IBM Research Report

Exploiting eDRAM Bandwidth with Data Prefetching: Simulation and Measurements

Valentina Salapura, José R. Brunheroto, Fernando Redígolo
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
Exploiting eDRAM bandwidth with data prefetching: simulation and measurements

Valentina Salapura, José R. Brunheroto, Fernando Redígolo

IBM Thomas J. Watson Research Center
Yorktown Heights, NY

{salapura,brunhe,fredigo}@us.ibm.com

Abstract

Future system design will exploit eDRAM both to provide greater on-chip cache capacity, and to address manufacturability concerns surfacing with recent technology generations. While eDRAM has excellent bandwidth and density, it offers comparatively slower access times.

Data prefetching offers an attractive solution to reduce access to a shared, large capacity eDRAM by reducing the average latency of accesses, and by allowing us to exploit the large bandwidth – in the form of wide data accesses – by making efficient use of accessed data.

In this work, we present the design space exploration which helped determine the design of the prefetch data cache in the Blue Gene/L supercomputer. We also evaluate simulation accuracy and the accuracy of the decisions made in the design process by comparing our simulation results with measurement results on actual Blue Gene systems. These experiments provide valuable validation for the decision process and our modeling environment, as they allow one to understand how well they track actual hardware behavior. Actual execution time measurements also include any system effects not modeled in our performance analysis environment, and confirm our selection of simulation parameters included in the model.

1 Introduction

Future microprocessor designs will require new design trade-offs to address new constraints on architectures. The increasing compute power available per chip from the use of chip multiprocessors is not matched by a commensurate increase in signal I/O bandwidth to satisfy the inherent memory bandwidth requirements, leading to a potentially unbalanced and inefficient design.

At the same time, the use of SRAM memories as on-chip memories to provide a significant reduction in bandwidth requirements is limited by its comparatively low density, and power dissipation. SRAM memories are also suffering from manufacturability constraints limiting future access speeds due to device variation [7, 11].

A promising solution to these multiple constraints is the adoption of embedded DRAM techniques for high-capacity high-density on-chip caches. Embedded DRAM uses logic fabrication technology to build the familiar 1T DRAM cell in a logic process, offering a significant increase in capacity per given unit area. While eDRAM offers attractive density and capacity per unit area, it has higher latency compared to SRAM-based on-chip solutions.

By using eDRAM in conjunction with chip-multiprocessor solutions, it is possible to deliver increased performance with reduced memory bandwidth requirements, and offer attractive system solutions transcending the constraints of current technology. Integrating large capacity eDRAM caches on chip makes high bandwidth access to high capacity on-chip storage a reality by offering both wide data paths, and higher on-chip transfer speeds. To deliver on the promise of this new memory hierarchy paradigm, we turned to prefetching to decouple application access latency from the technology, and use the available bandwidth for latency hiding.

With the introduction of the BlueGene systems, we provide the first system delivering on the promise of on-chip eDRAM for higher performance, lower cost, and higher reliability solutions [3]. To avoid the area cost and constraints imposed on SRAM in future technologies, we turn instead to an efficient prefetch cache architecture. In the BlueGene/L compute chip [5, 17, 18], instead of using a standard L2 SRAM-based cache, a small private prefetch cache is integrated in the memory hierarchy between a first level private 32KB data cache, and a 3rd level on-chip 4MB eDRAM cache shared between two processor cores on a chip. The work described here was instrumental in deciding the optimal prefetch architecture for BlueGene systems.

Unlike many previous studies, which have focused on application traces only, and may neglect the impact of the interaction between application software and operating sys-
tems, we study prefetch behavior for a set of compute intensive workloads using full system simulation. Brunheroto et al. [6] presents an initial evaluation of prefetch algorithms based on this simulation environment. Specifically, we compare an off-the-shelf Linux system modified to execute on BlueGene/L, and the optimized Blrts microkernel tailored specifically for the execution of BlueGene workloads.

By comparing our simulation results with measurement results on actual Blue Gene systems, we evaluate simulation accuracy and the decisions made in the design process. While prefetching is not a complete solution to memory latency issues, we believe that prefetching combined with high density on-chip eDRAM-based caches is an important aspect of a successful solution.

