Energy Characterization of Hardware-Based Data Prefetching

Yao Guo[1], Saurabh Chheda[2], Israel Koren[1], C. Mani Krishna[1], and Csaba Andras Moritz[1]

[1]*Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003*
[2]*BlueRISC Inc., Hadley, MA 01035*

Abstract

This paper evaluates several hardware-based data prefetching techniques from an energy perspective, and explores their energy/performance tradeoffs. We present detailed simulation results and make performance and energy comparisons between different configurations. Power characterization is provided based on HSpice circuit-level simulation of state-of-the-art low-power cache designs implemented in deep-submicron process technology. This is combined with architecture-level simulation of switching activities in the memory system. The results show that while aggressive prefetching techniques often help to improve performance, they increase energy consumption in most of the cases. In designs implemented in deep-submicron 100-nm BPTM process technology, cache leakage becomes one of the dominant factors of the energy consumption. We have, however, found that if leakage is optimized with recently-proposed circuit-level techniques, most of the energy degradation is due to prefetch-hardware related costs and unnecessary L1 data cache lookups related to prefetches that hit in the L1 cache. This overhead on the memory system can be as much as 20%.

1. Introduction

Prefetching has been proposed as a successful technique to hide memory latencies. Although considerable research has been focused on improving the performance of prefetching mechanisms, the impact of such prefetching techniques on processor energy efficiency remains unclear. In this paper, we present energy characterization of several hardware-based data prefetching techniques. Our purpose is to analyze the current and potential energy-related issues of data prefetching mechanisms and to motivate new development of energy-aware data prefetching techniques.

Both hardware [15], [3], [13], [14], [6] and software [11], [8], [9] techniques have been proposed for prefetching in recent years. Software prefetching is implemented by inserting explicit prefetch instructions into the executable code. Although there are no hardware requirements for software prefetching (prefetching instructions are supported by most contemporary microprocessors), the compiler process of inserting and scheduling prefetches is complicated. Hardware-based approaches are simpler since they do not require modification to executables. Although hardware prefetching requires extra prefetch hardware in a processor, such additional hardware requirements are typically small.

This paper evaluates several state-of-the-art hardware-based data prefetching techniques from an energy perspective, and explores their energy/performance tradeoffs. The prefetching techniques studied include:

- Two sequential prefetching (One Block Lookahead, OBL) approaches - a simple *sequential prefetching* (prefetch-on-miss) and *tagged sequential prefetching*;
- *Stride prefetching*, focusing on array-like structures, catches constant strides in memory accesses and prefetching using the stride information;
- *Dependence-based prefetching*, which is designed to prefetch on pointer-intensive programs containing linked data structures where no constant strides can be found.
- A new combined stride and dependence-based approach.

In this paper we consider 100-nm technologies: this is representative of such next generation process technologies. We present detailed simulation results on each prefetching technique and show performance and energy comparisons. We modify the SimpleScalar [4] simulation tool to implement the different prefetching techniques and collect statistics on performance as well as switching activity in memory systems. To estimate power consumption in the memory systems, we use state-of-the-art low-power cache circuits and simulate them using HSpice.

As expected, the results show that while aggressive prefetching techniques often help to improve performance, in most of the applications, they increase energy consumption by as much as 20%. In many systems [7], [10], this constitutes more than 10% increase in chip-wide energy consumption. In designs implemented in deep-submicron 100-nm BPTM process technology, cache leakage dominates the energy consumption. We have, however, found that if cache leakage is optimized with recently-proposed circuit-level techniques, most of the still remaining energy degradation is due to prefetch hardware related cost and unnecessary L1 data cache lookups related to prefetches that hit in the L1 cache. When the energy cost of off-chip accesses is increased to more pessimistic levels (e.g., due to very large load capacitances driven during off-chip accesses), the other energy effects become less visible.

The rest of this paper is structured as follows. Section 2 presents a brief introduction of the prefetching techniques we study in this paper. The power estimation of memory system aspects and experimental framework is presented in Section 3. Section 4 gives a detailed analysis of the experimental results. We conclude with Section 5.

2. Hardware-based Data Prefetching

2.1. Sequential Prefetching

Sequential prefetching schemes are based on the *One Block Lookahead* (OBL) approach, which initiates a prefetch for block *b*+1 when block *b* is accessed. OBL implementations differ based on what type of access to block *b* initiates the prefetch of *b*+1. We will evaluate two of the sequential approaches discussed by Smith [16] - *prefetch-on-miss sequential* and *tagged prefetching*.

