Energy and Performance Improvements in Microprocessor Design using a Loop Cache

Nikolaos Bellas[*], Ibrahim Hajj, Constantine Polychronopoulos, and George Stamoulis[†]
Department of Electrical & Computer Engineering
and the Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, IL 61801

Abstract

Energy dissipated in on-chip 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.

In this paper we extend the work proposed in [1], in which an extra, small cache (called *filter cache*) is inserted between the CPU data path and the L1 cache and serves to filter most of the references initiated from the CPU. In our scheme, the compiler is used to generate code that exploits the new memory hierarchy and reduces the possibility of a miss in the extra cache. Experimental results across a wide range of SPEC95 benchmarks show that this cache, which we call *L-Cache* from now on, has a small performance overhead with respect to the scheme without any extra caches, and provides substantial energy savings. The L-Cache is placed between the CPU and the I-Cache. The D-Cache subsystem is not modified. Since the L-Cache is much smaller, and, thus, has a smaller access time than the I-Cache, this scheme can also be used for performance improvements provided that the hit rate in the L-Cache is very high. In our experimental results, we show that the L-Cache does indeed improve performance in some cases.

1 Introduction

There has been a growing interest in reducing the power consumption of a processor in the past few years. The increasing prevalence of portable computing has promoted energy efficiency from concern primarily of circuit designers to an issue of general interest to the computer architecture community.

Some architectural techniques can improve performance and reduce energy dissipation at the same time. For example, any data or instruction locality enhancement technique decreases the miss rates in the L1 caches and, as a side effect, reduces the energy since the CPU does not access larger and more "expensive" caches.

On the other hand, most of the aggressive compiler transformations for speculative processors increase the power dissipated since they force the machine to execute speculatively a lot of redundant instructions from paths that may actually never be taken. The extra computation, which can be a substantial portion of the dynamic instructions, entails power waste although it greatly improves performance. This speculation overhead will probably increase in future processors as speculation techniques become more aggressive [2]

[*]DigitalDNA Systems Architecture Labs., Motorola Corp., Schaumburg, IL

[†]Intel Corporation, Santa Clara, CA

The paper by Kin [1] tries to improve the Energy·Delay product in the embedded processor families by introducing the filter cache: a small, extra cache with size less than 1KB. Provided that the locality of instructions, and the data reuse are large, the filter cache can satisfy most of the references by the CPU. The filter cache is smaller and more power-efficient than the L1 Caches, thereby reducing the average energy dissipated by a memory reference. The price to be paid, however, is larger miss rates and execution time. The authors report a decrease of 58% in the energy dissipated in the cache subsystems (both I-Cache and D-Cache), but an increase of 21% in the execution time for a set of multimedia benchmarks [3].

Our approach alleviates the negative effects on performance by having the compiler generate code that exploits the new memory hierarchy, and selecting statically which instructions are to be placed in the L-Cache. The CPU will then access the extra cache (L-Cache) only when one of these instructions are to be fetched, and it will bypass it otherwise. Since most of the programs tend to execute frequently only a small subset of their instructions, the L-Cache can be used to capture these instructions and provide them to the CPU. The compiler will restructure the code so that it is easy for the hardware to figure out whether it executes a "selected" or a "non-selected" instruction, and accordingly access either the L-Cache or the I-Cache.

The remainder of the paper is organized as follows: Section 2 details the compiler transformations necessary for our scheme, while section 3 describes the hardware support, and the energy estimation method we used for the caches. Section 4 presents simulation results for both energy and performance on a subset of SPEC95 benchmarks, and section 5 discusses an extension of the method for the integer benchmarks. The conclusion is given in section 6.

2 Compiler Enhancements

In our approach, the compiler is given the duty to select the appropriate part of the code to be placed in the L-Cache, as well as to restructure the code so that the conflicts between placed instructions are minimized. We use profile data from previous runs of the program to select the best instructions to be cached. The selection is done on a per basic block basis. After selection, the compiler lays out the program so that the selected basic blocks that need to be placed at the same time in the L-Cache do not map to the same L-Cache location. To achieve this, the compiler performs two stages:

- *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 locality problems in the I-Cache. In our experiments, we only perform function inlining for the integer benchmarks.

