Energy-Efficient Hardware Data Prefetching

Yao Guo, *Member, IEEE*, Pritish Narayanan, *Student Member, IEEE*, Mahmoud Abdullah Bennaser, *Member, IEEE*, Saurabh Chheda, and Csaba Andras Moritz, *Member, IEEE*

***Abstract*—Extensive research has been done in prefetching techniques that hide memory latency in microprocessors leading to performance improvements. However, the energy aspect of prefetching is relatively unknown. While aggressive prefetching techniques often help to improve performance, they increase energy consumption by as much as 30% in the memory system. This paper provides a detailed evaluation on the energy impact of hardware data prefetching and then presents a set of new energy-aware techniques to overcome prefetching energy overhead of such schemes. These include compiler-assisted and hardware-based energy-aware techniques and a new power-aware prefetch engine that can reduce hardware prefetching related energy consumption by 7–11×. Combined with the effect of leakage energy reduction due to performance improvement, the total energy consumption for the memory system after the application of these techniques can be up to 12% less than the baseline with no prefetching.**

***Index Terms*—Compiler analysis, data prefetching, energy efficiency, prefetch filtering, prefetch hardware.**

I. Introduction

In RECENT years, energy and power efficiency have become key design objectives in microprocessors, in both embedded and general-purpose microprocessor domains. Although extensive research [1]–[9] has been focused on improving the performance of prefetching mechanisms, the impact of prefetching techniques on processor energy efficiency has not yet been fully investigated.

Both hardware [1]–[5] and software [6]–[8], [10], [11] techniques have been proposed for data prefetching. Software prefetching techniques normally need the help of compiler analyses inserting explicit prefetch instructions into the executables. Prefetch instructions are supported by most contemporary microprocessors [12]–[16].

Hardware prefetching techniques use additional circuitry for prefetching data based on access patterns. In general, hardware prefetching tends to yield better performance than software prefetching for most applications. In order to achieve both energy efficiency and good performance, we investigate the energy impact of hardware-based data prefetching techniques, exploring their energy/performance tradeoffs, and introduce new compiler and hardware techniques to mitigate their energy overhead.

Our results show that although aggressive hardware prefetching techniques improve performance significantly, in most applications they increase energy consumption by up to 30% compared to the case with no prefetching. In many systems [17], [18], this constitutes more than 15% increase in chip-wide energy consumption and would be likely unacceptable.

Most of the energy overhead due to hardware prefetching comes from prefetch-hardware-related energy cost and unnecessary L1 data cache lookups related to prefetches that hit in the L1 cache. Our experiments show that the proposed techniques together could significantly reduce the hardware prefetching related energy overhead leading to total energy consumption that is comparable to, or even less than, the corresponding number for no prefetching. This achieves the twin objectives of high performance and low energy.

This paper makes the following main contributions.

- We provide detailed simulation results on both performance and energy consumption of hardware data prefetching.
  - We first evaluate in detail five hardware-based data prefetching techniques. We modify the SimpleScalar [19] simulation tool to implement them.
  - We simulate the circuits in HSPICE and collect statistics on performance as well as switching activity and leakage.
- We propose and evaluate several techniques to reduce energy overhead of hardware data prefetching.
  - A *compiler-based selective filtering* approach which reduces the number of accesses to prefetch hardware.
  - A *compiler-assisted adaptive prefetching* mechanism, which utilizes compiler information to selectively apply different hardware prefetching schemes based on predicted memory access patterns.
  - A compiler-driven filtering technique using a *runtime stride counter* designed to reduce prefetching energy consumption on memory access patterns with very small strides.
  - A *hardware-based filtering* technique applied to further reduce the L1 cache related energy overhead due to prefetching.
  - A *Power-Aware pRefetch Engine* (PARE) with a new prefetching table and compiler based location set analysis that consumes 7–11× less power per access com-

Manuscript received July 16, 2008; revised April 02, 2009. First published October 23, 2009; current version published January 21, 2011.

Y. Guo is with the Key Laboratory of High-Confidence Software Technologies (Ministry of Education), School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China (e-mail: yaoguo@sei.pku.edu.cn).

P. Narayanan is with University of Massachusetts, Amherst, MA 01003 USA (e-mail: pnarayan@ecs.umass.edu).

M. Bennaser is with Kuwait University, Safat 13060, Kuwait (e-mail: bennaser@eng.kuniv.edu.kw).

S. Chheda is with BlueRISC Inc., San Jose, CA 95134 USA (e-mail: chheda@bluerisc.com).

C. A. Moritz is with University of Massachusetts, Amherst, MA 01003 USA and also with BlueRISC Inc., Amherst, MA 01002 USA (e-mail: andras@ecs.umass.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2009.2032916

pared to previous approaches. We show that PARE reduces energy consumption by as much as 40% in the data memory system (containing caches and prefetching hardware) with an average speedup degradation of only 5%.

Compiler-based techniques for reducing energy overhead of hardware data prefetching are implemented using the SUIF [20] compiler framework. Energy and performance impact of all techniques are evaluated using HSPICE.

The rest of this paper is organized as follows. Section II presents an introduction to the prefetching techniques we evaluated and used for comparison. The experimental framework is presented in Section III. Section IV gives a detailed analysis of the energy overheads due to prefetching. Energy-efficient prefetching solutions are presented in Sections V and VI. Section VII presents the results. The impact of modifying architectural framework (out-of-order versus in-order architectures) and cache organization is discussed in Section VIII. The related work is presented in Section IX, and we conclude with Section X.

II. Hardware-Based Data Prefetching Mechanisms

Hardware-based prefetching mechanisms need additional components for prefetching data based on access patterns. Prefetch tables are used to remember recent load instructions and relations between load instructions are set up. These relations are used to predict future (potential) load addresses from where data can be prefetched. Hardware-based prefetching techniques studied in this paper include sequential prefetching [1], stride prefetching [2], dependence-based prefetching [3] and a combined stride and dependence approach [21].

A. Sequential Prefetching

Sequential prefetching schemes are based on the *One Block Lookahead* (OBL) approach; a prefetch for block $b+1$ is initiated when block b is accessed. OBL implementations differ based on what type of access to block b initiates the prefetch of $b+1$. In this paper, we evaluate two sequential approaches discussed by Smith [22]—*prefetch-on-miss sequential* and *tagged prefetching*.

Prefetch-on-miss sequential algorithm 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.

B. Stride Prefetching

Stride prefetching [2] 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 the most recently used load instructions. Each RPT entry contains the PC address of the load instruction, the memory address previously accessed by the instruction, 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 an associative hardware table which is accessed whenever a load instruction is detected. This hardware table normally contains 64 entries; each entry contains around 64 bits.

C. 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 stride prefetching.

One scheme for hardware-based prefetching on pointer structures is *dependence-based prefetching* [3] that detects dependencies between load instructions rather than establishing reference patterns for single instructions.

Dependence-based prefetching uses two hardware tables. The correlation table (CT) is 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 pointer-intensive Olden [23] benchmarks.

D. Combined Stride and Pointer Prefetching

In order to evaluate a technique that is beneficial for applications containing both array and pointer bashed accesses, a combined technique that integrates stride prefetching and pointer prefetching was implemented and evaluated. The combined technique performs consistently better than the individual techniques on two benchmark suites with different characteristics.

III. Experimental Assumptions and Methods

In this section, we describe in detail the experimental framework including processor pipeline, benchmarks, cache organization and leakage mitigation, cache power estimation and our methods for energy calculation. In the subsequent sensitivity analysis section, the impact of changing some of these assumptions, such as processor pipeline and cache organization is discussed.

TABLE I
PROCESSOR PARAMETERS

Processor speed	1GHz
Pipeline	4-way issue, Out-of-order
RUU size	16
Branch predictor	bimod, 2K entries

TABLE II
SPEC2000 AND OLDEN BENCHMARKS SIMULATED

Benchmark	Description
SPEC2000	
181.mcf	Combinatorial Optimization
197.parser	Word Processing
179.art	Image Recognition / Neural Nets
256.bzip2	Compression
175.vpr	Versatile Place and Route
Olden	
bh	Barnes & Hut N-body Algorithm
em3d	Electromagnetic Wave Propagation
health	Colombian Health-Care Simulation
mst	Minimum Spanning Tree
perimeter	Perimeters of Regions in Images

TABLE III
MEMORY PARAMETERS

L1 D-cache	32KB, CAM-tag, fully associative, 32B cache line
L1 I-cache	32KB, 2-way, 32B cache line
L1 cache latency	1 cycle
L2 cache	unified, 256KB, 4-way, 64B cache line
L2 cache latency	12 cycles
Memory latency	100 cycles latency + 10 cycles/word

TABLE IV
CACHE CONFIGURATION AND ASSOCIATED 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)		
tag	6.5	6.3
read	9.5	100.5
write	10.3	118.6
leakage	3.1	23.0
with reduced leakage	0.8	1.5

A. Experimental Framework

We implement the hardware-based data prefetching techniques by modifying the SimpleScalar [19] simulator. We use the SUIF [20] infrastructure to implement all the compiler passes for the energy-aware prefetching techniques proposed in Sections V and VI, generating annotations for all the prefetching hints which we later transfer to assembly codes. The binaries input to the SimpleScalar simulator are created using a native Alpha assembler. A 1-GHz processor with 4-way issue was considered. Table I summarizes processor parameters.