The prefetch-on-miss sequential algorithm simply initiates a prefetch for block *b*+1 whenever an access for block *b* results in a cache miss. If *b*+1 is already cached, no memory access is initiated. The tagged prefetching algorithm associates a bit with every cache line. This bit is used to detect when a line is demand-fetched or a prefetched block is referenced for the first time. In both cases, the next sequential block is prefetched.

OBL prefetching schemes are not as efficient as more recent schemes but they require relatively simple hardware. An OBL scheme was implemented in the HP PA7200 [5] which uses a modified version of tagged prefetching scheme and shows significant performance improvement for some benchmarks.

2.2. Stride Prefetching

Stride prefetching [3] monitors memory access patterns in the processor to detect constant-stride array references originating from loop structures. This is normally accomplished by comparing successive addresses used by memory instructions.

Since stride prefetching requires the previous address used by a memory instruction to be stored along with the last detected stride, a hardware table (called the *Reference Prediction Table*, RPT) is added to hold the information for only the most recently used load instructions. Each RPT entry contains the address of the load instruction, the address of this instruction as accessed previously, a stride value for those entries that have established a stride, and a state field used to control the actual prefetching.

Stride prefetching is more selective than sequential prefetching since prefetch commands are issued only when a matching stride is detected. It is also more effective when array structures are accessed through loops. However, stride prefetching uses a hardware table which normally contains 64 entries; each entry contains around 64 bits. This hardware table is accessed whenever a load instruction is detected.

2.3. Pointer Prefetching

Stride prefetching has been shown to be effective for array-intensive scientific programs. However, for general-purpose programs which are pointer-intensive, or contain a large number of dynamic data structures, no constant strides can be easily found that can be used for effective prefetching.

One scheme for hardware-based prefetching on pointer structures, called *dependence-based prefetching*, is proposed by Roth et al. [13]. Like stride prefetching, this scheme uses hardware tables to record the most recently executed load instruction. The difference is that this table is used to detect dependencies between load instructions rather than establishing reference patterns for single instructions.

Dependence-based prefetching requires the help of two hardware tables. The Correlation Table (CT) is the component responsible for storing dependence information. Each correlation represents a dependence between a load instruction that produces an address (producer) and a subsequent load that uses that address (consumer). The Potential Producer Window (PPW) records the most recent loaded values and the corresponding instructions. When a load commits, its base address value is checked against the entries in the PPW, with a correlation created on a match. This correlation is added to the CT.

PPW and CT typically consist of 64-128 entries containing addresses and program counters; each entry may contain 64 or more bits. The hardware cost is around twice that for stride prefetching. This scheme improves performance on many of the Olden [12] pointer-intensive benchmarks.

2.4. Combined Stride and Pointer Prefetching

One contribution of this paper is a combined stride and pointer prefetching techniques. Our objective is to evaluate a technique that is beneficial for applications containing both array and pointer based accesses.

We will show that the combined technique performs consistently better than the individual techniques on two benchmark suites with different characteristics. However, the hardware cost of this approach is higher since we need the hardware tables from both stride and dependence-based prefetching.

Table 1: *Baseline parameters*

Processor speed	1GHz
Issue	4-way, out-of-order
L1 D-cache	32KB, CAM-tag, 32-way, 32bytes cache line
L1 I-cache	32KB, 2-way, 32bytes cache line
L1 cache latency	1 cycle
L2 cache	unified, 256KB, 4-way, 64bytes cache line
L2 cache latency	12 cycle
Memory latency	100 cycles latency + 10 cycles/word

Table 2: *Prefetching hardware parameters*

Prefetching Scheme	Hardware required
Sequential	none
Tagged	1 bit per cache line
Stride	A 64-entry RPT
Dependence	A 64-entry PPW and a 64-entry CT
Combined	All three tables above

3. Experimental Assumptions

3.1. Experimental Framework

We implement the hardware-based data prefetching techniques by modifying the SimpleScalar [4] simulator. The parameters we used for simulation are listed in Table 1.

The hardware requirements for the prefetching schemes are shown in Table 2. For the three schemes that require hardware-based history tables, each entry of the hardware tables contains two 32 bits address value and some extra bits. For simplicity, we assume each entry has roughly the same size of 64 bits.

We randomly select a total of ten benchmark applications, five from SPEC2000 and five from Olden. The SPEC2000 benchmarks [1] use mostly array-based data structures, while the Olden benchmark suite [12] contains pointer-intensive programs that make substantial use of linked data structures. For SPEC2000 benchmarks, we fast forward the first one billion instructions and then simulate the next 100 million instructions. The Olden benchmarks are simulated to completion since they are relatively short.

