# Load Miss Prediction - Exploiting Power Performance Trade-offs

Konrad Malkowski, Greg Link, Padma Raghavan and Mary Jane Irwin
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA, 16802. Tel: 814-865-9505
E-mail:{malkowsk, link, raghavan, mji}@cse.psu.edu

***Abstract*— Modern CPUs operate at GHz frequencies, but the latencies of memory accesses are still relatively large, in the order of hundreds of cycles. Deeper cache hierarchies with larger cache sizes can mask these latencies for codes with good data locality and reuse, such as structured dense matrix computations. However, cache hierarchies do not necessarily benefit sparse scientific computing codes, which tend to have limited data locality and reuse. We therefore propose a new memory architecture with a *Load Miss Predictor (LMP)*, which includes a data bypass cache and a predictor table, to reduce access latencies by determining whether a load should bypass the main cache hierarchy and issue an early load to main memory. Our architecture uses the L2 (and lower caches) as a victim cache for data removed from our bypass cache. We use cycle-accurate simulations, with SimpleScalar and Wattch to show that our LMP improves the performance of sparse codes, our application domain of interest, on average by 14%, with a 13.6% increase in power. When the LMP is used with dynamic voltage and frequency scaling (DVFS), performance can be improved by 8.7% with system power savings of 7.3% and energy reduction of 17.3% at 1800MHz relative to the base system at 2000MHz. Alternatively our LMP can be used to improve the performance of SPEC benchmarks by an average of 2.9% at the cost of 7.1% increase in average power.**

## I. Introduction

Modern processors operate in the Gigahertz frequency range, are highly pipelined, with several levels of caches, and have the ability to issue multiple instructions in the same cycle. Such designs benefit scientific computing codes by enabling faster floating point computation rates, but only when there is data locality and reuse. However, in many *sparse* scientific codes representing scalable modeling and simulation applications [1], [2], the latency of memory accesses [3] presents a serious performance bottleneck. Such codes have limited data locality and reuse, and typically do not benefit from deep cache hierarchies and high clock frequencies. In this paper, we develop a unique memory architecture with a Load Miss Predictor (LMP) for improving the performance of such sparse applications by reducing effective load latencies. We also demonstrate how such an LMP can be designed to benefit sparse application without adversely impacting the performance of applications with significant data reuse and locality. Finally, we show how our LMP can be combined with Dynamic Voltage and Frequency Scaling (DVFS) [4] to reduce power while improving the performance of sparse scientific applications yielding significant reductions in energy consumption.

A large variety of scientific computing codes operate on either *dense* or *sparse* matrices [5]. An $N \times N$ dense matrix, stored in a traditional two dimensional data structure, has $N^2$ nonzero elements. On the other hand, an $N \times N$ sparse matrix typically has only $cN$ nonzero elements, where $c$ is a small constant in the range of $2 - 20$, and $N$ can be on the order of $10^9$. In such matrices, only nonzero elements are stored (with their corresponding index positions) in several one-dimensional arrays whose elements are typically accessed consecutively. Such sparse schemes lead to scalable formulations for many modeling and simulation applications [1], where large matrix sizes are required to capture the details occurring at micro-scale and calculate their effect on macro-scale events. Problem sizes are limited only by the available computational resources.

Multiprocessor systems are typically used to solve larger and more refined models. This in turn translates to a scaling of problem size with the number of processors; the single processor workload is selected to be the largest problem size that can fit in physical memory. This allows better efficiency to be maintained upon scaling to multiple processors where network latencies can dominate. Consequently, optimizing single processor performance is of considerable significance in high performance scientific computing.

Dense matrix operations, such as those in LINPACK [6], inherently have higher levels of data-locality and data-reuse, resulting in a relatively large-number of floating-point operations that can be performed per memory-access [7]. Such codes, for example in ATLAS [7], have optimized data access and reuse patterns to utilize deep cache hierarchies, multiple data paths and floating-point units, and high clock frequencies to achieve near peak execution rates [6]. However, sparse computations differ intrinsically from dense computations in these respects as discussed in Section II. As a consequence, they utilize only a fraction of the computing power of modern microprocessors despite sophisticated attempts at performance tuning [2], [8].

Technology scaling for microprocessor design has resulted in more and faster transistors on a chip. Thus, performance has increased, but at the cost of increasing chip power densities to

1-4244-0910-1/07/$20.00 ©2007 IEEE

the point where power is the real limiter for future scaling [9]. As chips approach their packaging thermal limits and cooling costs become prohibitive, power-aware design is starting to receive considerable attention in the high performance computing community [10], [11]. Sparse computations present interesting opportunities for power-aware high performance scientific computing [12] because, although they represent scalable formulations, they can achieve only a fraction of peak performance (despite extensive tuning). We conjecture that both performance and power improvements are possible by considering the co-evolution of architectural optimizations and tuned sparse matrix computations.

The remainder of this paper is organized as follows. Section II contains a brief overview of sparse applications and codes. Sections III, IV and V contain our main contributions. We develop our LMP architecture and discuss its implementation in Section III; in the last part, i.e. in Section VI, we discuss similarities and differences with earlier approaches. We present our methodology and empirical results respectively in Sections IV and V. We end with concluding remarks in Section VII.

## II. Sparse Scientific Computing Applications

Computational modeling and simulation is now widely used in research and industry to study problems in science and engineering and in industrial manufacturing and design. Examples of modeling applications include astrophysics [13], climate modeling [14], fluid flows [15], and many others. Many of these applications involve a model described by nonlinear Partial Differential Equations (PDEs) discretized in space and potentially evolving over time. These are solved using fully implicit Newton-Krylov schemes as well as semi-implicit approaches [5] which offer the advantage of faster convergence and numerical stability. The underlying computations are primarily *sparse*, i.e., involving large, often unstructured matrices with relatively few nonzero elements. A key feature common to these applications is that a given simulation includes many linear solutions involving sparse matrices, and the application time is dominated by the time to compute the sparse linear solution. In view of the important role of sparse applications, considerable research in scientific computing has been expended to improve the performance of sparse matrix operations [2], [8].