The contributions of this paper are: (1) analysis of prefetching potential in supercomputing applications, (2) an extensive simulation-based design space exploration of prefetch approaches for an on-chip eDRAM cache, (3) analysis of operating-system impact on prefetch effectiveness, and (4) validation of simulation results based on hardware measurements on a BlueGene/L system.

While we do not introduce any new prefetching scheme (that was not our intention), several ideas presented in this paper are novel: this is the first multiprocessor architecture implemented with shared eDRAM-based on-chip caches, and this is the first work to embed small private prefetch caches in the memory hierarchy to hide latency to eDRAM in a multiprocessor environment.

This paper is organized as follows: Section 2 describes the memory subsystem and prefetch cache architecture. Section 3 presents the simulation environment, our workloads and methodology. Section 4 describes simulation results for the design space leading to the selection of desirable prefetch algorithms. Section 5 validates the simulation results with hardware measurements obtained on a BlueGene/L system implementing the described architecture. We discuss related work in section 6, and draw our conclusions in section 7.

2 BlueGene memory subsystem architecture

The Blue Gene/L computer is a scalable system consisting of up to 65,536 nodes using IBM CMOS CU-11 technology. Each node is built around a single System-on-Chip based on the PowerPC 440 processor core and 9 or 18 SDRAM-DDR memory chips. The Blue Gene/L compute node contains two processors, each with a SIMD floating point unit, 32kB private L1 cache with 32B cache line size, private prefetching L2 cache with 128B prefetch buffer line size, and a shared 4MB L3 on-chip eDRAM cache with a 128B cache line size, as illustrated in Figure 1.

Private prefetch L2 caches provide decoupling of peak bandwidth requirements by buffering wide lines retrieved from the shared L3 cache. The L2 cache captures locality of reference by storing wide L3 cache lines, and satisfying multiple narrower L1 requests from the wide line buffers. This reduces latencies for L1 requests, and traffic for the L3 cache allowing efficient sharing of the L3 cache. The eDRAM consists of two single ported banks which can be accessed independently.

In our configuration, each core uses a prefetch cache, which serves multiple purposes:

- It serves to store fetched lines and capture locality from a narrow L1 fetch (32 bytes) to a wide L3 cache line (128 bytes).
- It detects data streaming behavior.
- It stores prefetched lines until they are referenced by the processor, or displaced by the replacement algorithm.

The memory subsystem of the BlueGene/L chip determined the sizing of the prefetch cache line. The prefetch cache implements a line size corresponding to the eDRAM cache line size at the next hierarchy level, capturing reference locality with an eDRAM line without requiring multiple accesses. This decision is key to reducing the number of accesses and allowing efficient sharing of the eDRAM-based on-chip L3 cache between two cores [14], as the 128B eDRAM cache line is four times the width of the 32B L1 data line. This allows us to capture more spatial reference context and to exploit the high bandwidth from the embedded DRAM-based next level cache.

We implement a small private prefetch cache between 32kB L1 caches and shared 4MB L3 eDRAM between two processors. The prefetch cache’s size is only 2kB per processor. It reduces execution time across a wide range of workloads by 10% on average. Comparing our memory system design with an L1 cache, L2 prefetch cache and a large eDRAM-based L3 cache to a traditional memory system design with an L1 cache and an SRAM-based L2 cache in terms of complexity, area, and power/performance, our design offers several advantages.

To fairly compare the two approaches, we could:

- use eDRAM as L2, with no prefetch cache: while keeping the cache size the same, it has longer la-
tency. These results correspond to our "no prefetching" scheme.

- keep L1 SRAM and L3 eDRAM caches, and add a standard L2 SRAM-based cache, having the same size as our prefetch cache: with 2kB size, our prefetch cache outperforms any traditional cache of the same size.

- remove eDRAM L3, use standard SRAM based L2: to keep chip area the same, 4MB eDRAM can be replaced by a 1MB SRAM shared between 2 processors. In addition, the eDRAM based solution requires about 4 time less power compared to 1/4 sized SRAM based solution. Power considerations are especially important for embedded systems, and systems of significant scale such as BlueGene.