3.2. Energy Evaluation

The memory system, including caches, consumes a significant fraction of total processor power. For example, the caches and translation look-aside buffers (TLB) together consume 23% of the total power in the Alpha 21264 [7], and the caches alone use 42% of the power in the StrongARM 110 [10]. In this paper, we will focus on the memory system power consumption. The L1 and L2 caches, off-chip memory accesses and prefetching hardware tables could amount to about half of the total processor energy consumption.

Prefetching schemes affect the energy efficiency of a processor in a number of ways. For example, unnecessary cache lookups and bus accesses due to redundant prefetching can severely increase energy consumption in the memory system. The most important energy consumption issues introduced by hardware prefetching include:

1) *Energy cost of prefetch hardware*. Most of the hardware techniques require extra hardware such as address history tables to record the recent memory access patterns that are used to make prefetching decisions. The hardware tables, although typically much smaller compared to caches, are significant sources of

energy consumption since they are normally accessed whenever a memory access (normally load) occurs.

2) *Extra tag lookups for the L1 Cache*. Whenever a prefetch command is initiated, the first thing the prefetch engine does is to check whether the data to be prefetched is in the L1 Cache. Although most of the time the prefetch attempts can be resolved in the L1 Cache, tag lookups still cost significant power.
3) *Extra memory accesses to L2 Caches*. This is where most of the actual prefetching happens, i.e., bring the data to the L1-Cache from L2 before it is accessed.
4) *Extra off-chip memory accesses*. Although most prefetch commands resolve in the L2 Caches, even the most conservative prefetching schemes issue unnecessary off-chip memory prefetches, which result in a slight increase in the traffic to off-chip main memory. As we will show later, the increase is normally less than 1%.

To accurately estimate power and energy consumption in L1 and L2 caches, we perform circuit-level simulations using HSpice. We base our design on a recently proposed low-power circuit [17] that we implemented in 100-nm BPTM (Berkeley Predictive Technology Model) technology. Our L1 cache includes the following low-power features: low-swing bitlines, local word-line, CAM-based tags, separate search lines, and a banked architecture. The L2 cache we evaluate is based on a banked RAM-tag design.

As we expect that implementations in 100-nm technology would have significant leakage, we apply a recently proposed circuit-level leakage reduction technique called asymmetric SRAM cells [2]. This is necessary because otherwise our conclusions would be skewed due to very high leakage power. The *speed enhanced cell* in [2] has been shown to reduce L1 data cache leakage by 3.8X for SPEC2000 benchmarks with no impact on performance. For L2 caches, we use the *leakage enhanced cell* which increases the read time for 5%, but it can reduce leakage power by at least 6X and by about 40X in the preferred state. In our evaluation, we assume speed-enhanced cells for L1 and leakage enhanced cells for L2 data caches, by applying the different asymmetric cell techniques respectively.

The power consumption for our L1 and L2 caches are shown in Table 3.

Table 3: *Cache configuration and power consumption*

Parameter	L1	L2
size	32KB	256KB
tag array	CAM-based	RAM-based
associativity	32-way	4-way
bank size	2KB	4KB
# of banks	16	64
cache line	32B	64B
Power (mW)		
P_tag	6.5	6.27
P_read	9.5	100.52
P_write	10.3	118.62
P_leakage	3.1	23.0
P_reduced_leakage	0.82	1.53

In a CAM-based cache (such as the L1 assumed in this paper) a CAM access and a data-array access are performed for each cache access. In a Ram-tag cache (such as the L2 assumed in this paper) multiple tag accesses and data-array accesses need to be completed, depending on associativity, for every access.

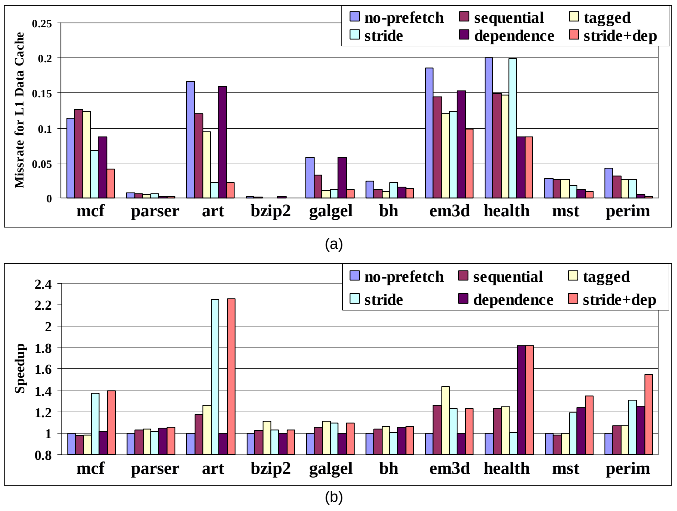


Figure 1: *Performance speedup for different prefetching schemes: (a) DL1 miss rate reduction; (b) IPC speedup.*

