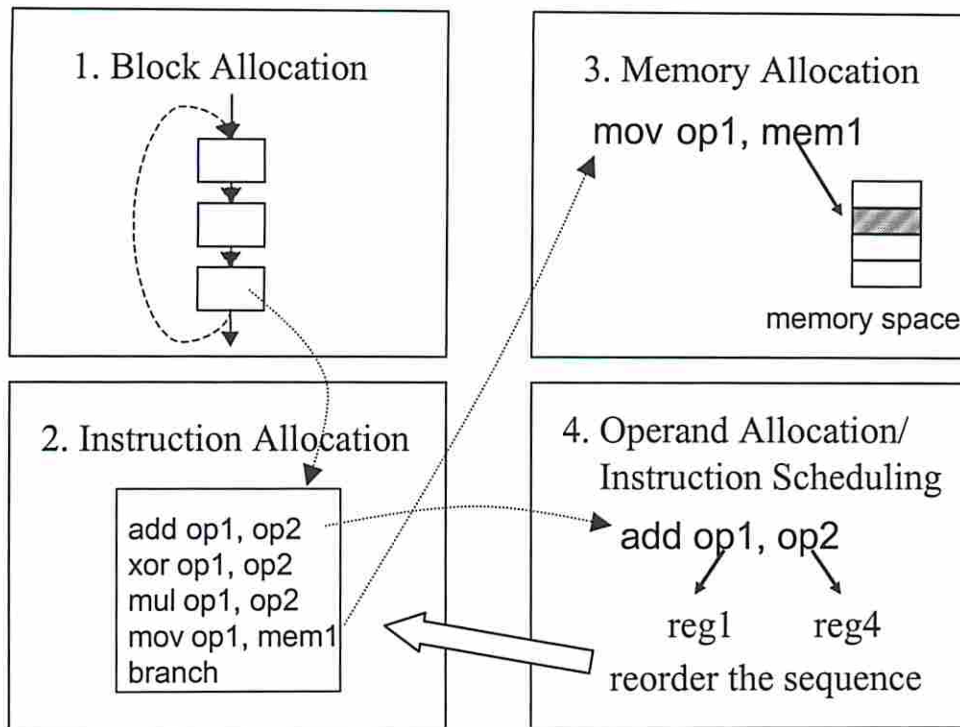


Figure 5. Synthesis procedure overview

The following is the summary description of the four synthesis phases (c.f. Figure 5):

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**: A set of instructions is allocated for each basic block in the *CFG*. Only the opcodes of these instructions are fixed in this phase; the operands and the order of instructions will be determined in the remaining phases. The characteristic sub-profile of this phase includes 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 characteristic sub-profile: CPI, mix of immediate-operand size, and average instruction length.

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 the average number of instructions between two branch instructions in the input trace.

4.2 Assumptions and Constraints

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

We limit the number of target addresses of a branch instruction to one. This enables us to solve the block allocation problem in Section 4.3 efficiently. The multiple target addresses occurs when the branch instruction takes the register as the target address, i.e. "`jump reg`" and the register takes more than one target address. This usually occurs in the switch statement of a C program. In typical C programs, indirect jump occurs less frequently compared to the other types of branch instructions; hence it is reasonable to drop it from our synthesis procedure. We also choose to keep the contents of the `reg` as a constant in the synthesized program such that the branch instruction will only have one target address. Note that the goal of our program synthesis is only to maintain the overall performance behavior, not the functional behavior of the input program.

Branch Decision Pattern	Branch Decision Sequence
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 branch decision sequences

Definition 4.1: *Branch decision sequence* of a basic block b is defined as the trace of the outcome of the branch instruction that is immediately before the exit point of the basic block b, when the synthesized *CFG* is simulated.

The branch decision sequences that are allowed during the program synthesis are listed in table 1; they are regular and periodic, and easy to analyze and synthesize.

Definition 4.2: A basic block b is defined as type i if its branch decision sequence follows branch decision pattern i listed in Table 1.

Consider the program segment in Figure 6.

