Architectural and Compiler Techniques for Energy Reduction in High Performance Microprocessors

Nikolaos Bellas, Ibrahim Hajj, Constantine Polychronopoulos, and George Stamoulis

Abstract— **In this paper, we focus on low-power design techniques for high-performance processors at the architectural and the compiler level. We focus mainly on developing methods for reducing the energy dissipated in the on-chip caches. Energy dissipated in caches represents a substantial portion in the energy budget of today's processors. Extrapolating current trends, this portion is likely to increase in the near future, since the devices devoted to the caches occupy an increasingly larger percentage of the total area of the chip.**

We propose a method that uses an additional mini cache located between the I-Cache and the CPU core, and buffers instructions that are nested within loops and are continuously otherwise fetched from the I-Cache. This mechanism is combined with code modifications, through the compiler, that greatly simplify the required hardware, eliminate unnecessary instruction fetching, and consequently reduce signal switching activity and the dissipated energy.

We show that the additional cache, dubbed *L-Cache*, is much smaller and simpler than the I-Cache when the compiler assumes the role of allocating instructions to it. Through simulation, we show that, for the SPECfp95 benchmarks, the I-Cache remains disabled most of the time, and the "cheaper" extra cache is used instead. We also propose different techniques that are better adapted to non-numeric, non loop-intensive code.

Keywords— **Special-lowpower99, Low-power-design, Memory, Power-consumption-model**

I. INTRODUCTION

In recent years, power dissipation has become a major design concern for the microprocessor industry. The shrinking device size, and the large number of devices packed in a chip coupled with the large operating frequencies, have led to unacceptably high levels of power dissipation.

The problem of the wasted power caused by unnecessary activity in various parts of the CPU during code execution has traditionally been ignored in code optimization and architecture design. Processor architects and compiler writers are concerned with system performance/throughput and they do little, if anything at all, to eliminate energy/power dissipation at this level. However, power dissipation is rapidly becoming the major bottleneck in today's systems integration and reliability. Modern microprocessors are large power consumers: the UltraSPARC-II from Sun consumes 58 W maximum power at 296 MHz, the Pentium Pro Processor consumes 35 W at 200 MHz, and the Alpha 21164PC chip from DEC consumes 32.5 W at 433 MHz.

In general, low-power and high performance are usually two conflicting goals at all levels of the design hierarchy. For example, one common technique for reducing power consumption is to lower the supply voltage. This reduction in supply voltage, however, results in slower circuits. Higher frequencies are desirable for high performance, but they increase power consumption. Higher activity (and thus utilization) could result in a larger throughput, but also in higher power. The excessive power consumption of today's processors is, in part, the outcome of very high utilization of their components.

In this paper we develop techniques for energy reduction that have little or no impact on performance. We focus on reducing the activity caused by the I-Cache subsystem which is one of the main power consumers in most of today's microprocessors. The on-chip L1 and L2 caches of the 21164 DEC Alpha chip dissipate 25% of the total power of the processor [1]. The StrongARM SA-110 processor from DEC, which targets specifically low-power applications, dissipates about 27% of the power in the I-Cache [2]. In the Pentium Pro processor, the instruction fecth unit (IFU) and the I-Cache contribute 14% to the total power consumed [3].

The reason for the high power consumption in the I-Cache subsystem is that the execution rate of a processor depends critically on the rate at which the instruction stream can be fetched from the I-Cache. The I-Cache should therefore be able to provide the data path of the machine with a continuous stream of instructions, and has therefore very high switching activity. In addition, the I-Cache drives large capacitance wires to the CPU core. What is more, today's caches constitute an ever increasing portion of the die area and the number of transistors of the processor.

The remainder of the paper is organized as follows: section II discusses related work and section III provides the motivation behind our approach. Section IV details the compiler transformations necessary for our scheme, while sections V and VI describe the hardware support, and the energy estimation method we used for the caches, respectively. Section VII presents simulation results for both energy and performance on a subset of SPEC95 benchmarks, and section VIII discusses an extension of the method for the integer benchmarks. The conclusion is given in section IX.

II. RELATED WORK

Power optimization at the architectural and software levels has attracted the interest of a number of researchers. A model that views power from the standpoint of the software that executes on a microprocessor and the activity that it causes, rather than from the traditional hardware standpoint has been proposed in [4] and tested in different architectures [5].

In [6] and [7] brief reviews of compiler techniques for power minimization are presented. As expected, standard compiler optimizations, such as loop unrolling, software pipelining, etc., are also beneficial for the reduction of energy since they reduce the running time of the code.

More recently, the impact of memory hierarchy in minimizing power consumption, and the exploration of data-reuse so that the power required to read or write data in the memory is reduced, are addressed in [8] and [9].

The filter cache [10] tackles the problem of large energy consumption of the L1 caches by adding a small, and thus more energy-efficient cache between the CPU and the L1 caches. The penalty to be paid in adding the filter cache is the increased miss rates and, hence, longer average memory access time. Although this might be acceptable for embedded systems for multimedia or mobile applications, it is not desirable for high-performance processors. The filter cache delivers an energy reduction of 58% for a 256-byte, direct mapped filter cache, while reducing performance by 21% for a set of multimedia benchmarks.

This work was supported by Intel Corp., Santa Clara, CA

N. Bellas is with the DigitalDNA Systems Architecture Laboratory, Motorola Inc., Schaumburg, IL

I. Hajj and C. Polychronopoulos are with the Coordinated Sciences Laboratory, Department Of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL

G. Stamoulis is with Intel Corp., Santa Clara, CA

In a similar work in [11], a mechanism is described which enables the by-pass of the I-Cache by storing the instructions within loops in an extra buffer. Only loops with no conditional branching can be accomodated with this method, since the mechanism assumes that all the instruction in a loop are stored in the extra buffer in the first iteration.