By keeping the prefetched data out of the L1 data cache, we avoid pollution of the L1 cache from prematurely fetching data into the cache and thereby potentially displacing data still needed by the processor. This is particularly important for carefully tuned algorithms that size their working sets to efficiently exploit the memory subsystem. Excessive prefetching in this environment can interfere with the delicate tuning performed to achieve peak performance.

Reusing an unmodified PowerPC core available as a hard macro was a project requirement. Any change to the core would incur a significant non-recurring engineering charge and would delay introduction of the BlueGene system, representing a net performance degradation against competing systems at the system availability date. Thus, using an unmodified first level PowerPC cache architecture extended with a second-level private application-tuned prefetch cache serves to leverage existing PowerPC cores and facilitate design reuse. This is especially advantageous in an environment where the L1 cache is tightly integrated with the load-store unit, and any changes to the L1 cache architecture would require changing the processor core. In addition, the entire core design has to be re-timed and re-validated due to the strong connectivity between the processor components. By employing an external prefetch cache, the tuned PowerPC 440 hard macro can be placed unmodified to achieve peak clock frequency without design rework, and low latency memory access.

In addition, the approach of having an external prefetch cache is advantageous because of contention for the L1 data cache port if the single data cache port is shared between the load-store unit and the prefetch unit. To overcome the contention problem, a two-port cache can be implemented, and but this implementation doubles the array.

### 2.1 Prefetch cache architecture

Figure 2 illustrates the architecture of the prefetch cache unit explored in this work and employed in the BlueGene/L compute chip. The prefetch cache consists of several components:

- Line buffers provide storage for demand-fetched and prefetch cache lines from eDRAM;
- A prefetch engine initializes prefetches, predicts the prefetch address and selects which line buffer to replace; and
- A stream detector unit detects reference patterns corresponding to data streams.

On each L1 data cache miss, the prefetch cache directory is checked. If the requested data is already available in the prefetch cache, it is forwarded to the L1 data cache. We have explored a number of approaches to detect streams. An initial set of experiments uses N-deep history queue for storing N prefetch cache address tags [15]. We refer to this history queue as the stream detection buffer. When the processor requests data which miss in the L1 data cache, the prefetching unit records the corresponding L2 address in the stream detection buffer. If the requested address matches an L2 address already recorded in the stream detection buffers, a stream is established.

When a stream is established, the first subsequent access to data resident in the line buffer triggers a prefetch request to be issued. In a prefetch request, one prefetch line (corresponding in size to four L1 cache lines) is fetched from the L3 cache and stored in the prefetch cache. For an L2 cache request which misses in the prefetch cache, the requested L3 cache line is fetched from the L3, and only the portion corresponding to the requested L1 cache line is forwarded to the L1 cache without storing the fetched L3 data line in a line buffer.

The alternative prefetching approach we have explored does not use stream detection buffers, but instead, issues a fetch request for each new L2 data cache request which is not satisfied in the prefetch cache, and also a prefetch request for the next line. Thus, this approach automatically starts prefetching a data stream based on only one request. We refer to this approach as optimistic prefetch stream detection.

The advantage of this approach is the ability to use the prefetch address tags associated with each line buffer also
as the address tracking method for identifying streams. This is advantageous because it reduces the number of state bits, which have to be maintained. As the optimistic prefetch approach uses a more aggressive prefetching strategy, it issues a higher number of prefetches to the L3.

The line buffers are managed as a fully associative cache. Once prefetched, the lines reside in the prefetch cache as long as no other request evicts or invalidates the entry. In the described architecture, each established stream uses effectively at least two entries in the prefetch cache: one entry to serve requests to the current L3 line, and another one to store prefetched data from the L3-cache.

Once a data stream has been detected, the condition to sustain the stream is that the prefetched line corresponding to the next address line has not been displaced by the time it is requested by the L1 cache. If new streams are detected and referenced more frequently, they will eventually displace older streams, which are no longer referenced.

We will explore tradeoffs in stream detection architecture, such as maintaining separate stream detection capabilities (stream detector), or using the line buffer tags for detecting streams (optimistic prefetching). We will also evaluate the impact of detection logic depth, number of line buffers, and impact of operating system on prefetch strategy in the next sections.

