# L1 Data Cache Decomposition for Energy Efficiency*

Michael Huang, Jose Renau, Seung-Moon Yoo, and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

## ABSTRACT

The L1 data cache is a time-critical module and, at the same time, a major consumer of energy. To reduce its energy-delay product, we apply two principles of low-power design: specialize part of the cache structure and break the cache down into smaller caches. To this end, we propose a new L1 data cache structure that combines a Specialized Stack Cache (SSC) and a Pseudo Set-Associative Cache (PSAC). Individually, our SSC and PSAC designs have a lower energy-delay product than previously-proposed related designs. In addition, their combined operation is very effective. Relative to a conventional 2-way 32 KB data cache, a design containing a 4-way 32 KB PSAC and a 512 B SSC reduces the energy-delay product of several applications by an average of 44%.

## 1. INTRODUCTION

Continuous technical advances are fueling the trend toward more sophisticated and powerful chip designs. Unfortunately, they also lead to increased energy consumption. Currently, chips have a higher power density than a hot plate; if the current trend holds, before 2010 they will have a power density close to a nuclear reactor [9].

Reducing the energy consumption of processor chips is not an easy task. No single module in the processor is solely responsible for most of the energy consumption. Rather, energy consumption is spread across different modules, including for example the data cache, instruction cache, clock, branch predictor, and instruction window. In this paper, we focus on energy-efficient designs for the L1 data cache.

It is well known that smaller caches consume less energy per access and are faster. Consequently, a potentially energy-efficient design for L1 is to partition the L1 cache into several, smaller caches that can be probed independently. One cache organization that can use this approach is the *Pseudo Set-Associative Cache (PSAC)*. A PSAC is a set-associative cache that has more than one hit time [1, 2, 4, 5, 11, 13, 19]. Recent examples of PSAC organizations that use cache partitioning are the Predictive Sequential Associative cache [4] and the Way-Predicting Set-Associative cache [11].

Another approach towards energy efficiency is to specialize the cache to handle certain types of references particularly well. If these references are frequent, specialization can significantly improve both the performance and the energy savings of the cache. One proposed type of specialization is the stack cache [7, 16], which is designed to handle stack accesses. Stack caches can be effective because stack accesses are numerous and have a typical behavior that can be exploited.

In this paper, we propose an L1 data cache design that combines new PSAC and stack cache designs. The new stack cache, which we call *Specialized Stack Cache (SSC)*, is a small cache with two pointers. Using an algorithm that is not time critical, the hardware uses these pointers to reduce unnecessary write backs and line fetches. The new PSAC schemes are based on the concept of the *Phased* cache [10], whereby a cache access first activates the tag array before trying to access the data array.

Our complete L1 data cache design is shown in Figure 1. Individually, our PSAC and SSC designs have a lower energy-delay product than previously-proposed related designs. In addition, their combined operation is very effective. If, instead of a conventional 2-way 32 KB data cache, we use one of our 4-way 32 KB PSAC and our 512 B SSC, we reduce the energy-delay product of several applications by an average of 44%.

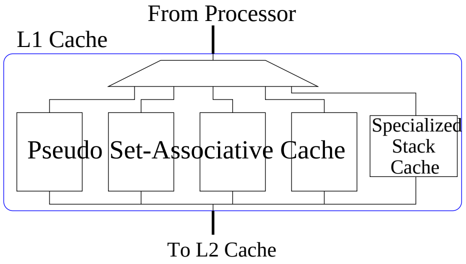

**Figure 1: Proposed L1 data cache organization.**

The paper is organized as follows: Section 2 describes our SSC; Section 3 describes our new PSAC organizations; Section 4 describes the setup that we use to evaluate the caches; Section 5 presents the evaluation; and Section 6 discusses related work.

## 2. SPECIALIZED STACK CACHE

Stack variables tend to have two interesting properties:

1. **It is easy to identify when a variable is dead.** A variable that resides in the stack can be live only as long as it is below the Top-Of-Stack (TOS) pointer. Once the TOS is lowered past the variable, the variable effectively becomes dead. This knowledge can be leveraged to reduce unnecessary write backs.

2. **It is easy to track whether or not a variable is initialized.** The typically small size of the stack makes it feasible to add simple hardware support to bound the range of memory lines that have *not* been displaced from the cache. This knowledge can be used on a write cache miss. If the write misses on a line whose address is inside this range, it is guaranteed that the line has not been initialized yet. As a result, the write miss can be serviced by simply allocating an empty line in the cache without fetching the line from L2 or memory.

*This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-0081307, and EIA-0072102, and by gifts from IBM and Intel.