The work in [3] focuses on excessive energy dissipation of high-performance, speculative processors that tend to execute more instructions than are actually needed in a program. The authors use the concept of branch prediction and confidence estimation [12] to detect when the CPU fetches and executes instructions from a speculative path that has a small probability to be taken. The CPU stops execution in the pipeline when there is a large probability of wrong path execution, and it resumes only when the actual execution path is detected.

III. MOTIVATION AND APPROACH

During a loop execution, the I-Cache unit frequently repeats its previous tasks over and over again: if the thread of control during program execution is caught in a loop, the I-Cache unit fetches the same instructions to the CPU core, and the ID decodes the very same instructions. The problem is that the IF unit does not operate in an efficient way with respect to power consumption, but it only tries to satisfy the demand of the execution units for high throughput, which is achieved through a fast first level (L1) instruction cache and high bandwidth buses between the cache and the CPU core. This approach works for performance, but it unnecessarily performs more work; thus, it dissipates more power than really needed.

Substantial power gains could be achieved if we could reduce the amount of instructions that the IF unit fetches, and subsequently disable the I-Cache system for all the time that it is not needed. The most usual method for disabling a unit is clock gating, i.e., not allowing the clock ticks to propagate changes to the output of the unit by ANDing them with a control signal.

This is the basic motivation of the architectural support that is proposed in this research as was also explained in [10] and [11]. All the instructions that belong to a loop can be fetched only the first time the thread of control passes through them. Subsequently, they can be stored in a special internal cache (the L-Cache) which is placed between the I-Cache and the CPU core. Each time the IF unit attempts to fetch an instruction from within the loop, the instruction that resides in this cache can be used instead. In the ideal case, the I-Cache unit can be shut down for the duration of the loop, as it does not need to operate, and its energy dissipation can be saved. Thus, this method exploits the locality in the instruction stream. To be energy prone, this mechanism only accesses the I-Cache and the L-Cache sequentially, i.e., in different clock cycles.

The approach advocated in our scheme relies on profile data from previous runs to select the best instructions to be cached. The unit of allocation is the basic block, i.e., an instruction is placed in the L-Cache only if it belongs to a selected basic block.[1] After selection, the compiler lays out the target program so that the selected blocks are placed contiguously before the non-placed ones. The main effort of the compiler focuses on placing the selected basic blocks in positions so that two blocks that need to be in the cache at the same time do not overlap in the L-Cache.

The compiler maximizes the number of basic blocks that can be placed in the L-Cache by determining their nesting and using their execution profile. The resulting hardware is very simple and most of the task is carried out by the compiler. We eliminate the need for a large L-Cache, thus greatly reducing the power requirements of the extra cache.

[1]A basic block is a sequence of instructions with no transfers in and out except possibly at the beginning or end.

IV. COMPILER ENHANCEMENTS

The selection of basic blocks to be inserted in the L-Cache is done by the compiler "statically", i.e., during compile time, and not "dynamically" during run time [13].

The optimization consists of two distinct phases:

- *function inlining*, in which the compiler tries to expose as many basic blocks as possible in frequently executed routines. This step should be done judiciously since function inlining can also create performance and locality problems in the I-Cache. This step aims at exposing as many basic blocks as possible in frequently executed routines. Our scheme assumes that no interprocedural basic block allocation can take place, i.e., at any given time, only basic blocks that belong to the same function can reside in the L-Cache. This precaution is taken since the compiler cannot know a-priori where the linker/loader will place the functions in the memory address space. Hence, each function in the source code is considered separately.
- *block placement*, the main stage of our method, in which the compiler selects, and then places the selected basic blocks so that the number of blocks that are placed at the same time in the extra cache is maximized. To that effect, the compiler avoids placing two blocks that have been selected to reside at the same time in the L-Cache in the same cache locations.

The reasoning behind our decision to choose a basic block as the basic unit of allocation and not a whole loop can be readily justified by considering the *control flow graph* (CFG) of a typical loop. In most cases, the loop contains basic blocks which are seldom executed during typical runs. These are blocks that take care of an exception condition or do error handling. If the whole loop was to be allocated in the L-Cache, these basic blocks would occupy space, but they would hardly ever be used. They would also disqualify frequently executed blocks from being cached.

The compiler seperates the selected basic blocks from the non-selected ones, and places all of them in the global address space. For example, consider the following code:

```
do 100 i=1, n
  B1;      # basic block
  if (error) then
     error handling;
  B2;      # basic block
100 continue
```

If the if-statement in the loop is placed between basic blocks B_1 and B_2 in the final layout of the code, it may create a conflict in the L-Cache. This will happen if the size of the L-Cache is smaller than the sum of the sizes of the basic blocks B_1, B_2 and the if-statement, but larger than the sum of the sizes of the basic blocks B_1 and B_2 alone. If we move the if-statement at the end, and place B_1 and B_2 one after the other, we effectively reduce the possibility of an overlap. We identify such cases and move the infrequently executed code away so that the normal flow of control is in a straight-line sequence. This entails the insertion of extra jump instructions to retain the original semantics of the code.

The block placement algorithm is delineated in Fig. 1. The object code and profile data for the original program are used as input to our tool. The output produced is an equivalent object code in which some of the basic blocks have been reordered and placed in specific memory locations. The following subsections give a detailed description for each of the steps of the method.

A. First step: Nesting computation

The control flow graph is built for each function of the original program in Step 1. Note that the program can be either the original one or the one that has been created after inlining. A

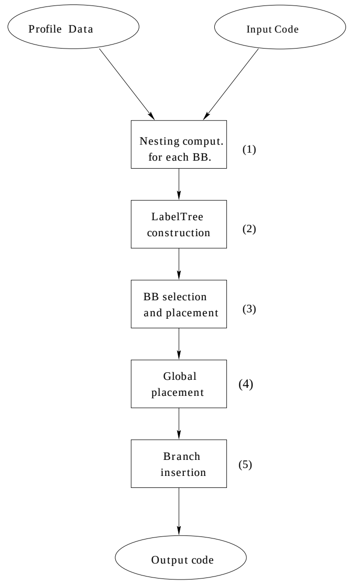

Fig. 1. Block placement overview.