The benchmarks evaluated are listed in Table II. The SPEC2000 benchmarks [24] use mostly array-based data structures, while the Olden benchmark suite [23] contains pointer-intensive programs that make substantial use of linked data structures. A total of ten benchmark applications, five from SPEC2000 and five from Olden were used. 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 except for *perimeter*, since they complete in relatively short time.

B. Cache Energy Modeling and Results

To accurately estimate power and energy consumption in the L1 and L2 caches, we perform circuit-level simulations using HSPICE. We base our design on a recently proposed low-power circuit [25] that we simulated using 70-nm BPTM technology. Our L1 cache includes the following low-power features: low-swing bitlines, local word-line, content addressable memory (CAM)-based tags, separate search lines, and a banked architecture. The L2 cache we evaluate is based on a banked RAM-tag design. Memory system parameters are summarized in Table III.

We fully designed the circuits in this paper for accurate analysis. CAM-based caches have previously been used in low power systems and shown to be very energy efficient [26], [27]. The key difference between CAM and RAM-tag-based approaches is that the CAM caches have a higher fraction of their power consumption from the tag check than data array access. A detailed RAM-tag-based analysis is part of our future work: we do not expect the prefetching results to be significantly different although clearly results will vary based on assumptions such as SRAM blocking of the data array in the RAM-tag caches as well as applications.

We apply a circuit-level leakage reduction technique called asymmetric SRAM cells [28]. This is necessary because otherwise our conclusions would be skewed due to very high leakage power. The *speed enhanced cell* in [28] has been shown to reduce L1 data cache leakage by 3.8× for SPEC2000 benchmarks with no impact on performance. For L2 caches, we use the *leakage enhanced cell* which increases the read time by 5%, but can reduce leakage power by at least 6×. 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 numbers of our L1 and L2 caches are shown in Table IV. 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 [29].

Each prefetching history table is implemented as a 64×64 fully-associative CAM array. This is a typical implementation of a prefetch history table [30], and is needed for performance/prefetching efficiency. HSPICE simulations show that the power consumption for each lookup is 13.3 mW and for each update is 13.5 mW. The leakage energy of prefetch tables are very small compared to L1 and L2 caches due to their small size (detailed power numbers based on HSPICE are shown in this paper).

IV. ANALYSIS OF HARDWARE DATA PREFETCHING

We simulated the five data prefetching techniques based on the experimental framework presented above. Simulation re-

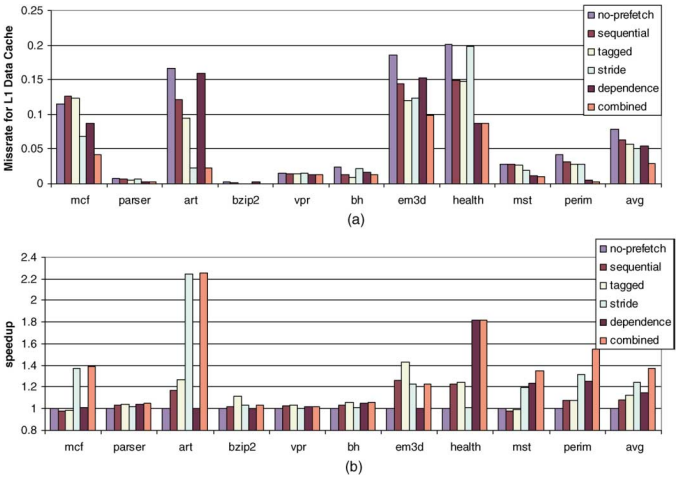

Fig. 1. Performance speedup: (a) L1 miss rate; (b) IPC speedup.

sults including performance improvement of data prefetching, the increase in memory traffic due to prefetching and the effect on energy consumption are thoroughly analyzed.

A. Performance Speedup

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

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% speedup on average) 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%. On the SPEC2000 benchmarks, the combined prefetching approach shows only marginal gains over the stride approach. The comparison between miss rate reduction in Fig. 1(a) and speedup in Fig. 1(b) matches our intuition that fewer cache misses means greater speedup.

The dependence based approach is much more effective for the five Olden pointer-intensive benchmarks in Fig. 1; the dependence-based approach eliminates about half of all the L1 cache misses and achieves an average speedup of 27%. Stride prefetching (14% on average) 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, the combined approach achieves the best performance speedup and is useful for general purpose programs which contain both array and pointer structures.

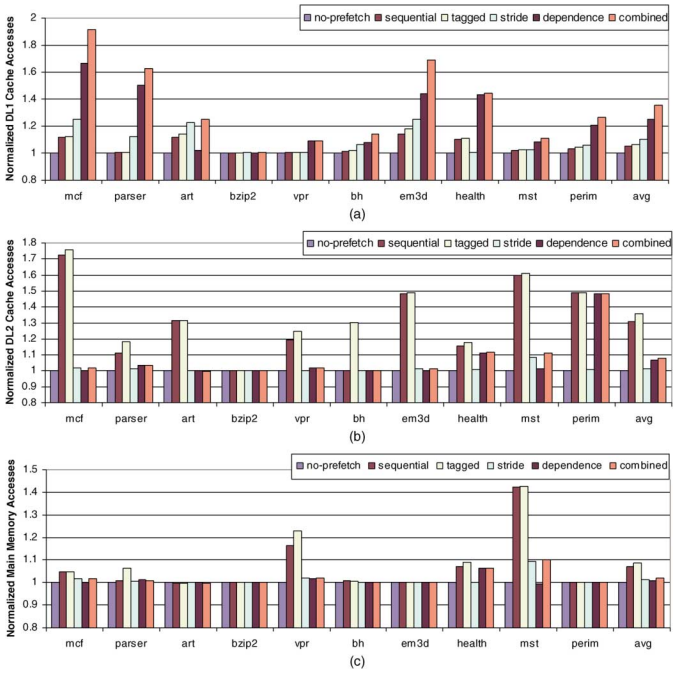

Fig. 2. Memory traffic increase for different prefetching schemes. (a) Number of accesses to L1 data cache, including extra cache-tag lookups to L1; (b) number of accesses to L2 data cache; (c) number of accesses to main memory.

B. Memory Traffic Increase and Tag Lookups: Major Sources of Energy Overhead

Memory traffic is increased because prefetched data are not always actually used in a later cache access before they are replaced. Useless data in higher levels of the memory hierarchy are a major source of power/energy consumption added by the prefetching schemes. Apart from memory traffic increases, power is also consumed when there is an 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 lookup in CAMs and tag-lookups plus data array lookups in RAM-tag caches) consumes power.

Fig. 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 combined stride and dependence based prefetching. However, the accesses to L2 only increase by 8% for the same scheme, showing that most of the L1 cache accesses are cache-tag lookups trying to prefetch data already present in L1.

Sequential prefetching techniques (both prefetch-on-miss and tagged schemes) show completely different behavior as they increase the L1 access by about 7% while resulting in a more than 30% average increase on L2 $\rightarrow$ L1 traffic. This is because sequential prefetching always tries to prefetch the next cache line which is more likely 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

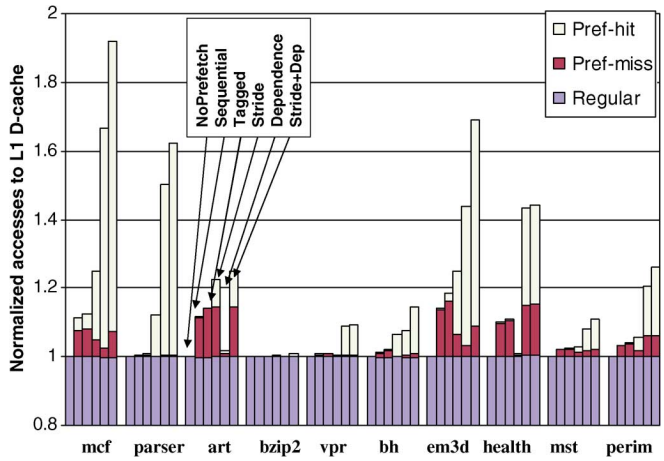

Fig. 3. Breakdown of L1 accesses, all numbers normalized to L1 cache accesses of baseline with no prefetching.

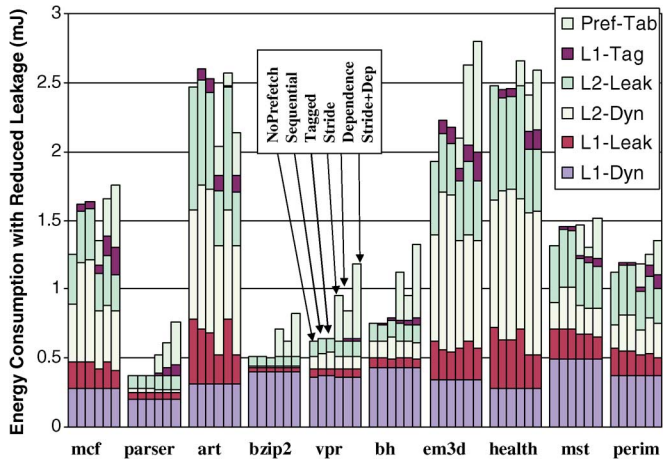

Fig. 4. Total cache energy consumption in out-of-order architectures with leakage reduction techniques applied.