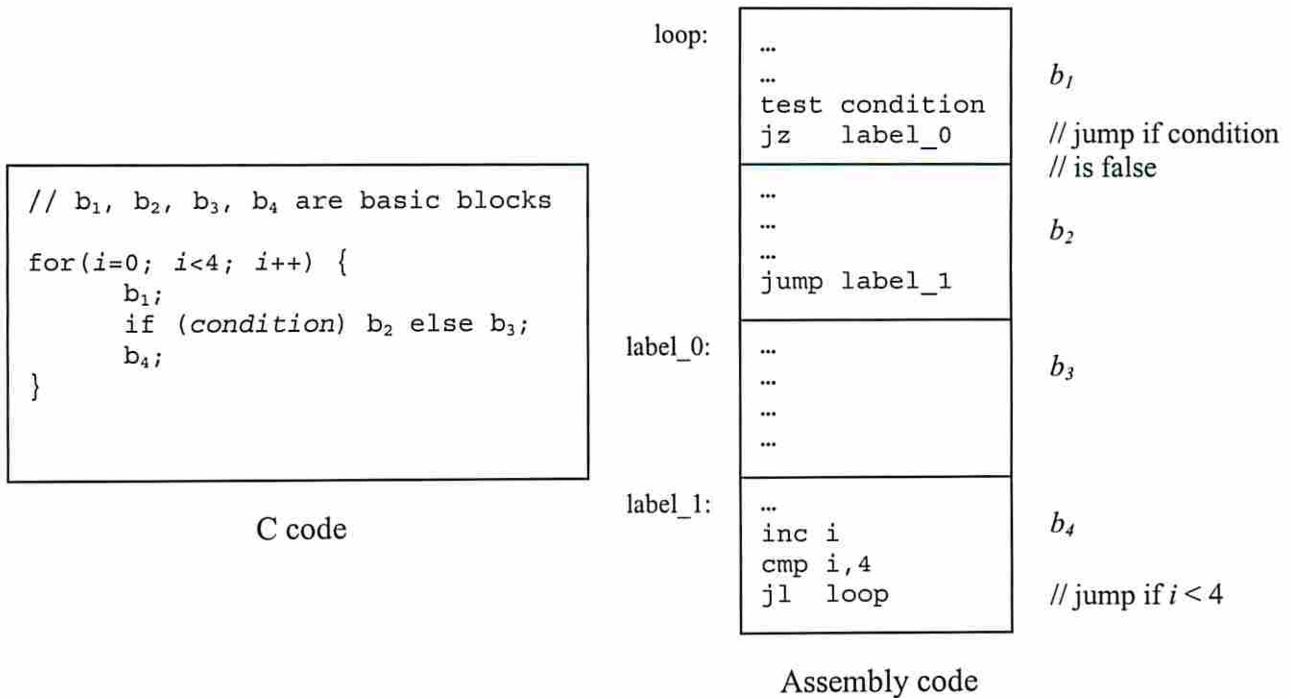


Figure 6. A for loop example

In the assembly code, the instructions involved in the **for** loop initialization are ignored and the instructions corresponding to the condition testing of the **if** statement are included at the end of b_1. Assuming the condition is true when loop count i is an odd number, the instruction trace becomes: b_1, b_3, b_4, b_1, b_2, b_4, b_1, b_3, b_4, b_1, b_2, b_4. We observe that the branch decision sequence of b_1 is taken, not taken, taken, not taken,….. Thus, b_1 is type 1. Similarly, we see that b_2 is type 2, b_3 is type 3, and b_4 is type 4. Experimental results show that a high synthesis quality can be achieved using only the branch decision sequences listed in the Table 1. Therefore, we do not allow other branch decision patterns in the

synthesized program. To reduce the number of variables in the block allocation formulation, we force the loop count in type 4 to be always an ***even*** number.

A *CFG G(B,E)* is a *loop-back CFG* if all blocks in the *CFG* are type 1 to type 3 except that the exit block is type 4 with branch targeting the entry point of the entry block of the *CFG.* 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 6 is b_1,b_3,b_4 (iteration 1), b_1,b_2,b_4 (iteration 2), b_1,b_3,b_4 (iteration 3), etc. Its period is two.

In our synthesis procedure, loop-back *CFG's* are concatenated to form the *CFG* of the synthesized program. The basic blocks within a loop-back *CFG* form a consecutive sequence in the output trace. This property localizes the analysis of the instruction-cache miss rate and the branch-prediction miss rate of the basic blocks, resulting in the use of linear only equations in the block allocation phase.

Only loop-back *CFG*'s with a period of at most two are used in our synthesis procedure. The period of two or less is chosen because it is sufficient for synthesizing the desired instruction-cache miss rate and branch-prediction miss rate (as demonstrated in the experimental results). If a larger period is used, we need to either limit the loop count to be a multiple of the period or use more variables in the problem formulation to account for the loop counts which are not multiples of the period. The former will compromise the compression ratio whereas the latter would significantly increase the complexity of the problem formulation.

4.3 Justification of Our Solution Techniques

As stated before, the program verification problem is undecidable. In our synthesis procedure, we build a new program starting with a set of templates and progressively refine it such that the control flow of the program remains predictable, while trying to match the characteristic profile. In Appendix A, we show that each phase of our four-phase synthesis procedure results in an NP-hard problem, which justifies the use of MILP or heuristics.

In each synthesis phase, there is a trade off between the algorithm complexity and the synthesis quality. In the block allocation phase, a mixed integer linear programming formulation is used because the input problem size is small enough to be solved efficiently using MILP solver. In the instruction allocation phase, although the 0-1 integer programming formulation is possible, the number of instruction slots and the number of instructions in the instruction set are too large to make the 0-1 integer programming

practical. So a greedy algorithm is used. The number of memory operands in the memory allocation phase is also too large for an exact algorithm to apply. In the operand assignment phase, we afford to use 0-1 integer programming to assign immediate operands because the number of immediate operands is small in most programs. The problem size is however too large to resort to an exact algorithm for register assignment and instruction scheduling. Hence, greedy approaches are applied.

4.4 Block Allocation

The block allocation is done in two steps: 1) design the set of *CFG* templates (macro block templates) as building blocks, 2) formulate the block allocation problem as a mixed integer linear programming (MILP) problem. At the end of this section, we show that in the MILP formulation, only twice as many integer variables as the number of macro block templates are needed. The number of macro block templates need not to be large, since it only has to be large enough to create different combinations of instruction-cache miss rate and branch-prediction miss rate. Our experimental results show that only eight macro block templates (c.f. Figure 13, Section 5) are enough to handle the SPEC 95 int benchmarks.

4.4.1 Macro Block Templates

A *macro block* is a *feasible loop-back CFG*, with period two or less, and with an annotated embedding method.

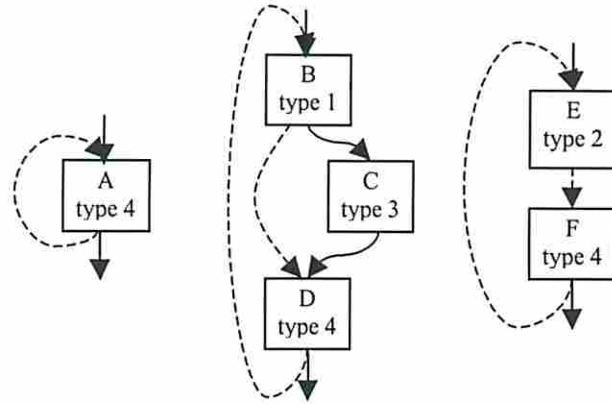
Definition 4.3: Given n macro blocks, a concatenation of these macro blocks is obtained by repeated application of pair-wise concatenation of their corresponding *CFGs* (c.f. Figure 7).

Observation 4.1: Any concatenation of any number of macro blocks creates a feasible CFG.

