RC24142 (W0612-003) December 1, 2006
Computer Science

IBM Research Report

Exploiting eDRAM Bandwidth with Data Prefetching: Simulation and Measurements

Valentina Salapura, Jose R. Brunheroto,
Fernando Redigolo, Alan Gara
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Exploiting eDRAM bandwidth with data prefetching: simulation and measurements

Valentina Salapura, José R. Brunheroto, Fernando Redígolo, Alan Gara

IBM Thomas J. Watson Research Center
Yorktown Heights, NY

Abstract

Compared to conventional SRAM, embedded DRAM (eDRAM) offers power, bandwidth and density advantages for the design of large on-chip cache memories. However, eDRAM suffers from comparatively slower access times than conventional SRAM arrays.

Data prefetching offers an attractive solution for the latency problem of a large capacity eDRAM cache, by reducing the average access latency. Moreover, data prefetching allows better exploitation of the large eDRAM bandwidth by making efficient use of the wide data accesses.

In this work, we present an exploration of design tradeoffs for the prefetch data cache in the Blue Gene/L® supercomputer. We also compare our simulation results to measurement results on actual Blue Gene systems. These experiments provide a validation for our modeling environment. Actual execution time measurements also include any system effects not modeled in our performance analysis environment, and confirm the 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 memory bandwidth via off-chip I/O. This may lead to a potentially unbalanced and inefficient design.

SRAM arrays are conventionally used as on-chip cache memories to obtain a significant reduction in I/O bandwidth requirements. However, the use of SRAM arrays is limited by the comparatively low density, and high power dissipation. SRAM memories are also suffering from manufacturability constraints limiting future access speeds due to device variation limiting the ability to accurately match FET devices of storage cells [8, 12].

A promising solution to these multiple constraints is the adoption of embedded DRAM (eDRAM) for high-capacity, high-density on-chip caches. Embedded DRAM merges DRAM and logic fabrication technologies to build the familiar 1T DRAM cell into a logic chip, and offers a significant increase in memory capacity per given unit area over SRAM, as well as low power operation and very wide data ports [13]. However, eDRAM typically will have a higher access latency than an SRAM-based solution.

Therefore, it was found necessary to deploy a prefetch scheme to decouple application access latency from eDRAM access latency, and use the available eDRAM bandwidth to hide latency.

The Blue Gene/L system is the first high performance computing system that delivers on the promise of on-chip eDRAM for increased performance at lower cost. The Blue Gene/L compute chip [5, 20, 21] has two processor cores. Each processor core incorporates a first level private 32kB instruction cache and a 32kB private data cache. Misses at the L1 cache level are given to a small private prefetch cache, acting as the L2 level. Each of the two L2 prefetch caches communicates with the 3rd level on-chip 4MB eDRAM cache, which is shared between the two processor complexes on the chip.

The work described here evaluates the prefetch architecture for Blue Gene systems.

Many previous studies have focused on application traces only and may have neglected the impact of the interaction between application software and operating system. In contrast, we study the prefetch behavior for a set of compute intensive workloads using full system simulation. Specifically, we compare an off-the-shelf Linux system modified to execute on Blue Gene/L, and the optimized CNK [16] microkernel tailored specifically for the execution of Blue Gene workloads.

Brunheroto *et al.* [6] presented an initial evaluation of the prefetch algorithms in this simulation environment.

By comparing our simulation results with measurements on actual Blue Gene systems, we evaluate simulation accuracy and the decisions made in the design process.

The contributions of this paper are: (1) an analysis of the prefetching potential in supercomputing applications,

(2) an extensive simulation-based design space exploration of prefetching approaches for an on-chip eDRAM cache, (3) an analysis of operating-system impact on prefetch effectiveness, and (4) a validation of simulation results with hardware measurements on a Blue Gene/L system.

While we do not introduce any new prefetch scheme, several ideas presented in this work 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 the access latency to eDRAM in a multiprocessor environment.

This paper is organized as follows: Section 2 describes the Blue Gene/L memory subsystem and prefetch cache architecture. Section 3 presents the simulation environment, our workloads and methodology. Section 4 analyzes simulation results for modeling the prefetch cache. Section 5 validates the simulation results with hardware measurements obtained on a Blue Gene/L system. We discuss related work in section 6, and draw our conclusions in section 7.

