A new scheme for I-Cache energy reduction in High-Performance Processors *

Nikolaos Bellas, Ibrahim Hajj and Constantine Polychronopoulos
Coordinated Sciences Laboratory, and
Electrical and Computer Engineering Department
University of Illinois at Urbana-Champaign, Urbana, IL 61801

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 today's microprocessors. To that effect, a number of research efforts have been devoted to architectural modifications that target power or energy minimization. Most of these efforts, however, involve a degradation in processor performance, and are, thus, deemed applicable only for the embedded, low-end market.

In this paper, we propose the addition of an extra, small cache between the I-Cache and the CPU, that serves to reduce the effective energy dissipated per memory access. In our scheme, the compiler generates code that exploits the new memory hierarchy and reduces the likelihood of a miss in the extra cache. We show that this is an attractive solution for the high-end processor market, since the performance degradation is minimal. We describe the hardware and compiler modifications needed to efficiently implement the new memory hierarchy, and we give the performance and energy results for most of the SPEC95 benchmarks. The extra cache, dubbed L-Cache from now on, is placed between the CPU and the I-Cache. The D-Cache subsystem remains as is.

1 Introduction

In the latest generations of microprocessors, an increasing number of architecture features have been exposed to the compiler to enhance performance. The advantage of this cooperation is that the compiler can generate code that exploits the characteristics of the machine and avoids expensive stalls We believe that such schemes can also be appliec for power/energy optimization by exposing the me ory hierarchy features in the compiler.

The filter cache [1] tackles the problem of larg energy consumption of the L1 caches by adding small, and, thus, more energy efficient cache be tween the CPU and the L1 caches. Provided tha the working set of the program is relatively small and the data reuse large, this "mini" cache can pro vide the data or instructions of the program an effectively shut down the L1 caches for long period during program execution. The penalty to be pai is the increased miss rates, and, hence, longer aver age memory access time. Although this might b acceptable for embedded systems for multimedi or mobile applications, it is out of the question fo high performance processors. The filter cache de livers an impressive energy reduction of 58% for 256-byte, direct mapped filter cache, while reduc ing performance by 21% for a set of multimedi benchmarks. Our approach has a very small per formance degradation with respect to the origina scheme without the filter cache, and smaller, bu still very large, energy gains.

We can alleviate the performance degradatio by having the compiler selecting statically, i.e. dur ing compile time, the parts of the code that are t be placed in the extra cache (the L-Cache), an restructuring the code so that it fully exploits th new hierarchy. The CPU will then access the L Cache only when instructions from the selecte part of the program are to be fetched, and it wi bypass it otherwise. Naturally, we want the mos frequently executed code to be selected by the com piler for placement in the L-Cache, since this i where most of the energy gains will come from.

The approach advocated in our scheme relie on the use of profile data from previous runs t select the best instructions to be cached. The uni of allocation is the basic block, i.e. an instruc tion is placed in the L-Cache only if it belongs t a selected basic block. After selection, the com piler lays out the target program so that the se

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

- 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 L-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.
- 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 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 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 satisfies all the above requirements. The extra branch instructions that might be needed to link it to its successor basic blocks will be an important overhead in this case.

The basic blocks are laid out in the memory address space so that all the selected basic blocks are placed contiguously before the non-selected ones. This arrangement greatly simplifies the hardware of the L-Cache as we will see in the next section. Branches are placed at the end of the blocks, if needed, to sustain the functionality of the code.

The user can trade off energy savings with delay increase by adjusting the thresholds as these were discussed in this section. For example, a smaller size threshold will probably lead to larger energy savings, and larger delay as well. As an extreme, the user can also completely disable the L-Cache, by forcing the compiler (through the user given thresholds) to generate the original code, or, on the other extreme, can emulate a filter cache scheme by selecting every basic block for placement in the L-Cache. The quality of the generated code can be determined by the user during compile time. Therefore, individual applications can choose from a range of caching policies.

