January 1997 UILU-ENG-97-2201
DAC-57

University of Illinois at Urbana-Champaign

Architectural Support for Power Reduction in High Performance Microprocessors

Nikos Bellas, Ibrahim Hajj, and
Constantine Polychronopoulos

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-97-2201 (DAC-57)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Intel
6c. ADDRESS (City, State, and ZIP Code) 1308 W Main St Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Santa Clara, CA 95052-8119
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Intel 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Santa Clara, CA 95-52-8119	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Architectural Support for Power Reduction in High Performance Microprocessors

12. PERSONAL AUTHOR(S)
Bellas, Nikos; Hajj, Ibrahim; Polychronopoulos, Constantine

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	97 01 30	20

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Low power, hardware/software codesign, computer architecture, compilers, microprocessors

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Prompted by demands in portability and low cost packaging, the microprocessor industry has started viewing power, along with area and performance, as a decisive design factor in todays microprocessors. Most of the research in recent years has focused on the circuit, gate and RT-levels of the design. In this paper, we focus on the software running on a microprocessor and we view the program as a power consumer. Our work concentrates on the role of the compiler in the construction of ''power-efficient'' code, and especially its interaction with the hardware so that unnecessary activity is saved. We propose a technique that effectively shuts-down the Instruction Fetch Unit (IF) of a processor when the execution thread is caught within a loop. This mechanism can have very substantial power savings, since the IF unit is the main power consumer in most of todays high-performance microprocessors.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 *Previous editions are obsolete.*

Architectural Support for Power Reduction in High Performance Microprocessors *

Nikos Bellas, Ibrahim Hajj,
and Constantine Polychronopoulos

Coordinated Science Laboratory
College of Engineering
University of Illinois at Urbana-Champaign

January, 28th 1997

Abstract

Prompted by demands in portability and low cost packaging, the microprocessor industry has started viewing power, along with area and performance, as a decisive design factor in todays microprocessors. Most of the research in recent years has focused on the circuit, gate and RT-levels of the design. In this paper, we focus on the software running on a microprocessor and we view the program as a power consumer. Our work concentrates on the role of the compiler in the construction of "power-efficient" code, and especially its interaction with the hardware so that unnecessary activity is saved. We propose a technique that effectively shuts-down the Instruction Fetch Unit (IF) of a processor when the execution thread is caught within a loop. This mechanism can have very substantial power savings, since the IF unit is the main power consumer in most of todays high-performance microprocessors.

*This work was supported by Intel Corp.

1 Introduction

In recent years, power dissipation has become a major design concern for the microprocessor industry. In CMOS and BiCMOS technologies, the chip components draw power supply current only during a logic transition. Although this is an attractive characteristic of these circuits, it makes the power consumption dependent on the switching activity inside these circuits. In other words, the same circuit will dissipate a different amount of power under different input vectors.

The problem of the power waste 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. On the other hand, power dissipation is rapidly becoming the major bottleneck in today's systems integration and reliability. No reliable industrial packaging technology exists nowadays that can handle more than 50 Watts. Modern microprocessors are indeed hot: the Power PC from Motorola consumes 8.5 W, the Pentium Processor 16 W, and the Alpha chip from DEC around 30 W [1].

A new 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 [2] and tested in different architectures [3, 4]. This methodology attempts to relate the power consumed by a microprocessor to the software that executes on it. This is different from the often used "bottom-up" approach in which power models are built using a layout, gate or RT-level model of each unit and the power consumption of the whole chip is the sum of the power consumed by each component unit.

A new instruction-level power model is proposed in these studies, in which the authors characterize each instruction of a given microprocessor in terms of the power it dissipates when it is executed [5]. This is the linking bridge between the low-level concept of power

dissipation and the high-level concept of software that runs on a microprocessor. It can also provide the means for power minimization through software techniques or through the interaction between software and hardware which has been unexploited thus far.

During the execution of a program, each instruction activates a number of units from the moment it is being fetched to the moment it retires. This activation causes power dissipation which can be different for each instruction since each one activates different modules of the CPU. Characterization of the power caused by each instruction can lead to a better understanding of the sources of power dissipation in the microarchitectures, to compiler techniques that generate "power-efficient" code, or even to co-design of the microarchitecture, the instruction set and the compiler for low-power microprocessors.

There has been little research done in the field of software-based power minimization. In [6], a brief review of some compiler techniques that are of interest in power minimization is presented. The problem of register allocation, which is central in the code generation phase of a compiler, is solved aiming at the minimization of switching activity in [7]. In [8], a Gray coding technique for the program counter of a processor is presented which causes less switching in the buses of the CPU. Also a heuristic for the scheduling of instructions in a dynamic scheduling machine is suggested so that the instruction which causes less switching is selected by the scheduler. These approaches, however, can achieve only a very limited power reduction in real, complex microprocessors with millions of transistors, sometimes at the expense of execution time.

Our approach, presented in this paper, aims at architectural level support for power reduction, coupled with compiler techniques which take advantage of the new hardware feature. We consider a technique which not only avoids negative impact on execution time, but which can potentially reduce it further. The organization of the paper is as follows: the next section gives a brief background of the problem of power minimization and motivates our work. Section 3 describes our approach in general terms, and Section 4 discuses some subtle implementation issues. Section 5 presents experimental results on

SPEC95 benchmarks and the paper is concluded with Section 6.