- *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 L-Cache is maximized.

The block placement technique is explained in detail in the next section.

2.1 Block Placement

Conceptually, the block placement stage consists of three steps.

In the first step, the compiler selects the basic blocks which will be placed in the L-Cache using the profile information from a previous run. These block are the most frequently executed blocks in the program that are nested within loops. A basic block is placed in the L-Cache 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.

Our algorithm selects basic blocks of the program using the *Control Flow Graph (CFG)* and the profile data. A block is selected by the compiler unless at least one of the following criteria is met:

- 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.
- Its execution frequency is smaller than a threshold, and is thus deemed unimportant. The execution frequency of a basic block is the percentage of the execution time of the program which is due to the execution of this basic block.
- It is not nested in a loop. There is no point in placing such a basic block in the L-Cache since it will be executed only once for each invocation of the function in which the basic block belongs.
- Even if its execution frequency is large, its *execution density* might be small. For example, a basic block that is located in a function which is invoked a lot of times might have a large execution frequency, but it might only be executed 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 other requirements. The extra unconditional branch instructions that might be needed to link it to its successor basic blocks will be an important overhead in this case.

In the second stage, 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
```

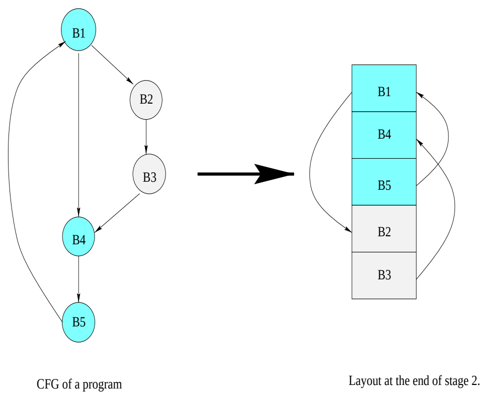

Figure 1: The CFG of a code, and the layout after the second stage

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 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 net effect of the second stage of the algorithm is that the selected basic blocks will be placed before the non-selected ones in the memory address space. Fig. 1 shows an example of the CFG of a portion of the code and the layout of the basic blocks after the second stage. Only basic blocks B_1, B_4, and B_5 have been selected in the first stage of the algorithm.

The structure of the program (as this is described by the CFG) is also important in minimizing the miss rate in the L-Cache. In the third stage, the compiler considers the nesting of the selected basic blocks and places them in the memory address space so that blocks that are in the same nesting do not map to the same L-Cache location. For example, B_6 in Fig. 2 should not map to the same L-Cache location as B_4 or B_5, since, in that case, the L-Cache would miss in every iteration of loop L_5 when it accesses B_6. The same goes for B_4 and B_5. On the other hand, B_1 and B_2 can overlap since B_2 is only executed when B_1 finishes.

The compiler uses the nesting information of the selected basic block, and tries to place them in such a position so that mapping conflicts in the L-Cache are minimized. Since the L-Cache is usually small, the reduction of mapping conflicts is a crucial step for the minimization of the misses in the L-Cache. The complete algorithm for the mapping of basic blocks in the L-Cache space area is given in [4]. The basic idea is that basic blocks that belong to different nesting levels should not map to the same position in the L-Cache. For example, B_4 and B_5 should not overlap in the L-Cache, as we explained in the previous paragraph.

The user has the ability to adjust the thresholds in the

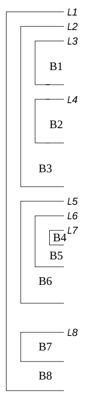

Figure 2: Nesting example

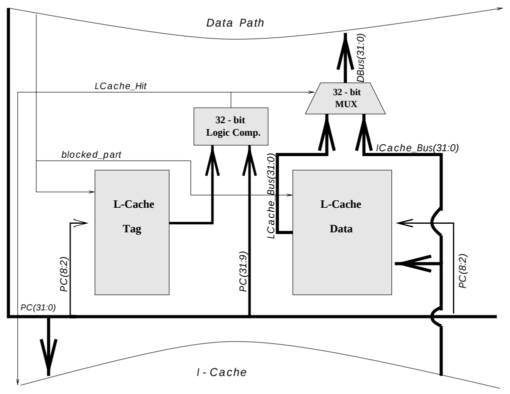

Figure 3: LCache organization

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. 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. Therefore, individual applications can choose from a range of caching policies.

3 Hardware Enhancements and Energy Estimation

The extra hardware needed to implement our scheme should allow the L-Cache to be bypassed when it executes basic blocks that are not selected by the compiler. The organization of the extra hardware is shown in Fig. 3. The L-Cache is filled with instructions from the higher levels of the memory hierarchy when it misses. It is assumed to be empty at the beginning of program execution.

The 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 case of a 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.