3 Methodology

As previously mentioned, we use full system simulation and two different operating systems to explore the effectiveness of stream prefetching for supercomputer applications, and the impact of the operating environment. Our system simulator is BGLsim [1], a full system simulator for the BlueGene/L system based on the Mambo PowerPC simulator [4]. BGLsim is an architecturally accurate simulator at the instruction-set level. BGLsim exposes all architected features of the hardware, including processors, floating-point units, caches, memory, interconnection, and other supporting devices. The simulator runs unmodified system and user software, as used on actual BlueGene hardware. An architectural simulation at the instruction-set level is several orders of magnitude faster than VHDL simulation at the logic design level, allowing exploring a large design space with real applications.

Our simulator allows running a range of complete and unmodified code, from simple self-contained executables to full Linux images. The simulator includes interaction mechanisms for inspecting the entire internal machine state, providing execution cycle estimates, and allowing more flexible and more detailed instrumentation than what is possible with real hardware. We have modified the simulator to include tracing capabilities [13].

In our experiments to evaluate the effectiveness of the prefetch cache and to explore design space trade-offs, we have used a trace driven cache model. We use the L1 address miss sequence (containing both application and operating system references) for a variety of numerically intensive applications running under the BlueGene/L compute node kernel (a single threaded OS) and Linux. To improve the simulation speed, we use a separate BlueGene/L cache model for the prefetch cache and the L3 cache level to evaluate prefetch cache configurations.

We have opted for a multi-module simulation environment which comprises of two modules, one the full system simulator with pseudo cycle accuracy that takes binary code as input, and the other based on traces which is much faster since it does not implement all the details. The second module is used to do the coarse design space exploration, yielding design parameters that we then evaluate in full detail using the first model. Since the prefetch cache is private to each processor, multiple processors do not affect prefetch hit rate in any way. There is a benefit to multi-processor designs due to the reduction in bandwidth coming out of the prefetch cache relative to the L1 cache, which is shown in the bandwidth reduction study.

The metrics we use to measure the performance improvement achieved by using the prefetch cache with various properties varied are prefetch cache hit rate, prefetch cache miss rate, and the execution time (as predicted by the BlueGene/L Pseudo Accurate Timing Model). The prefetch hit rate is the fraction of L1 data cache misses that hit in the prefetch cache divided by the total number of requests to the prefetch cache. A perfect prefetching scheme would hide the latency of the L3 cache, i.e., all memory accesses would be satisfied in the L2 prefetch cache and would have a prefetch cache hit rate of 100%.

The latency to the L3 cache is hidden if the line address corresponding to that address has already been prefetched and the data is already loaded into the prefetch cache. In our simulation model, we assume that the prefetch cache request to the L3 is satisfied within a constant L3 cache latency, while in the actual hardware the L3 cache latency varies depending on several factors (e.g. page already open, number of pending load requests).

In our experiments, we use a set of applications from the publicly available NAS [2] and Splash-2 [20] benchmark suites. These are well known benchmarks containing shared memory applications that have driven much research into shared memory architectures and cache-coherence protocols. We have opted to use these publicly available applications, as they are good representatives for a wide range of scientific applications. We concentrated our efforts on scientific computing intensive applications, as these were the target workloads for our BlueGene system.

Here, we report all of the NAS class S benchmarks, and the Splash-2 kernel applications (LU, Radix, FFT), and the ocean application. For Splash-2, we have used default settings resulting in a small footprint size.

For each of the benchmarks reported, we have executed a full application run, and we have collected the entire L1 miss sequence to determine prefetch opportunities. Tables 1 and 2 show the benchmarks used, the number of instructions executed during the run, and the number of L1 misses
We later explore a set of results, which combine the most realistic (under the design constraints of area, power, design complexity and so forth) and best performing parameters in each factor by varying one parameter at a time and setting the other parameters to a sufficiently large configuration.

We first vary the size of the stream detector buffers to determine the minimum size which yields a good prefetch cache hit rate. In order to determine how the sizing of the line buffers influences the prefetch cache hit rate, we vary the number of line buffers, and we also explore in detail the impact of using various replacement policies for the line buffers. In addition, we evaluate the impact of supporting bi-directional stream detection, which requires more complex hardware implementation, as opposed to prefetching only ascending order. We also analyze the impact of the operating system used.