In this paper we consider the following representative sparse codes to evaluate our LMP. We use the NAS benchmarks MG and CG with a "W" type workload [16]. Both benchmarks use a matrix vector multiplication routine that accounts for more than 90% of their execution time. Therefore, in our tests we additionally use a sparse matrix vector multiplication kernel from Sparsity (SMV) [2]. In our experiments, we consider two versions of SMV, an unoptimized form with no blocking (SMV-U) typical of many scientific codes and an optimized form with 2x1 blocking and loop unrolling (SMV-O) which decreases loads at the expense of a slight increase in floating point operations [2]. We report on the performance of SMV-U and SMV-O on the following four representative sparse matrices from structured mechanics: `bcsstk31`, `fdm2`, `qa8fm`, `msc23052`. The matrix properties including the name, dimension ($\times 10^3$), number of nonzeros ($\times 10^6$) and the percentage of nonzeroes relative to a dense matrix of the same dimension are: `bcsstk31`, 35.6, 1.2, .09%; `fdm2`, 32.1,.16, .01%; `qa8fm`, 66.1, 1.6, .03%; and `msc23052`, 23.0, 1.1, .21%. These matrices had been reordered using the Reverse Cuthill McKee [17] scheme to improve the locality of access in the source vector [12] as is commonly done for tuned scientific codes.

## III. Designing a LMP

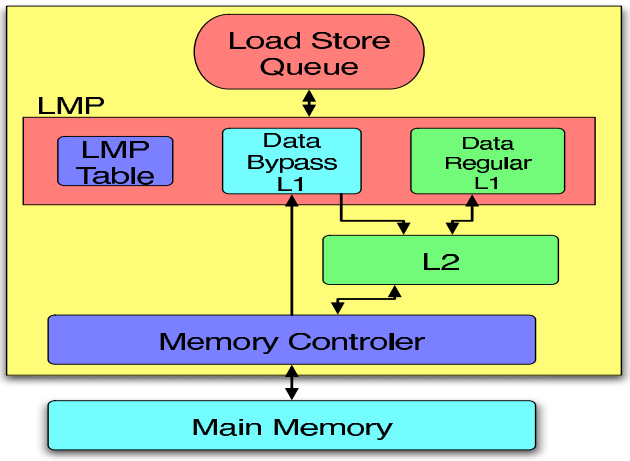


Fig. 1. LMP design.

Modern CPUs have deep cache hierarchies to mask the large latencies of accessing the main memory for codes with data reuse. However, when there is poor data reuse, as is the case in many sparse scientific codes, these elaborate cache systems actually add to the access latency as the caches are inevitably and unproductively accessed before each main memory access. As an example, for level 1 cache, level 2 cache, and main memory latencies of 1, 19 and 122 cycles respectively, bypassing the L2 cache on a miss could save 16% of the total latency. This effect will be further magnified with deeper cache hierarchies, especially with NUCA [18] caches where the longest bank miss latency can be significantly greater than the average cache latency. The rest of this section presents the design and implementation of our Load Miss Predictor to decrease penalties of such inevitable load misses.

### A. LMP design and operation

Figure 1 shows our proposed design, where the 32KB level-1 data cache (L1) is split into two portions, labeled *regular* and *bypass*. The *regular* portion is a low latency, highly associative cache typical of modern processors, 16KB in size, with a 32 byte long cache line. This portion of the L1 data cache has a direct connection to the level-2 (L2) cache. The *bypass* portion of the data L1 is also 16KB, and is directly connected to both the L2 cache and to the memory controller. We call it

the *Bypass L1* cache. Data is provided to the bypass cache from the main memory directly, and the L2 cache serves as its victim cache. The *Bypass L1* cache has a cache line width of 128 bytes, which is the cache-line width of the L2 cache as well as the amount of data transferred from the memory in one request. The cache-line width is based on the 128-bit wide memory bus and SDRAM transfer protocol [19], [20]. A prediction mechanism is added to this structure (the LMP Table in Figure 1). A detailed discussion and experimental justification for these design choices is presented in Section V.

When a load instruction is executed, its data request is sent to both the *Regular L1* and *Bypass L1* caches. When both caches miss the LMP table is consulted. The PC address of the load instruction is looked up in the LMP table much as in *branch prediction* technology. The LMP then determines whether the load should bypass (predicted miss), or proceed through the main cache hierarchy (predicted hit). When there is insufficient history for prediction, such as in newly encountered loads, they are directed through the main cache hierarchy, i.e., are predicted hits. Our predictor tracks only the history of loads that miss in both *Regular L1* and *Bypass L1*.

On predicted hits the load proceeds through the normal cache hierarchy. If the data is not found, the predictor encodes an incorrect prediction (predicted hit, cache miss) in the table. Otherwise, the predictor encodes a correct prediction in the table (predicted hit, cache hit). As long as the data is found anywhere in the cache hierarchy it will not be brought into the *Bypass L1*.