*ISLPED'01*, August 6-7, 2001, Huntington Beach, California, USA.
Copyright 2001 ACM 1-58113-371-5/01/0008 ...$5.00.

These two properties have been used by Lee *et al.* [16] to reduce unnecessary write backs and line fetches. They have proposed specialized microarchitecture for the stack. Their design is an 8 KB circular buffer (Section 6).

However, if energy efficiency is a major goal, a different design is called for. Instead of a large circular buffer, we would like a cache for stack references that both is small and has only simple extensions over a conventional cache design. Our solution to this problem is what we call the Specialized Stack Cache (SSC).

The SSC is connected in parallel to the rest of the L1 data cache as shown in Figure 1. It receives all the stack references issued by the processor. It has two registers, namely the TOS and the *Safe Region Bottom (SRB)* registers. The TOS contains the address of the top of the stack. It is used to reduce unnecessary write backs. Specifically, any line above the TOS is dead and, therefore, if it is displaced from the cache, it does not need to be written back to L2.

The SRB is used in combination with the TOS to reduce unnecessary line fetches. We want to set the SRB such that we can guarantee that no useful data in the range of addresses between the TOS and the SRB has been displaced from the SSC. Cache lines in this range of addresses can contain either initialized or uninitialized data. A stack cache write miss in this range of addresses means that the corresponding memory line is uninitialized. As a result, there is no need to fetch the line from L2 or memory.

To guarantee these conditions, the SRB is set as follows. When a process is scheduled, we set the SRB to the TOS. As the process runs, the TOS may move below or above the SRB. If it moves below (leaving the SRB outside the stack), we reset the SRB to the TOS. If it moves above, we do not change the SRB. However, if a dirty line between the TOS and the SRB addresses gets displaced from the SSC, we set the SRB to point to the displaced address plus one memory line closer to the TOS. Note that such a displaced dirty line may or may not contain live variables, but we have to be conservative and assume that it does. If, instead, a non-dirty line between the TOS and the SRB is silently displaced, no action is taken because it contained uninitialized data. With this support, it is guaranteed that no useful data in the range of addresses between the TOS and the SRB has been displaced from the SSC cache.

Fortunately, checking the TOS or SRB registers occurs off the critical path, and updating them occurs infrequently. Specifically, checking mostly occurs on write misses and on dirty line displacements: on a write miss, the SRB is checked to see if a line fetch can be avoided; on a dirty displacement, the TOS and SRB are checked to see if a line write back can be avoided and if the SRB needs to be updated. On a TOS move, the SRB is also checked to see if it needs to move.

Updates may occur in the following cases. The SRB may be updated in a dirty line displacement. Both the TOS and SRB may be updated on a stack pointer move. Finally, they are both set to the top of the stack when a process gets scheduled.

Finally, for this system to work, we also need two other supports. The first one is to keep the SSC virtually tagged. If we tagged the SSC with physical addresses, stack addresses could become noncontiguous, which would complicate address comparisons with the TOS and SRB. Fortunately, using a virtually-tagged SSC also saves energy because address translation is eliminated.

The second support is to ensure that all stack references go to the SSC. If some of them went to the rest of L1, our algorithm would not work properly. Consequently, we need to detect stack references. A good way to do so has been proposed by Bekerman *et al.* [3]. In the decode stage, instructions that use the stack pointer are identified. Their accesses are later forwarded to the SSC. Since not all the stack accesses use the stack pointer, a snooping mechanism is used to redirect the remaining stack accesses to the SSC. With this solution, only the accesses redirected by the snooping mechanism to the SSC require one additional cycle. According to [3], only 1.2% of the accesses need any redirection.

Overall, the resulting SSC is very energy efficient: both unnecessary write backs and unnecessary line fetches are reduced, the SSC can be quite small, and it is virtually tagged.

## 3. PSEUDO SET-ASSOCIATIVE CACHE

Pseudo Set-Associative Caches (PSAC) are set-associative caches that have more than one hit time [1, 2, 4, 5, 11, 13, 19]. One way to organize a PSAC is by combining a set of smaller caches (also called *ways*) that can be probed independently (Figure 1). In such designs, probing one of these small caches is likely to be faster and less energy-consuming than accessing a conventional set-associative cache. Recent examples of PSAC designs that are based on multiple smaller caches are the Predictive Sequential Associative cache [4] and the Way-Predicting Set-Associative cache [11]. In these schemes, on an access, a prediction mechanism selects which way to probe. If the probe hits, the access is satisfied with high speed and low energy consumption. Otherwise, further probing is necessary.

To predict which way to probe first, recent PSAC schemes use a steering table [4]. The table is a small cache where each entry has a pointer to the predicted way in the set. The table is indexed with some prediction source. Examples of prediction sources are the address of the load or store instruction inducing the access, or the number of the base register used by the instruction [4].