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 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. This way, the machine can figure out with only an extra comparison which portion of the code executes.

To imlement this, we add a new instruction to the instruction set architecture, called *"alloc"*, which marks the boundary between the placed and the non-placed code. This instruction is executed upon entry to a function, and is used to store this boundary address to the 32-bit register.

For the energy estimation in the I-Cache subsystem, we used an in-house cache energy model based on [5]. It is a detailed transistor-level model which uses run-time information for the caches (number of accesses, number of hits, misses, input signal statistics), and the complexity and internal cache organization (cache size, block size, associativity, banking). It is used for both the I-Cache and the L-Cache (or filter cache) [6].

4 Experimental Evaluation

In this section we describe the experimental results on energy savings and effects on performance of applying the proposed techniques to the SPEC95 benchmarks, and we compare our method with the filter cache method. We first outline the experimental setup.

4.1 Simulator Environment

To gauge the effect of our L-Cache in the context of a realistic processor operation, we simulated the MIPS2 ISA using the *MINT* [7] and the *SpeedShop* [8] 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 1 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 rowwise and columnwise to reduce the access time and the energy per access [5]. We use the tool *cacti*, described in [5], to estimate the access time of the on-chip caches, as well as the optimal banking that minimizes the access time.

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

Table 1: Memory subsystem configuration in the base machine

We considered a filter cache of 256 and 512 bytes, and block size that varied between 8 and 32 bytes. The L-Cache was also 256 and 512 bytes, and always 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 scenaria for the user given thresholds that guide the basic block selection and placement in the L-Cache (Table 2). 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 will force the tool to mark only the basic blocks that have at least 10 instructions, and so on.

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.

The next section delineates the energy and delay results for the filter and the L-Cache under the different scenaria described earlier. It also looks into potential performance gains using a faster clock, equal to the access time of a smaller I-Cache. We show that using our method, energy as well as delay can be simultaneously reduced, when the compiler assumes the role of statically allocating instructions to the L-Cache. For brevity, we only present the results for an extra cache of size 0.5KB.

4.2 Results

Using the configuration of Table 1, we performed an analysis across different organizations of the filter cache and the L-Cache. Table 3 shows the energy gains in the I-Cache subsystem for the three different L-Cache and filter 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 (or filter cache for the three last columns). A result

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 2: User given thresholds in the L-Cache experiments

less than one is good since it denotes an improvement in energy or delay with respect to the original scheme. Even a small delay penalty can be acceptable as long as the energy reduction is substantial.

The clock period was set equal to the access time of the D-Cache, which is the critical path in a lot of high performance processors. A 32KB, 2-way set associative D-Cache, with a 32 bytes block size has an access time of 11.4ns using a 0.8um feature size (as found using *cacti*).

Benchmark	*512 bytes extra cache*					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	32B
tomcatv	0.141	0.198	0.198	0.084	0.104	0.156
swim	0.139	0.145	0.173	0.092	0.114	0.174
su2cor	0.373	0.389	0.389	0.110	0.124	0.173
hydro2d	0.260	0.261	0.261	0.088	0.112	0.172
compress95	0.873	0.875	0.875	0.310	0.248	0.271
li	1	1	1	0.359	0.377	0.280
perl	0.934	0.94	0.949	0.421	0.32	0.308

Table 3: Normalized energy relative to the base machine for 512-byte extra cache

For the SPECfp95 benchmarks, a 0.5KB L-Cache is almost as successful as the filter cache in reducing the energy of the I-Cache subsystem especially when an aggressive scenario is followed. Filter caches with a 32 bytes clock size have large energy consumption per memory access, and they need a 32 byte large bus to connect them with the I-Cache. The filter caches capture all the instructions, no matter what their nesting is or how often they execute.