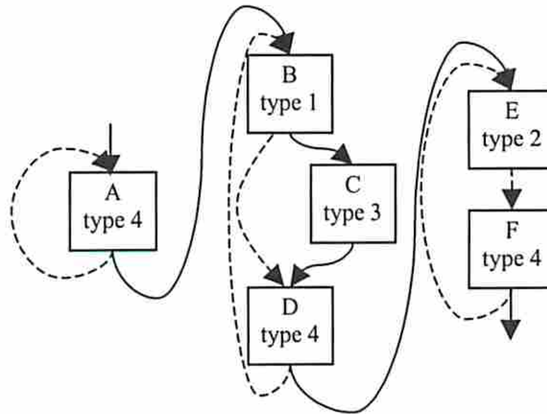
The number of instruction-cache misses of the synthesized program is given by:

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

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



(a) macro blocks



(b) concatenation of macro blocks

Figure 7. Concatenation of macro blocks

Different embedding methods can be chosen to achieve desired instruction-cache miss rate: sequential embedding which gives the minimal instruction-cache miss rate, skip embedding which increases *bubble* in Equation (1), and conflict embedding which increases *conflict* in Equation (1).

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

- a template for a loop-back *CFG* with a period of at most two
- embedding methods for all the basic blocks in the *CFG*
- average number of blocks *NB* which are being executed per loop
- a function returning the average number of instruction-cache lines fetched for *lp_cnt* loop iterations:

  $$Lines(lp_cnt) = C_2 \cdot lp_cnt + C_1$$

- a function returning the average number of branch-prediction misses for *lp_cnt* loop iterations:

  $$BP_{miss}(lp_cnt) = C_4 \cdot lp_cnt + C_3$$

Loop count, *lp_cnt*, is a parameter of the macro block template and should be an even number (limited by the constraint described in Section 4.2). $BP_{miss}(lp_cnt)$ and $Lines(lp_cnt)$ are linear functions of *lp_cnt* because of the periodic behavior of the *CFG*. Constants C_1, C_2, C_3, and C_4 are derived according to the specified embedding methods and the control types of the basic blocks in the *CFG*.

The following macro block template examples assume that instruction cache is 2-way set-associative with a line size of 32 bytes, instruction size of 4 bytes, and that every basic block has 4 instructions. In addition, the one-bit branch-prediction scheme (c.f. Figure 8) is assumed. A mis-prediction for a branch will reverse the prediction of the branch in Figure 8.

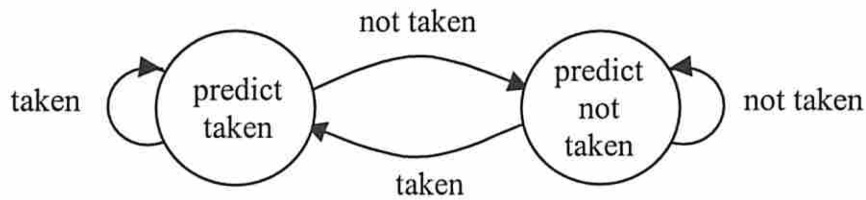
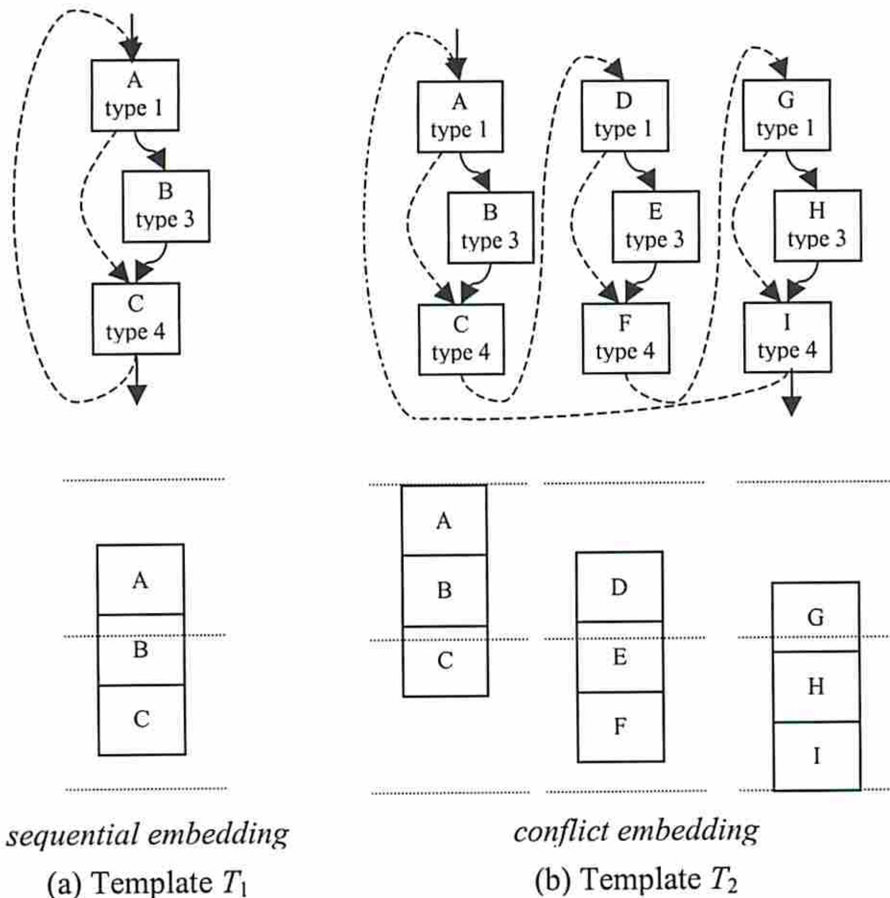


Figure 8. Branch-prediction Scheme



(a) Template T_1 (b) Template T_2

Figure 9. Macro block template examples (dotted lines represent the cache boundaries)

Sequential embedding is used in Figure 9(a):

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

Conflict embedding is used in Figure 9(b):

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

4.4.2 MILP Formulation

The block allocation problem is solved by concatenating macro blocks from the set of predefined macro block templates to minimize the sum of the differences of instruction-cache miss rate and branch-prediction miss rate between the input program and the synthesized program subject to the trace length constraint.

Let m denote the number of macro block templates in the synthesized program. Subscript i on a variable or a function shows that it is associated with the ith template. Let n_i be the number of macro block instantiations of the ith template; $lp_cnt_{i,j}$ denotes the loop count of the jth macro block of the ith template.