If an L1 miss occurs, energy is consumed not only in L1 tag-lookups, but also when writing the requested data back to L1. L2 accesses are similar, except that an L2 miss goes to off-chip main memory. Such an off-chip access consumes a significant amount of processor power. Rather than picking a single design-point, we choose a range of energy costs ranging from optimistic to pessimistic. We express the L2 miss energy as a function of L1 hit energy. We assume that an L2 cache miss consumes 32X to 512X single-word read energy of our L1 cache. A similar assumption has been made in [17]. The actual power consumed depends on how many bits are in transition and on the actual implementation/packaging choices.

Each prefetching history table is implemented as a 64×64 fully-associated CAM-array. The power consumption for each lookup or update to the table is roughly 7.3mW based on HSpice simulation. The leakage energy of these hardware tables will not be accounted because they are very small (512B) compared to L1 and L2 caches.

4. Results and Analysis

4.1. Performance Speedup

Performance speedup is the original, and still the primary goal, of prefetching. Figure 1 shows the performance results of different prefetching schemes. The first five benchmarks are array-intensive SPEC2000 benchmarks, and the last five are pointer-intensive Olden benchmarks. Figure 1(a) shows the reduction of DL1 miss-rate, and Figure 1(b) shows actual speedup based on simulated execution time.

As expected, the dependence-based approach does not work well on the five SPEC2000 benchmarks since pointers and linked data structures are not used frequently. But it still gets marginal speedup on three benchmarks (*parser* is the best with almost 5%).

Tagged prefetching (10%) does slightly better on SPEC2000 benchmarks than the simplest sequential approach, which achieves an average speedup of 5%. Stride prefetching yields up to 124% speedup (for *art*), averaging just over 25%. Combined prefetching is the best, but gives on the average only about 1.5% speedup compared to the stride approach. The comparison between miss rate reduction in Figure 1(a) and speedup in Figure 1(b) matches our intuition that fewer cache misses means greater speedup.

As for the five Olden pointer-intensive benchmarks in Figure 1, the dependence-based approach eliminates about half of all the L1 cache

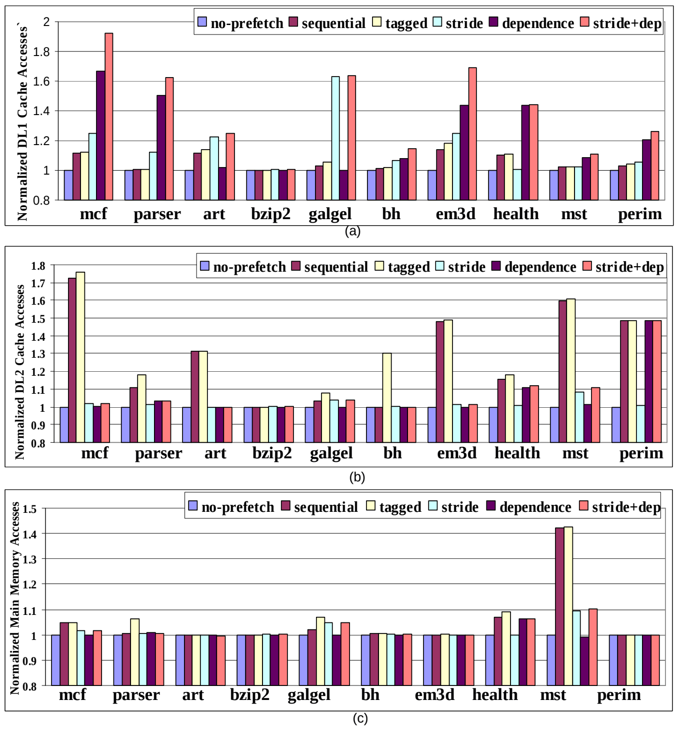


Figure 2: *Memory traffic increase for different prefetching schemes. (a) Number of accesses to L1 data cache, including extra cache-tag lookups to L1; (b) Number of accesses to L2 data cache; (c) Number of accesses to main memory.*

misses and achieves an average speedup of 27%. Stride prefetching (14%) does surprisingly well on this set of benchmarks and implies that even pointer-intensive programs contain significant constant-stride memory access sequences. The combined approach achieves an average of 40% performance speedup on the five Olden benchmarks.