L1 prefetch hits as shown in Fig. 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 with no real prefetching occurring.

In summary, from Fig. 3, L1 prefetching hits account for most of the increases in L1 accesses (70%–80%, on average). The extra L1 accesses will translate into unnecessary energy consumption.

C. Energy Consumption Overhead

Fig. 4 shows the total cache energy with leakage energy optimized by the leakage reduction techniques in [28]. The energy numbers presented for each column include (from bottom to top): L1 dynamic energy, L1 leakage energy, L2 dynamic energy, L2 leakage, L1 prefetching caused tag-checks, and prefetch hardware (history tables) energy cost.

As shown in the figure, 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

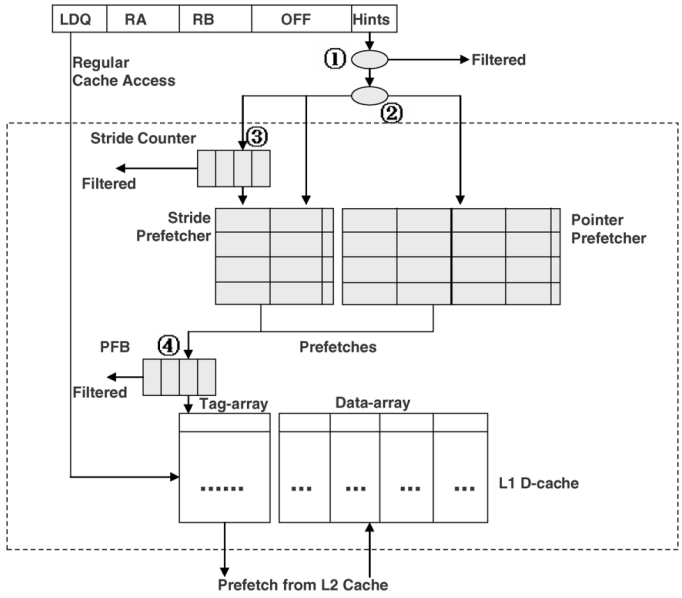

Fig. 5. Energy-aware prefetching architecture for general-purpose programs.

increased due to prefetching, the reduction in static leakage energy due to performance speedup can compensate somewhat for the increase in dynamic energy consumption.

Energy consumption for the hardware tables is very significant for all 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 dynamic energy overhead that results from combined prefetching. The reason is that prefetch tables are frequently searched and are also highly associative (this is needed for efficiency reasons).

The results in Fig. 4 show that on average, the prefetching schemes still cause significant energy consumption overhead even after leakage power is reduced to a reasonable level. The average overhead of the combined approach is more than 26%.

V. Energy-Aware Prefetching Techniques

In this section, we will introduce techniques to reduce the energy overhead for the most aggressive hardware prefetching scheme, the combined stride and pointer prefetching, that gives the best performance speedup for general-purpose programs, but is the worst in terms of energy efficiency. Furthermore, the following section (see Section VI) introduces a new power efficient prefetch engine.

Fig. 5 shows the modified prefetching architecture including four energy-saving components. The first three techniques reduce prefetch-hardware related energy costs and some extra L1 tag lookups due to prefetching [29]. The last one is a hardware-based approach designed to reduce the extra L1 tag lookups. The techniques proposed, as numbered in Fig. 5, are as follows:

1) *compiler-based selective filtering (CBSF) of hardware prefetches* approach which reduces the number of accesses to the prefetch hardware by only searching the prefetch hardware tables for selected memory accesses that are identified by the compiler;

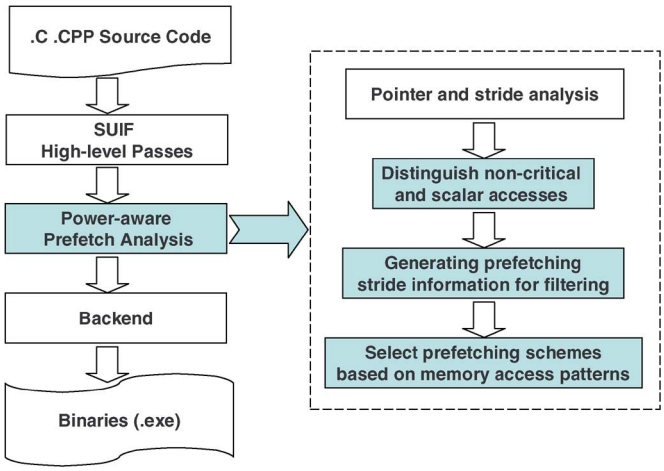

Fig. 6. Compiler analysis used for power-aware prefetching.

2) *compiler-assisted adaptive hardware prefetching (CAAP)* mechanism, which utilizes compiler information to selectively apply different prefetching schemes depending on predicted memory access patterns;
3) compiler-driven filtering technique using a *runtime stride counter (SC)* designed to reduce prefetching energy consumption on memory access patterns with very small strides;
4) hardware-based filtering technique using a *prefetch filter buffer (PFB)* applied to further reduce the L1 cache related energy overhead due to prefetching.

The compiler-based approaches help make the prefetch predictor more selective based on program information. With the help of the compiler hints, we perform fewer searches in the prefetch hardware tables and issue fewer useless prefetches, which results in less energy overhead being consumed in L1 cache tag-lookups.

Fig. 6 shows the compiler passes in our approach. Prefetch analysis is the process where we generate the prefetching hints, including whether or not to do prefetching, which prefetcher to choose, and stride information. A speculative pointer and stride analysis approach [30] is applied to help analyze the programs and generate the information needed for prefetch analysis. Compiler-assisted techniques require the modification of the instruction set architecture to encode the prefetch hints generated by compiler analysis. These hints could be accommodated by reducing the number of offset bits. We will discuss how to perform the analysis for each of the techniques in detail later.

A. Compiler-Based Selective Filtering (CBSF) of Hardware Prefetches

One of our observations is that not all load instructions are useful for prefetching. Some instructions, such as scalar memory accesses, cannot trigger useful prefetches when fed into the prefetcher.

The compiler identifies the following memory accesses as not being beneficial to prefetching.

- *Noncritical*: Memory accesses within a loop or a recursive function are regarded as critical accesses. We can safely filter out the other noncritical accesses.
- *Scalar*: Scalar accesses do not contribute to the prefetcher. Only memory accesses to array structures and linked data structures will therefore be fed to the prefetcher.

This optimization eliminates 8% of all prefetch table accesses on average, as shown in subsequent sections.

B. Compiler-Assisted Adaptive Hardware Prefetching (CAAP)

CAAP is a filtering approach that helps the prefetch predictor choose which prefetching schemes (dependence or stride) are appropriate depending on access pattern.

One important aspect of the combined approach is that it uses two techniques independently and prefetches based on the memory access patterns for all memory accesses. Since distinguishing between pointer and non-pointer accesses is difficult during execution, it is accomplished during compilation. Array accesses and pointer accesses are annotated using hints written into the instructions. During runtime, the prefetch engine can identify the hints and apply different prefetching mechanisms.

We have found that simply splitting the array and pointer structures is not very effective and affects the performance speedup (which is a primary goal of prefetching techniques). Instead, we use the following heuristic to decide whether we should use stride prefetching or pointer prefetching:

- memory accesses to an array which does not belong to any larger structure (e.g., fields in a C struct) are only fed into the stride prefetcher;
- memory accesses to an array which belongs to a larger structure are fed into both stride and pointer prefetchers;
- memory accesses to a linked data structure with no arrays are only fed into the pointer prefetcher;
- memory accesses to a linked data structure that contains arrays are fed into both prefetchers.

The above heuristic is able to preserve the performance speedup benefits of the aggressive prefetching scheme. This technique can filter out up to 20% of all the prefetch-table accesses and up to 10% of the extra L1 tag lookups.

C. Compiler-Hinted Filtering Using a Runtime Stride Counter (SC)

Another part of prefetching energy overhead comes from memory accesses with small strides. Accesses with very small strides (compared to the cache line size of 32 bytes we use) could result in frequent accesses to the prefetch table and issuing more prefetch requests than needed. For example, if we have an iteration on an array with a stride of 4 bytes, the hardware table may be accessed 8 times before a useful prefetch is issued to get a new cache line. The overhead not only comes from the extra prefetch table accesses; eight different prefetch requests are also issued to prefetch the same cache line during the eight iterations, leading to additional tag lookups.

Software prefetching would be able to avoid the penalty by doing loop unrolling. In our approach, we use hardware to accomplish loop unrolling with assistance from the compiler. The compiler predicts as many strides as possible based on static information. Stride analysis is applied not only for array-based memory accesses, but also for pointer accesses with the help of pointer analysis.