In an initial set of experiments, we have tried to isolate each factor by varying one parameter at a time and setting the other parameters to a sufficiently large configuration. We later explore a set of results, which combine the most realistic (under the design constraints of area, power, design complexity and so forth) and best performing parameters in combination.

### 4.1 Stream Detector Buffers

As previously mentioned, we have explored an approach to detect access streams using an N-deep history queue for storing N prefetch cache address tags. In this approach, a new stream is started only if an L2 cache request hits in the address history queue, requiring two requests to establish a data stream.

Figure 3 shows the behavior for a prefetch cache architecture with a stream detector mechanism with varying stream detector sizes ranging from 2 to 32 stream detection buffers to track address history, maintained in a FIFO organization, for both NAS and Splash-2 benchmarks.

Table 2 (a) shows that for the NAS benchmark using stream detector sizes above 16 does not significantly improve the hit rate, except for SP. For SP, adding more stream detection buffers continues to increase prefetch hit rate, as more of the distinct data streams referenced by SP can be kept in the prefetch cache. For most applications, though, 16 stream detection buffers are sufficient to detect all data streams in the application. For this simulation, we use a prefetch cache large enough to not limit the number of streams which can be established and maintained.

For the Splash-2 benchmark, as shown in Figure 3 (b) the hit rate is not significantly improved for stream detectors having a history queue deeper than 8. For some applications, like Radix and LU, the spacial locality of data is very high, so that the size of the stream detector does not change the prefetch cache hit rate. Based on this, we use a stream detector with 16 entries.

### 4.2 Prefetch Cache Size

In order to determine the optimal number of prefetch cache line buffers, we have varied their number from 7 to 31 while keeping the stream detector size fixed at 16. We change the number of line buffers in multiples of eight. One line buffer is used for buffering the data what are returned from L3 from demand fetches that are not buffered in L2 (e.g., L2 requests without an established stream), hence the odd number of line buffers available for stream prefetch. The results are illustrated in figure 4.

We observed that the effect of increasing the prefetch cache size is not linear regarding the hit rate. Choosing a prefetch cache size of 7 lines is clearly a not exploiting the full prefetch potential, and a cache size of 15 lines is a significantly better performing design point. For NAS benchmarks, selecting 23 line buffers increases the hit rate across all benchmarks on average by 2.7%, with the biggest benefit for the SP benchmark with a hit rate increase of 7%. A configuration with 31 line buffers (which doubles the area compared to 15 line buffers) only increases the hit rate for the SP benchmark.

For Splash-2 benchmarks, only the LU application benefits from increasing the number of line buffers to more than 15. The LU and SP applications have more streams, thus benefiting from a higher number of line buffers. As a result of this analysis, we chose a configuration with 15 line buffers which offers an attractive cost/performance tradeoff, given that further increases in line buffers offer only modest incremental performance gains at significant area cost.
Figure 3. Varying stream detector size across the NAS and Splash-2 benchmarks.

Figure 4. Varying the number of line buffers in the prefetch cache for NAS and Splash-2 benchmarks.

Figure 5. Prefetch cache miss rate for various line buffer replacement policies across NAS and Splash-2 benchmarks.
4.3 Prefetch Cache Replacement Policy

As previously described, the replacement policy captures reference behavior and helps to determine how streams are aged out of the prefetch cache to make room for new data lines. We have simulated a number of different replacement policies including the optimal replacement policy (one that requires future knowledge, therefore cannot be implemented in hardware) to show the theoretical upper bound for the stream detection.

We have explored and evaluated the following prefetch cache replacement policies:

- round-robin
- random
- least recently used (LRU)
- round-robin skipping most recently used (RRMRU)
- optimal (one that relies on future knowledge)

Whereas round-robin is simple to implement in hardware, this approach has a disadvantage that it can displace lines from the prefetch cache that have recently issued a prefetch request to the L3, for which the line is allocated, but the data is still in-flight from the L3 cache. To avoid this problem, we have explored a modification to round-robin where the three most recent requested lines are skipped (MRU=3).