When data requested by a load instruction at a particular PC address is repeatedly not found in the regular cache hierarchy, that load instruction will be marked as suitable for cache bypassing (predicted miss). On a predicted miss, a parallel request for an early load will be issued to the main memory by the *Bypass L1* and to the L2 cache by the *Regular L1*. If the L2 cache responds with data before the main memory does, the memory access will be canceled, the load will be marked as (predicted miss, cache hit), and the data will be stored in the *Regular L1* cache. Any data returned by the main memory as a result of the early load will be ignored. On the other hand, if the L2 does not respond with data, the main memory read access will be already underway, thus reducing the memory latency. The corresponding data will be stored in the *Bypass L1* cache, and the load will be marked as (predicted miss, cache miss). Both portions of L1 cache are checked for data on a store instruction to ensure cache consistency. As a result, a store will always proceed through the path used for loading the corresponding address.

Additional modifications to the processor are required to add the LMP table and bypass cache. Each load instruction has to be accompanied by either the complete PC address or a unique identifier in order to correctly map it into the predictor table. The bus connecting the L1 cache to L2 cache requires additional wires to communicate hits and misses to the load predictor. An additional wire is also required to control the tag and data look-up schemes in the cache hierarchy. In addition we need a mechanism such that during bypassed loads only cache tags are accessed, and data arrays are accessed only upon a hit. Finally, an additional bus is required to connect the *Bypass L1* to the main memory controller to initiate an early load.

Ideally, the prediction mechanism should be 100% accurate. However, in practice the design of the LMP must reflect a trade-off between accuracy, speed and hardware complexity. We therefore utilize a small 2-level predictor (L2H8) tracking 1024 independent load instructions, with an 8 bit hit/miss history per instruction (requiring 256 entries in the second level table). We believe this configuration offers a good trade-off between accuracy and hardware costs.

An important question is what to do with data displaced from the *Bypass L1* cache. We considered many replacement policies for data displaced from *Bypass L1* and chose a policy, where all cache lines that are displaced from the *Bypass L1* are written to the L2 cache whether clean or dirty ("all replaced victim"). Other possible policies are to write displaced dirty data directly to main memory ("no victim"), or to place only dirty displaced cache lines in the victim cache ("only dirty victim"). LMP design tradeoffs are discussed further in Section V.

We expect that our LMP design will have no negative impacts on clock cycle time, because our modifications to the CPU pipeline only require carrying additional bits, and thus a wider instruction. Our LMP could possibly introduce a 1 cycle penalty to L1 cache misses followed by access to L2 or main memory. However, we do not consider this penalty to be significant given the long latencies of main memory access.

## IV. Methodology

In this section, we describe our methodology for obtaining cycle accurate performance and power numbers through simulation.

We utilize SimpleScalar3.0d [21] and Wattch1.02d [22] with modifications to model power consumption and performance impacts of the LMP. Additionally, we developed an accurate model for the power and performance of DRAM type main memory by using data from Micron [19], [20] to simulate the DDR2 DRAM memory behavior and power consumption at a 333MHz bus clock frequency.

We start with a base processor architecture (henceforth referred to as **B**) that is similar to the processor and memory subsystem in one node of a modern superscalar cluster. Table I shows the configuration of our SimpleScalar processor. This is representative of modern processors, in particular RISC type processors like the PowerPC which are typically capable of issuing 4 instructions per cycle.

We model our processor power consumption in 130 nanometer technology. We obtain the 130 nanometer technology numbers by scaling [23] the power numbers already present in Wattch. We assumed a DVFS processor running in the frequency range of 1300–2000MHz with $V_{dd}$ between 1.07 – 1.5 Volts. The L1 cache latency is 1 cycle, L2 cache latency is 19 cycles, and memory latency is based on Micron, Inc. datasheets, and varies with CPU frequency [19], [20]. We

| | | | |
|---|---|---|---|
| -fetch:ifqsize | 4 | -cache:dl1 | 128:32:8:l |
| -fetch:mplat | 3 | -cache:dl1lat | 1 |
| -fetch:speed | 1 | -cache:dl2 | 1024:128:4:l |
| -bpred | bimod | -cache:dl2lat | 19 |
| -bpred:bimod | 2048 | -cache:il1 | 128:32:8:l |
| -decode:width | 4 | -cache:il1lat | 1 |
| -issue:width | 4 | -tlb:itlb | 16:4096:4:l |
| -issue:inorder | false | -tlb:dtlb | 32:4096:4:l |
| -issue:wrongpath | true | tlb:lat | 30 |
| -commit:width | 4 | -res:ialu | 3 |
| -ruu:size | 32 | -res:imult | 3 |
| -lsq:size | 8 | -res:memport | 1 |
| -res:fpalu | 3 | -res:fpmult | 3 |

TABLE I
WATTCH CONFIGURATION FOR BASE SYSTEM

modified Wattch to accommodate the additional functionality required by our architecture.

We use both PISA and Alpha configurations of SimpleScalar to enable the study of a larger set of codes. We use the Alpha configuration to evaluate the performance of SPEC benchmarks and PISA for the NAS benchmarks and sparse codes discussed in Section II. We ran all pre-compiled benchmarks from SPEC [24], for which we had a compiler and whose code ran on SimpleScalar. We also used SimPoints [25] to reduce the execution time of these simulations. The NAS benchmarks and SMV kernels were executed until completion, after being fast-forwarded past the initialization stage.

## V. EXPERIMENTAL RESULTS

In this section, we first discuss the LMP design trade-offs, and their impacts on performance, power and energy consumption for the SPEC 2000 and sparse benchmarks. We then consider the impact on performance and power of using our final LMP design on the SPEC 2000 benchmarks. We next consider these metrics in greater detail for the sparse codes described earlier in Section II. Unless otherwise stated, we report values of time, power and energy for each code relative to the base configuration **B** running at 2000MHz with a 512KB L2 cache. Values at the base configuration **B** are set to 1. Thus, for configurations with LMP (denoted as **B+LMP**), values that are less than 1 indicate relative improvements while values greater than 1 indicate degradations.