If the first probe misses, different policies are possible. One approach is to sequentially probe all the remaining ways until the data is found or an L1 miss is declared [13]. We call this scheme *Sequential*. A second policy, used in the Way Predicting Set Associative cache [11], is to simultaneously probe all the remaining ways. We call this scheme *Fall Back Regular (FallBackReg)*, since “it falls back to a regular” associative cache. Overall, *Sequential* tends to be slow and conserve energy while *FallBackReg* tends to be fast and energy-consuming.

To obtain a better balance between energy and overall speed, we propose two new PSAC schemes. These schemes are based on the concept of the *Phased* cache [10]. The phased cache is a set-associative cache where an access first activates all the tag arrays. If there is a match, only the data array in the correct way is subsequently activated. Consequently, relative to a conventional cache, the phased cache saves energy at the cost of extra delay.

The first PSAC scheme that we propose is called *Fall Back Phased (FallBackPha)*. In this scheme, the first probe activates a predicted way (tag and data array). If it misses, the cache acts like a phased cache: in the second attempt, all the remaining tag arrays are activated and, if a match is detected, the data array in the correct way is subsequently activated. Overall, this scheme favors energy savings at the expense of speed.

The second PSAC scheme that we propose is called *Predictive Phased (PredictPha)*. In this scheme, the first probe activates all the tag arrays and the data array of the predicted way. If the prediction is incorrect, at least we know which way (if any) has the correct data. In this case, the data array of the correct way is subsequently accessed. Overall, this scheme favors speed at the expense of energy consumption. Note, however, that this scheme can use a unified tag array structure for the whole PSAC, which can likely be designed to be more energy-efficient than separate tag arrays.

The proposed PSAC schemes can use different algorithms to index the steering table and to select a line for replacement on a miss. Due to lack of space, our evaluation section (Section 5) only considers a small subset of the design space. Specifically, to index the steering table, we use the address of the load or store instruction. Such information is available very early in the pipeline. It can be shown that this scheme is as accurate for our applications as the more complicated schemes in [4].

Initially, the table is loaded with pointers randomly pointing to the different ways with a uniform distribution. In a more sophisticated design, the compiler could analyze the program and provide heuristics to initialize the table. However, this is beyond the scope of this paper.

For the line replacement algorithm, we examine two choices. One approach is to always load the line into the way selected by the steering table. This scheme is not very flexible and may cause cache conflicts. To alleviate this problem, we also examine an adaptive scheme. The scheme uses MRU bits in each set of the PSAC. In this scheme, at line fill time, if the predicted way happens to be the MRU way, we instead load the line into another, randomly selected way and update the table entry accordingly. While this approach may induce a higher number of mispredictions, it may also reduce the miss rate. This approach is called *Adaptive*. Note that, since we are targetting high-associative PSAC systems, LRU information is unavailable.

## 4. EXPERIMENTAL SETUP

To evaluate our cache designs, we use detailed software simulations at the architectural level. The simulations are performed using a MINT-based execution-driven simulation system [15] that models out-of-order processors.

### 4.1 Architecture

The baseline architecture is an out-of-order processor with two levels of cache. The architecture loosely models a MIPS R10000 processor. Table 1 lists the parameters used in the simulation. While the latency numbers in the table correspond to an unloaded machine, we model contention in the whole system. For the processor chip, we assume 0.18 $\mu$m technology.

| Processor | Caches | Bus & Memory |
|---|---|---|
| Freq: 1 GHz<br>Issue width: 4<br>Dyn issue: yes<br>I-window size: 96<br>Ld/St units: 1<br>Int,FP units: 2,2<br>Pending Ld,St: 8,8<br>BR penalty: 4 cyc | L1 size: 32 KB<br>L1 OC,RT: 1,3 ns<br>L1 assoc: 2<br>L1 line: 32 B<br>L2 size: 512 KB<br>L2 OC,RT: 4,12 ns<br>L2 assoc: 8<br>L2 line: 32 B | Bus: split transaction<br>Mem: 1-channel Rambus<br>Bus width: 16 bits<br>DRAM bandwidth: 2 GB/s<br>Mem RT: 81 ns |

**Table 1: Baseline configuration. BR, OC, and RT stand for branch, occupancy, and round trip from the processor, respectively.**

### 4.2 Energy

To evaluate the different cache organizations under different applications, we use the *energy-delay product* metric. For all cache configurations, we calculate latencies and energies for reads, writes, line fills, write backs, and cache misses using an extension of the CACTI tool [18] that we developed called XCACTI. XCACTI scales down the technology parameters to match 0.18 $\mu$m technology. It also searches for configurations with the lowest energy-delay product, delay, or energy. The search can be optimized based on several timing constraints, the expected ratio of number of loads to stores, and the cache miss rate. Based on our observations of our applications, we optimize the L1 and L2 cache organizations for 25% of writes, while the stack caches are optimized for 50% of writes. Because we are evaluating energy-efficient designs, we use latched sense amplifiers instead of the ones from Wada in CACTI [18]. The tool also includes a phased cache model.