Figure 5 presents the effect of varying the replacement policy on the miss rate for the NAS and Splash-2 benchmarks, respectively. Across all applications, we can see that all replacement policies are positioned between the optimal replacement policy – which gives the theoretical upper bound for line buffer replacement – and the random replacement policy. As expected, LRU is the best choice for majority of applications, but we did not choose it because of its complex hardware implementation. RRMRU (round-robin with skipping the three most recently used lines) is as good as or better than round-robin replacement policy. In addition, this replacement policy is as simple to implement in hardware as round-robin, requiring only addition of two latches per line buffer to record the MRU status for the last three requests. As a result of this analysis, we have implemented RRMRU replacement policy.

4.4 Support for Bidirectional Streams

All results so far assume streams are only accessed in ascending address order. We have also explored whether bidirectional stream support (i.e., detecting and prefetching streams with positive and negative address strides) is beneficial for performance.

To implement bidirectional stream support, each line buffer stores an additional two bits to record the L1 cache line address of the first request. For the subsequent request to this prefetch cache line, the address of the request is compared to the saved data, and it is determined if the address is descending or ascending compared to the previous request. This information is stored in a stream direction bit associated with each line buffer. Based on this information, the next prefetch request is issued to access the ascending or descending address.

Figure 6 shows the effect of changing from an ascending stream detector to a bidirectional stream detector, using the RRMRU replacement algorithm.

One can observe that there is no significant benefit in using a bidirectional stream detector for these benchmarks, indicating that there are no significant negative stride accesses present in the applications of the NAS benchmarks. Also, scientific workloads in general do not show negative stride streams. Our model supporting bi-directional strides achieved minimal performance improvement, so we decided to remove negative stride support to simplify the prefetch engine.

4.5 Optimistic vs. Stream Detector Buffers

In order to evaluate the efficiency of the stream detection buffer, we have compared it against the optimistic prefetch approach, as described in 2.1. Figure 7(a) compares the miss rates for the optimistic and stream detector prefetch schemes. We observed that for some benchmarks (BT, FT and LU) the optimistic approach yields a lower miss rate, while for the other benchmarks both approaches present roughly the same miss rate.

To gain a better understanding of quantitative advantages of the stream detection buffer design, we also compare the execution times for the two prefetching approaches across the NAS benchmarks. The results are illustrated in Figure 7(b). We observed that the execution times for both approaches are remarkably similar across all the applications of the NAS benchmarks, with the largest difference in the execution times being 1.8%. The optimistic approach has
shorter execution times for the BT and FT applications, and stream detection buffer results in better execution time for the CG application.

As the optimistic prefetching approach uses a more aggressive prefetching strategy to increase the hit rate, we expected that the bandwidth requirements for the L3 will increase for the optimistic prefetch approach, as this approach issues a higher number of prefetches to the L3.

Figure 8 shows the normalized breakdown of L3 accesses for both prefetch approaches for the NAS benchmark suite. The choice of optimal prefetch algorithm depends on the workload. While some workloads (as exemplified by the FT benchmark) produce fewer accesses with the optimistic prefetcher, other workloads (as exemplified by the SP application) show a lower number of overall accesses with the stream detector.

The breakdown of L3 accesses into the categories for each approach gives more insight into this behavior. We classify the number of L3 accesses into two broad request categories, demand requests and prefetch requests. For the design with stream detection buffers, we classify demand fetches into two subcategories, a demand request, and a stream establishing demand request (i.e., a demand request hitting in the stream detection buffers and thereby causing a stream to be identified).

As is to be expected, prefetching with optimistic prefetching approach initiates a higher number of L3 prefetch accesses relative to the stream detection buffers. However, the number of demand accesses is smaller for the optimistic prefetching approach, resulting in a smaller total number of accesses.

For the cases where the optimistic prefetching achieves a lower number of accesses, the breakdown of the demand fetches for the stream detector shows the cause for the higher number of accesses using the more conservative stream detection logic: when a stream has not been detected, no buffer is available to store a wide L3 line for future accesses. As a result, two subsequent demand accesses, a first demand access to an L3 line, and a second, stream establishing demand access to the same L3 line, are performed before a stream is established. In comparison, the optimistic prefetching approach associates a stream with a demand buffer immediately and retains the entire L3 cache line for future accesses, thereby obviating the need for performing a second access to the same line.