2 Blue Gene memory subsystem architecture

Blue Gene/L is a scalable high performance computing system containing up to 65,536 nodes. Each node consists of a Blue Gene/L Compute chip surrounded by 9 or 18 SDRAM-DDR memory chips, which provide 512 MB or 1GB of external memory. The Blue Gene/L Compute (BLC) chip is a System-on-a-Chip built with IBM CMOS Cu-11 (130 nm) technology. As illustrated in Figure 1, the BLC chip contains two PowerPC 440 processor cores, each with a SIMD floating point unit. Each PowerPC 440 core contains a 32kB private L1 I-cache as well as a private 32kB L1 D-cache with 32B cache line size, and interfaces to a private prefetch L2 cache with 128B buffer line size. The L2 caches communicate with a shared 4MB L3 on-chip eDRAM cache with a 128B cache line size. The eDRAM is configured as two interleaved banks of 2 MB each [17].

The prefetch L2 cache decouples the number and timing of requests generated by the core from requests to the L3 as follows: the L2 cache stores wide L3 cache lines, and satisfies multiple narrower L1 requests. This reduces latencies for L1 requests and also reduces traffic for the L3 cache.

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

- It stores demand-fetched L3 cache lines as well as prefetched L3 cache lines.
- It detects data streaming behavior.

Note that the prefetch L2 cache implements a line size corresponding to the eDRAM cache line size at the next hierarchy level, capturing locality of reference within an eDRAM line. This decision is key to reducing the number of eDRAM accesses and allows efficient sharing of the eDRAM-based L3 cache between two cores.

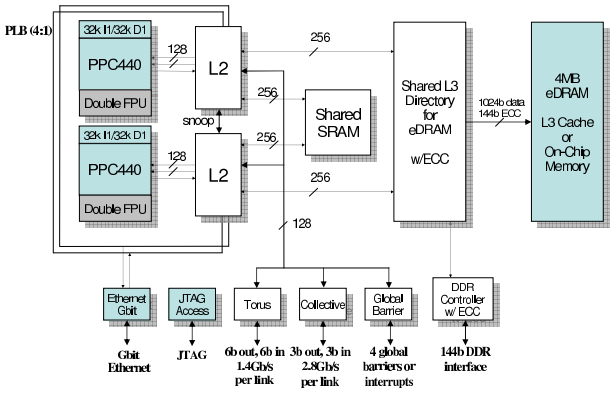


Figure 1. Blue Gene/L compute chip.

Comparing the Blue Gene/L memory system design to a traditional memory system design with an L1 cache and an SRAM-based L2 cache in terms of complexity, area, and power/performance, the present design offers several advantages.

There are several different ways to make this comparison:

- Remove the prefetch cache and use the eDRAM as L2: while keeping the cache size the same, it has longer latency. This configuration is part of our simulations and will be referred to in the rest of the article as the "no prefetch cache".
- Keep the L1 SRAM and L3 eDRAM caches, and add a standard L2 SRAM-based cache, having the same size (2kB total) as our prefetch cache. Such a small L2 cache would be ineffective. Placing a larger L2 cache would increase the chip area unacceptably.
- Remove eDRAM L3, use standard SRAM based L2. To keep the chip area the same, the 4MB eDRAM can be replaced by a 1MB SRAM. In addition, the eDRAM based solution requires about 4 time less power compared to a 1/4 sized SRAM based solution. Power considerations are especially important for embedded systems, and systems of significant scale such as Blue Gene.

It was a Blue Gene/L project requirement to reuse an unmodified PowerPC core available as a hard macro. Any change to the core would have incurred significant cost and would have delayed introduction of the Blue Gene system. The PowerPC 440 L1 cache is tightly integrated with the load-store unit, and any changes to the L1 cache architecture would have required changes to the processor core. In case of such a change, the entire core design would have had to be re-timed and re-validated. However, by deploying an external prefetch cache, the pre-tuned PowerPC 440 hard macro could be placed unmodified and could achieve peak clock frequency without design rework.