Table 2 shows the energy consumed in a cache access for different stack caches and for the baseline L1 and L2 caches. The baseline L2 cache is a phased cache. The numbers include only the energy consumed in the corresponding cache structure, and not in buses or buffers outside the cache. The table shows the case of a read hit, write hit, read miss, and write miss.

| Access Type | Energy Per Access (pJ) | | | | | |
|---|---|---|---|---|---|---|
| | Direct-Mapped Stack Cache | | | | 2-Way | 8-Way |
| | 256B | 512B | 1KB | 2KB | 32KB L1 | 512KB L2 |
| Read Hit | 130 | 146 | 166 | 195 | 735 | 3201 |
| Write Hit | 141 | 157 | 180 | 214 | 780 | 3775 |
| Read Miss | 130 | 88 | 144 | 146 | 735 | 535 |
| Write Miss | 17 | 20 | 23 | 30 | 79 | 535 |

**Table 2: Energy consumed in a cache access for different stack caches and for the baseline L1 and L2 caches.**

Table 3 shows the energy consumed in a cache access for different PSAC organizations and associativities. As usual, the numbers include only the energy consumed in the corresponding cache structure, and are organized according to the type of access. Since an access may involve several sequential probes, each access type in the table has several columns corresponding to the different probes. Each column is labeled with the latency in cycles taken to complete the corresponding probe. For comparison, we also show an L1 phased cache. In all the caches, a write checks the tag first and does not access the data array until a hit is detected. This phased write architecture saves energy with minimal performance degradation.

The TLB is modeled after the TLB of the MIPS R10000 processor. It uses 40-bit physical addresses, has 64 entries, and is fully associative. The best possible XCACTI configuration requires 141 pJ per read. Its access time is short enough for the fastest caches considered.

The steering table has 1024 entries. For a 4-way PSAC, the total size is 2 Kbits. This structure spends 59 pJ per read. For our applications, it can be shown that the energy spent reading the table is on average about 4% of the energy spent in instruction fetching in the baseline configuration.

We model a 1-channel Rambus memory system. Intel expects that such a system will consume about 1.2 W [12]. For a single channel at full bandwidth, we assume that each cache line fill consumes 48000 pJ.

### 4.3 Applications

For our experiments, we use a mix of multimedia, SpecInt, memory intensive, and pointer intensive applications. We compile them with IRIX MIPS 7.3 with -O2. Each application generates several millions of data references.

The applications are as follows. *BLAST* is a protein matching application. The algorithm tries to match an amino acid sequence sample against a large database of proteins.

*BSOM* is a neural network that classifies data. It uses only fixed-point arithmetic.

*CRAFTY* is from SpecInt 2000. We use a data set with a reduced search depth that produces about the same data cache, instruction cache, and TLB miss rates as the reference set.

*GZIP* is from SpecInt 2000. We use a reduced data set that generates about the same data cache, instruction cache, and TLB miss rates as the reference set.

| PSAC Organization | Energy Per Access (pJ) | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4-Way 32 KB PSAC | | | | | | | | | | | | | | 2-Way 32 KB PSAC | | | | | | | |
| | Read Hit | | | | Write Hit | | | | Read Miss | | | Write Miss | | | Read Hit | | Write Hit | | Read Miss | | Write Miss | |
| | 2cy | 4cy | 6cy | 8cy | 4cy | 6cy | 8cy | 10cy | 2cy | 4cy | 8cy | 2cy | 4cy | 8cy | 2cy | 4cy | 4cy | 6cy | 2cy | 4cy | 2cy | 4cy |
| Sequential | 288 | 442 | 596 | 750 | 317 | 364 | 411 | 458 | - | - | 616 | - | - | 188 | 427 | 688 | 472 | 533 | - | 522 | - | 122 |
| FallBackReg | 288 | 750 | - | - | 317 | 458 | - | - | - | 616 | - | - | 188 | - | Same as Sequential | | | | | | | |
| FallBackPha | 288 | - | 536 | - | 317 | - | 458 | - | - | 295 | - | - | 188 | - | Not Interesting | | | | | | | |
| PredictPha | 324 | 430 | - | - | 352 | - | - | - | 189 | - | - | 82 | - | - | 444 | 644 | 489 | - | 78 | - | 78 | - |
| Phased | - | 422 | - | - | 451 | - | - | - | 82 | - | - | 82 | - | - | - | 489 | 534 | - | 78 | - | 78 | - |

**Table 3:** **Energy consumed in an access for different PSAC organizations and associativities. Since a PSAC access may involve several sequential probes, each access type in the table has several columns corresponding to the different probes. Each column is labeled with the latency in cycles (*cy*) taken to complete the probe. For comparison, we also show an L1 phased cache.**

*MCF* is from SpecInt 2000. We use an increased version of the test input set.