Strides predicted as larger than half the cache line size (16 bytes in our example) will be considered as large enough since they will access a different cache line after each iteration. Strides smaller than the half the cache line size will be recorded and passed to the hardware. This is a very small eight-entry buffer used to record the most recently used instructions with small strides. Each entry contains the program counter (PC) of the particular instruction and a stride counter. The counter is used to count how many times the instruction occurs after it was last fed into the prefetcher. The counter is initially set to a maximum value (decided by cache_line_size/stride) and is then decremented each time the instruction is executed. The instruction is only fed into the prefetcher when its counter is decreased to zero; then, the counter will be reset to the maximum value.

For example, if we have an array access (in a loop) with a stride of 4 bytes, the counter will be set to 8 initially. Thus, during eight occurrences of this load instruction, it is sent only once to the prefetcher.

This technique reduces 5% of all prefetch table accesses as well as 10% of the extra L1 cache tag lookups, while resulting in less than 0.3% performance degradation.

D. Hardware-Based Prefetch Filtering Using PFB

To further reduce the L1 tag-lookup related energy consumption, we add a hardware-based prefetch filtering technique. Our approach uses a very small hardware buffer called the prefetch filtering buffer (PFB).

When a prefetch engine predicts a prefetching address, it does not prefetch the data from that address immediately from the lower-level memory system (e.g., L2 Cache). Typically, tag lookups on L1 tag-arrays are performed. If the data to be prefetched already exists in the L1 Cache, the request from the prefetch engine is dropped. A cache tag-lookup costs much less energy compared to a full read/write access to the low-level memory system (e.g., the L2 cache). However, associative tag-lookups are still energy expensive.

To reduce the number of L1 tag-checks due to prefetching, a PFB is added to store the most recently prefetched cache tags. We check the prefetching address against the PFB when a prefetching request is issued by the prefetch engine. If the address is found in the PFB, the prefetching request is dropped and it is assumed that the data is already in the L1 cache. If the data is not found in the PFB, a normal tag lookup is performed. The LRU replacement algorithm is used when the PFB is full.

A smaller PFB costs less energy per access, but can only filter out a smaller number of useless L1 tag-checks. A larger PFB can filter out more, but each access to the PFB costs more energy. To find out the optimal size of the PFB, a set of benchmarks with PFB sizes of 1 to 16 were simulated. our results show that an 8-entry PFB is large enough to accomplish the prefetch filtering task with negligible performance overhead.

PFBs are not always correct in predicting whether the data is still in L1 since the data might have been replaced although its address is still present in the PFB. Fortunately, results show that the PFB misprediction rate is very low (close to 0).

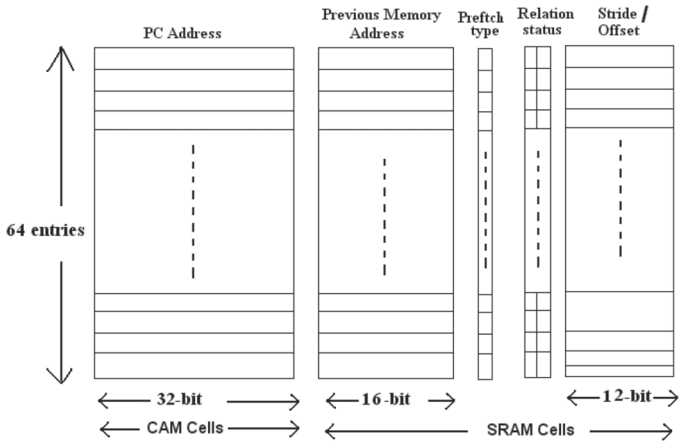

Fig. 7. Baseline design of the hardware prefetch table.

VI. PARE: A Power-Aware Prefetch Engine

The techniques presented in the previous section are capable of eliminating a significant portion of unnecessary or useless prefetching attempts. However, we have found that the energy overhead of prefetching is still pretty high, mainly because significant power is consumed in accessing the hardware table.

In this section, we propose a new power-aware data prefetching engine with a novel design of an indexed hardware history table [31]. With the help of the compiler-based location-set analysis, the proposed design could reduce power consumed per prefetch access to the engine.

Next, we will show the design of our baseline prefetching history table, which is a 64-entry fully-associative table that already uses many circuit-level low-power features. Following that we present the design of the proposed indexed history table for PARE. In the next section, we compare the power dissipations, including both dynamic and leakage power, of the two designs.

A. Baseline History Table Design

The baseline prefetching table design is a 64-entry fully-associative table shown in Fig. 7. In each table entry, we store a 32-bit program counter (the address of the instruction), the lower 16 bits of the previously used memory address (we do not need to store the whole 32 bits because of the locality property in prefetching). We also use one bit to indicate the prefetching type and two bits for status, as mentioned previously. Finally, each entry also contains the lower 12 bits of the predicted stride/offset value.

In our design, we use CAM for the PCs in the table, because CAM provides a fast and power-efficient data search function.

The memory array of CAM cells logically consists of 64 by 32 bits. The rest of the history table is implemented using SRAM arrays. During a search operation, the reference data are driven to and compared in parallel with all locations in the CAM array. Depending on the matching tag, one of the wordlines in the SRAM array is selected and read out.

The prefetching engine will update the table for each load instruction and check whether steady prefetching relationships have been established. If there exists a steady relation, the prefetching address will be calculated according to the relation

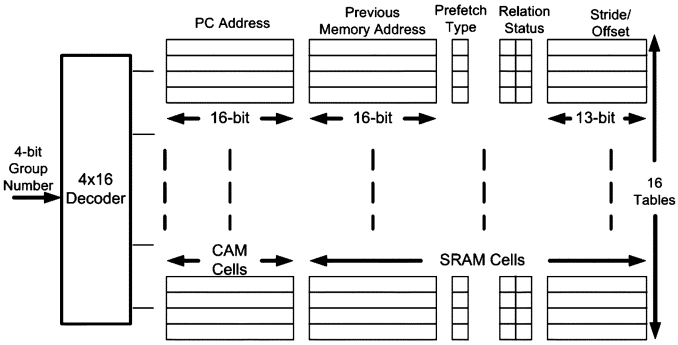

Fig. 8. Overall organization of the PARE hardware prefetch table.

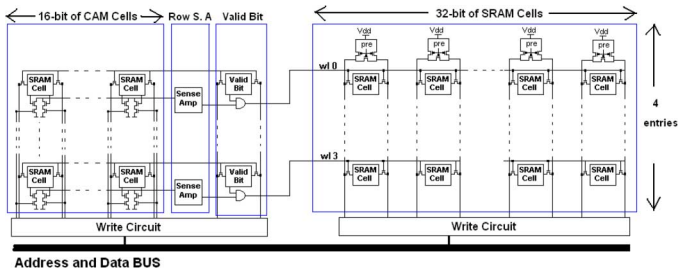

Fig. 9. Schematic for each small history table in PARE.

and data stored in the history table. A prefetching request will be issued in the following cycle.

B. PARE History Table Design

1) Circuits in PARE: Each access to the table in Fig. 7 still consumes significant power because all 64 CAM entries are activated during a search operation. We could reduce the power dissipation in two ways: reducing the size of each entry and partitioning the large table into multiple smaller tables.

First, because of the program locality property, we do not need the whole 32 bits PC to distinguish between different memory access instructions. If we use only the lower 16 bits of the PC, we could reduce roughly half of the power consumed by each CAM access.

Next, we break up the whole history table into 16 smaller tables, each containing only 4 entries, as shown in Fig. 8. Each memory access will be directed to one of the smaller tables according to their group numbers provided by the compiler when they enter the prefetching engine. The prefetching engine will update the information within the group and will make prefetching decisions solely based on the information within this group. The approach relies on new compiler support to statically determine the group number.

The group number can be accommodated in future ISAs that target energy efficiency and can be added easily in VLIW/EPIC type of designs. We also expect that many optimizations that would use compiler hints could be combined to reduce the impact on the ISA. The approach can reduce power significantly even with fewer tables (requiring fewer bits in the ISA) and could also be implemented in current ISAs by using some bits from the offset. Embedded ISAs like ARM that have 4 bits for predication in each instruction could trade off less predication bits (or none) with perhaps more bits used for compiler inserted hints.

Note that this grouping cannot be done with a conventional prefetcher. Without the group partition hints provided by compiler, the prefetch engine cannot determine which set should be searched/updated. In such a case, the entire prefetch history table must be searched, leading to higher energy consumption.

In the proposed PARE history table shown in Fig. 8, during a search operation, only one of the 16 tables will be activated. This is based on the group number provided by the compiler. We only perform the CAM search within the activated table, which is a fully-associative 4-entry CAM array, which is only a fraction of the original 64-entry table.

The schematic of each small table is shown in Fig. 9. Each small table consists of a 4×16 bits CAM array containing the program counter, a sense amplifier and a valid bit for each CAM row, and the SRAM array on the right which contains the data.

We use a power-efficient CAM cell design similar to [26]. The cell uses ten transistors that contain an SRAM cell and a dynamic XOR gate used for comparison. It separates search bitlines from the write bitlines in order to reduce the capacitance switched during a search operation.

For the row sense amplifier, a single-ended alpha latch is used to sense the match line during the search in the CAM array. The activation timing of the sense amplifier was determined with the case where only one bit in the word has a mismatch state.