The performance overhead of these cache configurations with respect to the original execution time is given in Table 4. This is a full chip simulation of R-4400 that takes into consideration the latency in the memory hierarchy, the structural hazards in the FPU, and the data dependency hazards both in the integer unit and the FPU.

Benchmark	*512 bytes extra cache*					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	32B
tomcatv	1	1	1	1.011	1.006	1.04
swim	1	1	1	1.011	1.006	1.004
su2cor	1	1	1	1.141	1.133	1.128
hydro2d	1	1	1	1.007	1.004	1.002
compress95	0.979	0.979	0.979	1.126	1.063	1.035
li	1	1	1	1.175	1.103	1.068
perl	1	1	1	1.211	1.131	1.084

Table 4: Normalized delay relative to the base machine for 512-byte extra cache

The most important advantage of the L-Cache with respect to the filter cache is the small performance overhead, which is vital for high performance machines. The performance overhead is due to the (small) miss rates in the L-Caches and the extra jump instructions that are inserted by the compiler as discussed previously. Filter cache configurations suffer from a much larger miss rate.

The previous tables identify the opportunity for performance gains if the designer exploits the smaller access time of the extra caches. By reducing the clock period delay along with energy can be simultaneously reduced. This concept is particularly attractive for our compiler-driven scheme, since it can benefit from the very high hit rate in the L-Cache.

We present one such modification. We set the clock period equal to the access time of the I-Cache and we modify the size of the I-Cache so that it becomes the critical path in the CPU. We present results for a direct-mapped I-Cache of 8KB, with a block size of 16 bytes. The new clock period is 7.91ns. Notice that, in the this case, the access time for the D-Cache becomes two clock cycles. If the L-Cache can satisfy most of the requests from the pipeline, then the smaller I-Cache will not severely affect the hit rate. In this experiment we make the implicit assumption that the critical path can be reduced to 7.91ns either by pipelining or because no other path in the original implementation had a delay larger than 7.91ns.

Again, we denote the execution time of the original configuration that uses no extra caches as unity, and normalize everything else with respect to that. The extra cache size varies again between 256 and 512 bytes. We should also note that other such ideas can be readily applicable for enhancing performance, by reducing the clock period. For example, the D-Cache, as opposed to the I-Cache, can be made smaller. Then, we need two clock cycles to access the larger I-Cache. Due to lack of space, we only present results for the first modification.

Benchmark	*512 bytes extra cache*					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	32B
tomcatv	0.818	0.812	0.812	0.820	0.816	0.810
swim	0.823	0.823	0.823	0.833	0.829	0.824
su2cor	0.784	0.784	0.784	0.799	0.794	0.789
hydro2d	0.785	0.784	0.784	0.789	0.787	0.783
compress95	0.860	0.860	0.860	0.961	0.917	0.898
li	0.907	0.907	0.907	1.039	0.990	0.955
perl	0.989	0.989	0.989	1.119	1.064	0.984

Table 5: Normalized delay relative to the base machine for a 512-byte extra cache, and a direct-mapped, 8KB I-Cache with block size of 16 bytes

By applying this framework to our simulator, we observed that energy as well as delay can be reduced (Table 5). This is especially true for the L-Cache, since the compiler can guide the hardware to access either the L-Cache or the I-Cache, and avoid the unacceptably high miss rates of the filter cache for some programs.

Integer benchmarks do not perform well under this selection algorithm. Most of the basic blocks in the SPECint95 benchmarks are not nested, and, hence, cannot be placed in the L-Cache during execution. From a performance point of view, however, the L-Cache is still preferable to a filter cache in a processor that runs integer code. In the next section, we analyze an alternative approach for the selection and placement of basic blocks in the L-Cache, that is better suited to non-numeric code.

5 Modified scheme for integer benchmarks

The previous methodology was based on the detection of nested basic blocks in loops that are executed a large number of times. These basic blocks became candidates for compiler-driven placement in the L-Cache. Figure 4 shows why the algorithm fails for some integer benchmarks.