*MP3D* is an MP3 decoder. We use mpg123 version 0.59r, which is the fastest available UNIX GPL MP3 decoder. We reproduce three different samples, namely voice, cd, and hifi.

*MP3E* is an MP3 encoder. We use lame3.85, which is fast and widely used in the MP3 community. We encode two different samples, namely voice and cd.

*TREE* is the *Treeadd* pointer-intensive, recursive application from the Olden suite.

To estimate the working set of the applications, Figure 2 computes the miss rates of each application for several plain L1 data caches. The caches are 2-way set-associative. The figure does not show *MCF* because of its high miss rate. From the figure, we see that it is realistic to use a 32 KB L1 in our baseline system: L1 captures most of the working set of the applications but the system can still benefit from an L2 cache.

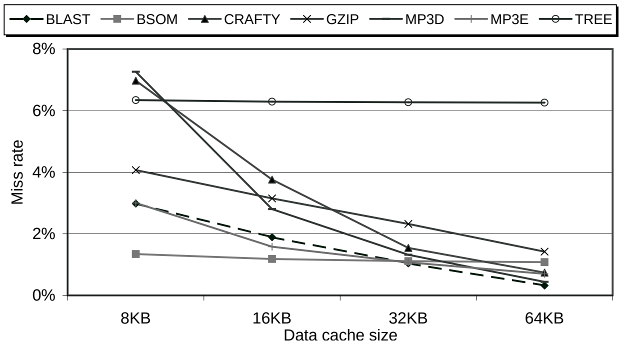

**Figure 2:** **Application miss rates for different data cache sizes.**

# 5. EVALUATION

In this section, we evaluate our SSC and PSAC organizations, both individually and combined. As indicated before, we use as a metric the *energy-delay product*. In this metric, *delay* is the execution time of the application, and *energy* is the energy consumed in the data memory hierarchy. The latter includes the TLB, the L1 and L2 caches, and the main memory. Different applications spend a different percentage of their energy in the data memory hierarchy. In our application suite, this percentage ranges from 15% to 55% and is on average 30%. In all the figures, the results are normalized to those of the baseline architecture described in Table 1.

## 5.1 SSC Analysis

The benefit of using a small cache is two-fold: faster access and lower energy consumption. Stack references have far more spatial locality than the rest of the data references. Therefore, a small structure fulfills the role of stack cache nicely. In our simulations, we find that the miss rate barely changes for stack cache sizes larger than 2 KB. Moreover, XCACTI shows that beyond 2 KB, the cache cannot be accessed in one processor cycle for 1 GHz. Thus, we only show data for stack cache sizes between 256 B and 2 KB.

In Figure 3, we use our baseline 32 KB L1 cache and we add a small stack cache for stack references: either an SSC or a plain write back cache (WB). We use different stack cache sizes as shown in the legend. The bars are ordered from left to right in the order of decreasing energy-delay products. The bars correspond to the average of the applications.

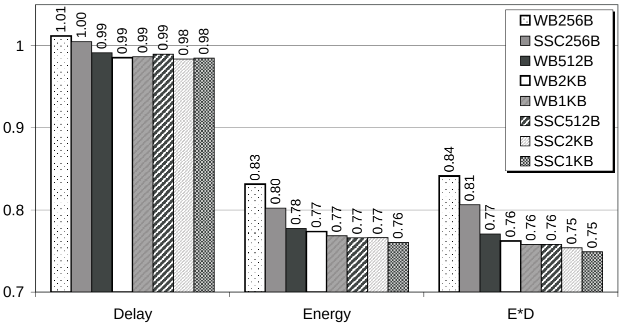

**Figure 3:** **Delay, energy, and energy-delay product of the baseline system with a stack cache. WB stands for plain write back stack cache.**

Figure 3 shows that adding a small structure to cache stack references is very energy efficient. Although the performance does not improve much, the energy savings range from 17% to 24%. The result is a much reduced energy-delay product.