### A. LMP design trade-offs

Key trade-offs during the LMP design stage included the sizes of the *Regular L1* and *Bypass L1* caches, the L2 cache size, the predictor mechanism type and size, and the *Bypass L1* data replacement policy.

To address the *Bypass L1* data replacement policy trade-offs we created an "ideal" predictor. Our "ideal" predictor was a 2-level predictor, with 1024 16-bit hit/miss history entries in the first table. The second table consisted of 65536 bimodal counters. Instead of splitting the 32KB L1 data cache of the base configuration in two 16KB halves (half for the *Bypass L1* and half for the *Regular L1*), in our "ideal" LMP predictor we made both parts 32KB in size. As mentioned in Section III we considered three options for replaced data movement: "no victim", "only dirty victim" and "all dirty victim". The "all replaced victim" policy showed no performance degradations for SPEC 2000 benchmarks, as shown in Figure 2, and was chosen as the data replacement policy for our LMP design.

The "ideal" predictor described earlier has a significant hardware footprint, large power consumption, and it would add a significant delay to the memory subsystem. We considered the performance of simpler predictor designs, including the "bimodal" predictor consisting of a 2048-entry table of bimodal counter, and the "L2H8" predictor. Our experiments indicate that both designs consume less power than the "ideal" predictor (figures omitted for brevity). Their performance is compared against the "ideal" predictor in Figure 3. The "L2H8" predictor closely follows the performance of the "ideal" predictor at much lower design complexity and it is thus our choice for the LMP.

Changing the *Bypass L1* and *Regular L1* cache sizes from 32KB to 16KB, so that the total number of data cache bytes (*Bypass L1* + *Regular L1*) is equal to the base configuration, does not have a significant impact on the average performance of the SPEC 2000 benchmarks. The geometric mean of performance improvements, as shown in Figure 4, ranged between 2.5% to 4% for 32-1 (32KB regular, 1KB bypass) and 32-32 (32KB regular, 32KB bypass) cache configurations respectively, with the smaller cache configurations consuming less power and energy. For example, the 16-16 cache configuration consumes 7% less power and 5% less energy, than the 32-32 configuration, relative to a base configuration (figure omitted for space).

The observed performance fluctuations for the **mgrid**, **crafty** and **gzip** benchmarks were investigated further. The **crafty** and **gzip** benchmarks are cache bound codes, and decreasing the amount of *Regular L1* available resulted in more L1 cache misses. The SPEC **mgrid** code follows the same algorithm as the NAS MG benchmark, which is investigated together with our sparse codes in the following paragraphs.

The change in the LMP cache configuration has a varied impact on performance of our selected sparse codes. Results in Figure 5 (top subfigure) show that NAS MG is very sensitive to the cache size of the *Bypass L1* cache. As the size decreases from 32KB to 16KB the performance improvement reduces from 18.3% to 13.1%. Further reduction of *Bypass L1* cache size to 4KB and 1KB results in significantly smaller performance gains. The CG and SMV kernels are not as affected by the *Bypass L1* cache size changes (less than 2%). The average power consumption for the sparse codes is shown in Figure 5 (bottom subfigure). The 32-32 cache configuration consumes nearly 5% more power on average than the 16-16 configuration, and 2% more energy on average, thus making the 16-16 configuration a good compromise for both SPEC and sparse applications (figures omitted to save space).

Figure 6 shows the impacts of our final LMP configuration

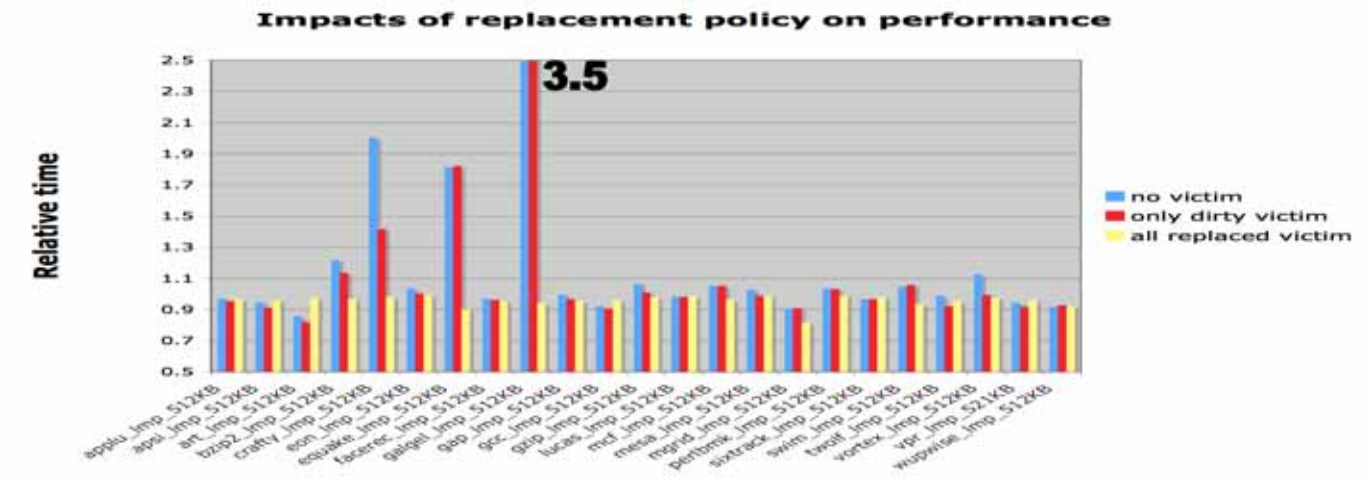


Fig. 2. Impact of LMP bypass cache replacement policy on SPEC2000 performance with "Ideal" LMP. "No victim" (left bar), "Only dirty victim" (middle) and "All replaced victim" (right). Values are relative to base configuration **B** at 2000MHz set to 1.