Each word has the valid bit which indicates whether the data stored in the word will be used in search operations. A match line and a single-ended sense amplifier are associated with each word. A hit/miss signal is also generated: its *high* state indicating a *hit* or *multiple hits* and the *low* state indicating *no hits* or *miss*.

Finally, the SRAM array is the memory block that holds the data. Low-power memory designs typically use a six-transistor (6T) SRAM cell. Writes are performed differentially with full rail voltage swings.

The power dissipation for each successful search is the power consumed in the decoder, CAM search and SRAM read. The power consumed in a CAM search includes the power in the match lines and search lines, the sense amplifiers and the valid bits.

The new hardware prefetch table has the following benefits compared to the baseline design:

- the dynamic power consumption is dramatically reduced because of the partitioning into 16 smaller tables;
- the CAM cell power is also reduced because we use only the lower 16 bits of the PC instead of the whole 32 bits;
- another benefit of the new table is that since the table is very small (4-entry), we do not need a column sense amplifier. This also helps to reduce the total power consumed.

However, some overhead is introduced by the new design. First, an address decoder is needed to select one of the 16 tables. The total leakage power is increased (in a relative sense only) because while one of the smaller tables is active, the remaining 15 tables will be leaking. However, results show that the PARE design overcomes all these disadvantages.

2) Compiler Analysis for PARE: This section presents the compiler analysis that helps to partition the memory accesses into different groups in order to apply the new proposed PARE history table.

We apply a location-set analysis pass to generate group numbers for PARE after the high-level SUIF passes.

Location-set analysis is a compiler analysis similar to pointer alias analysis [32]. By specifying locations for each memory objects allocated by the program, a location set is calculated for each memory instruction. A key difference in our work is that we use an approximative runtime-biased analysis [30] that has no restrictions in terms of complexity or type of applications. Each location set contains the set of possible memory locations which could be accessed by the instruction.

The location-sets for all the memory accesses are grouped based on their relationships and their potential effects on the prefetching decision-making process: stride prefetching is based on the relationship within an array structure, while dependence-based pointer prefetching is based on the relationship between linked data structures.

The results of the location-set analysis, along with type information captured during SUIF analysis, give us the ability to group the memory accesses which relate during the prefetching decision-making process into the same group. For example, memory instructions that access the same location-set will be put in the same group, while the instructions accessing the same pointer structure will also be put in the same group.

Group numbers are assigned within each procedure, and will be reused on a round-robin basis if necessary. The group numbers will then be annotated to the instructions and transferred to the SimpleScalar simulator via binaries.

VII. Results and Analysis

This section details the evaluations of all the previously mentioned energy-aware techniques. We first show the results by applying each of the techniques individually; next, we apply them together.

A. Compiler-Based Filtering

Fig. 10 shows the results for the three compiler-based techniques, first individually and then combined. The results shown are normalized to the baseline, which is the combined stride and pointer prefetching scheme without any of the new techniques.

Fig. 10(a) shows the number of prefetch table accesses. The compiler-based selective filtering (CBSF) works best for *parser*: more than 33% of all the prefetch table accesses are eliminated. On average, CBSF achieves about 7% reduction in prefetch table accesses. The compiler-assisted adaptive prefetching (CAAP) achieves the best reduction for *health*, about 20%, and on average saves 6%. The stride counter filtering (SC) technique removes 12% of prefetch table accesses for *bh*, with an average of over 5%. The three techniques combined filter out more than 20% of the prefetch table accesses for five out of ten benchmarks, with an average of 18% across all applications.

Fig. 10(b) shows the extra L1 tag lookups due to prefetching. CBSF reduces the tag lookups by more than 8% on average; SC removes about 9%. CAAP averages just over 4%. The three techniques combined achieve tag-lookup savings of up to 35% for *bzip2*, averaging 21% compared to the combined prefetching baseline.

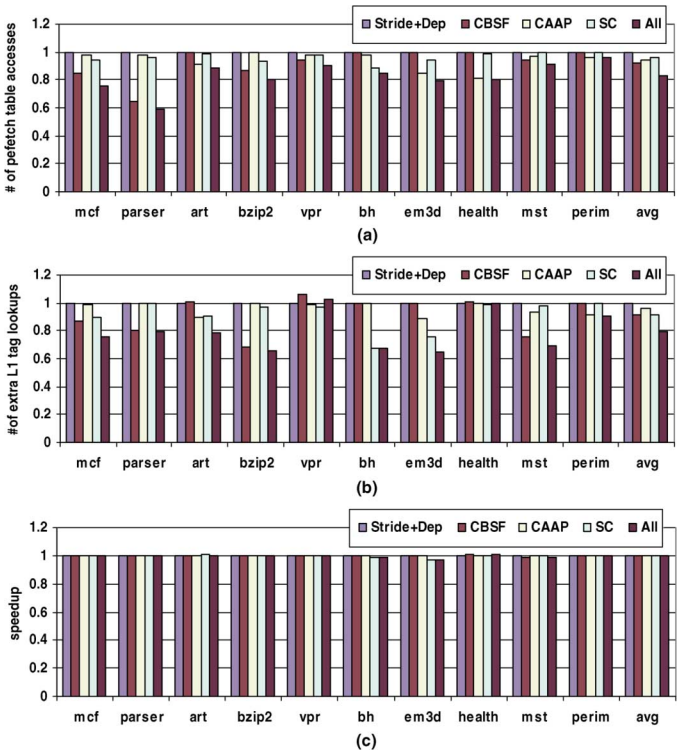

Fig. 10. Simulation results for the three compiler-based techniques: (a) normalized number of the prefetch table accesses; (b) normalized number of the L1 tag lookups due to prefetching; and (c) impact on performance.

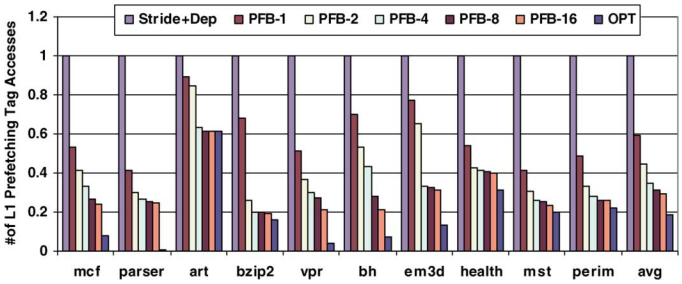

Fig. 11. Number of L1 tag lookups due to prefetching after applying the hardware-based prefetch filtering technique with different sizes of PFB.

The performance penalty introduced by the three techniques is shown in Fig. 10(c). As shown, the performance impact is negligible. The only exception is *em3d*, which has less than 3% performance degradation, due to filtering using SC.

B. Hardware Filtering Using PFB

Prefetch filtering using PFB will filter out those prefetch requests which would result in L1 cache hits if issued. We simulated different sizes of PFB to find out the best PFB size, considering both performance and energy consumption aspects. Fig. 11 shows the number of L1 tag lookups due to prefetching after applying the PFB prefetch filtering technique with PFB sizes ranging from 1 to 16.

As shown in the figure, even a 1-entry PFB can filter out about 40% of all the prefetch tag accesses (on average). An 8-entry

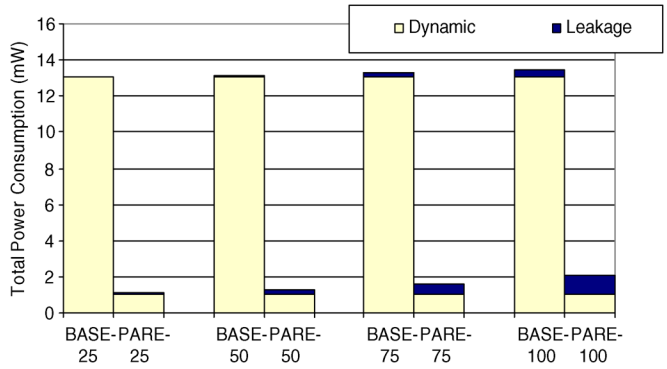

Fig. 12. Power consumption for each history table access for PARE and baseline designs at different temperatures (°C).

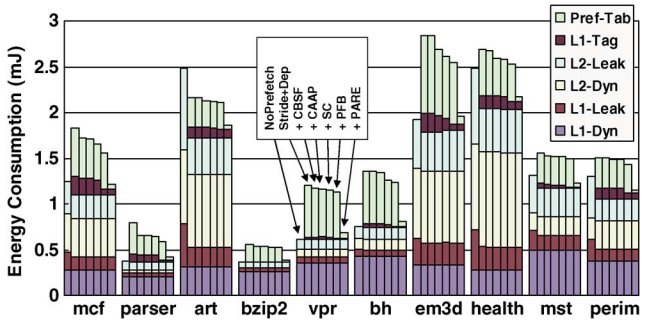

Fig. 13. Energy consumption in the memory system after applying different energy-aware prefetching schemes.

PFB can filter out over 70% of tag-checks with almost 100% accuracy. Increasing the PFB size to 16 does not increase the filtering percentage significantly. The increase is about 2% on the average compared to an 8-entry PFB, while the energy cost per access doubles.