2 Background and Motivation

The power consumed by a CMOS circuit (Figure 1) is given by the formula:

$$P = \frac{1}{2} V_{dd}^2 f C_{out}, \tag{1}$$

where V_{dd} is the supply voltage, f is the clock frequency, and C_{out} is the physical capacitance at the output of the circuit. However, (1) assumes that the output of the circuit switches and charges the output capacitance in every clock cycle. This is not always the case, since a change at the input of a circuit does not always propagate to the output. Therefore, a more accurate formula is:

$$P = \frac{1}{2} V_{dd}^2 f C_{out} p, \tag{2}$$

where p is the probability that the output of the circuit will toggle during a clock cycle. The term fp expresses the number of output toggles per unit time at the output of the circuit. Therefore, the power consumed in a circuit depends on the toggling of its output or, more generally, the power consumed by a unit depends on the activity within it.

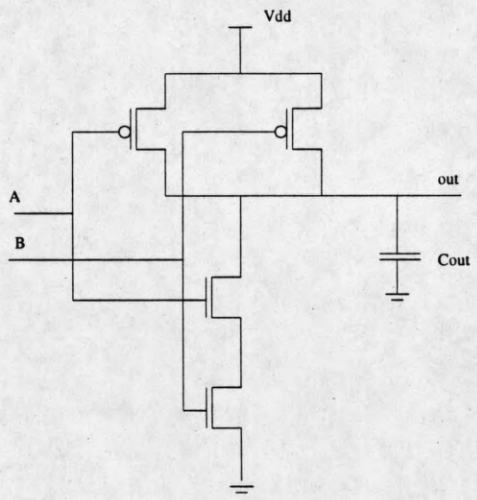


Figure 1: CMOS circuit

Larger and more active units in a microprocessor are expected to consume more power, and should therefore be the target of power minimization. Most of the techniques that have been investigated in the area of low power design try to minimize the physical capacitance C_{out} and the probability p of switching. Different circuit design techniques, logic and high-level synthesis techniques have been proposed that tackle the problem from the lower levels of the design hierarchy.

An additional problem that designers have to cope with is that low power and high performance are two conflicting goals at all levels of the design hierarchy. For example, one of the main low power optimizations is the reduction of the supply voltage to a circuit. This reduction, however, results in slower circuits and larger clock cycles. Higher frequencies are desirable for high performance but not for power. Higher activity (and thus utilization) results in a larger throughput but higher power as well. The excessive power consumption of today's processors is in part the outcome of very high utilization of its components.

Our work, on the other hand, focuses on the critical aspect of hardware/software co-design which is widely accepted as an effective means for performance optimization, yet it is so far unexploited in low power design. We are targeting the activity caused by the Instruction Fetch (IF) unit which is the main power consumer in most of todays microprocessors. The reason is that the execution rate of a processor depends critically on the rate at which the instruction stream can be fetched from the Instruction Cache (I-Cache). The IF unit should therefore be able to provide the datapath of the machine with a continuous stream of instructions, and has therefore very high activity. In addition to that, it has to drive large capacitance wires to the I-Cache.

3 Architectural Support for Power Optimization

From the previous discussion, it follows that the IF unit of processors should be our main focus for drastic power cuts. In order for such optimizations to be attractive, they should

not have a negative impact on performance. During the progran execution, the IF unit frequently repeats its previous tasks over and over again: if a program is caught in a loop, the IF unit will fetch the same instructions to the CPU core, and the ID will decode 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 IF unit. This approach works for performance but it unnecessarily performs more work than is needed, and thus it dissipates a lot of power.

To illustrate this point, and also introduce our modification, let us refer to the following code written in a MIPS-like format:

```
        add    r1,r0,r0
        addi   r2,r0,#100
label1: addi   r3,r0,#20
label2: lw     r4,0(r5)
        add    r1,r1,r4
        addi   r5,r5,#4
        subi   r3,r3,#1
        bnez   r3,label2
        subi   r2,r2,#1
        bnez   r2,label1
```

There are only ten different instructions in this program but the IF unit will fetch $100*20*5+100*3+2 = 10,302$ instructions in the ideal case if no false branch predictions were made. Substantial power gains could be achieved if we could reduce the amount of

instructions that the IF unit fetches, and subsequently disable it for all the time that 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.

This is the basic motivation of the architectural support that is proposed in this paper. All the instructions that belong to the inner loop can be fetched and decoded only the first time the thread of control passes through them. Subsequently, they can be stored in a special internal cache which is placed between the decoder and the later stages of the instruction pipeline. Each time the branch prediction logic instructs the IF unit to fetch an instruction from within the loop, the already decoded instructions that reside in this cache can be used instead. In the ideal case, the IF unit can be shut down for the duration of the loop, as it does not need to operate, and its power dissipation can be saved. The same is true for the on-chip I-Cache as well.

It is worth noting that no performance degradation can result from this enhancement. On the contrary, since the time-consuming phases of fetch can be avoided, the performance can potentially be increased. The amount by which performance can be increased depends on how aggressively one pursues such architectural enhancements—for example how large this cache should be. The only difference between the new microarchitecture and a conventional one is the source that supplies the pipeline with instructions during loop execution, and the subsequent clock gating of the IF unit.

4 Implementation and Compiler Support

Let us consider, how the proposed enhancement can be implemented, and in particular how the compiler and hardware can manage this special cache, which we refer to as Loop Cache (L-Cache). In Figure 2, an implementation of our scheme is shown in a generic, five-stage RISC processor. The advantages of this scheme are more prominent in CISC architectures where variable instruction formats result in higher power consumption during the IF stage.