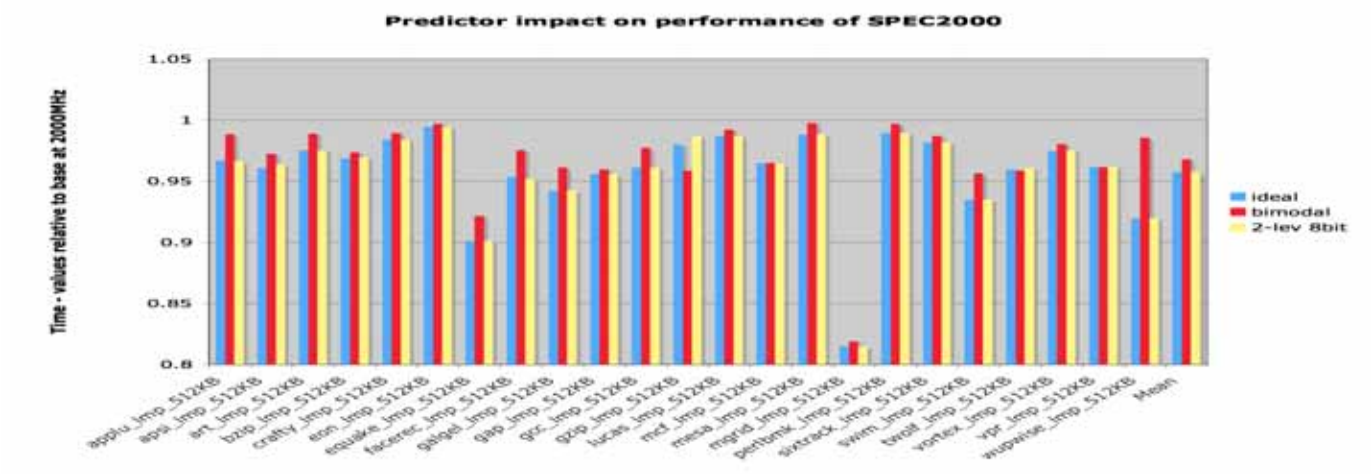


Fig. 3. Impact of "ideal" (left bar), "bimodal" (middle bar) and "2 level 8 bit (L2H8)"(right bar) predictors on execution time of SPEC 2000 benchmark suite at 2000MHz. Values are relative to base configuration **B** at 2000MHz set to 1.

("L2H8" predictor, 16-16 cache configuration, "all replaced victim" replacement policy) on performance (left), power (middle) and energy (right) consumption of SPEC 2000 benchmarks. Relative values are shown for (**B + LMP**) compared to base **B** at 2000MHz. The addition of the LMP increases the performance by an average of 2.9%. *Mgrid*, which is representative our sparse matrix application domain, benefits at significantly higher level of 12%.

The gains in performance come at the expense of increased hardware cost and thus system power. Observe that adding the LMP results in a 5% increase in energy and 7% increase in average system power. These increases are the result of more frequent memory accesses, the addition of the 16KB *Bypass L1* cache, prediction hardware, concurrent accesses to both *Bypass L1* and *Regular L1* caches, and concurrent accesses to L2 cache tag arrays during early memory reads.

Figure 7 shows the impact of using LMP on performance, power and energy for the NAS CG, MG codes and SMV-U, SMV-O. Observe that SMV-U benefits most with a 16.7% reduction in time, with SMV-O a close second with 15.6%. These improvements in execution time are primarily from decreases in the average load latencies (labeled as LSQ lat.) by 14.2%. On average, the performance of these four codes improves by 14%. However, power increases substantially by 13.6% on average, while the energy decreases slightly by an average of 2.3%, due to shortened running time but higher power consumption.

### *B. Co-optimizing performance, power and energy for scientific computing*

We now consider in detail how our final LMP can be used to improve the performance of sparse scientific codes, and how power can be reduced by using dynamic voltage and frequency scaling (DVFS) [4] with the LMP. Use of DVFS with the base