We also show the ideal situation (OPT in the figure), where all the prefetch hits are filtered out. For some of the applications, such as *art* and *perim*, the 8-entry PFB is already very close to the optimal case. This shows that an 8-entry PFB is a good enough choice for this type of prefetch filtering.

As stated before, PFB predictions are not always correct: it is possible that a prefetched address still resides in the PFB but it does not exist in the L1 cache (has been replaced). Based on our evaluation, although the number of mispredictions increases with the size of the PFB, an 8-entry PFB makes almost perfect predictions and does not affect performance [29].

C. PARE Results

The prefetch hardware history table proposed was designed using the 70-nm BPTM technology and simulated using HSPICE with a supply voltage of 1 V. Both leakage and dynamic power are measured. Fig. 12 summarizes our results showing the breakdown of dynamic and leakage power at different temperatures for both baseline and PARE history table designs.

From the figure, we see that leakage power is very sensitive to temperature. The leakage power, which is initially 10% of the total power for the PARE design at room temperature (25 °C), increases up to 50% as the temperature goes up to 100 °C. This is because scaling and higher temperature cause subthreshold leakage currents to become a large component of the total power dissipation.

The new PARE table design proves to be much more power efficient than the baseline design. The leakage power consumption of PARE appears to more than double compared to the baseline design, but this is simply because a smaller fraction of transistors are switching and a larger fraction are idle. The dynamic power of PARE is reduced dramatically, from 13 to 1.05 mW. Consequently, the total power consumption of the prefetch history table is reduced by 7–11×. In the energy results presented next, we used the power consumption result at 75 °C, which is a typical temperature of a chip.

D. Energy Analysis With All Techniques

Fig. 13 shows the energy savings achieved. The techniques are applied in the following order: CBSF, CAAP, SC, PFB, and PARE. The figure shows the energy consumptions after each technique is added.

Compared to the combined stride and pointer prefetching, the CBSF shows good improvement for *mcf* and *parser*, with an average reduction of total memory system energy of about 3%.

The second scheme, CAAP, reduces the energy consumed by about 2%, and shows good improvement for *health* and *em3d* (about 5%).

The stride counter approach is then applied. It reduces the energy consumption for both prefetch hardware tables and L1 prefetch tag accesses. It improves the energy consumption consistently for almost all benchmarks, achieving an average of just under 4% savings on the total energy consumption.

The hardware filtering technique is applied with an 8-entry PFB. The PFB reduces more than half of the L1 prefetch tag lookups and improves energy consumption by about 3%.

Overall, the four filtering techniques together reduce the energy overhead of the combined prefetching approach by almost 40%: the energy overhead due to prefetching is reduced from 28% to 17%. This is about 11% of the total memory system energy (including L1, L2 caches, and prefetch tables).

Finally, we replace the prefetching hardware with the new PARE design and achieve energy savings of up to 8× for the prefetching table related energy (the topmost bar). After the incorporation of PARE, the prefetching energy overhead becomes very small (top bar, less than 10% for all applications). When combined with the effect of leakage reduction due to performance improvement, half of the applications studied show a total energy decrease after energy-aware data prefetching techniques applied (12% decrease for *health*).

E. Performance Aspects

Fig. 14 shows the performance statistics for the benchmarks after applying each of the five techniques proposed, one after another. We can see that there is little performance impact for the four prefetch filtering techniques. On average, the three compiler-based filtering and PFB only affect the performance benefits of prefetching by less than 0.4%.

On average, PARE causes a 5% performance benefit reduction compared to the combined prefetching scheme that consumes the most energy. However, the energy savings achieved

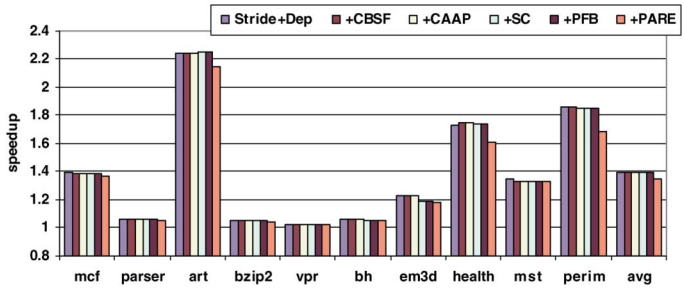

Fig. 14. Performance speedup after applying different energy-aware prefetching schemes.

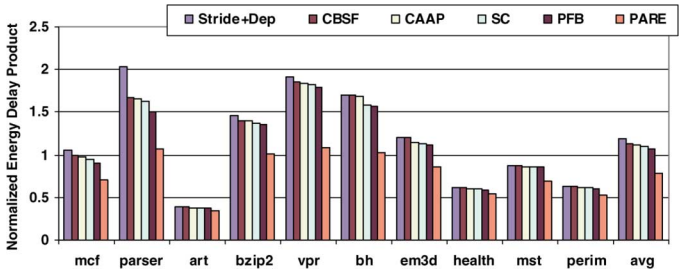

Fig. 15. EDP with different energy-aware prefetching schemes.

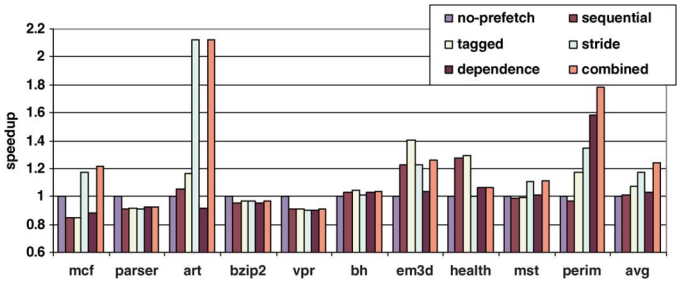

Fig. 16. Performance speedup for in-order architectures.

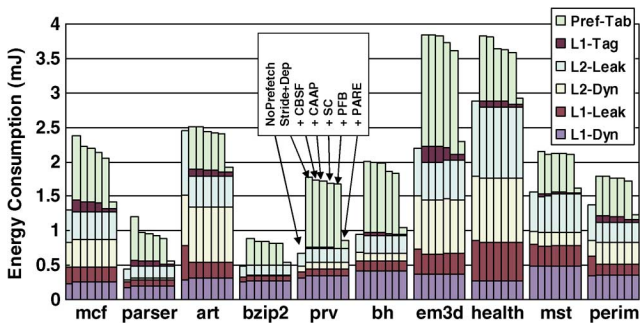

Fig. 17. Energy consumption in the memory system after applying different energy-aware prefetching schemes for in-order architectures.

from PARE are very significant. The proposed schemes combined yield a 35% performance improvement on average compared to no prefetching.

F. Energy-Delay Product (EDP)

EDP is an important metric to evaluate the effectiveness of an energy saving technique. A lower EDP indicates that the energy saving technique evaluated can be considered worthwhile because the energy saving is larger than the performance degradation (if any).

The normalized EDP numbers of the proposed energy-aware techniques are shown in Fig. 15. All numbers are normalized to the case where no prefetching techniques are used. Compared to the combined stride and pointer prefetching, the EDP improves by almost 48% for *parser*. On average, the four power-aware prefetching techniques combined improve the EDP by about 33%.

Six out of the ten applications have a normalized EDP less than 1 with all power aware techniques applied. Four applications have a normalized EDP slightly greater than 1. However, it is important to note that the EDP with PARE is still much lower than the EDP with the combined prefetching technique for all applications. This is due to the considerable savings in energy achieved with minimal performance degradation.

The average EDP with PARE is 21% lower than with no prefetching. The normalized EDP results show that data prefetching, if implemented with energy-aware schemes and hardware, could be very beneficial for both energy and performance.

VIII. Sensitivity Analysis

In this section, we change the experimental framework (pipelines, memory organization) and analyze the impact of hardware prefetching and various energy aware techniques. New experiments include: evaluating the impact of energy-aware techniques on in-order architectures and the impact of varying cache sizes. We find that our energy-aware techniques continue to be applicable in significantly reducing the energy overhead of prefetching in these scenarios.

A. Impact on in-order architectures

While most microprocessors use multiple-issue out-of-order execution, many mobile processors use in-order pipelines. Energy conservation in these systems is of paramount importance. Therefore, the impact of prefetching and energy-aware techniques on four-way issue, in-order architectures was extensively evaluated. In these simulations all other processor and memory parameters were kept identical to Tables I and III, respectively. Fig. 16 shows performance speedup of all hardware prefetching schemes against a scheme with no prefetching for in-order architectures. In general, in-order architectures show similar performance trends as out-of-order architectures with hardware data prefetching. However, the actual performance benefits of prefetching are somewhat lesser; average performance improvement is around 24% for the combined (stride + dependence) approach, compared to 40% for out-of-order architectures.

Fig. 17 shows the energy savings for in-order execution with all techniques applied. We see that CBSF, CAAP, SC, and PFB together improve the total memory subsystem energy by 8.5%. This is somewhat less than the out-of-order case, where the corresponding number was 11%.