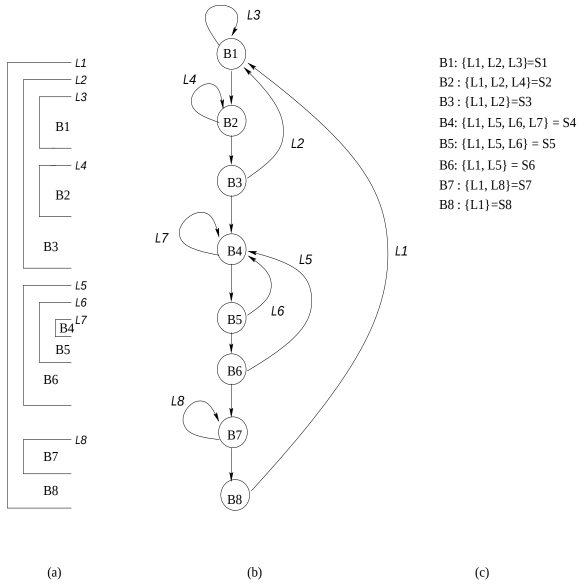

Fig. 2. First step of block placement.

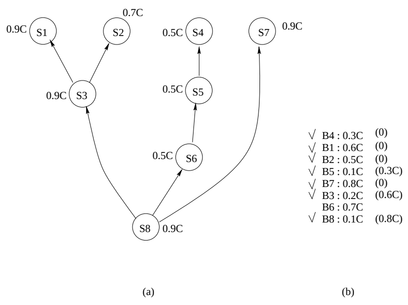

Fig. 3. LabelTree.

node in the CFG can have none, one, or more than one predecessors, and at most two successors. This is the case when there is a branch instruction at the end of the basic block. We introduce a slight modification in our CFG: although a procedure is normally considered as having only one entry, we generalize this as follows: if there is a function call within a procedure, the return from this function is declared as a new entry to the procedure. The reason for this modification is that we do not want to place basic blocks across procedures. Each procedure, upon entry, will assume that nothing is in the L-Cache from its caller. In other words, basic blocks within a loop which has a function call will not be eligible for caching. This restriction aims at freeing the linker from a possible burden when it maps a function body to the memory space. Some linkers try to map routines that call each other frequently onto contiguous memory addresses to increase the locality of accesses. An inter-function basic block allocation would pose additional constraints to the linker.

Next, in the same step, the tool finds the loops and the nesting for every basic block [14]. A $LabelSet(B)$ for every basic block B is the set of loops to which B belongs. If B is not nested, $LabelSet(B) = \{\}$. If B is enclosed in loops L_1, L_2 and L_3, then $LabelSet(B) = \{L_1, L_2, L_3\}$. These are the same sets used in [15]. In Fig. 2, an example is given to describe the data structures used and the information produced during the first step of the algorithm. A loop nesting is shown in 2(a), the corresponding CFG in 2(b), and the LabelSets in 2(c).

B. Second step: LabelTree construction

In Step 2, we construct a directed acyclic graph (DAG) using the LabelSets as follows: the nodes are the different LabelSets found in the previous step. There is an arc between two such LabelSets $< l_1, l_2 >$ if l_1 is a proper subset of l_2 (Fig. 3a). Our data structure, dubbed *LabelTree*, is a tree since no basic block can have two different nestings.

The LabelTree describes the nesting relationship between basic blocks. Basic blocks in the same path in the LabeleTree belong to the same nesting, although in different depths. Basic blocks that are near the leaves of the LabelTree are deeply nested, whereas basic blocks that are near the root are not.

C. Third step: Basic block selection and placement

Step 3 takes over the main part of our allocation algorithm (Fig. 4).

A well-known NP-complete problem is that of placing objects with a given value and weight into a knapsack so that the total value of the placed objects is maximized under the constraint that their weight does not exceed the capacity of the knapsack. We only expect to find a good heuristic which will place the most frequently executed basic blocks in the L-cache provided that their size is smaller than the size of the L-Cache.

The algorithm scans the basic blocks in descending order of execution frequency. Hence, the most important blocks are the first to be considered and have a greater chance to be placed in the L-Cache. For every node in the *LabelTree* we designate a size, which denotes the position in the L-Cache where a basic block of the node should be placed in every step of the algorithm. The size should always be less than or equal to the cache size; otherwise the current basic block cannot be placed in the L-Cache.