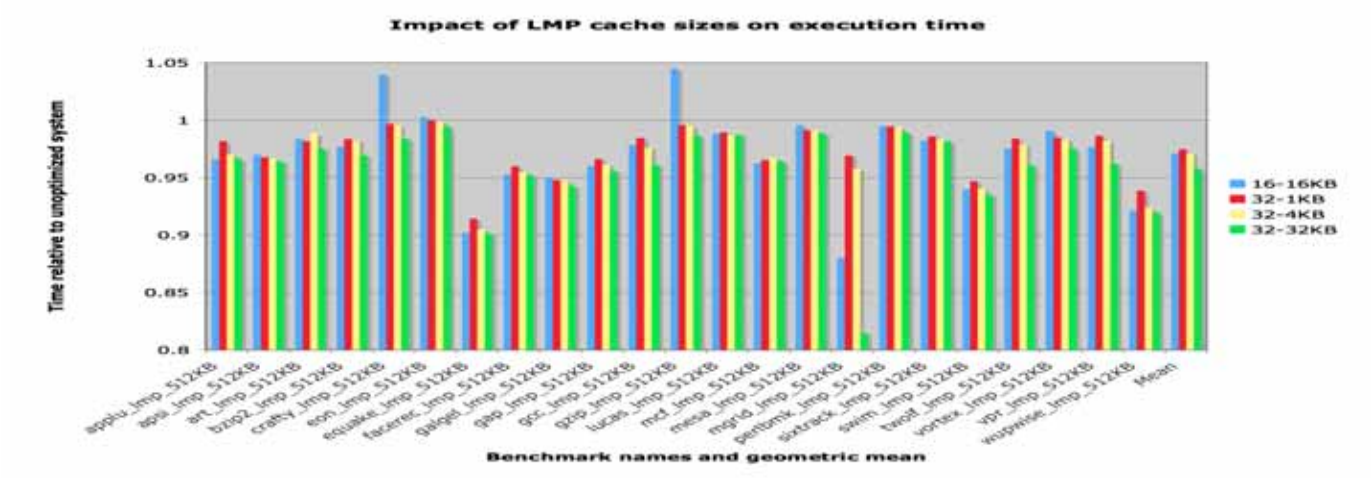


Fig. 4. Impact of the Regular L1 and Bypass L1 data cache sizes on execution of spec2000 benchmarks. 16-16KB means 16KB *Regular L1* and 16KB *Bypass L1*. Values are relative to unoptimized base configuration **B** at 2000MHz set to 1.

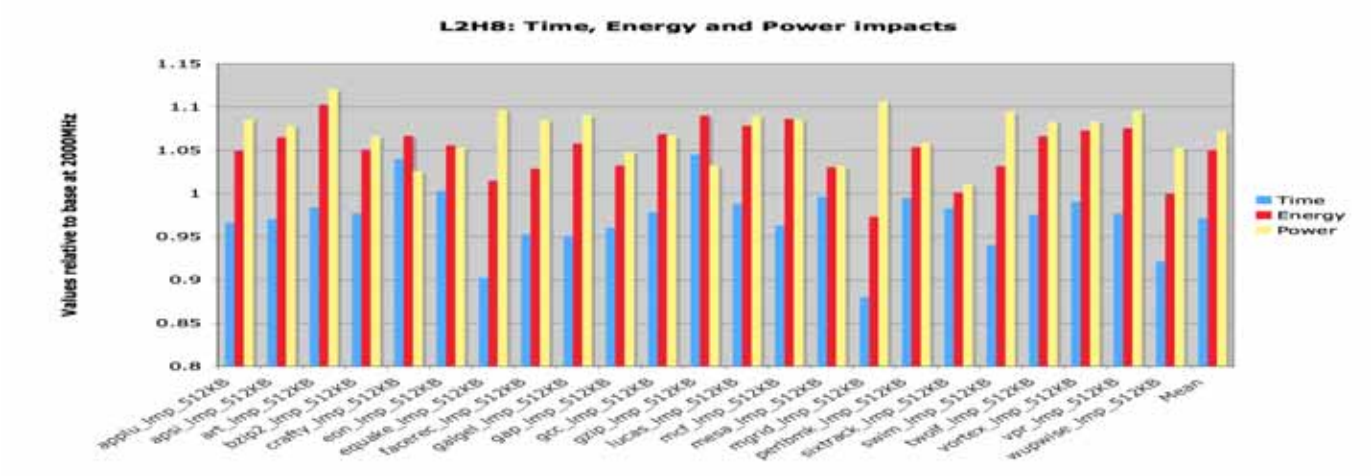


Fig. 6. Impact of our final LMP configuration on execution time, average power and energy consumption of SPEC 2000 benchmark suite at 2000MHz. Values are relative to base configuration **B** at 2000MHz set to 1.

architecture alone results in reduced performance at significant system power and energy savings. The latter can be used to potentially offset the power and energy increases when the LMP is added while retaining its performance benefits.

In Figure 8, we show relative values for time, power and energy for our four sparse benchmarks. The LMP offsets the performance degradation from frequency scaling down to the extent that even at 1600MHz show performance improvements. Corresponding power and energy savings are in excess of 20%. At 1400MHz performance improvements for SMV-U and SMV-O are in the range 5 to 10% with power reductions over 35% and energy reductions of over 40%. We summarize these power performance tradeoffs in Figure 9 indicating the mean relative values of time, power and energy for the base configuration **B** with DVFS and the enhanced configuration with LMP with DVFS (denoted by **B+LMP**) .

These plots clearly indicate that without the LMP, it is not possible to save power without degrading performance. Observe that **B+LMP** configuration at 1800MHz reflects nearly equal improvements in performance and power for over 15% improvements in energy. Without the LMP, performance degrades by nearly 7% at equivalent energy levels.

## VI. Related research

The importance of reducing memory access latencies is reflected in a rich set of earlier results towards faster loads [26]–[29].

Tyson [26] et al. considered techniques for improving effective cache hit rates by using static and dynamic techniques to determine whether data should be cached or not, thus reducing cache pollution. This work is similar to our approach in the sense that they use prediction mechanisms to avoid caching data that is not being reused. A key difference is that we use the prediction technology to lower the effective load latency