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

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

Let the BL' denote the average block length of the output trace, the $iCache'_{miss_rate}$, and BP'_{miss_rate} are calculated by:

$$iCache'_{miss_rate} = \frac{\#\text{ of lines loaded into cache}}{\#\text{ of instructions in the output trace}} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} Lines_i(lp_cnt_{i,j})}{TB' \cdot BL'} \quad (2)$$

$$BP'_{miss_rate} = \frac{\#\text{ of branch_prediction misses}}{\#\text{ of branch instructions in the output trace}} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} BP_{miss,i}(lp_cnt_{i,j})}{TB'} \quad (3)$$

Given a fixed output trace length, and assuming that BL' is equal to (or very close to) the average block length of the input trace, TB can be treated as a constant. Assuming both TB' and BL' are constants, equations (2) and (3) become linear.

We can then find out $lp_cnt_{i,j}$ by solving an MILP problem with the objective function :

$$w_1 \left| BP_{miss_rate} - BP'_{miss_rate} \right| + w_2 \left| iCache_{miss_rate} - iCache'_{miss_rate} \right| \quad (4)$$

The weight coefficients w_1 and w_2 are determined based on the ratio of the per-instruction energy penalty incurred by a branch prediction miss and that by an instruction cache miss (as stated in Section 3.2, this energy penalty can be estimated by using standard RT-level power estimation tools). After the MILP is solved, the block length of each basic block is decided by an iterative greedy algorithm so as to make the BL' very close to BL. More details about this formulation are provided in Appendix B.

4.5 Instruction Allocation

The instruction allocation phase has two goals: 1) allocating instructions to form the control flow specified in the previous phase, 2) matching the instruction mix of the input trace. Hence, the instruction allocation is divided into two steps: *control-flow-related* allocation and *control-flow-unrelated* allocation.

Assume there are p instructions $I_1, I_2, \ldots, I_p$ in the instruction set. $Freq(I_i)$ and $Freq'(I_i)$ are the execution frequencies of the instruction I_i in the input program and the synthesized program, respectively. The goal is to minimize of the total difference between instruction frequencies of the two programs, i.e., to minimize the objective 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) operands.

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

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

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

The instruction mix of input trace is shown in table 2. Assume there are only four instruction slots, with equal execution frequency, in the synthesized program. A straightforward allocation would yield: "**mov** reg->mem", "**mov** mem->reg", "**add** r1,r2->r3", and "**sub** r1,r2->r3".

Next, we group instructions with similar energy cost together as shown below:

- "**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 by group matching:

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

The second sequence is obviously more representative of the instruction profile given in table 2. We conclude that instruction grouping is necessary and helpful in maximizing the synthesis quality subject to a fixed compression ratio.

Let the instruction set grouped into k groups, G_1, G_2, ..., 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 a two-level, lexicographic cost function:

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

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

Control-flow-related allocation is based on the basic block templates for each branch decision pattern (c.f. Table 1). The *basic block template* specifies what instruction should be added to achieve the desired branch decision pattern. Two example templates are shown in Figure 10. By alternating the value in the branch decision flag, a type 1 branch decision pattern is achieved. In our approache, a greedy iterative improvement algorithm is used to select a proper template among the set of pre-defined basic block templates for each basic block.

```
// mem1 stores 0 initially
...
...
not mem1->mem1
jnz taken_target
// jump if mem1 != 0
```

```
// reg1 stores 1 initially
...
...
xor reg1,1->reg1
jz  taken_target
// jump if reg1 == 0
```

Figure 10. Basic block templates for branch decision pattern 1

Control-flow-unrelated allocation is used to assign instructions to the unoccupied instruction slots in each basic block. Again, a greedy iterative improvement algorithm is used to allocate control-flow-unrelated instructions while minimizing the foresaid lexicographic objective function.

4.6 Memory Allocation

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 difference between data-cache miss rates of the input program and the synthesized program:

$$w_3\left|dCache_read_{miss_rate} - dCache_read'_{miss_rate}\right| + w_4\left|dCache_write_{miss_rate} - dCache_write'_{miss_rate}\right| \quad (5)$$

Again, w_3 and w_4 are determined based on the ratio of the per-instruction energy penalty of a cache-read miss versus a cache write miss.

An instruction with scalar access will always access the same memory location; an instruction with vector access will always change its memory location. The vector access, which is used to create higher cache misses, can be implemented by taking loop counter as the offset of the referenced memory address.

According to the control flow requirement, the memory accesses for instructions in a basic block are classified into one of four types: *exclusive read, exclusive write, shared read,* and *shared write.* Memory references within the control-flow-related instructions are marked as exclusive read or write. All the other memory references are classified as shared read/write.

```
        // mem2 stores 10 initially
        ...
        mov r1->mem1
        ...
loop:
        ...
        dec mem2
        jnz loop      // jump if mem2 != 0
```

Figure 11. Exclusive-read type of memory

For example, in Figure 11, the instruction "`mov mem2->r2`" is initialized to the loop count 10. Since the contents of `mem2` must not be changed by other memory references before or during the loop where it resides, `mem2` is labeled as exclusive read. Similarly, the instruction "`not mem1->mem1`" in Figure 10 is used to flip the content of `mem1` such that the branch decision sequence is alternately taken. To avoid the possibility of other memory references changing the content of `mem1` during its lifetime, `mem1` needs to be marked as exclusive read/write.

During their lifetime, the exclusive-read/write memory references should never share the same memory space with other memory references that may potentially destroy the control-flow-related information. For example, in Figure 11, if `mem2` shares the same location with `mem1`, then the content of the loop counter will be destroyed. Note that for memory references that assume some initial value at the program startup, their lifetimes start from the very beginning of the program execution. For example, `mem2` in Figure 11 is initialized to value 10, which means that its lifetime starts from the beginning of the program execution and ends when the control flow exits the loop shown in Figure 11.

Definition 4.4: *Two memory operands are compatible if they can share the same memory location without destroying the control-flow-related information.*

Our greedy algorithm first finds an initial allocation with minimum date-cache-read/write 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).

It then 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.

4.7 Operand Assignment / Instruction Scheduling

This phase is divided into two steps. The first step can be skipped if the target processor has fixed instruction length. In step 1, immediate operands are assigned and the register operands are classified according to the average instruction length of the input program. In step 2, register operands are selected

from their classification in step 1 and the instructions are shuffled to match the CPI of the input program.