The first step is to propagate the effect of the size of the basic block under consideration towards the leaves of the tree rooted at node N (DOWN_TRAV()). Suppose, for example, that the current basic block is B_3 in Fig. 3a. Both nodes B_1 and B_2 have already been considered and placed in the L-Cache. The size of B_3 added to the $max(size(B_1), size(B_2))$ should not exceed the cache size C. If this is the case, B_3 is placed in the L-Cache. In other words, B_3 will remain in the L-Cache while B_1 and B_2 are executed, and it will not be replaced. This step aims at placing B_3 in a different L-Cache position from both B_1 and

```
void Allocate(LabelTree T, CacheSize C)
  /* T root of the LabelTree, C size of L-Cache */
   for every node N in T set size(N) = 0;
   for every basic block BB in the program in descending order
     of number of times executed do
      let N be the node in T that BB corresponds to;
         /* N is unique */
      if N is the root next;
          /* we don't place BB which are not nested */
      old_size(N) = size(N);
      fit = DOWN_TRAV(N, BB, C);
      if (fit == FALSE) then continue;   /* next basic block */
      UP_TRAV(N);
     put the basic block BB in the L-Cache in position old_size(N);
  end for;
end Allocate;

boolean DOWN_TRAV(TreeNode N, BasicBlock BB, CacheSize C)
   if (DOWN_TRAV_FIRST(N, BB, C) then
      DOWN_TRAV_SEC(N);
      return true;
   else
      return false;
end DOWN_TRAV;

flag = true;
boolean DOWN_TRAV_FIRST(TreeNode N, BasicBlock BB, CacheSize C)
   if (N != NULL and flag) then
      if (size(N) + size(BB) > C) then
         flag = FALSE;
      else
         temp_size(N) = size(N) + size(BB);
      end if;
      for all children of N do
         DOWNTRAV_FIRST(N->child);
   end if;
end DOWN_TRAV_FIRST;
void DOWN_TRAV_SEC(TreeNode N)
   if (N != NULL) then
      size(N) = temp_size(N);
      for all children of N do
         DOWN_TRAV_SEC(N->child);
   end if;
end DOWN_TRAV_SEC;

procedure UP_TRAV(TreeNode )
   while (N->parent) do
      N = N->parent;
      size(N) = max(size(N->child)) among all children;
   end while;
```

Fig. 4. Placement Algorithm.

B_2. If B_3 overlapped with them, it would have to be fetched from the I-Cache instead, since it would be replaced by B_1 or B_2 after being executed. This technique maximizes the number of basic blocks that are placed in the cache and avoids conflicts between them.

If $max(size(B_1), size(B_2)) + size(B_3) > C$, the placement of B_3 is not possible, and the algorithm continues with the next basic block.

Subsequently, the algorithm calls UP_TRAV() which propagates the effect of the new placement to the outer blocks. This, in effect, reduces the chance of the outer blocks to be placed in the L-Cache, which is not bothering at all, since we are mostly interested in the inner, most frequently executed blocks. In Fig. 3a, the annotated *LabelTree* for the example in Fig. 2 is given with the final placement of the basic blocks in 3b. All the blocks except B_6 are placed in the L-Cache (the positions are in the parentheses and are with respect to the beginning of the L-Cache).

The algorithm is greedy because it tries to accumulate as many important basic blocks as possible in the L-Cache. In the case where the most frequently executed basic blocks are the most deeply nested, the algorithm will succeed in putting all of them in the L-Cache provided that the size of each one is smaller than the cache size.

In practice, we only consider a fraction of the basic blocks of the program, i.e., the ones with a substantial contribution to the execution time. This will speed up the algorithm significantly. We rule out any basic block with execution time less than a user-defined threshold. The complexity of the algorithm is $O(($number of basic blocks$)*h)$, where h is the height of the *LabelTree*. The maximum *LabelTree* height is 2^d, where d is the depth of the deepest nested loop (usually a small integer).

A basic block will not be selected for placement in algorithm *Allocate()* if any of the following is true:

- The algorithm finds that the basic block was too large to fit in the L-Cache. This can be either because the size of the block is larger than the cache size, or because it cannot fit at the same time with other, more important, basic blocks. The algorithm described in this section is used to implement this criterion.
- Its execution frequency is smaller than a threshold, and is thus deemed unimportant.
- It is not nested in a loop. There is no gain in placing such a basic block in the L-Cache since it will be executed only once for each invocation of its function.
- Even if its execution is large, its *execution density* might be small. For example, a basic block that is located in a function which is invoked many times might have a large execution frequency, but it might only be executed a few times for every function invocation. We define the execution density of a basic block as the ratio of the number of times it is executed to the number of times that the function in which it belongs is invoked.
- Finally, a very small basic block is not placed in the L-Cache even if it passes all the above criteria. The extra jump instructions that might be needed to link it to its successor basic blocks will be an important overhead in this case.

A basic block is placed in the L-Cache only if it is expected to stay there for a long period of time without getting replaced. This in effect decouples the communication between the I-Cache and the L-Cache and reduces the traffic between them.

C.1 Example

We refer to Fig. 3 to show how the algorithm works. We consider the basic block with the largest contribution in the execution time first; in that case B_4, which belongs to LabelSet S_4. Basic block B_4 is the first to be considered and can be placed in the L-Cache without any conflict. We set the variable *size* of S_4 equal to the size of B_4, i.e., $0.3C$. We also set the *size* of all the LabelSets between S_4 and the root to $0.3C$.

We continue with B_1, which belongs to S_1. Basic block B_1 can also be placed in the L-Cache since no conflict arises. The *size* variable of S_1, S_3 and S_8 are set to $0.6C$. Next, B_2 is placed in the L-Cache, but the *size* of S_3 and S_8 do not change.

Basic block B_5 is not in a leaf; therefore, we need to use the $DOWN\_TRAV()$ function to propagate the effects of the inclusion of B_5 on its descendants. Since $size(S_4) + size(B_5) = 0.4C < C$, we can place B_5 in the L-Cache. We also set $size(S_5) = size(S_6) = 0.3C + 0.1C = 0.4C$. We continue in this manner, and we only select a basic block if it does not create a conflict with a block that has already been selected. We notice that B_6 cannot be placed in the L-Cache because $size(B_6) + size(S_5) > C$.

D. *Fourth and fifth steps: Global placement in the memory*

Step (4) in our methodology is the placement of the basic blocks in the global address space. The algorithm takes as input the placement of the basic blocks with respect to the L-Cache and tries to minimize the necessary space as much as possible. Extra jump instructions are inserted in Step (5) to retain the semantics of the original program.

In Fig. 5, a complete example of the original and the restructured CFG is shown for the code of Fig. 2. Blocks B_1 and B_2 will overlap in the L-Cache since B_2 will be executed only when the loop of B_1 exits. On the other hand, if B_3 overlapped with B_2 or B_1, it would miss in every execution of the $B_3 \rightarrow B_1$ loop.