Our proposed SSC system delivers an energy-delay product that is always lower than the one delivered by the WB system. The difference becomes smaller as the SSC increases in size. The reason is that larger caches have lower miss rates and, therefore, fewer write backs and line fills. Overall, if we try to balance performance, energy, and area cost, we recommend using a 512 B SSC.

Figure 3 only shows the average of all the applications. In applications with frequent stack activity like MP3E, the benefits of SSC over plain WB are significant. Figure 4 repeats Figure 3 for the MP3E application only. We can see that the SSC system delivers an energy-delay product that is substantially lower than the one delivered by the WB system. In no application of our suite does the system with SSC have a higher energy-delay product than the system with WB.

Finally, software optimizations specific to stack caches are an interesting issue not applicable to traditional architectures. With a stack cache (SSC or WB), stack accesses are faster and spend less energy. We can exploit this difference by making sure that variables

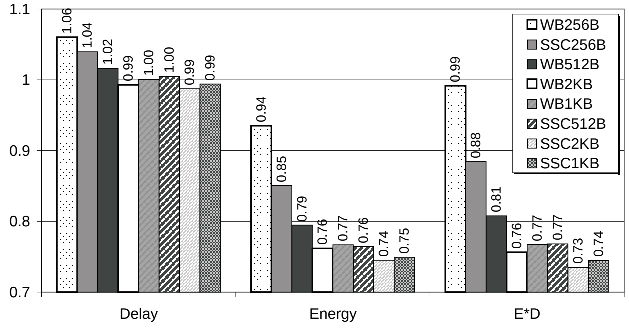

**Figure 4:** **Delay, energy, and energy-delay product of a system with a stack cache for the MP3E application.**

that are used as temporaries inside a subroutine as declared as automatic inside the subroutine. Of course, over-utilizing the stack can be bad. Large structures allocated in the stack can increase the miss rate, negating the benefit. In our application suite, the two MP3 applications have structures that can be bigger than the stack cache itself. We changed these structures into static variables so that their accesses do not go to the stack cache. With a 512 B SSC or WB cache, these changes improve the energy-delay product by an average of 19% and 14% in MP3D and MP3E, respectively. These changes have no impact on the baseline architecture.

## 5.2 PSAC Analysis

We simulate all the PSAC schemes and algorithms discussed in Section 3. For clarity, Figure 5 only shows the most representative schemes. It excludes the schemes with adaptive replacement and the 2-way phased cache.

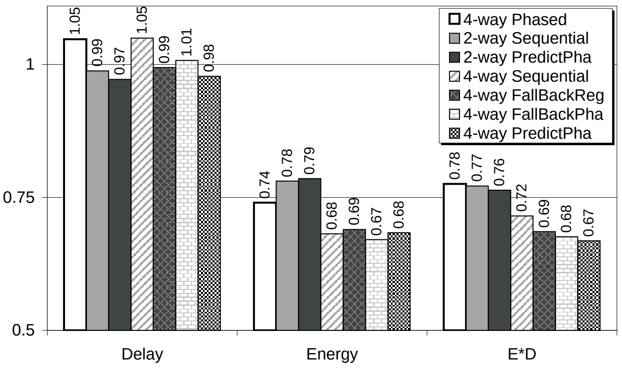

**Figure 5:** **Delay, energy, and energy-delay product of systems with different PSAC organizations.**

From Figure 5 we can make several observations. First, 4-way PSAC structures have a lower energy-delay product than 2-way ones because smaller structures are more energy efficient. Second, the phased cache suffers from low performance. Thus, despite its low energy consumption, it has a poor energy-delay product. Third, *Sequential* is not as good as other schemes; for 4-way set-associative systems, its energy-delay product is 7% higher than *PredictPha*'s. Finally, *PredictPha* has the best energy-delay product. The difference in energy-delay product between *PredictPha* and the other optimized schemes (*FallBackPha* and *FallBackReg*) is small.

Not shown in the figure is the impact of adaptivity in the line replacement algorithm as described in Section 3. Although adaptive schemes always have a lower miss rate than their non-adaptive counterparts, they often have a higher misprediction rate. The resulting energy-delay product changes little. Consequently, we do not consider adaptivity further.

## 5.3 Combination

Figure 6 shows the effect of using an SSC and a PSAC simultaneously. For the PSAC, we use *PredictPha* with an 8 KB bank in each way. We vary the number of ways from 3 to 4. The last five bars in each group add an SSC of size ranging from 256 B to 2 KB. As usual, the bars are relative to the baseline system of Table 1.

If we focus on the energy-delay product bars, we see that, even if we have a PSAC as our L1, we still want to add an SSC. Comparing Figure 6 to Figure 3, we see that the energy-delay product reductions delivered by an SSC and by a PSAC are not fully additive. However, a system combining a PSAC and a 512 B SSC delivers the best balance between performance, energy consumption, and area cost.