Step 1.

Immediate operand assignment:

In the Pentium instruction set, the immediate operand can be a byte, word (16bit), or double world (32bit). The immediate operand length should be chosen such that it matches the mix of the immediate operand size of the input program. 0-1 integer programming is used to assign immediate operands since the number of immediate operands is small.

Register classification:

In the Pentium instruction set, there are two different lengths of encoding for "`mov reg,mem`" type of instructions [7]. If `reg` is `eax`, then the opcode is one-byte long; if `reg` is one of `{ebx,ecx,edx,ebx,esp,ebp,esi,edi}`, the opcode is two-bytes long. This allows one to assign a register operand to certain register class according to the resulting instruction encoding length. The number of register operands is usually large, so a greedy algorithm is used. We first minimize the average instruction length and then increase it by changing the class assignments of some registers to match the target average instruction length.

Step 2.

Pipeline stalls are architecture-dependent effects, which may be caused by very different reasons when we consider different architectures. To be able to match the CPI, we need to design rules that can incrementally adjust the pipeline-stall rate, and provide a performance model to calculate the CPI given an instruction sequence.

1. Rules to increase/decrease CPI (pipeline stalls).

There are three types of hazards that cause the pipeline stalls [9]:

- structural hazards arising from resource conflicts
- data hazards arising from data dependencies

- control hazards arising from the branch instruction when the instructions at the target address are not pre-fetched.

Unlike control hazards, which are already fixed after the block allocation, structural hazards and data hazards are not fixed at this point. To increase the data hazards, we create data dependency between adjacent instructions. As an example, consider two instructions inside a basic block as shown in Figure 12. We assume that, as a result of register classification in Step 1, register `reg3` is assigned to the class of physical registers `{eax,ebx,ecx}` while `reg4` is assigned to the class of physical register `{eax}`. We can then assign physical register `eax` to both `reg3` and `reg4` to create data dependency.

```
...
add reg1,reg2->reg3
mov reg4->mem1
...
```

Figure 12. Read after write dependency

To increase the structural hazards, we introduce resource conflicts. The Pentium processor, for instance, can issue only one multiplication operation at a time, whereas it can issue up to two logic operations within the same clock cycle. As a result, two logic operations can be paired together to increase the CPI, while splitting two multiplication operations will decrease the CPI. Other rules for the data and structural hazards can be found in [6].

2. Performance model for the instruction pipeline.

Because the rules for increasing or decreasing pipeline stalls have non-linear effects on the CPI, we need to have a performance model for instruction pipeline to calculate the exact CPI. This performance model is in fact a fast instruction pipeline simulator, which is part of the architectural simulator for the microprocessor.

In this step, we repeatedly apply rules to adjust the pipeline-stall rate until the CPI of the input program is matched. After each adjustment, the performance model is evaluated to obtain accurate characteristic values for the synthesized program.

5. Experimental Results

To verify the effectiveness of the proposed approach, we 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 the 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 [6].
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 much more accurate. This enables us to test our approach on both large and small test programs.

Eight integer SPEC95 programs are used as our benchmark. Instruction traces of these programs are first captured. 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 new programs.

In block allocation, eight macro block templates (c.f. Figure 13) are used. We first try to find the largest compression ratio such that instruction-cache miss rate and branch-prediction miss rate are well matched. In the synthesized *CFG*, 48-52 basic blocks are allocated to make sure that there is enough flexibility in the subsequent phases to achieve a high synthesis quality. Note that for a program with lower instruction-cache miss rate, the compression ratio need to be set smaller to be able to meet the specified instruction-cache miss rate. This is because the loop counts of macro blocks need to be large enough to lower the instruction-cache miss rate. In the remaining phases, maximum quality is the synthesis goal.

The instruction set is partitioned into 30 groups in the instruction allocation phase. Immediate operand assignment is skipped because almost all the immediate operands, in the benchmark programs, are 4-bytes long. The register classification is still performed to match the average instruction size. The runtime for synthesizing each program is about 20 minutes on a Pentium machine. The traces of these synthesized programs are smaller than the input traces by 3-5 order of magnitude as shown in Chart 1.