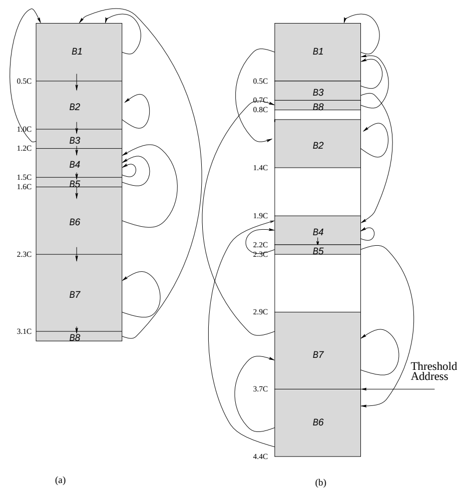

Fig. 5. CFG restructuring example.

The user has the ability to adjust the thresholds in the selection of the basic blocks in the first stage, and trade off performance degradation with power savings. For example, a smaller basic block frequency threshold will select more basic blocks for placement, leading to larger energy savings, and, possibly, to a larger delay, since these basic blocks will need extra jump instructions to retain the semantics of the code and will create larger conflicts in the L-Cache. In the extreme case, the user can either select every basic block to be placed in the extra cache, or can disable the L-Cache altogether. In the former case, the scheme emulates a filter cache organization, whereas, in the latter case, it emulates the original scheme that has no extra cache. These two extremes are subsets of our compiler-driven scheme. The method is very flexible, and individual applications can choose from a range of caching policies.

V. HARDWARE ENHANCEMENTS

In addition to the compiler enhancement, our scheme requires extra hardware for the implementation of the L-Cache scheme. This is shown in Fig. 6.

The *program counter* (PC) is presented to the L-Cache tag at the beginning of the clock cycle. The L-Cache tag will only output its tag if the *blocked_part* signal is on. This signal is generated by the instruction fetch unit (IFU), and its meaning is explained later. In that case, the comparator checks for a match, and if it finds one, it instructs the multiplexer to drive the contents of the L-Cache in the data path. At the same time, the data portion of the L-Cache asserts its output and sends the new instruction to the data path. The I-Cache is disabled for this clock cycle, since the signal *blocked_part* is on.

In the case of an L-Cache miss (*LCache_Hit* is off), the I-Cache controller activates the I-Cache in the next clock cycle and gets the referenced instruction from there. At the same time, this instruction is transfered to the L-Cache. Note that the L-Cache and I-Cache are only accessed sequentially and never in parallel. If *blocked_part* = off, the I-Cache controller activates the I-Cache without waiting for the *LCache_Hit* signal. In this way, the L-Cache can be bypassed without a delay penalty.

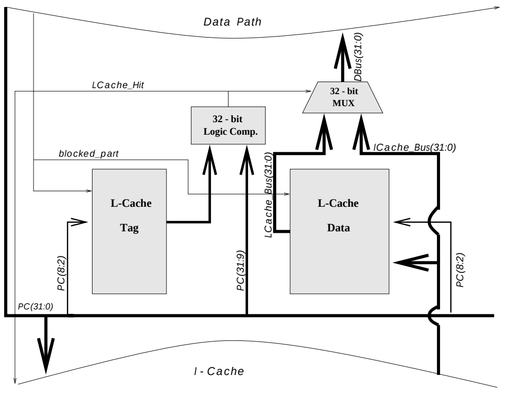

Fig. 6. L-Cache organization.

Recall that the compiler has already laid out the code so that the basic blocks that are destined for the L-Cache are placed before the others. A 32-bit register is used to hold the address of the first non placed block in the main memory layout. If during program execution the PC has a value less than that address, the 32-bit comparator will set *blocked_part* = on, else this signal will be set to off. In the former case, the machine will attempt to access the L-Cache first, whereas, in the latter case, it will bypass the L-Cache and it will try to fetch the instruction from the I-Cache. This way, the machine can figure out which portion of the code executes with only an extra comparison.

This simplification is only possible because of the way that the code has been restructured in the compilation phase. Notice also that if *blocked_part* = on, the L-Cache can still miss; this will happen, for example, when the basic block to be placed in the L-Cache has not been executed before, i.e., the first time the thread of control passes through it. Therefore, the tag portion of the L-Cache is still needed.

Finally, we extend the instruction set architecture (ISA), and we add an instruction called *alloc* which marks the boundary between the selected and the non selected code. This extra instruction is used to store the address of the first non-placed instruction in the 32-bit register, as described in the previous section, and it is the first instruction to be executed upon entry in a procedure. The ID stage of the pipeline will decode the instruction and place the address in the register. There is only one such instruction per function, and its effect on performance is negligible.

The comparator can have a negative impact on the clock cycle of the machine if its latency cannot be hidden. The actual impact (if any) depends on the specific machine, and whether the comparator will be placed in the critical path of the processor. In any case, the effect on performance should be considered in a real system during the performance versus energy trade-offs.

VI. ENERGY ESTIMATION

TABLE I
Memory subsystem configuration in the base machine.

Parameter	Configuration
L1 I-Cache	32KB/32/1/1/4/8
L1 D-Cache	32KB/32/2/1/4/8

We have developed our cache energy model based on the work by Wilson and Jouppi [16] in which they propose a timing analysis model for SRAM-based caches [17]. Our model uses runtime information of the cache utilization (number of accesses, number of hits, misses, input statistics, etc.) gathered during simulation, as well as complexity and internal cache organization parameters (cache size, block size, associativity, banking, etc.). A 0.8 μm technology with 3.3 volts voltage supply is assumed. These models are used for the estimation of energy in both the I-Cache and the L-Cache.