Compared to a hypothetical addition of an internal prefetch unit to the PowerPC 440 core, the approach of having an external prefetch cache is advantageous because

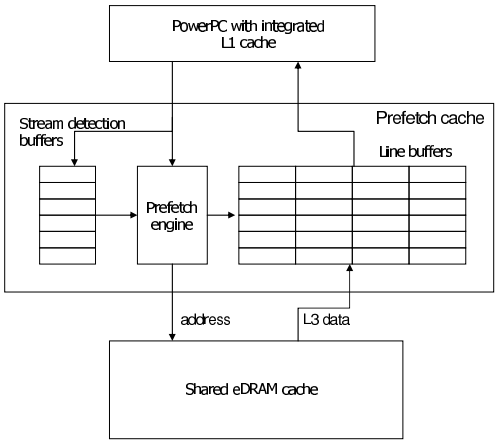


Figure 2. Prefetch cache architecture.

of contention for the L1 data cache port if the single data cache port would have been shared between the load-store unit and the internal prefetch unit. To overcome the contention problem, a two-port cache could have been be implemented, but this would have nearly doubled the array size.

By keeping the prefetched data out of the L1 data cache, we avoid pollution of the L1 cache from prematurely fetching data which could potentially displace 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. Extraneous prefetching in this environment can interfere with the delicate tuning performed to achieve peak performance.

2.1 Prefetch Cache Architecture

Figure 2 illustrates the architecture of the prefetch cache unit explored in this work and deployed as a L2 data cache in the Blue Gene/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. For an L1 cache request which misses in the prefetch L2 cache, only a quarter of the addressed L3 cache line is fetched from the L3. Only the portion corresponding to the requested L1 cache line is buffered in the L2 cache (in a portion of one line buffer dedicated for this) before it is forwarded to the L1 cache.

We have modeled a number of approaches to detect streams. An initial set of experiments uses an N-deep history queue for storing N prefetch cache address tags [18]. We refer to this history queue as the stream detection buffer.

When the processor requests data which miss in the L1 data cache, the prefetch unit records the corresponding L2 address in the stream detection buffer (step 1). If the requested address matches an L2 address already recorded in the stream detection buffers, but the address tag does not match, the requested L3 cache line is fetched and stored in a line buffer (step 2) and a stream is established.

Once a stream is established, the first subsequent access to data resident in the line buffer triggers a prefetch request to be issued (step 3). 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.

An alternative prefetching approach 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 prefetching uses a more aggressive prefetch 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, along with the impact of the operating environment.

Our system simulator is BGLsim [7], a full system simulator for the Blue Gene/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 Blue Gene hardware. An architectural simulation at the instruction-set level is several orders of magnitude faster than VHDL simulation at the logic design level, allowing exploration of a large design space with real applications. While the full system simulator can simulate a multi-node system, in this work we use it for simulation of a single Blue Gene/L node.

BGLsim can run a range of unmodified codes, from simple self-contained executables to full Linux images. The simulator includes interaction mechanisms for inspecting the entire internal machine state. It allows more flexible and more detailed instrumentation than what is possible with real hardware. We have modified the simulator to include tracing capabilities [15].

In our experiments, we have developed 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 Blue Gene/L compute node kernel (a single threaded OS) and Linux. To improve the simulation speed, we use a separate cache model for the prefetch cache and the L3 cache level.

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. The pseudo cycle accurate simulator provides execution cycle estimates. The trace simulator is much faster since it does not implement all the details. This 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.

We have opted for a single processor simulation, as having multiple processors on a chip do not affect the prefetch hit rate in any way. The reason for this is that prefetch cache is private to each processor, an there is no inter-processor interaction for prefetch cache.

There is a benefit to multi-processor designs due to the reduction in bandwidth requirements when using the prefetch cache relative to the L1 cache, which is shown in the bandwidth reduction study.

The metrics we use to characterize the prefetch cache performance are prefetch cache hit rate, prefetch cache miss rate, and the execution time (as predicted by the Blue Gene/L Pseudo Accurate Timing Model) [1]. 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 prefetch scheme would minimize the impact of 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%.

NAS	Instructions	L1 Misses	Misses per 1000 instructions
BT	547,414,050	30,788,712	56.24
CG	349,304,498	19,824,670	56.75
FT	645,116,212	37,248,944	57.74
IS	30,697,133	564,715	18.40
LU	238,891,062	10,934,076	45.77
MG	56,399,797	2,897,583	51.38
SP	273,988,939	20,660,969	75.41

Table 1. NAS Benchmarks Characteristics

Splash	Instructions	L1 Misses	Misses per 1000 instructions
LU	57,687,452	343,118	5.95
FFT	60,373,803	712,177	11.80
Radix	87,116,807	582,659	6.69
Ocean	30,005,066	1,843,293	61.43

Table 2. Splash-2 Benchmarks Characteristics

In case of a prefetch cache miss, we assume in our simulation model that the subsequent request to the L3 is satisfied with 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). All experiments were done with the L1 cache in write-back mode.