4.6 Prefetch Cache Performance Characteristics

In order to evaluate the efficiency of the prefetch cache, we compare the two prefetch schemes – stream detector buffer and optimistic prefetching – with application perfor-
Figure 9. Normalized execution time for the optimistic prefetch cache, stream detector prefetch cache, for disabled prefetch, and for no prefetch cache across the NAS benchmarks.

Figure 10. Normalized breakdown of eDRAM accesses for the optimistic prefetch cache, stream detector prefetch cache, and disabled prefetch cache across the NAS benchmarks.

formance results obtained when prefetching is disabled.

Figure 9 compares the normalized execution time for the two prefetching approaches and with prefetch disabled. The prefetch cache reduces execution time for both prefetch methods by 12% on average. The biggest performance improvement due to the prefetch cache is achieved for the CG benchmark (22%), whereas for the LU benchmark the performance improvement is only 2%.

Simulation results also show a performance benefit when using prefetching, versus just exploiting multi-line buffers without prefetch engine. While multi-line buffers reduce execution time on average by about 10%, prefetching provides an additional 2%–5% performance improvement across all applications compared to an architecture with line buffers without prefetching.

The second important aspect is reducing the number of accesses to the eDRAM, to reduce contention for the eDRAM cache port by the two processor cores, the network interface and the memory controller. Figure 10 shows the normalized breakdown of eDRAM accesses for both prefetch approaches, and with prefetch disabled for the NAS benchmark suite. The prefetch cache reduces the number of eDRAM accesses significantly, on average by 60% across all NAS benchmarks. This is caused by the fact that for scientific application most references are streams, and thus buffering of the eDRAM data in wide 128B prefetch cache lines dramatically reduces the number of requests needed.

The two prefetch schemes show remarkably similar characteristics in terms of execution time and eDRAM accesses. Although the hit rate of the optimistic prefetching scheme is higher, the overall execution time obtained by running a pseudo-cycle accurate version of BGLsim (taking into account the latency to the L3 and the pending requests to the L3) is equivalent to the stream detector buffers scheme. As area requirements for implementing stream detectors is only a fraction of area required for line buffers (resulting in increased latch count of less than 3%), and show higher hit rate for several applications, we have implemented both prefetching methods.

4.7 Operating System Impact

Finally, we have explored the impact of using different operating systems on prefetch cache performance. We compare two basic models, representing a full-fledged multi-threaded UNIX operating system (using the Linux operating system), and a streamlined single-threaded kernel solution (based on the compute node kernel Blrts employed in BlueGene/L).

The BlueGene/L high performance runtime system’s kernel implements an identical virtual address space to physical address translation. This linear mapping ensures that an application’s access patterns in virtual address space are reflected in physical address space available to the memory subsystem. Thus, address mapping discontinuities does not interfere with the operation of the prefetch engine.

In comparison, a standard Linux kernel uses a 4k page size. As a result of establishing page translations in response to demand paging, the kernel will map the continuous virtual address space to discontinuous physical 4k pages as illustrated in Figure 11. At each page boundary, the prefetch engine continues to prefetch from the contiguous physical address which may not match the actual access pattern in virtual address space. Thus, streams have to be re-established and bandwidth and access efficiency is lost at every page transition. Figure 12 compares the impact of memory allocation policies in Linux and blrts on the prefetch cache hit rate.

Because the 64-entry TLB cannot contain the entire address space for memory and I/O devices of a Blue Gene
node with small pages, additional degradation is introduced when TLB entries must be reloaded. This is particularly expensive in an environment without hardware-managed TLBs where each TLB miss will cause a trap to the operating system. This has been mitigated somewhat in more recent versions of the Linux kernel with the introduction of large page support.

5 Hardware measurements

Based on the evaluations described here, the BlueGene/L system architecture implements a 15 entry prefetch cache with a choice of both prefetch algorithms to efficiently prefetch data from the high-bandwidth on-chip eDRAM cache.

To verify our simulation results, and to choose the appropriate configurations for future systems based on this design, we have performed extensive empirical performance analysis of applications. For comparison with the design

space exploration work reported during the concept phase, we report hardware measurement results for the public NAS benchmarks.