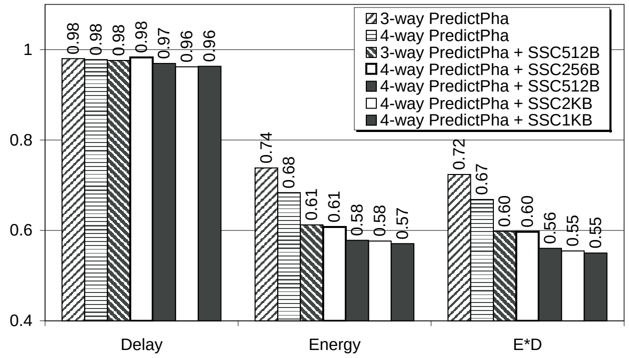

**Figure 6:** **Delay, energy, and energy-delay product of systems with combinations of SSC and PSAC structures. For the PSAC, we use *PredictPha* with different numbers of ways. The last five bars in each group add an SSC.**

For designs in which chip real estate is an issue, Figure 6 also shows an interesting tradeoff. The first three bars in each group correspond to a 3-way 24 KB PSAC, a 4-way 32 KB PSAC, and a combination of a 3-way 24 KB PSAC and a 512 B SSC, respectively. We can see that reducing the PSAC size from 32 KB to 24 KB (second and first bars) increases the energy-delay product by 7%. However, adding a 512 B SSC to the 24 KB PSAC (third bar), not only makes up for the loss, but actually improves the energy-delay product by 10% over the 32 KB PSAC. Consequently, a smaller PSAC combined with an SSC is a relatively good solution if real state is tight.

## 5.4 Energy Breakdown

Figure 7 shows the breakdown of the total energy consumption in the data memory hierarchy across different applications. The breakdowns are shown for different L1 organizations: the baseline, a 4-way 32 KB *PredictPha* PSAC, the baseline plus a 512 B SSC, and a 4-way 32 KB *PredictPha* PSAC plus a 512 B SSC. The energy is broken down into main memory, L2, L1 outside the SSC, and SSC. As we can see, our PSAC and SSC designs target what is usually the largest energy consumption chunk in the data memory hierarchy. They reduce it effectively.

# 6. RELATED WORK

Proposals for special hardware to handle the top of the stack, or even a hardware stack, are not uncommon. For example, Ditzel and

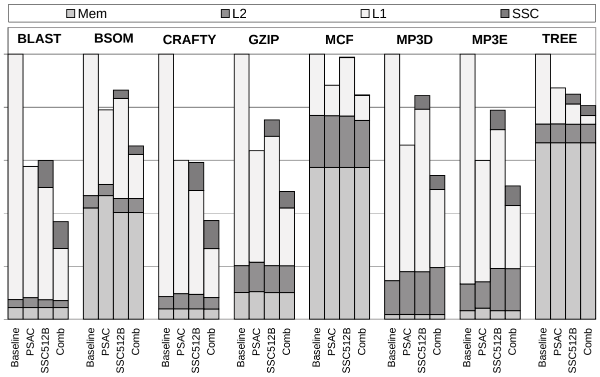

**Figure 7: Breakdown of the total energy consumed in the data memory hierarchy. *PSAC* is a 4-way 32 KB *PredictPha*, while *Comb* is *PSAC* plus a 512 B SSC.**

McLellan [8] proposed using a large register file to simulate a stack. However, many of such designs are performance driven. Our approach, instead, is to exploit stack properties for energy efficiency, which leads to different designs.

Perhaps the design closest to ours is the circular buffer for the stack proposed by Lee *et al.* [16]. The buffer exploits the stack reference properties discussed in Section 2 to save line fills and write backs. Its goal is to improve performance, rather than to do it in an energy-efficient manner, as our SSC. As a result, the buffer is very large (8 KB), and includes two status bits per word. The large size is needed to avoid conflicts. Our SSC, instead, is very small (512 B), is organized as a plain cache, and adds two pointers that perform simple, low-energy operations.

Cho *et al.* proposed a plain stack cache for high-performance processors [6, 7]. The stack cache is accessed by an additional load/store unit to simplify coherence between multiple load/store units. They did not evaluate their design for energy-efficiency. Later, Lee and Tyson proved that such a design is good for low energy-delay product in embedded processors [17]. We show that an SSC has a lower energy-delay product than a plain stack cache. Furthermore, our analysis suggests that for high-frequency processors, a small SSC (512 B) is the most desirable design.