The utilization parameters are available from the simulation of the memory hierarchy. The cache layout parameters, such as transistor and interconnect physical capacitances, can be obtained from previous layouts, from libraries, or from the final layout of the cache itself. We use the numbers given in [16] for a 0.8 μm process.

VII. EXPERIMENTAL EVALUATION

A. Simulator Environment

We evaluated the effectiveness of our software/hardware enhancements by examining the energy savings on a set of SPEC95 benchmarks. The benchmarks were compiled with the MIPSpro compiler using the -O2 optimization flag. Hence, we enabled all the traditional optimizations but we disabled any interprocedural analysis and inlining. That was necessary in order to test our own inlining heuristic.

To gauge the effect of our L-Cache in the context of a realistic processor operation, we simulated the MIPS2 instruction set architecture (ISA) using the *MINT* [18] and the *SpeedShop* [19] tool suites. MINT is a software package for instrumenting and simulating binaries on a MIPS machine. We built a MIPS2 simulator on top of MINT which accurately reflects the execution profile of the R-4400 processor. Table I describes the memory subsystem configuration as (cache size / block size / associativity / cycle time / latency to L2 cache in clock cycles / transfer bandwidth in bytes per clock cycles from the L2 Cache). Both I-Cache and D-Cache are banked both row-wise and columnwise to reduce the access time and the energy per access [16]. We use the tool *cacti*, described in [16], to estimate the access time of the on-chip caches, as well as the optimal banking that minimizes the access time.

The L-Cache was 256 and 512 bytes, and had a block size of 4 bytes, i.e. the size of a MIPS instruction. A larger block size does not significantly increase the hit rate of the L-Cache, whereas it negatively affects the dissipated energy per access. The L2 unified cache is off-chip and its energy dissipation is not modeled.

We also experimented with different scenarios for the user-given thresholds that guide the basic block selection and placement in the L-Cache (Table II). A more aggressive scenario results in larger energy gains at the expense of larger performance degradation. A frequency threshold of 0.01%, for example, will force the tool to mark for placement only basic blocks that have an execution time of at least 0.01% of the total execution time of the program. A size threshold of 10 instructions will force the tool to mark only the basic blocks that have at least 10 instructions, and so on. Different parameters are selected for the FP and integer programs based on the different features of these programs. Our experimental methodology was as follows. First, we ran the benchmarks to collect the profile data. The data were used to drive the inline and the block placement heuristics. The tool, along with the restructuring of the body of the program, selected various statistics regarding the quality of the generated code. SpeedShop was used for profiling and the MIPSpro compiler was used for compilation and code optimization. The actual simulation was done using MINT. Function inlining was used only for the SPECint95 benchmarks. Through experimentation, we found out that inlining is more beneficial when only leaf functions are absorbed; hence, we limit our inlining procedure to consider only leaf functions.

B. Results

The percentage of dynamic instructions that cause the machine to access the L-Cache in the course of program execution is shown in Table III. This access may result in either a hit or a miss. This percentage is high for all the SPECfp95 benchmarks, reflecting the efficacy of our approach for these programs. As expected, a larger L-Cache is more succesful in storing basic blocks and therefore in disabling the I-Cache for a larger period of time. In some cases, even a small L-Cache is capable of effectively shutting-down the I-Cache for the duration of the program execution. The law of diminishing returns applies here as well, since a very large L-Cache (1024 instructions) is usually as succesful as smaller ones. In most cases, a 256 instruction L-Cache approximates the performance of an infinite size L-Cache.

On the other hand, most integer benchmarks do not have a large number of basic blocks that can be cached in the L-Cache. They are also insensitive to the cache size variation, which is to be expected since the basic blocks of integer programs are generally small. Most of the basic blocks of the SPECint95 benchmarks are not nested, or they are nested within a loop that contains a function call; hence, they cannot be included in the L-Cache. Integer programs with complex control flow graphs, such as interpreters, compilers and so on, have a large number of different paths in the CFG. These benchmarks have the worst behavior. Benchmarks with a more regular structure (compression programs, simulators, etc.) are better suited to our algorithm.

Table IV shows the energy gains in the I-Cache subsystem for the three different L-Cache configurations. The numbers are normalized with respect to the energy dissipation of the original scheme. The energy in the modified configurations is due to both the I-Cache and L-Cache. A result less than one is desirable since it denotes an improvement in energy or delay with respect to the original scheme.

The performance overhead of these cache configurations with respect to the original execution time is given in Table V. This is a full chip simulation that takes into consideration the latency in the memory hierarchy, the structural hazards in the FPU, and the data dependency hazards in both the integer unit and the FPU.

A very important feature of the L-Cache approach is the small performance overhead, which is vital for high performance machines. The performance overhead is due to the miss rates in the L-Caches and the extra jump instructions that are inserted by the compiler as discussed previously. The scheme gains benefit from the very high hit rates of the compiler-managed L-Cache.

An optimal L-Cache has a size of 128 instructions (i.e., 0.5-Kbytes) for the FP benchmarks. Small caches are not very succesful in disabling the I-Cache. Larger caches, on the other hand, have larger energy dissipation per access, yet not a much better hit rate than average sized caches. The energy dissipation drops as the size increases, but it goes up again for the larger caches.

VIII. MODIFIED SCHEME FOR INTEGER BENCHMARKS

Integer benchmarks do not perform well under the loop-based selection algorithm as we have explained in the previous section. Most of the basic blocks in the SPECint95 benchmarks are not nested; hence, they cannot be placed in the L-Cache during execution.

The previous methodology was based on the detection of nested basic blocks in loops which did not contain function calls. These basic blocks were candidates for compiler-driven placement in the L-Cache. As is evident from the experimental results, the method is not succesful for a large category of integer benchmarks, such as interpreters and compilers. Figure 7

TABLE II

User-given thresholds in the L-Cache experiments.