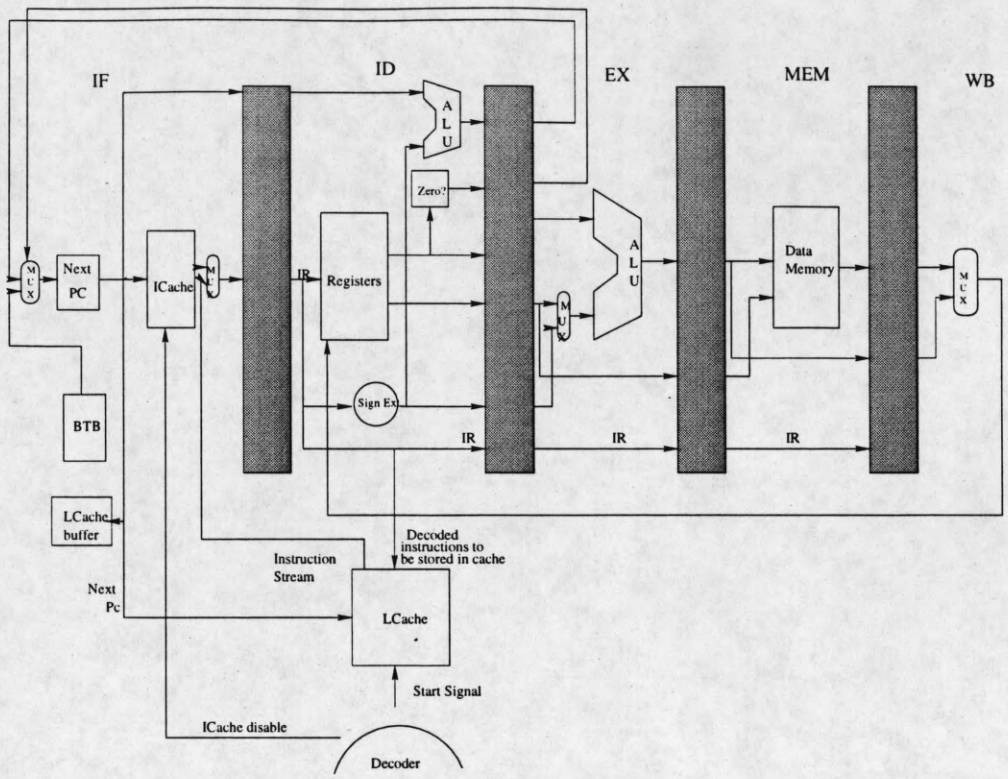


Figure 2: Block diagram of a proposed implementation

We propose a simple and flexible solution which exploits the compiler's knowledge of the structure of a program. In this work we describe a possible static implementation, which relies on the capability of the compiler to detect "good" candidate loops for caching, i.e. loops whose instructions can be stored in the L-Cache. Innermost loops with a number of instructions that can fit in the L-Cache are our main target. It is therefore a compiler-driven, or static method, which determines which instructions will be cached during compile-time.

The compiler inserts a special instruction in the header of such a loop to instruct the hardware to store the instructions of the loop in the L-Cache during the first loop iteration. In the previous example, the compiler inserts this additional instruction before the fourth instruction (the lw instruction) which instructs the processor to store the next five instructions (the inner loop) in the L-Cache. The compiler detects the most promising candidate loops during the code generation phase using static analysis or profiling techniques from

previous runs. The second time that the loop is executed the instructions are fetched from the L-Cache and not from the I-Cache. No performance or power penalty is paid except for the execution of this additional instruction. In the general case, however, the gains are significant, especially in CPU-intensive applications where most of the execution time is spent in loops.

The above scheme assumes that the execution of an iteration of the loop is always sequential without any branches, i.e. the *control flow graph (CFG)* of the loop is a single node. In the general case, however, one can have a loop like the one in Figure 3. There might be branches in the loop and therefore the first iteration of the loop might not pass through all the instructions and store them in the cache. Even worse, today's high performance compilers use techniques like loop-unrolling extensively which increases the size of a loop and creates additional basic blocks within it. Loop-unrolling makes the allocation of a whole loop body into the L-Cache more difficult. In addition to that, many basic blocks of the loop are not executed frequently, and therefore their caching in the L-Cache would be inefficient.

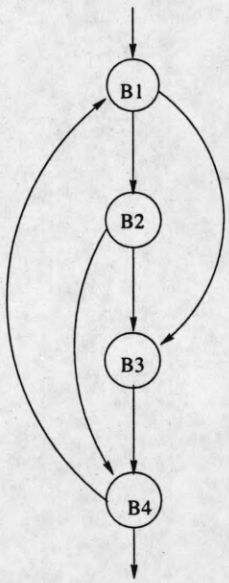


Figure 3: CFG of a loop

Prompted by this observation we modify slightly our initial scheme: we now insert this special instruction on a per basic block basis, rather than on a per loop. This greatly simplifies the allocation and de-allocation of the L-Cache to basic blocks and makes the bookkeeping for the allocation of more than one basic blocks in the L-Cache easier. Ideally, the compiler detects the most frequently executed basic blocks within a loop and will inserts the additional instruction *"alloc bb_addr, n"* at the beginning of the loop. The *bb_addr* operand refers to the address of the first instruction of the basic block to be cached, and n to the number of instructions in this basic block. With this scheme, all the special allocation instructions for basic blocks located in a loop will be placed at the header of the loopa, and will thus execute only only once per loop.