Figure 13 shows the normalized measured execution times for the two prefetch schemes and with L2 disabled for each NAS benchmark, and compares the measured execution times to simulated execution times. The hardware measurements confirm the trends shown by simulations of a significant improvement in performance due to the use of the prefetch cache. Both hardware results and simulation are normalized to itself (e.g., hardware results are normalized using hardware optimistic prefetch results, and simulation results are normalized using optimistic prefetch simulation results) to eliminate systematic deviation between simulator and hardware measurements. This confirms the relative accuracy of the model to select the optimal design point.

Figure 14 plots the ratio of simulated execution time over the corresponding measured hardware execution time. The simulation results are typically within 10 to 20% of final performance, including all system effects, operating system interaction, and so forth. We note that the simulation results are also conservative in projecting both the baseline performance, and even more so, in the modeled improvements.

The simulation error compares very favorably to the only other work published on correlating simulated results against actual hardware measurements for the FLASH system [9]. This confirms the quality of our simulation environment, and the decision to go with a full system simulator for the BlueGene system.

6 Related work

Fetching data from memory before the processor needs it has been widely employed and explored. The underlying idea is to overlap memory access time with computation, and thus to improve processor performance by re-
prefetching performs poorly when non-sequential memory access patterns are encountered.

Gschwind and Pietsch [10] prefetch into stream buffers under program control. In this approach, prefetch streams are identified by prefetch register FIFOs, and software can specify arbitrary stride.

The PowerPC architecture supports data stream prefetching into the L1 cache with appropriate data stream touch instructions. However, these approaches require significant investment by the programmer (or appropriate compiler support) to specify the streams.

Lee et al. [21] evaluate the performance of several prefetching cache architectures for multimedia applications.

Sequential prefetching techniques perform poorly for sequences of irregular access patterns, as in pointer chains, where the code follows a serial chain of loads. The approach described in [8] uses a pointer cache to assist prefetching for pointer load sequences.

In prior work, Gibson et al. [9] have evaluated the effectiveness of the FLASH simulation environment, and correlated hardware and simulation results.

Puzak et al. [16] discuss prefetching metrics, and analyze the potential for prefetching in SPECcpu and OLTP workloads. Here, we concentrate on workloads with established regular access patterns in compute intensive applications with regular memory access patterns.

7 Conclusion

This study presents an exhaustive analysis of design space options specifically for supercomputer class applications with full system simulation, and considering operating system impact by comparing different operating systems (a full-featured Linux and a custom-tailored single-threaded lightweight kernel specially designed for the BlueGene/L supercomputer). Unlike previous studies based on application-only traces, we have used full system simulation to get representative cache miss behavior including OS interaction, and used the full L1 miss sequences to explore the prefetch cache design space.

The simulation results guided the design space exploration that lead to the design of the BlueGene/L memory subsystem, and the efficient use of embedded DRAM caches in a PowerPC chip-multiprocessor. We have also validated the simulation environment against hardware measurement, and confirmed the accuracy and high quality of our performance evaluation environment.

In traditional architectures prefetch cache lines and L1 cache lines were of the same size, and careful prefetching was important so not to pollute prefetch lines. This equation has changed for the architectures where prefetch cache lines are significantly wider than the L1 cache lines, such as in the BlueGene/L architecture. Having high bandwidth between shared eDRAM cache and a prefetch cache capable of sustaining both processors’ memory requests elimi-
nates the L3 throughput bottleneck, and thus enables efficient data stream prefetching reducing the execution time.

We have obtained measurement results for a range of our design study options from Blue Gene hardware, and analyzed the decisions and framework used to obtain our decisions. We show good correlation of models with actual hardware measurements, and that our modeling efforts to ensure conservative modeling of proposed extensions were successful.

8 Acknowledgments

The Blue Gene/L project has been supported and partially funded by the Lawrence Livermore National Laboratories on behalf of the United States Department of Energy, under Lawrence Livermore National Laboratories Subcontract No. B517552. The authors would like to thank John-David Wellman and Michael Gschwind for their help in the preparation of this paper.

References