Experiments	*Frequency Thres.*		*Size Thres.*		*Exec. Density Thres.*	
	FP	INT	FP	INT	FP	INT
Aggressive (a)	0.01%	0.01%	5	5	5	5
Less Aggressive (b)	0.5%	0.5%	10	5	10	5
Moderate (c)	1%	1%	20	5	20	5

TABLE III

L-Cache utilization statistics: percentage of instructions that cause an access to the L-Cache.

Bench.	*L-Cache size*					
	32 in.	64 in.	128 in.	256 in.	512 in.	1024 in.
tomcatv	15.2%	40.2%	90.0%	99.8%	99.9%	99.9%
swim	0.1%	72.7%	91.7%	99.9%	99.9%	99.9%
su2cor	10.0%	53.3%	66.9%	98.2%	99.2%	99.2%
hydro2d	22.6%	39.4%	39.4%	94.5%	94.5%	94.5%
go	7.4%	7.4%	7.4%	7.4%	7.4%	7.4%
compress	15.5%	15.5%	15.5%	15.5%	15.5%	15.5%
li	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
perl	8.7%	8.7%	8.7%	8.7%	8.7%	8.7%

TABLE IV

Normalized energy relative to the base machine for a 256-byte and 512-byte extra L-Cache.

Benchmark	*256 B L-Cache*			*512 B L-Cache*		
	(a)	(b)	(c)	(a)	(b)	(c)
tomcatv	0.565	0.622	0.622	0.141	0.198	0.198
swim	0.312	0.318	0.347	0.139	0.145	0.173
su2cor	0.498	0.511	0.511	0.373	0.389	0.389
hydro2d	0.418	0.438	0.493	0.260	0.261	0.261
go	0.952	0.974	0.986	0.951	0.974	0.986
compress95	0.873	0.875	0.875	0.873	0.875	0.875
li	1	1	1	1	1	1
perl	0.934	0.94	0.949	0.934	0.94	0.949

TABLE V

Normalized delay relative to the base machine for a 256-byte and 512-byte extra L-Cache.

Benchmark	*256 B L-Cache*			*512 B L-Cache*		
	(a)	(b)	(c)	(a)	(b)	(c)
tomcatv	1	1	1	1	1	1
swim	1	1	1	1	1	1
su2cor	1	1	1	1	1	1
hydro2d	1	1	1	1	1	1
go	1.012	1.012	1.012	1.017	1.013	1.013
compress95	0.980	0.979	0.979	0.979	0.979	0.979
li	1	1	1	1	1	1
perl	1	1	1	1	1	1

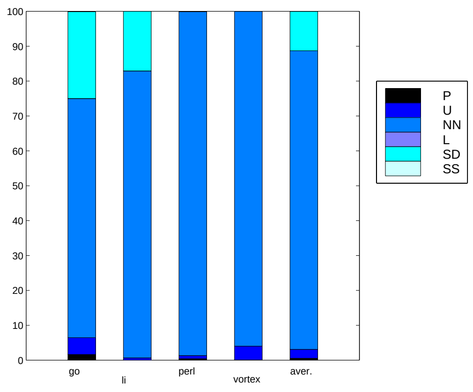

Fig. 7. Instruction placement results for the SPECint95 benchmarks with a 512-byte L-Cache.

gives insight into the failure of the algorithm for some of the integer benchmarks.

Shown is the classification of the dynamic mix of instructions for the most troublesome SPECint95 benchmarks for a 0.5-Kbyte L-Cache. An instruction belongs to one of the six following categories: “P” if it has been selected by the algorithm to be positioned in the L-Cache, “U” if it is in a basic block with a small execution frequency (unimportant), “NN” if it is in a block with large execution frequency but not nested in a loop, “SD” if it is in a nested block with large execution frequency but small execution density, “SS” if it belongs to a nested block with large frequency and execution density but small size, and “L” if it satisfies all the above criteria but does not fit in the L-Cache. For this experiment, the frequency threshold is $\frac{1}{10000}$ of the execution time of the program, the execution density threshold is five executions per function invocation, and the size threshold is five instructions.

The single most important reason that disqualifies the basic blocks of the integer benchmarks from being cached is nesting. Most of the basic blocks do not belong to a loop, or they belong to a loop that has a function call (85% of them). More than 10% of the basic blocks have small execution density.

The problem seems to be inherent to the structure of integer programs, especially when they are written in C/C++. This programming methodology favors small sections of sequential code, procedural abstraction (many functions), and lack of very deeply nested loops. The execution time is distributed among a larger number of basic blocks, many of which do not execute many times per function invocation. An alternative approach for selection of blocks for the L-Cache is therefore appropriate for these programs.

The proposed solution selects a function and places its most important basic blocks permanently in the L-Cache. In other words, they are not replaced when the thread of control leaves the function. Naturally, we select the function with the largest contribution in the execution time, as this has been designated by the profile data. The method consists of two steps as before.

A. Function inlining and block placement

Before placement, our method performs function inlining to maximize the gains of this approach. The function with the largest execution time may contain function calls to other functions. If these functions are inlined, the contribution of the original function in the total execution time will increase.

After inlining, the heuristic selects the most frequently executed basic blocks of the inlining function. This selection is based on user-given compiler options. If all these basic blocks of the function fit in the L-Cache, the block placement algorithm will proceed to place them all. The size of the L-Cache is therefore important, unlike in the loop-based heuristic in which the integer benchmarks were almost insensitive to L-Cache size variations.

In general, the problem can take the form of the *0-1 Knapsack* problem which is NP-complete [20]: Given a finite set U of basic blocks bb, each one with a size $s(bb)$, a value $n(bb)$ which is the number of executed instructions in bb, and a positive L-Cache size C, find a subset $U' \subseteq U$ of basic blocks such that $\sum_{bb \in U'} s(bb) \leq C$ and such that $\sum_{bb \in U'} n(bb)$ is as large as possible. Since a basic block can either be placed in the L-Cache or not (we cannot place part of the block), an optimal solution requires exponential time in the number of basic blocks.