In summary, for array-intensive programs, stride prefetching does reasonably well and dependence-based pointer prefetching is not very effective. However, for pointer-intensive programs, both stride and dependence-based approaches do sufficiently well. The combined approach achieves the best performance speedup due to prefetching. In general, the combined technique is useful for general purpose programs which contain both array and pointer structures.

4.2. Memory Traffic Increase

Memory traffic is increased because not all the data we prefetch from the next level are useful (i.e., not all they are actually used by a later access before they are replaced from the cache). In most cases, some useless data is prefetched into the higher level of the memory hierarchy; these are a major source of power/energy consumption added by the prefetching schemes. Apart from memory traffic increases, power is also consumed when we attempt to prefetch the data that already exists in the higher level cache. In this case, the attempt to locate the data (e.g., cache-tag lookups) consumes power.

Figure 2 shows the number of accesses going to different levels in the memory hierarchy. The numbers are normalized to the baseline with no prefetching. On average, the number of accesses to L1 D-cache increases almost 40% with the most aggressive prefetching scheme. However, the accesses to L2 only increase by 8% for the same scheme, showing that most of the L1 cache accesses are only cache-tag lookups trying to prefetch data already present in L1.

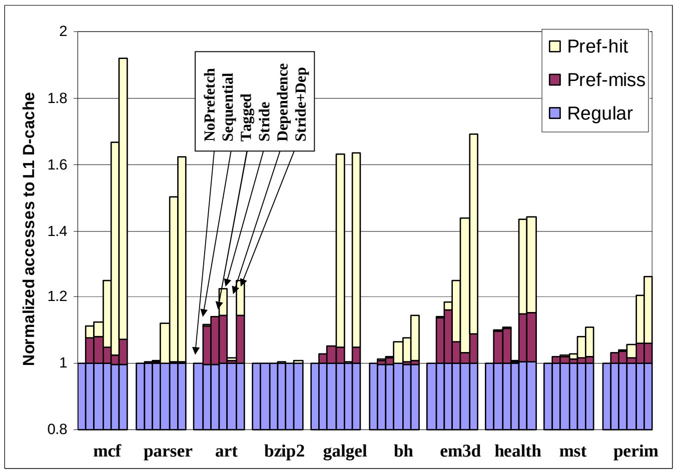


Figure 3: *Breakdown of L1 Accesses, all numbers normalized to L1 cache accesses of baseline with no prefetching.*

Sequential prefetching techniques (both prefetch-on-miss and tagged schemes) show completely different behavior as they increase the L1 access for only about 7% while resulting in a more than 30% average increase on L2->L1 traffic. The explanation for this is that sequential prefetching always tries to prefetch the next cache line which has a much greater chance to miss in L1. Main memory accesses are largely unaffected in the last three techniques, and only increase by 5-7% for sequential prefetching.

As L1 accesses increase significantly for the three most effective techniques, we break down the number of L1 accesses into three parts: regular L1 accesses, L1 prefetch misses and L1 prefetch hits, shown in Figure 3. The L1 prefetch misses are those prefetching requests that go to L2 and actually bring cache lines from L2 to L1. While the L1 prefetch hits stand for those prefetching requests that hit in L1 and no real prefetching occurs.

From Figure 3, we can see that L1 prefetching hits account for most of the increases in L1 accesses. On average, 70-80% of all the increases come from extra L1 prefetching hits, which may result in significant energy overhead, while being almost useless for performance speedup. The extra L1 accesses will obviously translate into unnecessary energy consumption.

4.3. Energy Consumption

We use the power numbers shown in Section 3 to calculate the energy consumption.

4.3.1. Cache energy consumption

Figure 4 shows the dynamic energy consumption for L1 and L2 caches and prefetching tables. For most of the benchmarks, the L1 dynamic energy (excluding prefetching overhead) is not affected significantly. The L2 dynamic energy is increased in proportion to the L2 memory traffic increase shown in Figure 2(b). Prefetching related energy overhead on L1 cache is quite small for sequential prefetching, but more significant for the other three prefetching approaches. This part of the energy overhead is proportional to the prefetch-related L1 access increase shown in Figure 3.