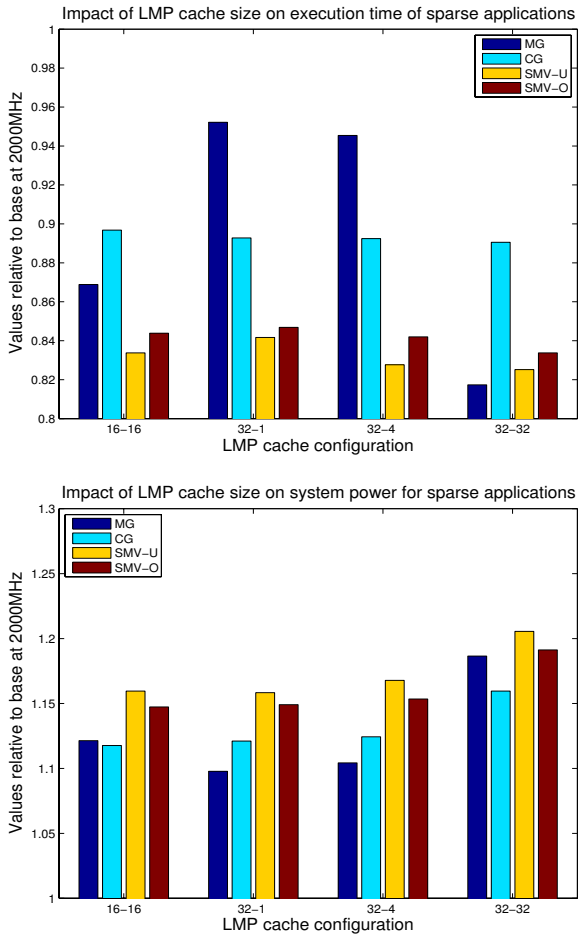


Fig. 5. Impact of Regular L1 and Bypass L1 cache size on performance (top subplot) and power (bottom subplot) of sparse benchmarks. X axis labels can be decoded in the following way: 16-16 means 16KB *Regular L1* and 16KB *Bypass L1*. Values are relative to base configuration **B** at 2000MHz set to 1.

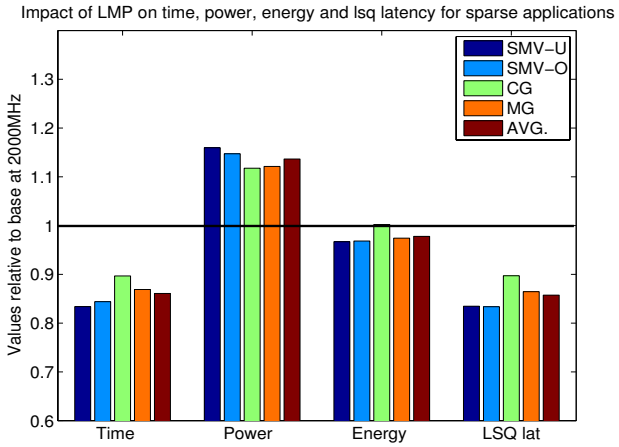


Fig. 7. Impact of LMP on execution time (1st bar group), power (2nd bar group), energy (3rd bar group) and on average load latency (4th bar group) for SMV-O, SMV-U, MG, and CG. Results are shown relative to values observed for each code at the base configuration **B** at 2000MHz set to 1.

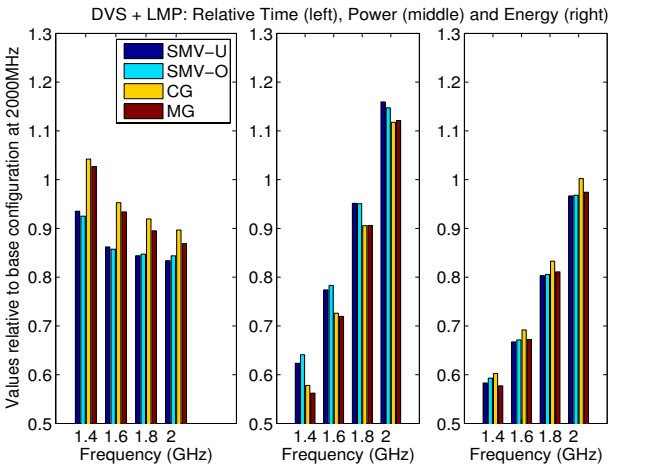


Fig. 8. Impact of LMP and DVFS on execution time (left subplot), power (middle subplot) and energy (right subplot) of SMV-U (1st bar), SMV-O (2nd bar), CG (3rd bar) and MG (4th bar) sparse benchmarks. Results are shown relative to the base configuration **B** at 2000MHz set to 1.

by bypassing the cache, however the bypassed data can stay resident in the *Bypass L1* cache for future reuse.

Memik, et al. [27] propose schemes to predict quickly if data at a given address is already stored in cache or not. Thus, they provide a faster but possibly less accurate look-up mechanism at every cache level. If the data is determined to not exist at a particular level a full access is avoided at that level. Thus they can reduce latencies by bypassing some cache levels. Our approach is different in that it is not data address centric, but rather instruction centric to predict a complete bypass of the cache hierarchy based on past hit/miss history.

Yehia, et al. [28] present a scheme which is similar in the sense that they use branch prediction for load instructions. However they focus on pairs of load instructions representing indirect memory references. Upon detection of such a pair and a miss prediction, they replace (through hardware) the second load by a new "load squared" instruction in which indirect address resolution is performed by memory-side logic. Such an approach may not be effective for the sparse scientific codes of interest to us. This is primarily because such codes avoid indirect memory references and most loads are performed to consecutive memory locations as discussed in Section I.

Our work is also peripherally related to the following. Software cache bypassing schemes were discussed by Chi [29]. Energy savings for scientific applications were considered by Choi, et al. [4] and Freeh, et al. [30]. Additionally, prefetching techniques were discussed by Lin, et al. [31]. Effects of prefetchers on performance and power of sparse applications were investigated by authors in [12].