More than one basic blocks can reside in the L-Cache at any time. The space needed for their storage is allocated when the thread of control passes from the *alloc* instruction at the header of the loop. The compiler does not allocate a basic block if the L-Cache does not have enough entries to accommodate it. The de-allocation of all the basic blocks that currently reside in the L-Cache is done explicitly using the instruction *"dealloc"* at the end of the outermost loop that contains all the blocks currently in L-Cache.

This method of static allocation of basic blocks has some disadvantages stemming from the fact that the decisions are taken at compile and not at execution time. It is very difficult for the compiler to make the optimal allocation of basic blocks given a fixed L-Cache size. This problem is NP-complete (similar to the 0-1 knapsack problem) even if the compiler knew a-priori the execution profile of the program. In a pathological scenario the L-Cache would fill up with rarely executed basic blocks and it would be used scarcely. However, this whole scheme serves the purpose of power reduction. In other words, it should be simple and clean and should avoid excessive switching. A sub-optimal solution with respect to L-Cache hit rate can still result into a near-optimal solution with respect to power minimization. A more intricate caching scheme with dynamic allocation and de-allocation, might have resulted in a power consuming cache which would increase the total activity. The extra

activity caused by the dynamic scheme, combined with the more complex hardware support would most likely deteriorate rather than improve the method with respect to power.

In addition to this compiler enhancement, our scheme needs extra hardware for the implementation of the L-Cache scheme. The L-Cache itself should satisfy the following criteria:

- It should have a small number of entries (compared to the other on-chip caches). Since the L-Cache will only store frequently executed basic blocks which have a small number of instructions, it is not necessary to be large. Moreover, large caches dissipate more power since their access requires the switching of longer and more capacitive wires. A realistic size of the L-Cache is determined experimentally in the next sections for a specific target platform.

- It is preferable to use a cache organization that does not create much redundant computation for accessing a cache line. For example, in a fully associative organization, a large number of concurrent comparisons of tags take place, although only the result from one line is used. This concurrency is done for faster cache access but it consumes more power. From this perspective, a set associative organization or a direct mapped organization is preferable.

Note that the organization and the management of the L-Cache is much simpler than the organization of the I and D-Caches. There is no need for block replacement since the locality of the instructions of the loop will ensure that they will not be mapped into the same block. In this case, a direct mapped cache organization suffices. If we allow the caching of basic blocks across different subroutines, then a conflict may arise, and a set associative cache should be used instead. Under all scenarios, the following is always true: once an instruction is placed into the L-Cache, it cannot be removed until explicitly deallocated using the *"dealloc"* instruction. Also, no optimization techniques like instruction prefetching are useful since they will not have any effect on power dissipation.

These design criteria are selected because cache line replacement would likely stress the clock cycles and increase the critical path delay: if line replacement were allowed, the processor would have to search the L-Cache first, and then decide where to take the next instruction from. In case of an L-Cache miss, the instruction would be fetched from the I-Cache. In that case, the search would have to be done sequentially (that is, first check the L-Cache and then the I-Cache) in order to have any power savings if the L-Cache access were a hit. This double access would deteriorate the performance, especially in today's high clock rate processors. Using our scheme, we can save the searching of the L-Cache as we will describe shortly.

Along with the L-Cache, we need a buffer to hold information about the basic blocks currently stored in the L-Cache. There is an entry for each basic block in the L-Cache. Each such entry contains the range of the addresses of the instructions of the basic block that corresponds to this entry. In other words, the address of the first and the of the last instruction.

4.1 Determining next instruction supplier

During program execution, the flow of control will reach the instruction *alloc bb_addr, n* just before a loop which contains a basic block marked for caching is entered. The execution of this instruction will cause the processor to allocate n entries in the L-Cache, as well as one entry in the buffer that will correspond to this block. At that point, the range of addresses of the basic block are stored in the buffer. If the basic block is located in an inner loop, the special instruction is inserted at the header of the outermost loop that encompasses the basic block.

The processor checks if a basic block has an entry in the buffer when it starts executing this block. If not, then it fetches its instructions from the I-Cache as usual. When the flow of control reaches for the first time a basic block which has an entry in the buffer, the instructions are fetched from the I-Cache to the execution pipeline, and after decoding are

also stored to the L-Cache. The processor determines the first execution of a basic block in a loop using an extra bit in the buffer of this block. This is set when the *alloc* instruction is executed, and reset the first time that the basic block is entered.

In subsequent executions of the basic block, the bit is already reset, and the processor sets a flag to denote that the source of the instructions is now the L-Cache rather than the I-Cache. The flag will remain set as long as the processor executes in the specific basic block. It is reset when the Program Counter(PC), which is controlled by the branch prediction logic, starts fetching instructions outside the basic block. As long as the flag is set, the processor will ignore the I-Cache and will search the L-Cache. The instructions of this basic block are guaranteed to be in the L-Cache in their decoded form. Therefore, the I-Cache can be disabled for the duration of the basic block, resulting in large power savings. When the thread reaches the *"dealloc"* instruction, the buffer is cleared and the contents of the L-Cache are invalidated.

In high performance microprocessors the IF unit is more complex and contains buffers to decouple the fetching of the instruction from the decoding process. The gains will be more substantial since more units can be disabled during basic block execution. These units are normally active every cycle and have to drive large capacitances since they read data from the I-Cache. For example, instruction prefetching can be disabled since all the necessary instructions are within the CPU core (in the L-Cache).