Energy consumption for the hardware tables are very significant for all the three prefetching techniques using hardware tables. On average, the hardware tables consume almost the same amount of energy as regular L1 caches accesses for the combined prefetching. Typically this portion of energy accounts for 60-70% of all the

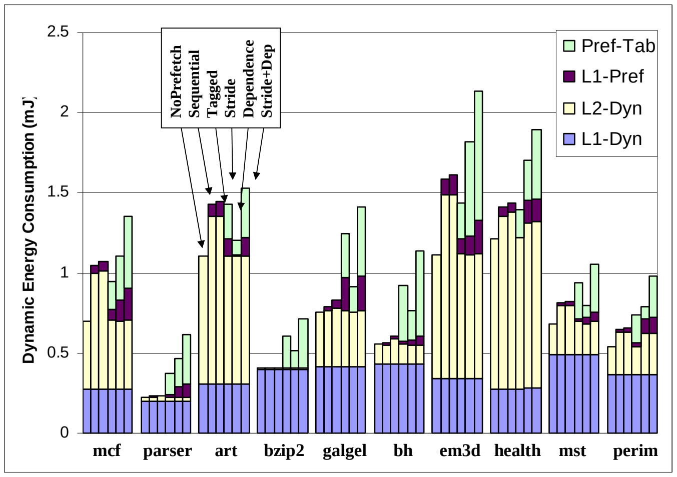


Figure 4: *Total cache energy consumption without considering leakage energy.*

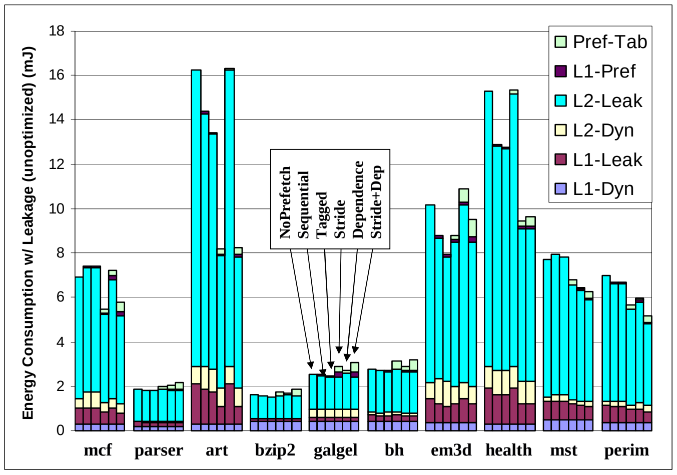


Figure 5: *Total cache energy consumption with unoptimized leakage energy accounted.*

dynamic energy overhead that result from combined prefetching. The reason is that prefetch tables are frequently looked up and are also highly associative. Their energy consumption is similar to a tag lookup in a highly optimized banked low power cache.

For the most aggressive combined prefetching approach, the prefetching energy overhead almost doubles the total dynamic energy (baseline with no prefetching) for some applications (such as *mcf* and *em3d*), and is 76% on the average. For the other prefetching techniques, there is a 25% increase for sequential prefetching, and about 38% for both stride and dependence schemes. This shows that while complicated prefetching algorithms can achieve greater speedups, they can significantly increase the overall energy consumption.

Figure 5 shows the total cache energy consumption with leakage energy also accounted. Leakage energy is proportional to program runtime and thus decreases linearly with speedup: higher speedup will reduce the leakage energy consumption.

In this figure, the total energy consumption for caches is dominated by L2 leakage because of the large size (256KB) of the L2 cache. As we can see, for most of the applications, the relative prefetching

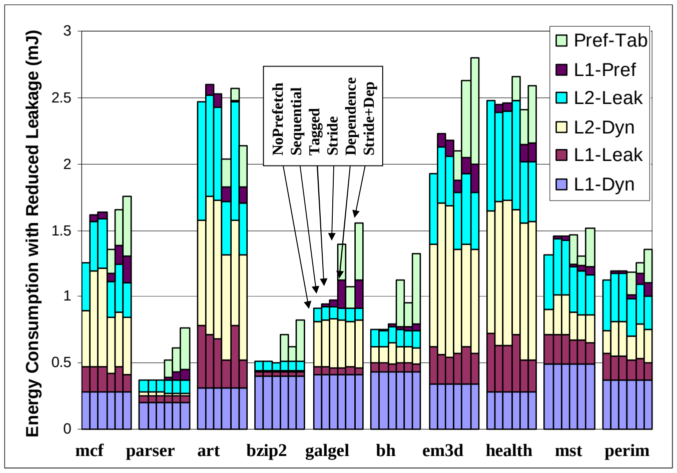


Figure 6: *Total cache energy consumption with leakage reduction techniques applied.*

overhead shown in Figure 4 has been significantly reduced after the leakage energy is taken into account.