## VII. Conclusions

We have developed a Load Miss Predictor (LMP) hardware addition which is effective at reducing memory access laten-

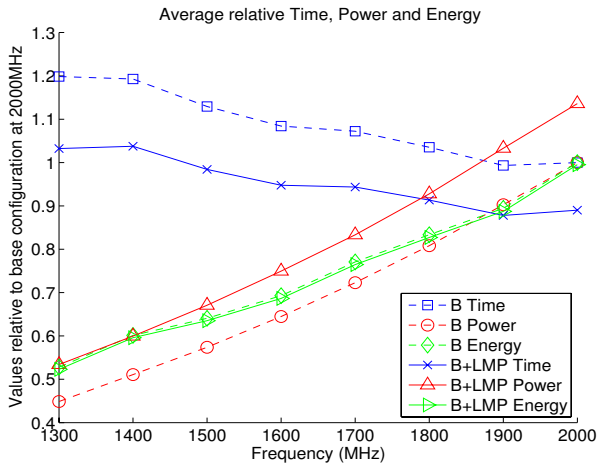


Fig. 9. Relative average time, power and energy, using DVFS with (solid lines) and without (dashed lines) LMP for 1300MHz to 2000MHz The average is taken over the results for SMV-U, SMV-O, MG and CG). Results are shown relative to the base configuration **B** at 2000MHz.

cies for sparse applications which have limited data locality and reuse. On average, execution time is reduced by 14% when this feature is utilized with a 13.6% increase in system power. A fraction of the improvements in time can be traded off for substantial power and energy savings by using DVFS. For example, on decreasing frequency to 1800MHz from 2000MHz system power is reduced by approximately 7.3% and energy is reduced by 17.3%, while maintaining 8.7% improvements in time. These results are promising and they indicate the potential for power-aware design optimizations suitable for high performance scientific computing. A natural extension of this work would be to evaluate the impact of combining our LMP with other static and dynamic schemes [26], [27].

## REFERENCES


[1] D. Keyes. Terascale Optimal PDE Simulations (TOPS) Center. http://tops-scidac.org/, 2004.

[2] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*, pages 127–136. Springer-Verlag, 2001.

[3] D. A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004.

[4] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10004. IEEE Computer Society, 2004.

[5] Michael T. Heath. *Scientific Computing:An Introductory Survey, Second Edition*. McGraw Hill, 2002.

[6] J. Dongarra. Top500 list and the LINPACK benchmark. http://www.top500.org/lists/linpack.php.

[7] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project, 2000.

[8] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IMB Journal of Research and Development*, 41(6):711–72, 1997.

[9] P. Gelsinger. GigaScale integration for teraops performance – challenges, opportunities, and new frontiers, 2004. DAC 2004 Keynote. Talk and slides at http://www.dac.com.

[10] M. Huang, J. Renau, Seung-Moon Yoo, , and J. Torrellas. L1 data cache decomposition for energy efficiency. In *International Symposium on Low Power Electronics and Design (ISLPED)*, August 2001.

[11] U. Ko, P. Balsara, and A.K. Nanda. Power and performance optimization for on-chip multi level cache hierarchies in microprocessors. *IEEE Transactions on VLSI Systems*, pages 299–308, 1998.

[12] K. Malkowski, I. Lee, P. Raghavan, and M.J. Irwin. Conjugate gradient sparse solvers: Performance-power characteri stics. In *Proceedings of the 20th IEEE International Parallel and Distributed Symposium, IPDPS'06, Second High-Performance, Power-Aware Computing Workshop*, April 2006.

[13] B. Fryxell, K. Olson, P. Ricker, F. Timmes, M. Zingale, D. Lamb, P. MacNeice, R. Rosner, J. Truran, and H. Tufo. FLASH: An adaptive-mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophysical J. Supplement*, 131:273–334, 2000. (see http://www.asci.uchicago.edu).

[14] J. W. Larson, B. Norris, E. T. Ong, and et. al. Components, the Common Component Architecture, and the climate/weather/ocean community. In *84th American Meteorological Society Annual Meeting*. American Meteorological Society, 2004.

[15] T.J. Chung. *Computational fluid dynamics*. Cambridge University Press, 2002.

[16] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. NAS parallel benchmark results. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 386–393, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[17] J. A. George and J. W-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1981.

[18] C. Kim, D. Burger, and S.W. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. *Micro, IEEE*, 23:99 – 107.

[19] Micron Inc. Micron 256Mb: x4, x8, x16 DDR2 SDRAM data sheet MT47H16M16BG-37E, 2004. http://www.micron.com/products/dram/ddr2sdram/part.aspx?part=MT47H16M16% BG-37E.

[20] Micron Inc. Micron Technical Note TN-47-04 Calculating Memory System Power for DDR2, 2004. http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf.

[21] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[22] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94. ACM Press, 2000.

[23] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[24] F. Bower. Notes on running spec2000 benchmarks with simplescalar. http://arch.cs.duke.edu/spec2000.html, 2005.

[25] B. Calder. Simpoint. http://www-cse.ucsd.edu/~calder/simpoint/, 2006.

[26] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 93–103. IEEE Computer Society Press, 1995.

[27] G. Memik, G. Reinman, and W.H. Mangione-Smith. Just say no: benefits of early cache miss determination. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*.

[28] S. Yehia, J.-F. Collard, and O. Temam. Load squared: adding logic close to memory to reduce the latency of indirect loads with high miss ratios. *SIGARCH Comput. Archit. News*, 33(3):17–24, 2005.

[29] Chi-Hung Chi and Henry Dietz. Improving cache performance by selective cache bypass. In *System Sciences, 1989. Vol.I: Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, volume 1, pages 277–285, Jan 1989.

[30] V. W. Freeh, F. Pan, N. Kappiah, D. K. Lowenthal, and R. Springer. Exploring the energy-time tradeoff in mpi programs on a power-scalable cluster. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 4.1, 2005.

[31] W.-F. Lin, S. K. Reinhardt, and D. Burger. Designing a modern memory hierarchy with hardware prefetching. *IEEE Trans. Comput.*, 50(11):1202–1218, 2001.