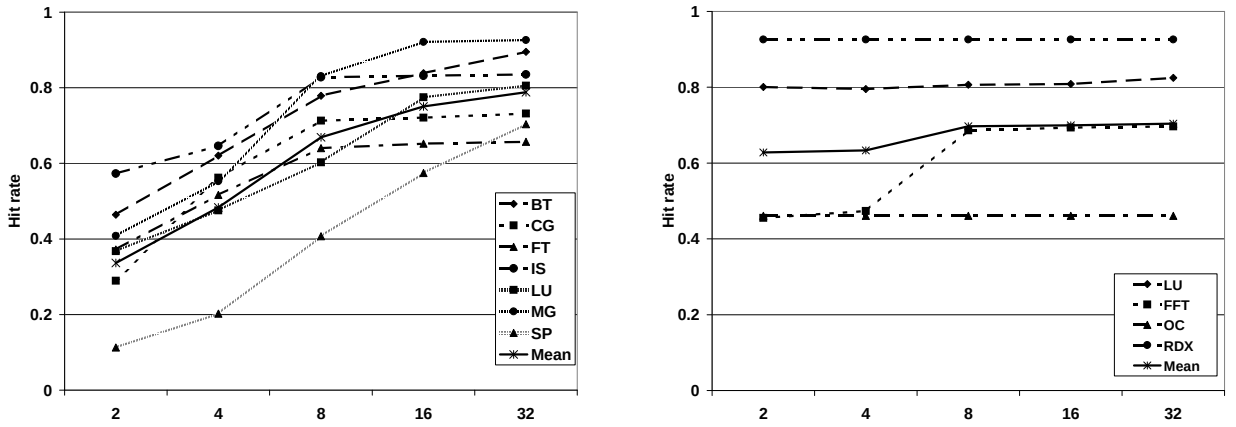
In our experiments, we use a set of applications from the publicly available NAS [2] and Splash-2 [23] benchmark suites. These are well known benchmarks containing shared memory applications that have driven much research into shared memory architectures and cache-coherence protocols. We have opted to use these publicly available applications, as they are good representatives of a wide range of scientific applications. We concentrated our efforts on scientific computing intensive applications, as these were the target workloads for the Blue Gene system.

Here, we report on 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 data miss sequence to determine prefetch opportunities. Tables 1 and 2 show the benchmarks used, the number of instructions executed during the run, the number of L1 data cache misses in absolute number, and the number of L1 data cache misses per 1000 instructions which represent the total prefetch opportunity.

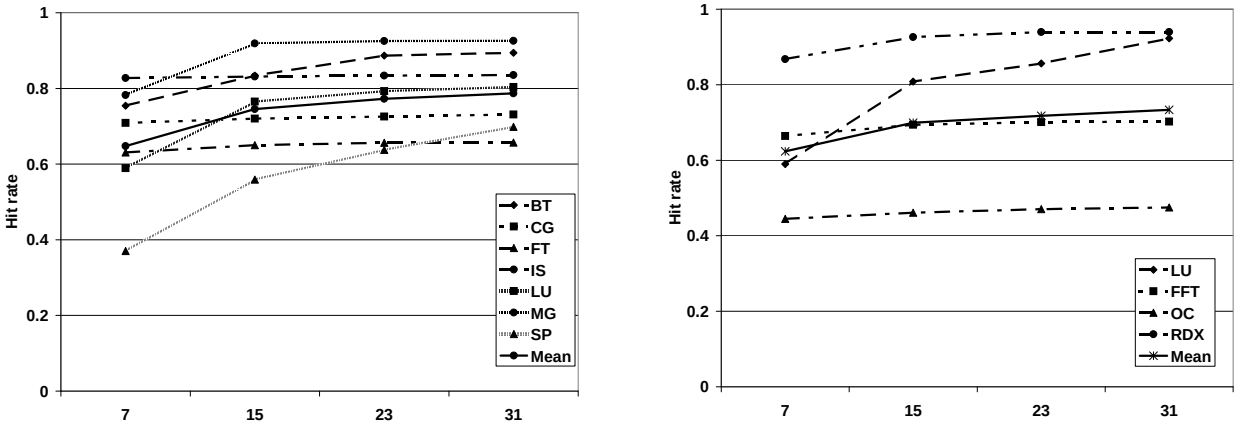
4 Experiments and Simulation Results

We model the prefetch cache to optimize the prefetch cache hit rate and execution time. We study the impact of