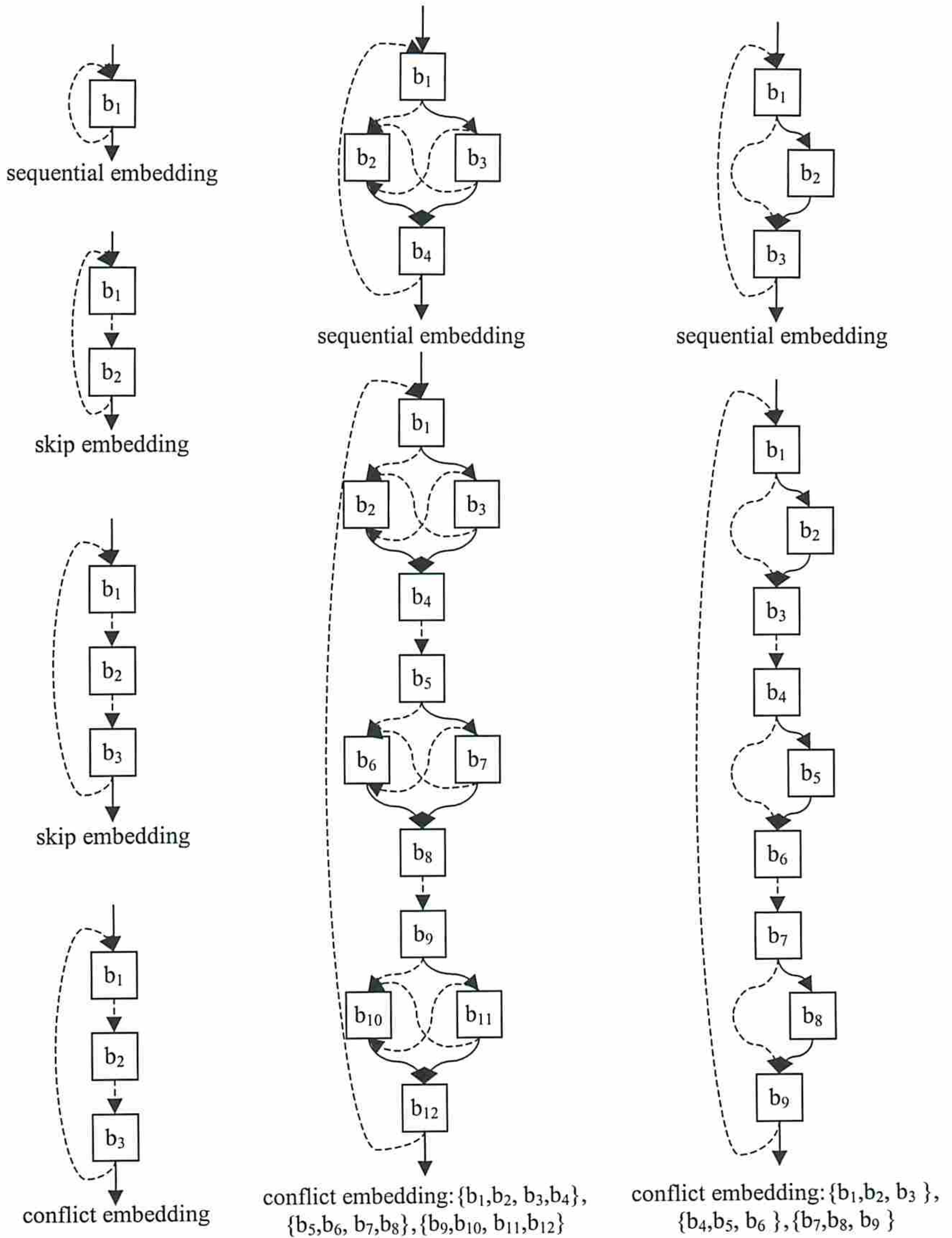


Figure 13. Eight macro block templates

Experimental data is collected on the Pentium processors running at 90MHZ with 2.9V power supply. To measure only the power consumption of the Pentium processor, a special board provided by Intel is used to plug in the processor socket such that the power supply connections from the motherboard to the processor are split. This setup enables us to measure the current that goes into the Pentium processor while executing the programs. Note that only the energy consumed by the Pentium processor (including on-chip cache) is measured and reported, and the energy consumed by external cache, chipset, and memory modules is excluded.

Table 3 shows the synthesis qualities. The entries in table 3 correspond to the percentage error between the same parameters (for example, *iCache hit rate*) in the original program and the synthesized program. The last row of the table gives the average percentage error over all benchmarks. Chart 2 shows the energy per instruction for the original program (*EPI)* and the energy per instruction for the synthesized program (*EPI')*. The average error of *EPI'* is 2.9%, which shows the accuracy of proposed profile-driven program synthesis.

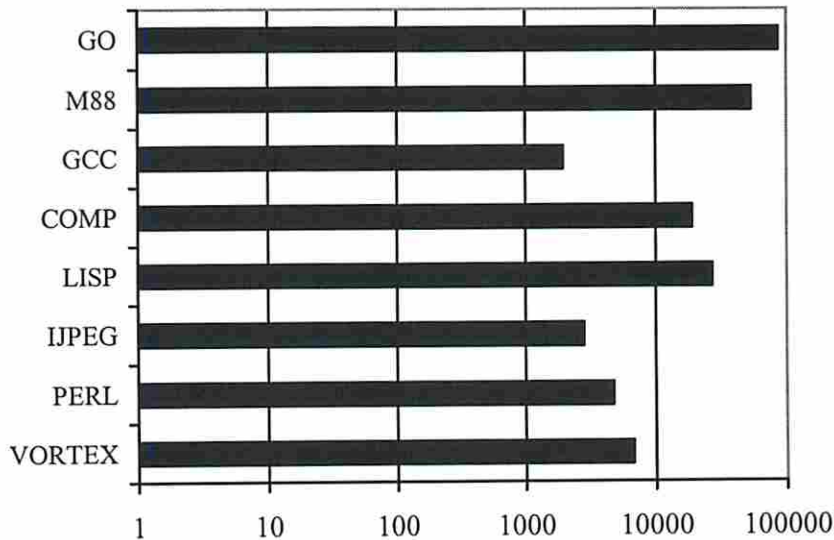


Chart 1. Compression Ratio

	iCache hit rate err %	*BP hit rate* err %	*inst. mix* err%	*dCache read hit rate* err %	*dCache write hit rate* err %	*avg. inst. size* err%	*CPI err %*
GO	0.9	2.4	1.2	0.2	0.4	0.3	1.0
M88K	0.2	0.8	1.9	0.6	1.0	0.8	0.9
GCC	1.2	0.5	2.9	0.05	0.7	0.6	0
COMP	0.1	0.2	0.6	2.9	0.4	0.2	2.9
LISP	0.2	0.6	1.6	0.1	1.2	0.5	4.5
IJPEG	0.01	0.2	1.9	0.1	0.1	0.3	2.9
PERL	1.2	1.1	2.6	0.04	0.3	0.4	1.7
VORTEX	0.5	0.1	4.0	0.1	3.9	0.5	3.8
Ave Err %	0.54%	0.74%	2.1%	0.51%	1.0%	0.5%	2.2%