Several PSAC designs have been proposed which can be implemented as a set of smaller, independently-probed caches. They differ in the algorithms used for way selection and line replacement. Many of the proposals are for 2-way PSAC systems, including the Hash Rehash [1], Column Associative [2], Predictive Sequential Associative [4], and MIPS R10000 [19] caches.

The PSAC schemes with higher associativity tend to use MRU information in the target set to select the order to probe the ways. For example, Kessler *et al.* sequentially probe the ways from MRU to LRU [13]. The Way-Predicting Set-Associative cache probes the MRU way first, and then the remaining ways in parallel [11]. A similar approach is used by Chang *et al.* [5]. Using MRU information necessarily introduces a serialization step right after we know the address to load or store.

All the mentioned proposals except [11] are evaluated for performance. The system in [11] is evaluated for energy-delay product using a simplistic model. In other related work, Kim *et al.* [14] calculate the energy consumed by several 2-way PSAC systems, which they call *Multiple-Access Caches*.

Our work in this paper differs in several ways. First, we focus on PSAC systems with associativity higher than 2. Secondly, we propose two new PSAC systems based on the phased cache. Finally, we evaluate many schemes under the energy-delay product metric using a very detailed model based on an extension to CACTI.

## 7. CONCLUSIONS

This paper presented a design for a high-performance, energy-efficient L1 data cache. The cache combined new designs of a stack cache and a PSAC. Overall, we recommend an L1 design with a small SSC and a 4-way PSAC. Relative to a conventional 2-way 32 KB data cache, a 512 B SSC with a 4-way 32 KB *PredictPha* PSAC reduced the energy-delay product of our applications by an average of 44%. Furthermore, in an area-constrained design, we recommend to use a smaller PSAC and still keep the small SSC. For example, we can use a 3-way 24 KB PSAC with a 512 B SSC. Finally, we found that changes in the line replacement algorithm to ignore the prediction scheme and not replace MRU lines have a negligible impact on the energy-delay product.

## 8. REFERENCES

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] A. Agarwal and S. Pudar. Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches. In *Int'l Symposium on Computer Architecture*, pages 179–190, May 1993.

[3] M. Bekerman, A. Yoaz, F. Gabbay, S. Jourdan, M. Kalaev, and R. Ronen. Early Load Address Resolution Via Register Tracking. In *Int'l Symposium on Computer Architecture*, pages 306–315, June 2000.

[4] B. Calder, D. Grunwald, and J. Emer. Predictive Sequential Associative Cache. In *Int'l Symposium on High Performance Computer Architecture*, pages 244–253, February 1996.

[5] J. H. Chang, H. Chao, and K. So. Cache Design of a Sub-Micron CMOS System/370. In *Int'l Symposium on Computer Architecture*, pages 208–213, June 1987.

[6] S. Cho, P. Yew, and G. Lee. Access Region Locality for High-Bandwidth Processor Memory System Design. In *Int'l Symposium on Microarchitecture*, pages 136–146, November 1999.

[7] S. Cho, P. Yew, and G. Lee. Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor. In *Int'l Symposium on Computer Architecture*, pages 100–110, May 1999.

[8] D. Ditzel and H. McLellan. Register Allocation for Free: The C Machine Stack Cache. In *Architectural Support for Programming Languages and Operating Systems*, pages 48–56, March 1982.

[9] P. Gelsinger. Microprocessors for the New Millennium - Challenges, Opportunities and New Frontiers. In *Int'l Solid-State Circuits Conference*, February 2001.

[10] A. Hasegawa et al. SH3: High Code Density, Low Power. *IEEE Micro*, pages 11–19, Dec 1995.

[11] K. Inoue, T. Ishihara, and K. Murakami. Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption. In *Int'l Symposium on Low-Power Electronics and Design*, pages 273–275, 1999.

[12] Intel Corporation. *Mobile Power Guidelines 2000, Rev 1.0*, 1998.

[13] R. Kessler, R. Jooss, A. Lebeck, and M. Hill. Inexpensive Implementation of Set-Associativity. In *Int'l Symposium on Computer Architecture*, pages 131–140, June 1989.

[14] H. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Multiple Access Caches: Energy Implications. In *IEEE CS Annual Workshop on VLSI*, pages 53–58, April 2000.

[15] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, October 1998.

[16] H. Lee, M. Smelyanskiy, C. Newburn, and G. Tyson. Stack Value File: Custom Microarchitecture for the Stack. In *Int'l Symposium on High-Performance Computer Architecture*, pages 5–14, January 2001.

[17] H. Lee and G. Tyson. Region-Based Caching: An Energy-Delay Efficient Memory Architecture for Embedded Processors. In *CASES 2000*, November 2000.

[18] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal on Solid-State Circuits*, 31(5):677–688, May 1996.

[19] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–41, April 1996.