As with out-of-order architecture, PARE significantly reduces the prefetching related energy overhead compared to the combined prefetching approach without any energy aware

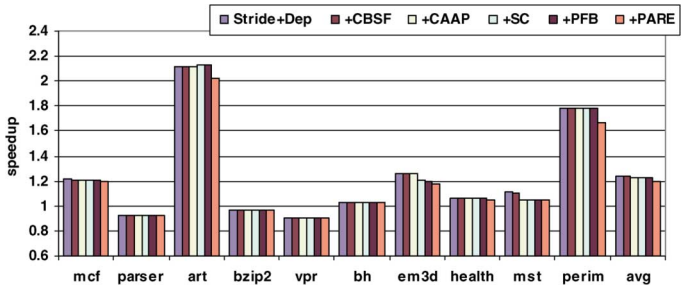

Fig. 18. Performance speedup after applying different energy-aware prefetching schemes (in-order architectures).

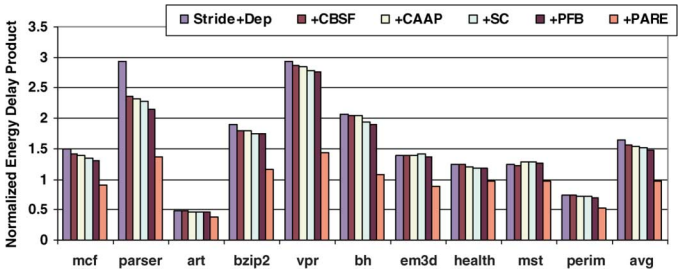

Fig. 19. Normalized energy-delay product with different energy-aware prefetching schemes (in-order architectures).

techniques. In cases where a significant performance improvement was shown (*art* and *perim*), the total energy with PARE is less than with no prefetching due to a significant improvement in leakage energy. In *em3d, health, mst* the total energy with PARE is less than 5% higher than the case with no-prefetching, implying that almost all prefetch related energy overhead can be reclaimed using the PARE engine and compiler techniques.

Fig. 18 shows the performance impact for in-order architectures with energy-aware techniques included. For *art, perim*, which have the most significant performance improvement, the compiler techniques have almost no impact on performance. With all techniques incorporated, the average reduction in performance benefit from the combined (stride + dependence) scheme is around 3%–4%. However, the energy savings far outweigh the small performance benefit decrease, similar to what was shown in out-of-order architectures. The PARE scheme has a 20% speedup compared to the no-prefetching baseline.

Fig. 19 shows the normalized energy delay products for in-order execution. On average, EDP improves by 40% for PARE over the combined (stride + dependence) prefetching scheme for the benchmarks studied. These results indicate that the various hardware prefetching and energy efficient techniques are equally applicable to out-of-order as well as in-order architectures.

B. Impact of Larger Caches Sizes

The impact of increasing cache sizes is discussed in this section. The experiments detailed here assume 128 kB IL1 and DL1 caches and 1 MB DL2. We estimate the leakage and dynamic power for these caches based on the following assumptions:

- *Leakage Power:* Leakage power increases linearly with cache size, e.g., 128 kB DL1 and IL1 caches consume 4× leakage power compared to a 32 kB cache.

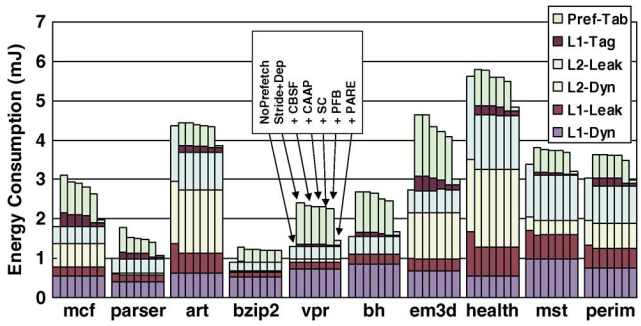

Fig. 20. Impact on energy consumption in the memory system for 128 kB L1 and 1 MB L2—dynamic power scaled by 2×

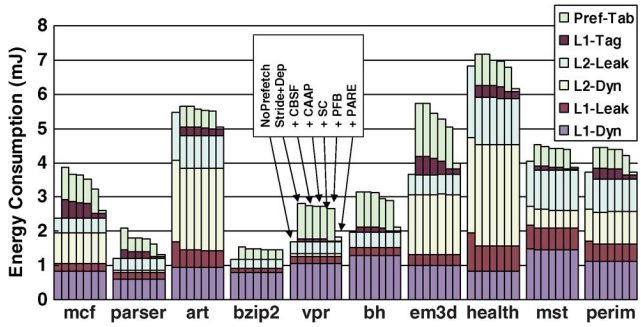

Fig. 21. Impact on energy consumption in the memory system for 128 kB L1 and 1 MB L2—dynamic power scaled by 3×.

- *Dynamic Power:* By using cache blocking and maintaining the same associativity, dynamic power can be subject to less than a linear increase in relation to cache size. However, the additional components introduced (e.g., larger decoders and wiring) will cause increased power dissipation. For this analysis, we consider two different dynamic power estimations: in one case the 128 kB cache dynamic power is 2× that of the 32 kB cache, and in the other it is 3×. While the actual power dissipation will depend on circuit and cache organization, we consider these to be representative scenarios. We estimate dynamic power for the 1 MB DL2 in the same fashion.
- *Prefetch Tables:* Prefetch tables are assumed identical to ones used in earlier sections and consume the same amount of power.

Fig. 20 shows the energy consumption for the benchmarks with all techniques incorporated assuming that dynamic energy scales by 2×. The average energy savings over the combined (stride + dependence) approach with all compiler techniques and PFB is 8%, and 26% with PARE. The energy numbers assuming a 3× increase in dynamic power are very similar (see Fig. 21), PARE improves energy over the combined approach by 23%. However, average total energy with all energy saving techniques in both cases is around 4% higher than the total energy without prefetching. The primary reason is that leakage energy improvements are smaller, because performance improvement of prefetching is diminished (see Fig. 22) due to fewer cache misses. However, for larger problem sizes, the performance benefits of prefetching are expected to be larger.

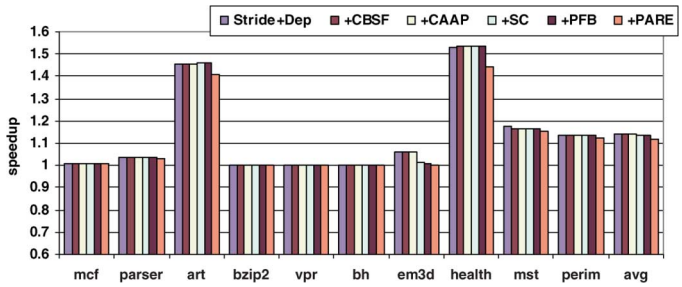

Fig. 22. Impact on performance speedup for larger cache size.

IX. Related Work

Compiler-assisted techniques for prefetching have been explored by various groups. In general, these use profiling as an effective tool to recognize data access patterns for making prefetch decisions. Luk *et al.* [33] use profiling to analyze executable codes to generate post-link relations which can be used to trigger prefetches. Wu [34] proposes a technique which discovers regular strides for irregular codes based on profiling information. Chilimbi *et al.* [35] use profiling to discover dynamic hot data streams which are used for predicting prefetches. Inagaki *et al.* [36] implemented a stride prefetching technique for Java objects. Most of this prefetching research focuses on improving performance, instead of energy consumption. Furthermore, our techniques are in the context of hardware data prefetching.

To reduce memory traffic introduced by prefetching, Srinivasan *et al.* propose a static filter [37], which uses profiling to select which load instructions generate data references that are useful prefetch triggers. In our approach, by contrast, we use static compiler analysis and a hardware-based filtering buffer (PFB), instead of profiling-based filters.

Wang *et al.* [38] also propose a compiler-based prefetching filtering technique to reduce traffic resulting from unnecessary prefetches. Although the above two techniques have the potential to reduce prefetching energy overhead, there are no specific discussions or quantitative evaluation of the prefetching related energy consumption, thus we cannot provide a detailed comparison with their energy efficiency.

Moshovos *et al.* proposes Jetty [39], an extension over snoop coherence that stops remotely-induced snoops from accessing the local cache tag arrays in SMP servers, thus saving power and reducing bandwidth on the tag arrays. The purpose of our hardware filtering (PFB) is also saving power on the tag arrays, but in a different scenario.

Chen [11] combines Mowry's software prefetching technique [8] with dynamic voltage and frequency scaling to achieve power aware software prefetching. Our hardware-based prefetching approaches and energy-aware techniques can be complementary to this software prefetching approach. Furthermore, the scope for voltage scaling is diminished in advanced CMOS technology generations [40] where voltage margins are expected to be lower.

X. Conclusion

This paper explores the energy-efficiency aspects of hardware data-prefetching techniques and proposes several new techniques and a PARE to make prefetching energy-aware. PARE reduces prefetching related energy consumption by 7–11×. In conjunction with a net leakage energy reduction due to performance improvement, this may yield up to 12% less total energy consumption compared to a no-prefetching baseline. While the new techniques may have a very small reduction in performance benefits compared to a scheme with prefetching but no energy-aware techniques, they still maintain a significant speedup (35% in out-of-order, and 20% in in-order architectures) compared to the no-prefetching baseline, thereby achieving the twin goals of energy efficiency and performance improvement.