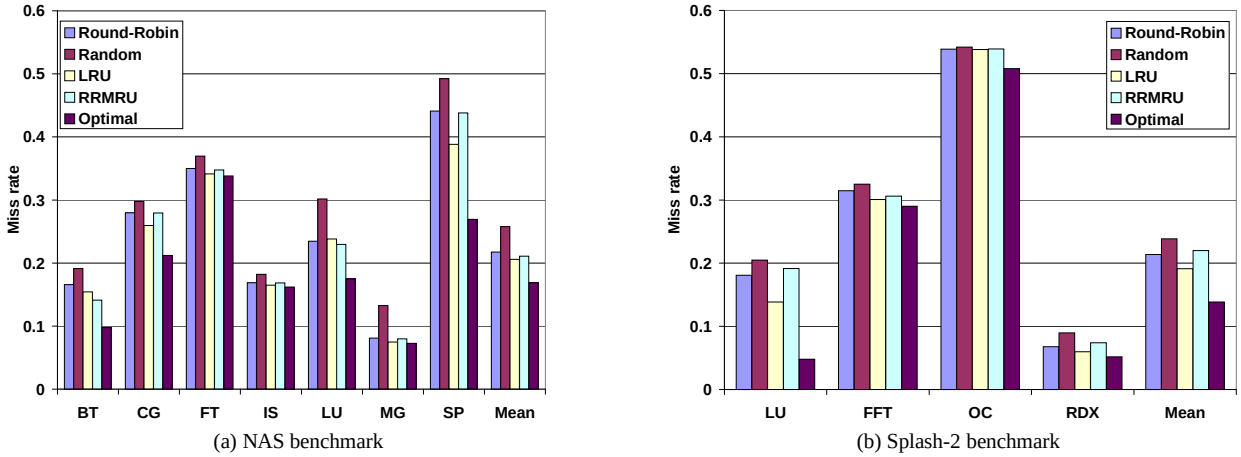
(a) NAS benchmark (b) Splash-2 benchmark

Figure 3. Varying stream detector size across the NAS and Splash-2 benchmarks.



(a) NAS benchmark (b) Splash-2 benchmark

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



(a) NAS benchmark (b) Splash-2 benchmark

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

several design parameters, and two operating systems.

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 in ascending address order. We also analyze the impact of the operating system used. To fully understand the impact of prefetching on the overall memory subsystem, we determine the impact of prefetching on the memory bandwidth to the shared L3 cache.

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. For this simulation, we use a prefetch cache large enough to not limit the number of streams which can be established and maintained.

Figure 3(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 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 spatial 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, a stream detector with 16 entries seem to be the best design choice.

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 that 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 on it rate is not linear. Choosing a prefetch cache size of 7 lines is clearly 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 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. Given that further increase in line buffers offers only modest incremental performance gains at significant area cost, a configuration with 15 line buffers offers an attractive cost/performance tradeoff.

4.3 Prefetch Cache Replacement Policy

The replacement policy determines how streams are aged out of the prefetch cache to make room for new data lines. We have modeled 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 modeled 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 was not chosen in the actual Blue Gene/L hardware design 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.

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 a subsequent request to this prefetch cache line, the address of the new request is compared to the saved data, and it is determined if the new 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 access patterns with negative strides present in these benchmarks. Also, scientific workloads in general do not show negative stride streams. Our model supporting bi-directional strides achieved minimal performance improvement.

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 prefetching, 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.