3 Hardware modifications and Power Estimation

In addition to the compiler enhancement, our scheme needs extra hardware for the implementation of the L-Cache scheme. The extra hardware is shown in Fig. 2.

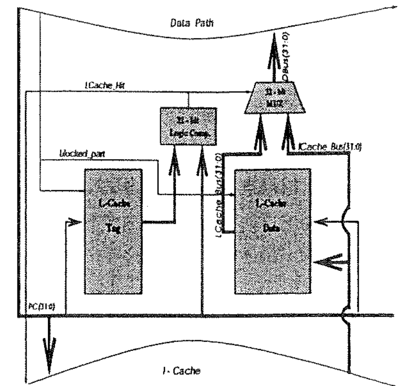

Figure 2: LCache organization

The organization of the L-Cache itself is depicted in Fig. 2. It needs the data and the tag part, an extra comparator for the tag comparison, and a 32-bit multiplexer which drives the data from the L-Cache or the I-Cache to the data path pipeline.

The functionality of the L-Cache is as follows: The *PC* is presented to the L-Cache tag at the beginning of the clock cycle. The L-Cache will only get activated if the *"blocked_part"* signal is on. This signal is generated by the IF unit, and its meaning is explained in the following paragraphs. 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. 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 is activated in the next clock cycle and provides the data. The I-Cache is accessed if the L-Cache misses, whereas the L-Cache is accessed only when *"blocked_part"* = on. If *"blocked_part"* = off, the I-Cache controller activates the I-Cache without waiting for the *"LCache_Hit"* signal. The two caches are always accessed sequentially and not in parallel.

We extend the ISA, and we add an instruction called *"alloc"* which has a J-type MIPS format. It is inserted by our tool and contains the address of the first non-placed block of the function. There is one such instruction for every function of the program and the address of *"alloc"* is stored when a function is entered. During the execution of the code in the function, if the *PC* has a value less than that address, the *"blocked_part"* signal is set else this signal will be set to off. This way, the machine can figure out which portion of the code executes with only an extra comparison.

We have developed our cache energy model based on [3]. This is a transistor level model which uses the run-time characteristics of the cache to estimate the energy dissipation of its main components [4]. A 0.8um technology with 3.3 Volt power supply is assumed. The cache energy is

5 Conclusions

We believe that, since performance is the most important objective of today's high-end microprocessors, no energy reduction technique will be acceptable, unless it only marginally affects the execution time, or its overhead can be hidden by other compiler/architectural techniques. If this is the case, even a moderate energy reduction will be welcome.

This paper presents a new, modified version of the filter cache in which the compiler and the extra hardware cooperate to decrease energy consumption. The compiler can select only the most important parts of the code to be placed in the extra cache, and can direct the hardware to probe the extra cache only when this code is to be fetched to the data path. The method is adaptive, since the user can aggressively pursue energy reductions to the expense of performance, or vice versa, by providing different compilation options.

References

[1] Johnson Kin, Munish Gupta and William Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," in *IEEE International Symposium on Microarchitecture*, pp. 184–193, Dec. 1997.

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

[3] S. Wilson and N. Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches," DEC WRL Technical Report 93/5, July 1994.

[4] N. Bellas, *PhD Thesis in progress*. 1998.

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

Benchmark	256 bytes extra cache					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	3
tomcatv	0.686	0.705	0.739	0.551	0.497	0.
swim	0.242	0.244	0.262	0.243	0.329	0.
su2cor	0.701	0.734	0.827	0.537	0.485	0.
hydro2d	0.418	0.438	0.493	0.397	0.413	0.
mgrid	0.943	0.980	0.986	0.600	0.530	0.
applu	0.736	0.755	0.840	0.541	0.490	0.
turb3d	0.618	0.622	0.839	0.426	0.430	0.
apsi	0.758	0.867	0.957	0.554	0.498	0.
wave5	0.768	0.822	0.822	0.576	0.517	0.
FP aver.	0.652	0.685	0.752	0.492	0.465	0.
go	0.948	0.955	0.955	0.612	0.545	0.
m88ksim	0.818	0.822	0.856	0.617	0.566	0.
compress95	0.890	0.897	0.897	0.569	0.523	0.
li	0.975	0.978	0.976	0.651	0.586	0.
INT aver.	0.908	0.913	0.921	0.612	0.555	0.