References

[1] A. J. Smith, "Sequential program prefetching in memory hierarchies," *IEEE Computer*, vol. 11, no. 12, pp. 7–21, Dec. 1978.
[2] J. L. Baer and T. F. Chen, "An effictive on-chip preloading scheme to reduce data access penalty," in *Proc. Supercomput.*, 1991, pp. 179–186.
[3] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Proc. ASPLOS-VIII*, Oct. 1998, pp. 115–126.
[4] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," in *Proc. ISCA-26*, 1999, pp. 111–121.
[5] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless content-directed data prefetching mechanism," in *Proc. ASPLOS-X*, 2002, pp. 279–290.
[6] T. Mowry, "Tolerating latency through software controlled data prefetching," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, Mar. 1994.
[7] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger, "Spaid: Software prefetching in pointer- and call-intensive environments," in *Proc. Micro-28*, Nov. 1995, pp. 231–236.
[8] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *Proc. ASPLOS-VII*, Oct. 1996, pp. 222–233.
[9] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "Memory-system design considerations for dynamically-scheduled processors," in *Proc. ISCA-24*, 1997, pp. 133–143.
[10] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. ASPLOS-V*, Oct. 1992, pp. 62–73.
[11] J. Chen, Y. Dong, H. Yi, and X. Yang, "Power-aware software prefetching," *Lecture Notes Comput. Sci.*, vol. 4523/2007, pp. 207–218, 2007.
[12] D. Bernstein, D. Cohen, A. Freund, and D. E. Maydan, "Compiler techinques for data prefetching on the PowerPC," in *Proc. PACT*, Jun. 1995, pp. 19–26.
[13] 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 J.*, vol. 47, no. 1, pp. 25–33, Feb. 1996.
[14] G. Doshi, R. Krishnaiyer, and K. Muthukumar, "Optimizing software data prefetches with rotating registers," in *Proc. PACT*, Sep. 2001, pp. 257–267.
[15] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 12, pp. 24–36, Mar./Apr. 1999.
[16] V. Santhanam, E. H. Gornish, and H. Hsu, "Data prefetching on the HP PA8000," in *Proc. ISCA-24*, May 1997.
[17] M. K. Gowan, L. L. Biro, and D. B. Jackson, "Power considerations in the design of the alpha 21264 microprocessor," in *Proc. DAC*, Jun. 1998, pp. 726–731.
[18] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, G. W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, and S. C. Thierauf, "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor," *Digit. Techn. J. Digit. Equip. Corp.*, vol. 9, no. 1, pp. 49–62, 1997.
[19] D. C. Burger and T. M. Austin, "The Simplescalar tool set, Version 2.0," Univ. Wisconsin, Madison, Tech. Rep. CS-TR-1997-1342, Jun. 1997.
[20] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. Lam, and J. L. Hennessy, "SUIF: A parallelizing and optimizing research compiler," Comput. Syst. Lab., Stanford Univ., Stanford, CA, Tech. Rep.CSL-TR-94-620, May 1994.

[21] Y. Guo, S. Chheda, I. Koren, C. M. Krishna, and C. A. Moritz, "Energy characterization of hardware-based data prefetching," in *Proc. Int. Conf. Comput. Des. (ICCD)*, Oct. 2004, pp. 518–523.
[22] A. J. Smith, "Cache memories," *ACM Comput. Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.
[23] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 2, pp. 233–263, Mar. 1995.
[24] SPEC, "The standard performance evaluation corporation," 2000. [Online]. Available: http://www.spec.org
[25] M. Bennaser and C. A. Moritz, "A step-by-step design and analysis of low power caches for embedded processors," presented at the Boston Area Arch. Workshop (BARC), Boston, MA, Jan. 2005.
[26] M. Zhang and K. Asanovic, "Highly-associative caches for low-power processors," presented at the Kool Chips Workshop, Micro-33, Monterey, CA, Dec. 2000.
[27] R. Ashok, S. Chheda, and C. A. Moritz, "Cool-mem: Combining statically speculative memory accessing with selective address translation for energy efficiency," in *Proc. ASPLOS-X*, 2002, pp. 133–143.
[28] N. Azizi, A. Moshovos, and F. N. Najm, "Low-leakage asymmetric-cell SRAM," in *Proc. ISLPED*, 2002, pp. 48–51.
[29] Y. Guo, S. Chheda, I. Koren, C. M. Krishna, and C. A. Moritz, "Energy-aware data prefetching for general-purpose programs," in *Proc. Workshop Power-Aware Comput. Syst. (PACS'04) Micro-37*, Dec. 2004, pp. 78–94.
[30] Y. Guo, S. Chheda, and C. A. Moritz, "Runtime biased pointer reuse analysis and its application to energy efficiency," in *Proc. Workshop Power-Aware Comput. Syst. (PACS) Micro-36*, Dec. 2003, pp. 1–15.
[31] Y. Guo, M. Bennaser, and C. A. Moritz, "PARE: A power-aware hardware data prefetching engine," in *Proc. ISLPED*, New York, 2005, pp. 339–344.
[32] R. Rugina and M. Rinard, "Pointer analysis for multithreaded programs," in *Proc. PLDI*, Atlanta, GA, May 1999, pp. 77–90.
[33] C.-K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, and R. Cohn, "Profile-guided post-link stride prefetching," in *Proc. 16th Int. Conf. Supercomput. (ICS)*, Jun. 2002, pp. 167–178.
[34] Y. Wu, "Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching," in *Proc. PLDI*, C. Norris and J. B. Fenwick, Jr., Eds., Jun. 2002, pp. 210–221.
[35] T. M. Chilimbi and M. Hirzel, "Dynamic hot data stream prefetching for general-purpose programs," in *Proc. PLDI*, C. Norris and J. B. Fenwick, Jr., Eds., Jun. 2002, pp. 199–209.
[36] T. Inagaki, T. Onodera, K. Komatsu, and T. Nakatani, "Stride prefetching by dynamically inspecting objects," in *Proc. PLDI*, Jun. 2003, pp. 269–277.
[37] V. Srinivasan, G. S. Tyson, and E. S. Davidson, "A static filter for reducing prefetch traffic," Univ. Michigan, Ann Arbor, Tech. Rep. CSE-TR-400-99, 1999.
[38] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, "Guided region prefetching: A cooperative hardware/software approach," in *Proc. ISCA*, Jun. 2003, pp. 388–398.
[39] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi, "JETTY: Filtering snoops for reduced energy consumption in smp servers," in *Proc. HPCA-7*, 2001, p. 85.
[40] B. Ganesan, "Introduction to multi-core," presented at the Intel-FAER Series Lectures Comput. Arch., Bangalore, India, 2007.

Yao Guo (S'03–M'07) received the B.S. and M.S. degrees in computer science from Peking University, Beijing, China, and the Ph.D. degree in computer engineering from University of Massachusetts, Amherst.

He is currently an Associate Professor with the Key Laboratory of High-Confidence Software Technologies (Ministry of Education), School of Electronics Engineering and Computer Science, Peking University. His research interests include low-power design, compilers, embedded systems, and software engineering.

Pritish Narayanan (S'09) received the B.E. (honors) degree in electrical and electronics engineering and the M.Sc. (honors) degree in chemistry from the Birla Institute of Technology and Science, Pilani, India, in 2005. He is currently working toward the Ph.D. degree in electrical and computer engineering from the University of Massachusetts, Amherst.

Currently, he is a Research Assistant with the Department of Electrical and Computer Engineering, University of Massachusetts. He was previously employed as a Research and Development Engineer at IBM, where he worked on process variation and statistical timing analysis. His research interests include nanocomputing fabrics, computer architecture, and VLSI.

Mr. Narayanan was a recepient of the Best Paper Award at ISVLSI 2009. He has served as a reviewer for IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS and the IEEE TRANSACTIONS ON NANOTECHNOLOGY.

Mahmoud Abdullah Bennaser (M'08) received the B.S. degree in computer engineering from Kuwait University, Safat, Kuwait, in 1999 and the M.S. degree in computer engineering from Brown University, Providence, RI, in 2002, and the Ph.D. degree in computer engineering from University of Massachusetts, Amherst, in 2008.

He is an Assistant Professor with the Computer Engineering Department, Kuwait University. His research interests include computer architecture, and low-power circuit design.

Saurabh Chheda received the M.S. degree in computer engineering from University of Massachusetts, Amherst, in 2003.

He is currently the SoC Processor Architect with Lattice Semiconductor, where he is involved in the research and development of innovative microprocessor architectures for programmable devices.

Csaba Andras Moritz (M'85) received the Ph.D. degree in computer systems from the Royal Institute of Technology, Stockholm, Sweden, in 1998.

From 1997 to 2000, he was a Research Scientist with Laboratory for Computer Science, the Massachusetts Institute of Technology (MIT), Cambridge. He has consulted for several technology companies in Scandinavia and held industrial positions ranging from CEO, to CTO, and to founder. His most recent startup company, BlueRISC Inc, develops security microprocessors and hardware-assisted security solutions. He is currently a Professor with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst. His research interests include computer architecture, compilers, low power design, security, and nanoscale systems.