We apply a greedy approximation algorithm which works as follows: we order the set U of basic blocks by the "key": $\frac{n(bb)}{s(bb)}$ so that $\frac{n(bb_1)}{s(bb_1)} \geq \frac{n(bb_2)}{s(bb_2)} \geq \cdots \geq \frac{n(bb_n)}{s(bb_n)}$. Starting with U' empty, we proceed sequentially through the list, each time adding a basic block bb whenever the sum of the sizes of the blocks already in U' and bb does not exceed C.

In addition, we perform another greedy procedure in which the list has been sorted using only the number of cycles $n(bb)$ of each basic block, so that $n(bb_1) \geq n(bb_2) \geq \cdots \geq n(bb_n)$. The best solution among the two is selected. A near optimal solution is obtained using this approach in our experiment.

B. Experimental evaluation of the modified scheme

In the new experiments we did not set any size or density constraints. Since the basic blocks are placed in the L-Cache when the selected function is executed for first time and remain there afterwards, it s not necessary to pose extra limitations in their selection.

The memory hierarchy subsystem is described in Table I. The energy gains of the L-Cache is given in Table VI. The results are very encouraging for benchmarks that have poor performance under the initial method. Similar results are obtained for most of the integer benchmarks that do not score well under the old scheme (e.g. *130.li, 134.perl*).

TABLE VI

Normalized energy relative to the base machine for a 256-byte and 512-byte L-Cache, using the modified scheme for integer benchmarks.

Benchmark	*256-byte L-Cache*	*512-byte L-Cache*
compress95	0.808	0.776
li	0.286	0.269
perl	0.823	0.823

The execution time overhead is negligible in this scheme for an L-Cache of 0.5-Kbyte. This is because the hit rate is almost 100% and the L-Cache is large enough to accommodate all the important basic blocks of a function.

IX. CONCLUSIONS

In this research, we have developed techniques for hardware/software co-design in high-performance processors that result in energy/power reduction at the system level. To that effect, we make a more judicious use of one of the most power-consuming modules of a CPU, the I-Cache. The techniques we descibed are orthogonal to the standard circuit or gate level techniques that are traditionally used by designers to reduce power and can therefore be used to further reduce power consumption without impairing performance.

Since performance is the most important objective of today's high-end microprocessors, no energy reduction technique will be acceptable, unless it has only a marginal negative effect on the execution time, or unless its overhead can be hidden by other compiler/architectural techniques. If this is the case, even a moderate energy reduction will be welcome.

Most of the energy gains in high-performance and embedded processors alike will be extracted from the high level of the design flow, when the designers have not yet committed to major design decisions. Major energy gains can be obtained if the compiler and the hardware are designed with low energy in mind.

References

[1] J. Edmondon, "Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor," *Digital Technical Journal*, vol. 7, no. 1, pp. 119–135, 1995.

[2] D. Dobberpuhl, "The design of a high-performance low-power microprocessor," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 11–16, 1996.

[3] S. Manne, D. Grunwald, and A. Klauser, "Pipeline gating: Speculation control for energy reduction," in *Proceedings of the International Symposium of Computer Architecture*, pp. 132–141, 1998.

[4] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 437–445, Dec. 1994.

[5] V. Tiwari, S. Malik, A. Wolfe, and T.C. Lee, "Instruction level power analysis and optimization of software," *Journal of VLSI Signal Processing*, vol. 13, Aug. 1996.

[6] V. Tiwari and S. Malik and A. Wolfe, "Compilation techniques for low energy: An overview," in *Proceedings of the IEEE Symposium on Low Power Electronics*, Oct. 1994.

[7] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh, "Techniques for low energy software," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 72–75, Aug. 1997.

[8] J. Diguet, S. Wuytack, F. Catthoor, and H. De Man, "Formalized methodology for data reuse exploration in hierarchical memory mappings," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 30–35, Aug. 1997.

[9] S.Wuytack, F.Catthoor, L. Nachtergaele, and H. De Man, "Power exploration for data dominated video applications," in *Proceedings of the International Symposium of Low Power Electronics and Design*, 1996.

[10] J. Kin, M. Gupta, and W. Mangione-Smith, "The filter cache: An energy efficient memory structure," in *Proceedings of the International Symposium on Microarchitecture*, pp. 184–193, Dec. 1997.

[11] R. Bajwa, M. Hiraki, H. Kojima, D. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki, "Instruction buffering to reduce power in processors for signal processing," *IEEE Transactions on VLSI Systems*, vol. 5, pp. 417–424, Dec. 1997.

[12] E. Jacobsen, E. Rotenberg, and J. Smith, "Assigning confidence to conditional branch prediction," in *Proceedings of the International Symposium on Microarchitecture*, pp. 142–152, 1996.

[13] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis, "Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 70–75, Aug. 1998.

[14] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools.* Reading, MA: Addison–Wesley, 1986.

[15] S. McFarling, "Program optimization for instruction caches," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 16–27, June 1989.

[16] S. Wilson and N. Jouppi, "An enhanced access and cycle time model for on-chip caches," tech. rep., DEC WRL 93/5, July 1994.

[17] N. Bellas, I. Hajj, and C. Polychropoulos, "A detailed, transistor-level energy model for SRAM-based caches," in *Proceedings of the International Symposium on Circuits and Systems*, 1999.

[18] J. E. Veenstra and R. J. Fowler, "MINT: A front end for efficient simulation of shared-memory multiprocessors," in *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 201–207, 1994.

[19] *SpeedShop User's Guide.* Silicon Graphics, Inc., 1996.

[20] M. Garey and D. Johnson, *Computers and Intractability.* New York, NY: W.H.Freeman & Co., 1979.