Table 3. Experimental Results

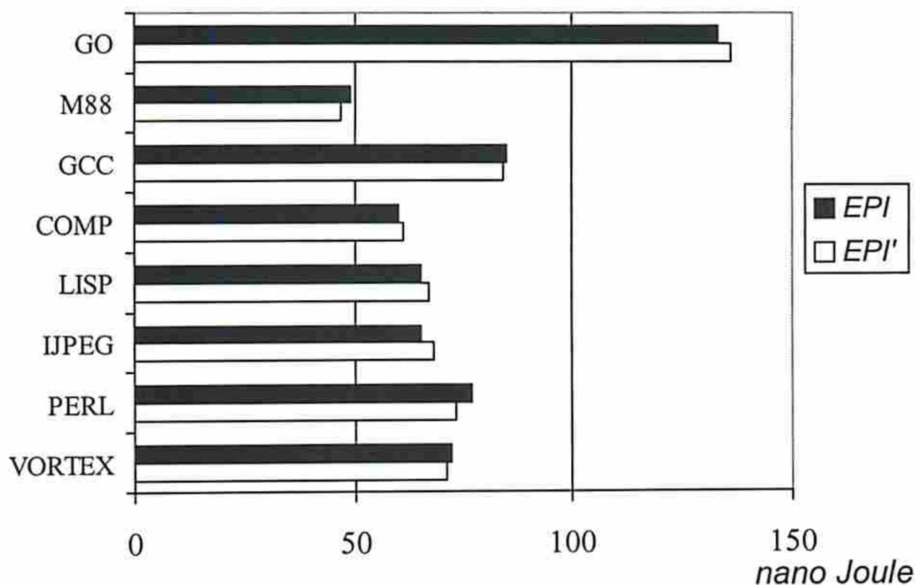


Chart 2. ***EPI*** **vs** ***EPI'***

6. Conclusion

We have presented a new technique for evaluating the power dissipation of application programs running on a high performance microprocessor. The first step of this technique was to identify a characteristic set, which is sufficient to capture the microprocessor power and performance behavior. A trace-driven architecture simulation was then run through the instruction trace to collect the characteristic profile. MILP and heuristic rules were used to transform predefined macro block templates

into a working program after four synthesis phases. Experimental results showed the effectiveness of the proposed technique on an Intel Pentium processor running SPEC int 95 benchmarks.

References

[1] Toshinoriato, Y. Otaguro, M. Nagamatsu, and H. 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] C-L Su, C-Y Tsui, and A. Despain. "Low power architectures design and compilation techniques for high performance processors." In *Proceedings of IEEE COMPCON*, pp. 489-498, 1994

[3] V. Tiwari, S. Malik, A. Wolfe, and T-C Lee. "Instruction level power analysis and optimization of software". In *Journal of VLSI Signal Processing,* Aug/Sept, 1996.

[4] C-Y Tsui, R. Marculescu, D. Marculescu, M. Pedram, "Improving the efficiency of power simulators by input vector compaction". In *Proceedings of Design Automation Conference*, pp.165-168, 1996.

[5] D. Marculescu, R. Marculescu, M. Pedram, "Stochastic Sequential Machine Synthesis Targeting Constrained Sequence Generation", In *Proceedings of Design Automation Conference*, pp.696-701, 1996.

[6] Intel Corporation, "Pentium Processor Family Developer's Manual", *Volume 1:Pentium Processors,* 1996.

[7] Intel Corporation, "Pentium Processor Family Developer's Manual", *Volume 3: Architecture and Programming Manual,* 1996.

[8] M. R. Garey, D. S. Johnson, "Computer and Intractability, A Guide to the Theory of NP-Completeness", *W.H. Freeman and Company*, 1979.

[9] J. Hennessy, D. Patterson, "Computer Architecture: A Quantitative Approach", *Morgan Kaufmann Publishers Inc.*, 1990.

[10] P. Landman and J. Rabaey, "Power estimation for high-level synthesis", In *Proceedings of IEEE European Design Automation Conference*, pages 361-366, Feb. 1993.

[11] R. Burch, F. N. Najm, P.Yang, and T. Trick, "A Monte Carlo approach for power estimation." In *IEEE Transactions on VLSI*, Volume 1, pages 63-71, March 1993.

Appendix A. Complexity Analysis for Program Synthesis

We show that each phase of the program synthesis procedure results in an NP-hard problem. The formulations given below are simplified versions of the problems encountered in each phase; that is, some of the secondary non-linear constraints are ignored. The proof of the NP-hardness of the original problems is immediate (proof by restriction, see [8, p.63]).

The subset sum problem, the graph k-colorability problem, and resource constrained scheduling problem are iterated from [8]. These problems will not be included in the final manuscript; they are included for the convenience of the reviewers.

Subset sum problem:

Instance: Finite set A, size $s(a) \in Z^+$ for each $a \in A$, positive integer B.
Question: Is there a subset $A' \subseteq A$ such that the sum of the sizes of the elements in A' is exactly B?
Complexity: NP-complete

Graph K-Colorability:

Instance: Graph $G=(V,E)$, positive integer $K \leq |V|$.

Question: Is G K-colorable, i.e., does there exist a function $f:V \rightarrow \{1,2,\ldots,K\}$ such that $f(u) \neq f(v)$ whenever $\{u,v\} \in E$?

Complexity: NP-complete for all fixed $K \geq 3$.

Resource Constrained Scheduling Problem:

Instance: Set T of tasks, each having length $l(t)=1$, number $m \in Z^+$ of processors, number $r \in Z^+$ of resources, resource bounds B_i, $1 \leq i \leq r$, resource requirement $R_i(t)$, $1 \leq R_i(t) \leq B_i$, for each task t and resource i, and an overall deadline $D \in Z^+$.
Question: Is there a m-processor schedule σ for T that meets the overall deadline D and obeys the resource constraints, i.e., such that for all $u \geq 0$, if $S(u)$ is the set of all $t \in T$ for which $\sigma(t) \leq u \leq \sigma(t) + l(t)$, then for each resource i the sum of $R_i(t)$ over all $t \in S(u)$ is at most B_i.

Complexity: NP-complete for $r=1$, $m=2$, and the partial order $\angle$ on I is a forest.

A.1 Block Allocation

Input: A finite set M of macro-block templates, linear functions $f_i(n_i, lp_cnt_i)$, and $g_i(n_i, lp_cnt_i)$, which return the number of cache misses and branch prediction misses for the ith template in M. Arguments n_i and lp_cnt_i denote the number of instantiations and total loop counts for ith template, respectively. C_{miss} and B_{miss} are the number of target cache misses and branch prediction misses to match.

Objective: Find n_i and lp_cnt_i for each template in M such that $\left|\sum_i f_i(n_i, lp_cnt_i) - C_{miss}\right| + \left|\sum_i g_i(n_i, lp_cnt_i) - B_{miss}\right|$ is minimized.