The implementation we propose using the compiler is an example of how the hardware and the software can co-operate to reduce unnecessary activity in the CPU. The use of the compiler can offer a very inexpensive solution to the caching of decoded loop instructions. More elaborate solutions that offer greater flexibility can be incorporated, but the power overhead of the extra hardware should not exceed the gains.

5 Experimental Results

The method described in the previous section was tested on the SPEC95 benchmarks. We determined whether the bechmark programs are amenable to these modifications and what is the potential for power savings in them. We compiled, ran and profiled a set of benchmark programs using the MIPS compiler in a R4000 microprocessor. The next figures show the reduction in the number of instruction fetch/decode operations for each of the programs tested.

The left four bars denote the percentage of instructions in the dynamic mix that are cached during program execution. Each bar corresponds to a different cache size scenario. Infinite caches, and caches with 32, 128, and 1024 entries have been tested. All experiments assume a direct-map cache organization. The cache line size is 4 bytes, and it can store a MIPS instruction. The right four bars denote the percentage of the clock cycles for which the processor fetches the decoded form of instructions from the L-Cache and thus disable the I-Cache. The experiment also assumes that the compiler will statically determine the best basic blocks to be cached. This is an assumption which might be violated in practice.

For example, 94.83% of the executed instructions in tomcatv can be placed in the L-Cache if the L-Cache has 1024 entries or is infinite. This corresponds to the 92.67% of the execution time of this benchmark. The cache with 1024 entries corresponds to a 4K cache, which is very large for our purposes. Results for infinite caches are used as a theoretical upper bound and show the ability of the benchmark under consideration to be cached during execution.

From the results, it turns out that a cache with 1024 entries has the same characteristics as an infinite size cache. Smaller L-Caches have capacity conflicts and do not perform as well. A cache with 128 or 256 entries seems to be a satisfactory trade-off between power savings and area.

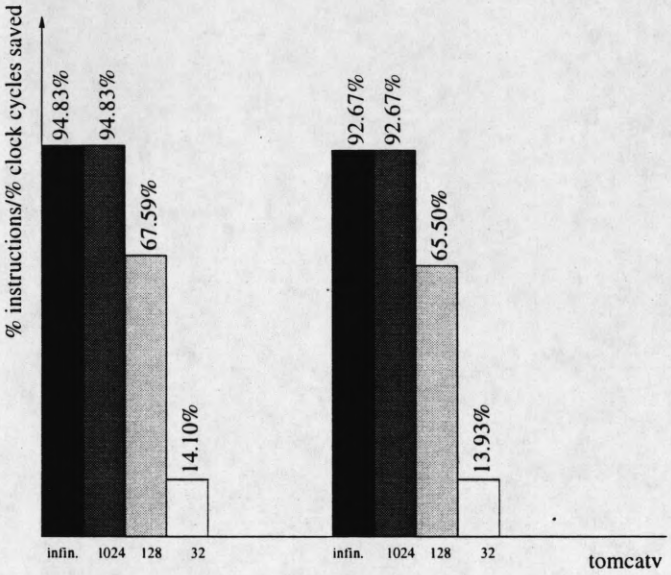
As expected, the floating point benchmarks have different behavior than the integer

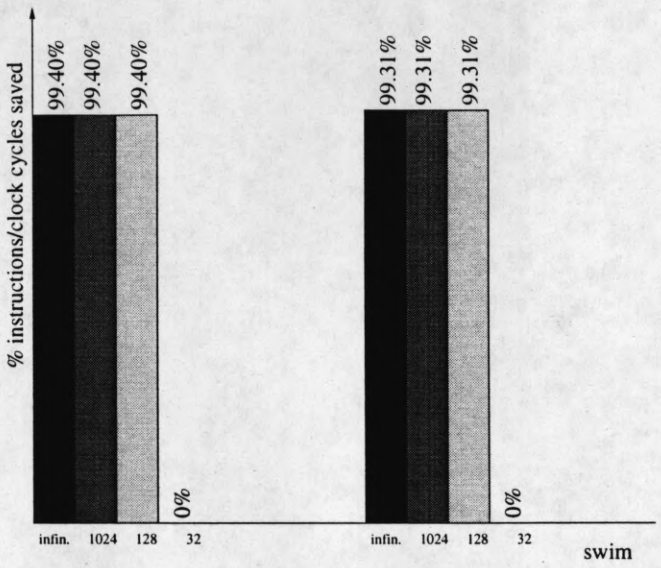
benchmarks. Since they are more CPU-intensive, and spend most of their execution time in loops, they are better apted to our scheme. The integer benchmarks, on the other hand, are less sensitive to the cache size since the size of the basic blocks is smaller, and the capacity conflicts are fewer. It is worth noting that the loop-unrolling optimization that is performed by the compilers in machines that use static scheduling, is detrimental in schemes with small L-Cache sizes. Machines that use dynamic scheduling to eliminate pipeline stalls will probably need smaller L-Caches to achieve a satisfactory performance.