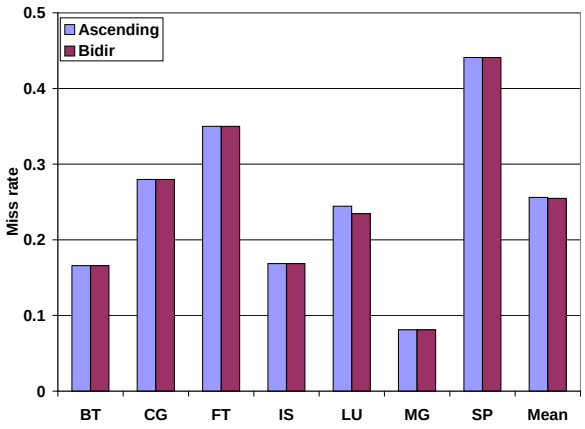


Figure 6. Bidirectional stream support using NAS benchmarks.

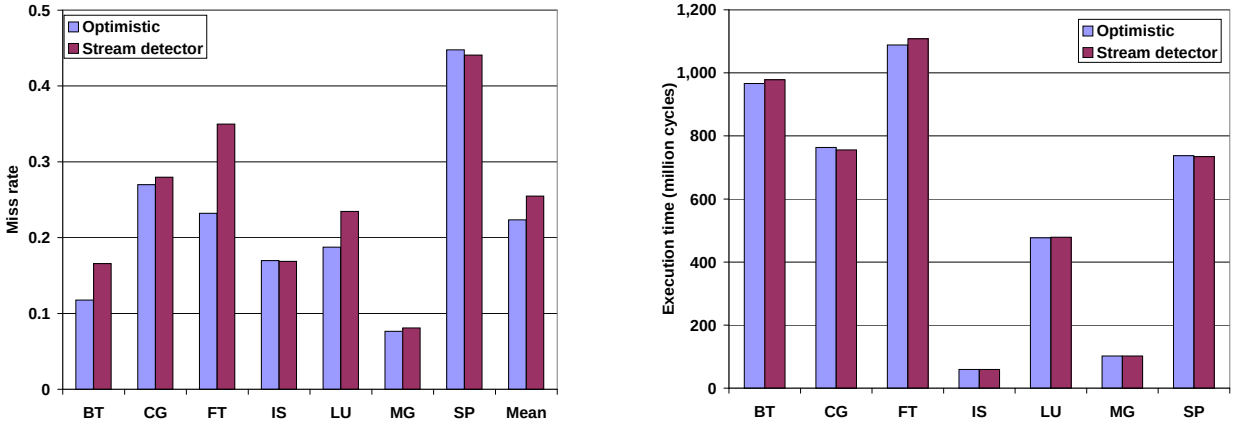
To gain a better understanding of quantitative advantages of the stream detection buffer design, we also compare the execution times for the two prefetching approaches across the NAS benchmarks. The results are illustrated in Figure 7(b). We observed that the execution times for both approaches are remarkably 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 uses a more aggressive prefetch strategy to increase the hit rate, we expected that the bandwidth requirements for the L3 will increase for the optimistic prefetching, 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, optimistic prefetching initiates a higher number of L3 prefetch accesses relative to the stream detector approach. However, the number of demand accesses is smaller for optimistic prefetching, resulting in a smaller total number of accesses.



(a) Miss rate (b) Execution time

Figure 7. Optimistic and stream detector buffers prefetch comparison for the applications from the NAS benchmark.

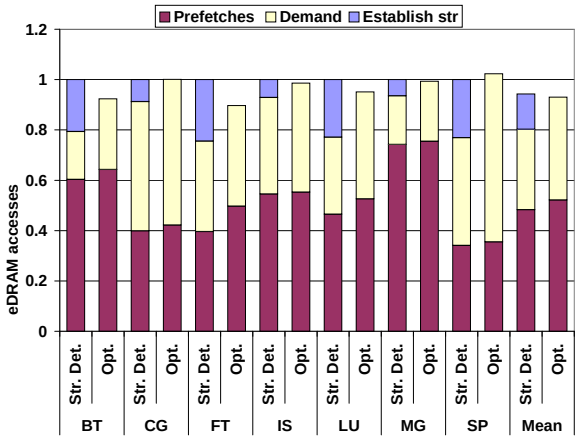


Figure 8. Normalized breakdown of eDRAM accesses for the optimistic vs. stream detector prefetching across the NAS benchmarks.