Table 2: Normalized energy relative to the b
machine for 256-byte

Benchmark	512 bytes extra cache					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	3
tomcatv	0.692	0.692	0.745	0.306	0.369	0.
swim	0.259	0.261	0.279	0.236	0.325	0.
su2cor	0.295	0.308	0.367	0.272	0.343	0.
hydro2d	0.263	0.259	0.299	0.241	0.326	0.
mgrid	0.225	0.245	0.279	0.249	0.330	0.
applu	0.666	0.689	0.787	0.333	0.446	0.
turb3d	0.431	0.432	0.696	0.350	0.387	0.
apsi	0.594	0.716	0.856	0.442	0.436	0
wave5	0.669	0.723	0.725	0.448	0.447	0
FP aver.	0.455	0.481	0.559	0.320	0.379	0
go	0.945	0.955	0.955	0.562	0.511	0
m88ksim	0.816	0.822	0.856	0.595	0.547	0
compress95	0.893	0.900	0.900	0.400	0.425	0
li	0.975	0.978	0.976	0.510	0.491	0
INT aver.	0.907	0.914	0.922	0.517	0.494	0

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

Benchmark	256 bytes extra cache					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	
tomcatv	1.002	1.001	1.000	1.322	1.168	1
swim	1.000	1.000	1.000	1.027	1.016	1
su2cor	1.001	1.001	1.000	1.308	1.157	1
hydro2d	1.038	1.032	1.023	1.174	1.091	1
mgrid	1.009	1.004	1.003	1.367	1.198	1
applu	1.080	1.052	1.032	1.314	1.165	1
turb3d	1.014	1.014	1.003	1.203	1.108	1
apsi	1.022	0.998	0.993	1.318	1.168	1
wave5	1.040	1.038	1.038	1.346	1.187	1
FP aver.	1.023	1.016	1.010	1.264	1.140	1
go	1.016	1.011	1.011	1.349	1.199	1
m88ksim	1.023	1.024	1.023	1.375	1.227	1
compress95	1.023	1.021	1.021	1.340	1.193	1
li	0.997	0.998	0.998	1.407	1.244	1
INT aver.	1.015	1.014	1.013	1.368	1.216	1

Table 4: Normalized delay relative to the base
chine for 256-byte extra cache

Benchmark	512 bytes extra cache					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	
tomcatv	1.000	1.000	1.000	1.083	1.049	1
swim	1.002	1.000	1.000	1.017	1.010	1
su2cor	1.010	1.003	1.001	1.050	1.026	1
hydro2d	1.052	1.030	1.026	1.021	1.011	1
mgrid	1.009	1.004	1.004	1.028	1.014	1
applu	1.240	1.137	1.100	1.236	1.122	1
turb3d	1.037	1.033	1.018	1.126	1.067	1
apsi	1.040	1.012	0.987	1.211	1.111	1
wave5	1.027	1.038	1.038	1.220	1.122	1
FP aver.	1.046	1.029	1.019	1.110	1.059	1
go	1.019	1.011	1.011	1.301	1.170	1
m88ksim	1.023	1.024	1.023	1.350	1.207	1
compress95	1.023	1.021	1.021	1.174	1.101	1
li	0.997	0.998	0.998	1.272	1.159	
INT aver.	1.015	1.014	1.013	1.274	1.159	

Table 5: Normalized delay relative to the base
chine for 512-byte extra cache