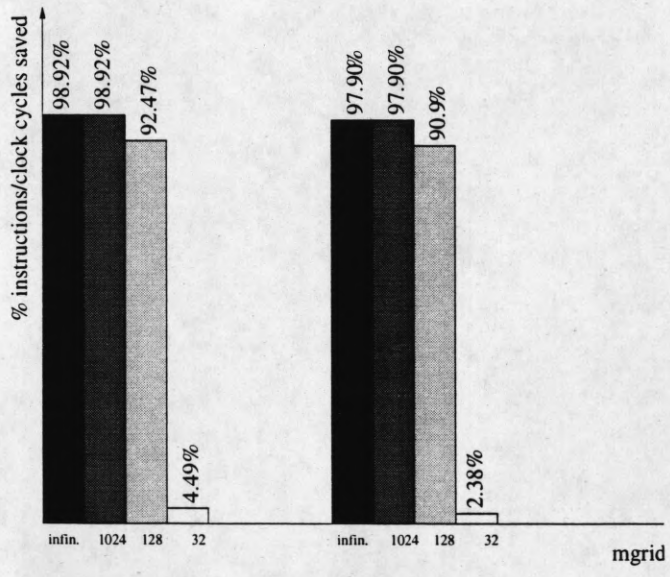
6 Conclusions

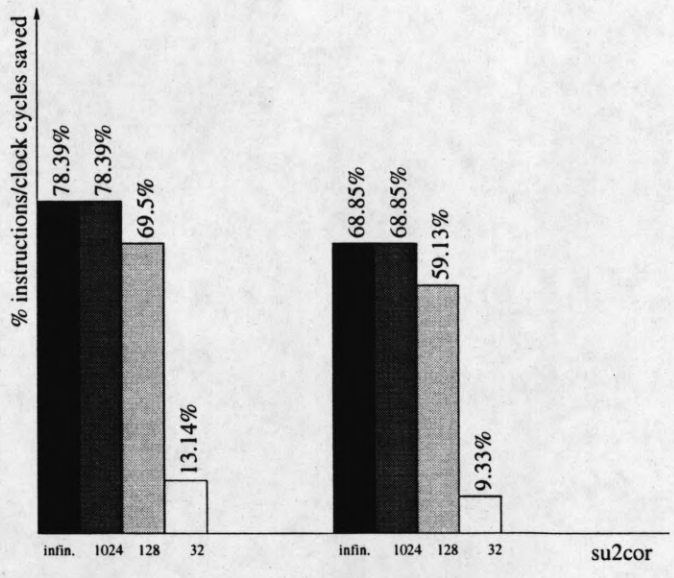
This paper presented a design paradigm for hardware/compiler co-design that targets activity minimization in a processor. These techniques are orthogonal to the standard circuit or gate level techniques that are traditionally used by designers to reduce power and can therefore be used to further reduce power consumption without impairing performance. This paradigm describes a more judicious use of the IF unit of a processor when the flow of control is caught within a loop.

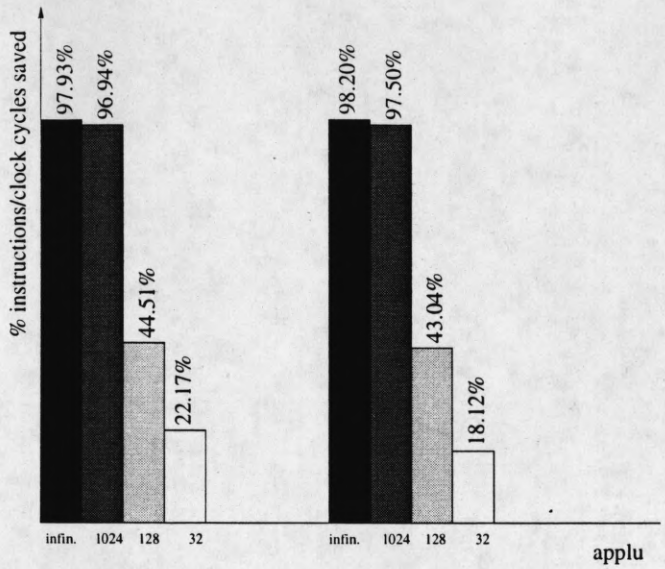
The gains from this modifications can be very important for machines with a very high power consumption in the IF unit. The savings are dependent on the structure of the program, and can be maximized for scientific computations with regular loop patterns.

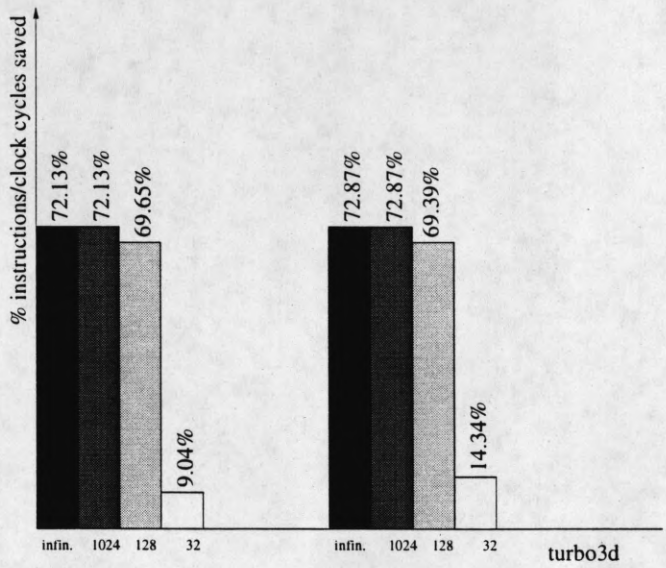

% instructions/% clock cycles saved
94.83%
94.83%
67.59%
14.10%
92.67%
92.67%
65.50%
13.93%
infin. 1024 128 32
infin. 1024 128 32
tomcatv