With no leakage optimization, sequential prefetching saves on average about 10% of the total energy, stride prefetching about 17% and the combined approach results in almost 24% energy savings. The results show that prefetching schemes which have a better performance speedup also save energy when leakage energy increases to a certain level in deep sub-micron technologies.

However, leakage energy could be reduced significantly by techniques such as asymmetric SRAM cells [2]. Figure 6 shows the total cache energy after applying the above leakage reduction techniques. The dynamic hit energy dominates some of the benchmarks with higher IPC; however, the leakage energy still dominates in some programs, such as *art*, which have a higher L1 miss rate and thus a longer running time. Although both L1 and L2 cache access energy are significantly increased due to prefetching, the static (leakage) energy reduction due to performance speedup can compensate for at least some portion of the increase in dynamic energy consumption.

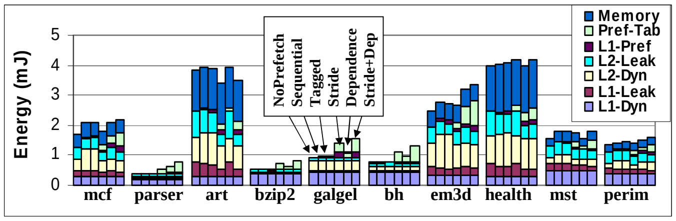
The results in Figure 6 show that on average, the prefetching schemes still cause relatively significant energy consumption overhead when leakage consumption is reduced to a reasonable level. The average increase of the combined approach is more than 26%, and about 11% increase for stride prefetching.

4.3.2. Energy cost for off-chip accesses

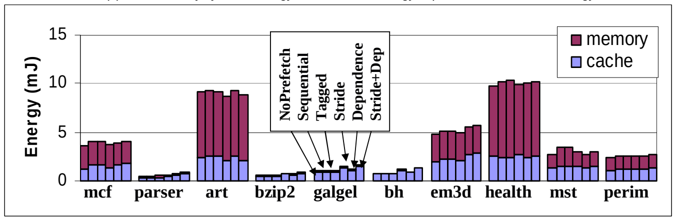
To estimate the energy consumption within the processor for driving off-chip memory accesses, we use similar assumptions as in [17]. We assume that an L2 cache miss consumes 32-512X single-word read energy of the L1 D-cache. Our results, including energy consumption for both caches and off-chip memory access related power, are shown in Figure 7.

Figure 7(a) shows the situation where L2 miss energy cost is 32X of L1 hit energy. The prefetching energy overhead is quite significant for many applications, averaging 7% for sequential prefetching, 8% for stride prefetching and more than 20% for the combined approach.

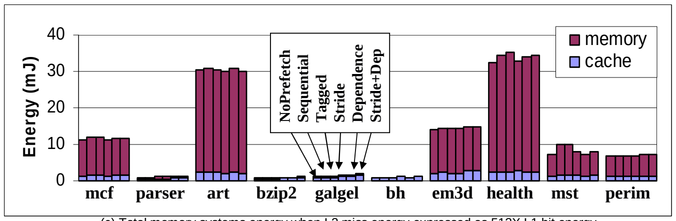
When the off-chip memory costs goes up to 128X, as shown in Figure 7(b) the prefetching overhead stays at 7% for sequential techniques, but drops to almost half for the last three schemes, averaging about 11% for combined prefetching. If the off-chip memory costs



(a) Total memory systems energy when L2 miss energy expressed as 32X L1 hit energy

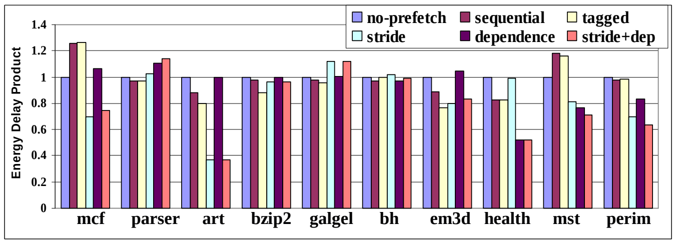


(b) Total memory systems energy when L2 miss energy expressed as 128X L1 hit energy