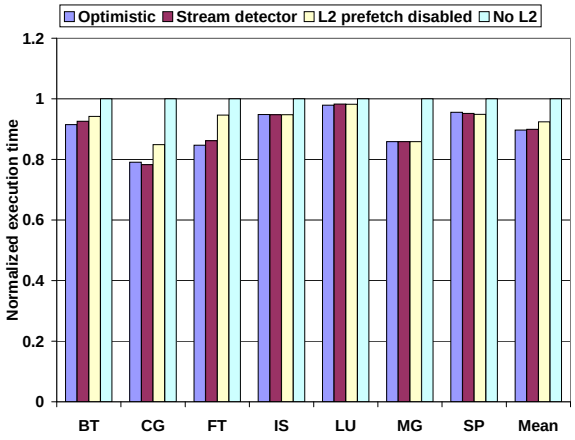


Figure 9. Normalized execution time for the optimistic prefetch cache, stream detector prefetch cache, L2 with disabled prefetching, and without L2 prefetch cache configurations across the NAS benchmarks.

For the cases where 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 allocated 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 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 performance results obtained when prefetching is disabled, but L2 line buffers are used. We also compare these schemes with the configuration without L2 prefetch cache.

Figure 9 compares the normalized execution time for these four approaches. 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

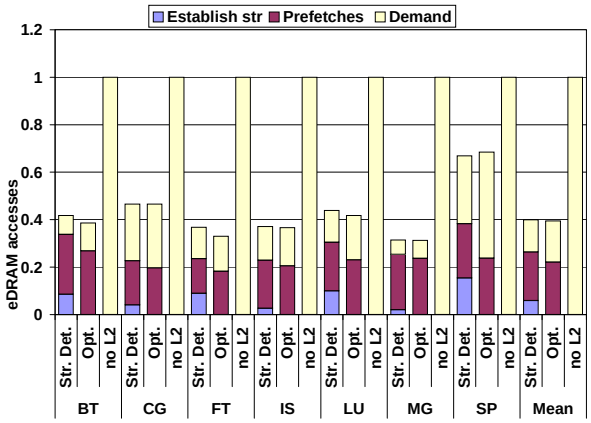


Figure 10. Normalized breakdown of eDRAM accesses for the optimistic prefetch cache, stream detector prefetch cache, and disabled prefetch cache across the NAS benchmarks.

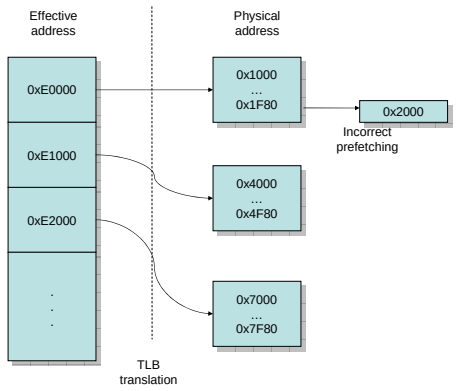


Figure 11. Linux page translation and prefetching

2%.

Simulation results also show a performance benefit when using prefetching, versus just exploiting multi-line buffers without the 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 L2 prefetch caches, the network interface and the memory controller. Figure 10 shows the normalized breakdown of eDRAM accesses for both prefetch approaches, and without L2 prefetch cache 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 prefetch 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.

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 multithreaded UNIX operating system (Linux), and a streamlined single-threaded kernel solution (the compute node kernel CNK employed in Blue Gene/L [16]).

CNK implements static mapping of virtual address to physical address. This linear mapping ensures that an application's access patterns in virtual address space are reflected in the physical address space available to the memory subsystem.

In comparison, a standard Linux kernel uses a 4kB page size. As a result of establishing page translations in response to demand paging, the kernel will map a continuous virtual address space to discontinuous physical 4kB 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 CNK on the prefetch cache hit rate.

With small pages, the 64-entry TLB of the PowerPC440 processor core cannot contain the entire address space for memory and I/O devices of a Blue Gene node. 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 an exception to the operating system. This effect has been mitigated somewhat in more recent versions of the Linux kernel with the introduction of large page support.

5 Hardware Measurements

The Blue Gene/L compute chip implements a 15 entry prefetch cache with a choice of both prefetch algorithms (stream detection and optimistic prefetching).