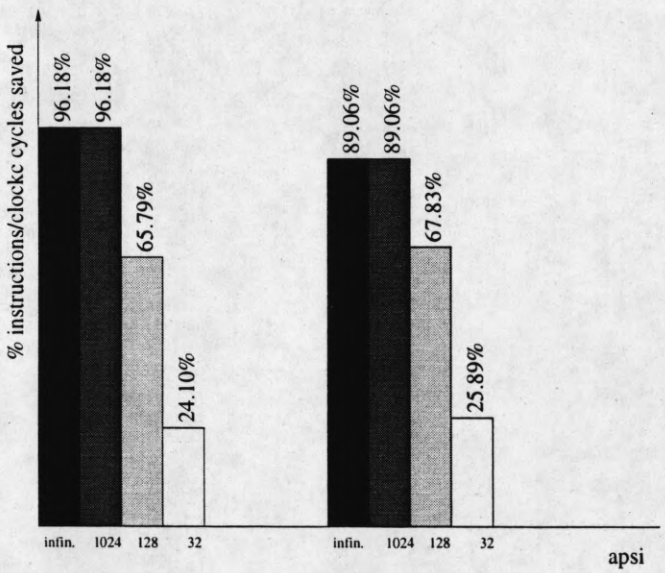

% instructions/clock cycles saved
99.40%
99.40%
99.40%
0%
99.31%
99.31%
99.31%
0%
infin. 1024 128 32
infin. 1024 128 32
swim

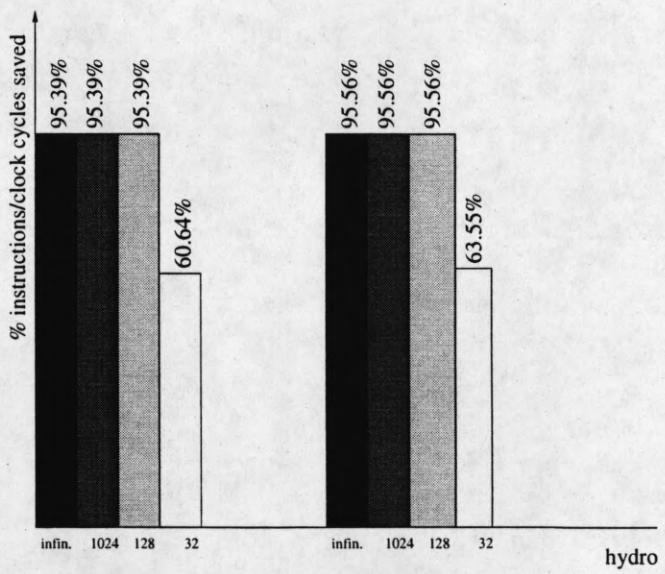

% instructions/clock cycles saved
98.92%
98.92%
92.47%
4.49%
97.90%
97.90%
90.9%
2.38%
infin. 1024 128 32
infin. 1024 128 32
mgrid

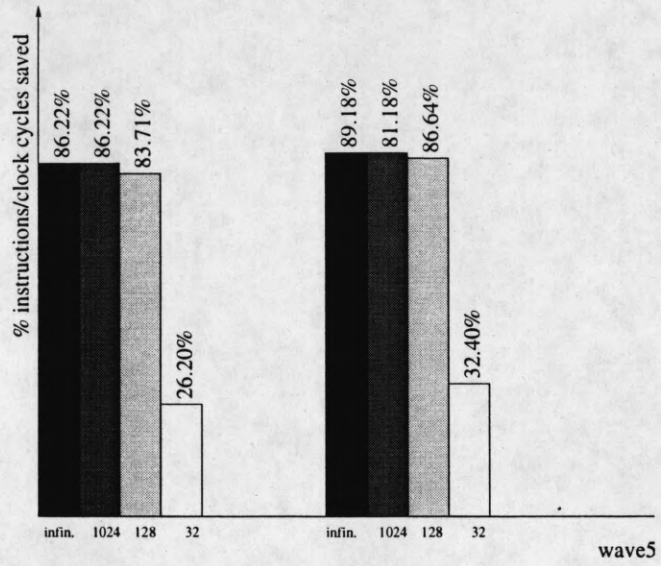

% instructions/clock cycles saved
78.39%
78.39%
69.5%
13.14%
68.85%
68.85%
59.13%
9.33%
infin. 1024 128 32
infin. 1024 128 32
su2cor

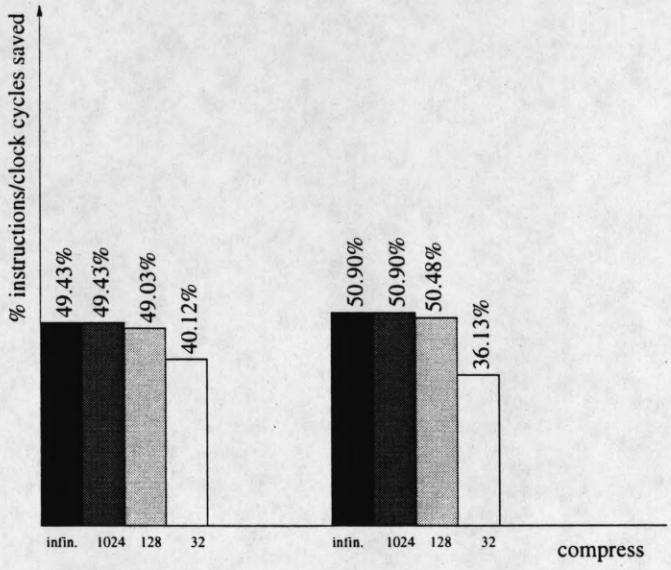

% instructions/clock cycles saved
97.93%
96.94%
44.51%
22.17%
98.20%
97.50%
43.04%
18.12%
infin. 1024 128 32
infin. 1024 128 32
applu