Complexity: We restrict lp_cnt_i to constant values, n_i to be either 0 or 1, and simplify the objective to minimize only the first term, $\left|\sum_i f_i(n_i, lp_cnt_i) - C_{miss}\right|$. Because f_i is a linear function, we can rewrite the object function as :

$$\left|\sum_i c_i n_i - C'_{miss}\right| \quad \text{(A.1)}$$

where c_i is a constant. We form a subset M' by including the ith elements from M exactly when n_i equals 1.

The decision problem then becomes: Is there a subset $M' \subseteq M$ such that the sum of the c_i's is equal to C'_{miss}? The decision problem is thus equivalent to the subset sum problem, which is NP-complete. Therefore, the block allocation problem is NP-hard.

A.2 Instruction Allocation

Input: Set T of instruction types and set S of instruction slots where each slot s in S has a known execution count *count(s)*; the target count *Count(t)* for each instruction type t in T is also given.

Objective: Assign an instruction type in T to each instruction slot in S such that $\sum_{t \in T}\left|\sum_{s \in S} type(s,t)count(s) - Count(t)\right|$ is minimized, where the *type(s,t)* equals 1 if instruction type t is assigned to slot s, 0 otherwise.

Complexity: If we simplify the objective function to $\left|\sum_{s\in S} type(s,t)count(s) - Count(t)\right|$, where t is a fixed instruction type, then it becomes the same form as Equation (A.1). This means the instruction allocation problem is also NP-hard.

A.3 Memory Allocation

Input: Set M of memory operands, the access count *count(m)* for each m in M, the computability constraints on M, and the target number of cache misses A.

Objective: Find a memory address, *address(m)*, for each memory operand in M such that |*CacheMiss* - *A*| is minimized, where *CacheMiss* is a non-linear function that depends on *count(m)* and *address(m)* for every m in M.

Complexity:

An interference graph *G(M,E)* can be formed according to the compatibility constraint (see Definition 4.4) where E is a set of edges $\{(m_1,m_2)|\ m_1$ is not compatible with m_2, for every m_1,m_2 in $M\}$. Let *CacheMiss* equal the number of different memory addresses accessed by the program. The problem, then, becomes the graph *k*-colorability problem, which is NP-complete. Therefore the memory allocation problem is NP-hard.

A.4 Operand Allocation

The problem is similar to instruction allocation. It is therefore NP-hard.

A.5 Instruction Scheduling

Input: Set I of instructions, partial order relations on I induced by the *CFG* and the data (register) dependencies, number $m \in Z^+$ of pipelines, and number $r \in Z^+$ of resources. A is the target number of execution clock cycles to match.

Objective: Find a total order for I such that $|t\text{-A}|$ is minimum, where t is the total execution time for the given instruction sequence to be executed on m pipelines subject to the specified resource constraint.

Complexity: The problem is the same as the resource constrained scheduling problem. Furthermore, the partial order relations induced by the *CFG* and the data dependencies can form a forest and the decision

problem is NP-complete for even m=2 and r=1. Consequently, the instruction scheduling problem is NP-hard.

Appendix B. MILP Formulation for Block Allocation

From Equation (4), the objective function is:

$$w_1\left|BP_{miss_rate} - BP'_{miss_rate}\right| + w_2\left|iCache_{miss_rate} - iCache'_{miss_rate}\right|$$

The objective function can be written as follows:

$$w_1(X_1 + X_2) + w_2(Y_1 + Y_2)$$

subject to the following constraints:

$$iCache_{miss_rate} - iCache'_{miss_rate} + X_1 - X_2 = 0 \quad \text{(B.1)}$$

$$BP_{miss_rate} - BP'_{miss_rate} + Y_1 - Y_2 = 0 \quad \text{(B.2)}$$

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

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 the target output trace length L_l. The nonzero tolerance on the output trace length often leads to better results compared to a fixed length constraint.

From Equations (2) and (3), the $iCache'_{miss_rate}$, and BP'_{miss_rate} are calculated by:

$$iCache'_{miss_rate} = \frac{1}{TB' \cdot BL'} \sum_{i=1}^{m} \sum_{j=1}^{n_i} Lines_i(lp_cnt_{i,j}) = \frac{1}{TB' \cdot BL'} \left(\sum_{i=1}^{m} \sum_{j=1}^{n_i} C_{2,i} \cdot lp_cnt_{i,j} + \sum_{i=1}^{m} C_{1,i} \cdot n_i \right) \quad \text{(B.3)}$$

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

Let $lp_cnt_i = \sum_{j=1}^{n_i} lp_cnt_{i,j}$ Note that lp_cnt_i and n_i must satisfy $lp_cnt_i \geq 2 \cdot n_i$ to have a feasible solution.

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

The objective function becomes $\left[w_1(X'_1 + X'_2) + w_2(Y'_1 + Y'_2)\right] / TB'$

Because TB' is bounded by a tight trace length constraint: $\frac{min_length}{BL'} \leq TB' \leq \frac{max_length}{BL'}$, it is nearly constant and can thus be removed from the objective function.

The objective function is approximated by:

$$w_1(X'_1 + X'_2) + w_2(Y'_1 + Y'_2).$$

Substituting Equations (B.3) and (B.4) into Equations (B.1) and (B.2):

$$\sum_{i=1}^{m}(iCache_{miss_rate} \cdot NB_i - C_{2,i}/BL') \cdot lp_cnt_i - \sum_{i=1}^{m}\frac{C_{1,i} \cdot n_i}{BL'} + X'_1 - X'_2 = 0 \tag{B.5}$$

$$\sum_{i=1}^{m}(BP_{miss_rate} \cdot NB_i - C_{4,i}) \cdot lp_cnt_i - \sum_{i=1}^{m}C_{3,i} \cdot n_i + Y'_1 - Y'_2 = 0 \tag{B.6}$$

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

min $w_1(X'_1 + X'_2) + w_2(Y'_1 + Y'_2)$

subject to

Equations (B.5) and (B.6)

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

$lp_cnt_i = 2 \cdot r_i \qquad 1 \leq i \leq m$ (forcing lp_cnt_i to be an even number)

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

$n_i, r_i \in Z^+$

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