The figure displays the classification of the dynamic instructions for the most troublesome SPECint95 benchmarks for a 0.5KB 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 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 to the L-Cache. For this experiment, the frequency threshold is chosen 0.01% of the execution time of the program, the density threshold 5 executions per function invocation, and the size threshold 5 instructions.

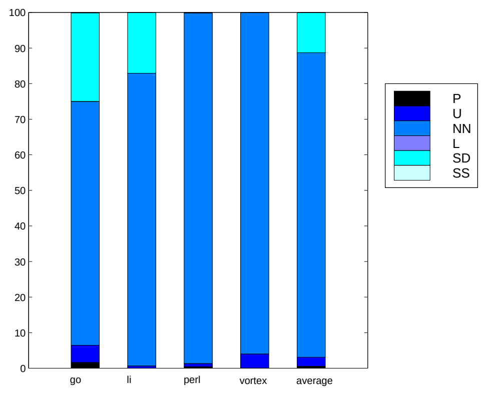

Figure 4: Instruction placement results for the SPECint95 benchmarks with a 0.5KB L-Cache

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 a function call (85% of them). In more than 10% of the cases, the basic blocks have small execution density.

The problem seems to be inherent to the structure of integer programs, especially when those are written in C/C++. This programming methodology favors small sections of sequential code, procedural abstraction, 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 more 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 to the execution time as this has been designated by the profile data. The method consists of two steps:

First, our method performs function inlining to maximize the gains of this approach. For example, 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 will increase and a larger number of basic blocks will be exposed in that function.

After inlining, the heuristic selects the most frequently executed function. If all the important basic blocks of the function fit in the L-Cache, the block placement algorithm will proceed to place them all. A heuristic is used in case the L-Cache cannot accommodate all the important basic blocks at the same time.

In that case, 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.

During execution, the L-Cache will be instructed to store all the selected basic blocks when the thread of control passes through them for the first time. These basic blocks will remain in the L-Cache thereafter. The hardware mechanism to achieve this are identical to the loop-based algorithm.

5.1 Experimental Evaluation of the Modified Scheme

The energy gains of the L-Cache are given in Table 6. The results are very encouraging for benchmarks that have poor performance under the initial method. On average, the energy dissipated in the I-Cache/L-Cache subsystem is 84.5% of the energy in the original I-Cache subsystem with almost no performance overhead. 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*). In the new experiments we did not set any size constraints for the selected basic blocks.

The execution time overhead is negligible in this scheme for an L-Cache of 0.5KB. 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.

Benchmark	*Energy*	*Delay*
compress95	0.776	0.979
li	0.869	0.981
perl	0.823	1

Table 6: Normalized energy and delay relative to the base machine for a 512-byte extra cache, using the modified scheme for integer benchmarks

6 Conclusions

This paper presented a paradigm for hardware/compiler codesign that targets activity minimization in a processor. These techniques 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. This paradigm describes a more judicious use of the I-Cache unit of a processor when the flow of control is caught within a loop. The compiler is given the responsibility to restructure the code. The aim is to minimize the overlap between basic blocks that are selected to be placed in an extra cache. This cache can be used instead of the larger, more capacitive, and more energy consuming I-Cache. The hardware mechanism that supports the scheme was shown to be very simple.

Since performance is the most important metric in today's high performance processors, no energy reduction technique will be attractive unless it has only a marginal effect on performance, or its overhead can be hidden by other compiler/architectural techniques. If this is the case, even a moderate energy reduction will be welcome.

References

[1] 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.

[2] M. Lipasti and J. P. Smith, "Superspeculative microarchitecture for beyond ad 2000," *IEEE Computer*, vol. 30, pp. 59–66, Sept. 1997.

[3] C. Lee and M. Potkonjak and W.H. Mangione-Smith, "Mediabench: A tool for evaluating multimedia and communication systems," in *Proceedings of the International Symposium on Microarchitecture*, Dec. 1997.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] G. Kane and J. Heinrich, *MIPS RISC Architecture*. Englewood-Cliffs, NJ: Prentice Hall, 1992.