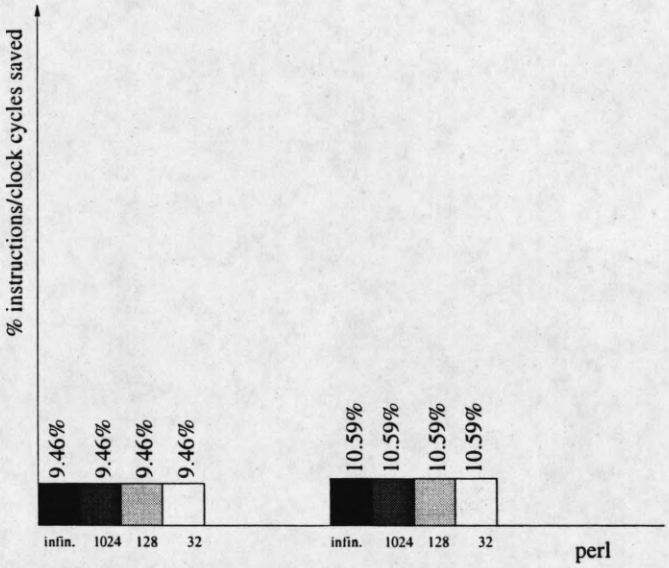

% instructions/clock cycles saved
72.13%
72.13%
69.65%
9.04%
72.87%
72.87%
69.39%
14.34%
infin. 1024 128 32
infin. 1024 128 32
turbo3d

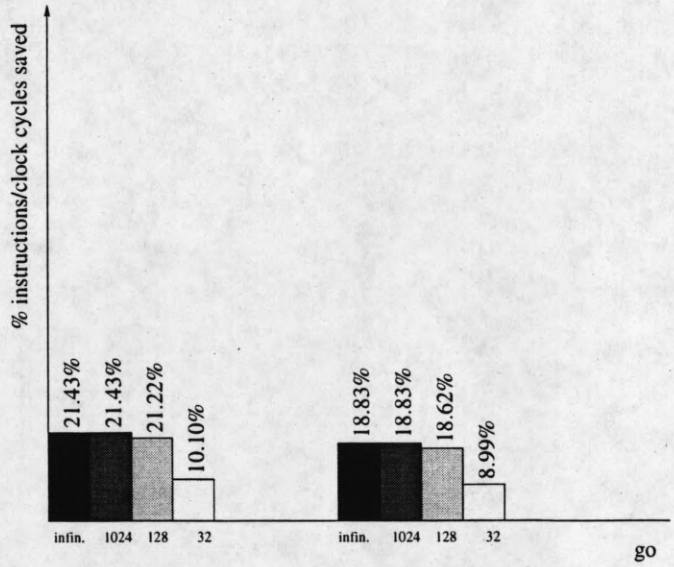

% instructions/clockc cycles saved
96.18%
96.18%
65.79%
24.10%
89.06%
89.06%
67.83%
25.89%
infin. 1024 128 32
infin. 1024 128 32
apsi


% instructions/clock cycles saved
95.39%
95.39%
95.39%
60.64%
95.56%
95.56%
95.56%
63.55%
infin. 1024 128 32
infin. 1024 128 32
hydro


% instructions/clock cycles saved
86.22%
86.22%
83.71%
26.20%
89.18%
81.18%
86.64%
32.40%
infin. 1024 128 32
infin. 1024 128 32
wave5


% instructions/clock cycles saved
49.43%
49.43%
49.03%
40.12%
50.90%
50.90%
50.48%
36.13%
infin. 1024 128 32
infin. 1024 128 32
compress


% instructions/clock cycles saved
9.46%
9.46%
9.46%
9.46%
10.59%
10.59%
10.59%
10.59%
infin. 1024 128 32
infin. 1024 128 32
perl


% instructions/clock cycles saved
21.43%
21.43%
21.22%
10.10%
18.83%
18.83%
18.62%
8.99%
infin. 1024 128 32
infin. 1024 128 32
go

References

[1] F. Najm, "A Survey of Power Estimation Techniques in VLSI circuits", *IEEE Transactions on VLSI Systems*, vol.2, pp. 446-455, Dec. 1994.

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

[3] V. Tiwari and T. Lee, "Power analysis of a 32-bit embedded microcontroller," in *Asia and South Pacific Design Automation Conference*, (Chiba, Japan), Apr. 1995.

[4] T. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power analysis and low-power scheduling techniques for embedded dsp software," in *International Symposium on System Synthesis*, (Cannes, France), Sept. 1995.

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

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

[7] J. Chang and M. Pedram, "Register allocation and binding for low power," in *Design Automation Conference*, pp. 29–35, IEEE/ACM, 1995.

[8] C. L. Su, C. Y. Tsui, and A. Despain, "Low power architecture design and compilation techniques for high performance processors," in *IEEE COMPCON*, pp. 489–498, IEEE/ACM, 1994.