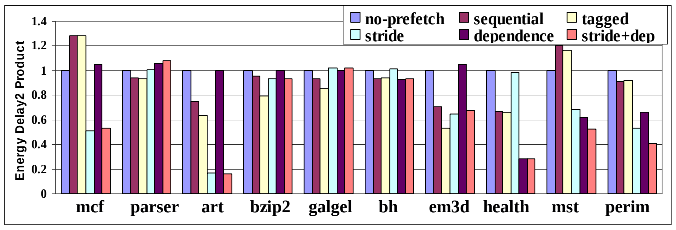


(c) Total memory systems energy when L2 miss energy expressed as 512X L1 hit energy

Figure 7: *Total energy consumption for memory systems with varying L2 miss energy cost.*



(a)



(b)

Figure 8: *Energy-delay product for different prefetching techniques. (1) Energy-delay product; (2) Energy-delay^2 product. In both figures, we assume that the leakage reduction techniques are applied and the off-chip memory energy cost is 32× L1 hit energy.*

were to increase to a pessimistic 512X as shown in Figure 7(c), the energy overhead of prefetching drops to less than 5%.

4.3.3. Energy-delay product

Finally, we show in Figure 8 the energy-delay and energy-delay^2 product normalized to the baseline (no prefetching) using the assumption that L2 miss energy is 32X L1 hit energy.

In most cases, we note that both energy-delay and energy-delay^2 products improve with effective prefetching techniques that achieve a large enough performance speedup. The energy-delay product improves by more than 20% for the combined prefetching, while the energy-delay^2 improves by almost 35%. This is important since by choosing a design point with lower voltage, this could be converted into energy efficiency. Nevertheless, we believe that more energy-focused prefetching algorithms and architectures should be developed to achieve energy efficiency even at unchanged voltage levels.

According to this figure, extra energy cost by complicated prefetching techniques are worthwhile for some applications such as the combined prefetching approach on *mcf* and *em3d*.

5. Conclusion

This paper studies the energy consumption issues related to data prefetching. In deep-submicron process technologies, memory system energy is dominated by the leakage component unless effective leakage reduction techniques are used. As feature sizes continue to decrease, leakage power will constitute an increasing fraction of the total energy consumption, favoring aggressive prefetching techniques. However, with successful leakage control, the problem shifts back to tuning the level of prefetch aggressiveness; otherwise the energy cost of prefetching will be dominated by the overhead from the prefetching hardware energy consumption and from extra L1 lookups when prefetching requests resolve at L1 Cache.

Clearly, for low-power processors, choosing the correct prefetching technique with good speedup and less energy overhead will be very important. New power-aware prefetching techniques are needed to reduce the energy overhead without decreasing the performance benefits of data prefetching.

References

[1] SPEC2000 benchmarks, http://www.spec.org.
[2] N. Azizi, A. Moshovos, and F. N. Najm. Low-leakage asymmetric-cell sram. In *ISLPED'02*, pages 48–51, 2002.
[3] J. L. Baer and T. F. Chen. An effictive on-chip preloading scheme to reduce data access penalty. In *Supercomputing 1991*, pages 179–186.
[4] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, Univ. of Wisc., Madison, June 1997.
[5] K. K. Chan, C. C. Hay, J. R. Keller, G. P. Kurpanek, F. X. Schumacher, and J. Zheng. Design of the HP PA 7200 CPU. *Hewlett-Packard Journal*, 47(1):25–33, Feb. 1996.
[6] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless content-directed data prefetching mechanism. In *ASPLOS-X*, pages 279–290, 2002.
[7] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *Design Automation Conference(DAC-98)*, pages 726–731, June 1998.
[8] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. Spaid: software prefetching in pointer- and call-intensive environments. In *Micro-28*, pages 231–236, 1995.
[9] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *ASPLOS-VII*, pages 222–233, 1996.
[10] J. Montanaro and et. al. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *Digital Technical Journal*, 9(1), 1997.
[11] T. Mowry. *Tolerating Latency Through Software Controlled Data Prefetching*. PhD thesis, Dept. of CS, Stanford Univ., Mar. 1994.
[12] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.
[13] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *ASPLOS-8*, pages 115–126, Nov. 1998.
[14] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *ISCA-26*, pages 111–121, 1999.
[15] A. J. Smith. Sequential program prefetching in memory bierarchies. *IEEE Computer*, 11(12):7–21, Dec. 1978.
[16] A. J. Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.
[17] M. Zhang and K. Asanovic. Highly-associative caches for low-power processors. In *Kool Chips Workshop, Micro-33*, Dec. 2000.