To verify our simulation results, we have performed extensive empirical performance analysis of applications. Here, we report hardware measurement results for the se-

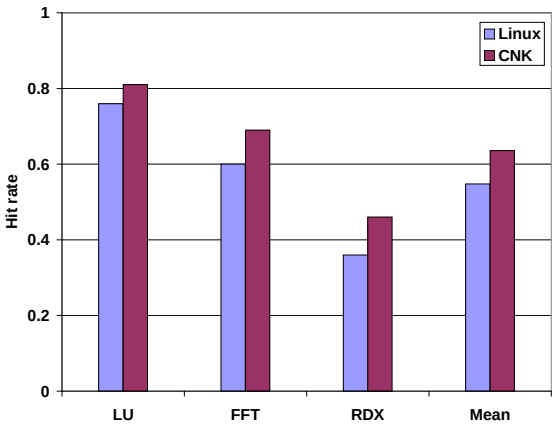


Figure 12. Splash-2 benchmarks on CNK and Linux.

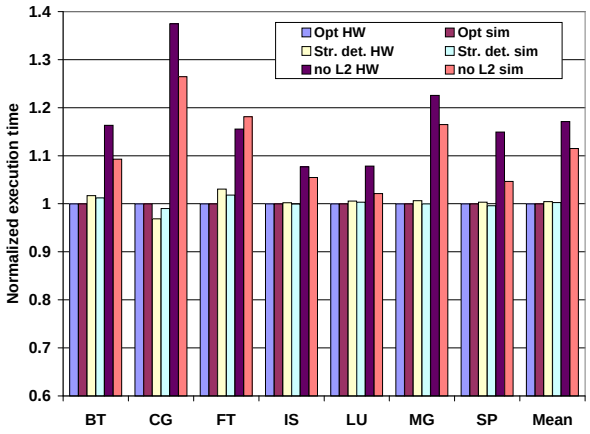


Figure 13. Hardware measurement execution time and simulated execution time (normalized) for the NAS benchmark.

rial NAS benchmarks.

Figure 13 shows the normalized measured execution times for three configurations implemented in hardware, and compares the measured execution times to simulated execution times for each NAS benchmark. The three implemented hardware configurations are the two prefetch schemes and L2 disabled, in which L2 is bypassed. 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 deviations between simulator and hardware measurements. This confirms the relative accuracy of the model to select the optimal design point.

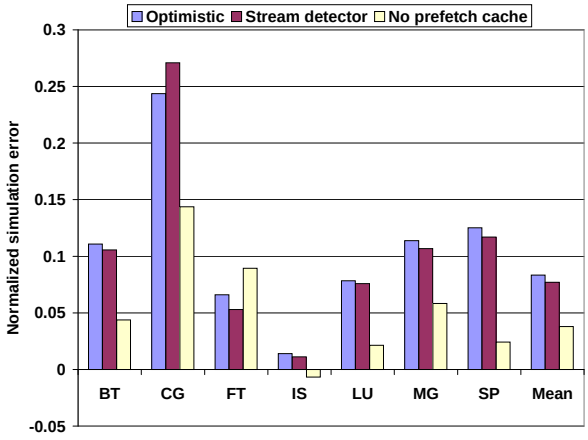


Figure 14. Simulation error against hardware measurements for the NAS benchmark.

Figure 14 plots the simulation error expressed as a difference of simulated execution time in cycles and measured hardware execution time over the corresponding measured hardware execution time. The simulation results are typically within 10 to 20% of measurements obtained on the actual hardware, 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 [10]. This confirms the quality of our simulation environment, and the decision to go with a full system simulator for the Blue Gene system.

6 Related Work

Fetching data from memory before the processor needs it has been a widely deployed and explored concept. The underlying idea is to overlap memory access time with computation, and thus to improve processor performance by reducing the number of stall cycles. Ideally, only data which are needed are prefetched so the data are ready to be used when the processor needs them. However, by prefetching too many unneeded data into the data cache, available memory bandwidth for other participants on the memory bus is reduced. Moreover the data cache gets polluted as prefetched data can displace useful data.

All prefetch schemes can be grouped into three prefetching techniques: hardware prefetching, software prefetching, and hybrid techniques. Generally, hardware prefetching techniques do not need modification of existing executables – prefetching is completely transparent from the software point of view – and can be implemented with relatively simple hardware. Software prefetching is generally

based on application properties obtained during compilation time and/or run time. While it requires no hardware support, the application suffers additional overhead, like code expansion, runtime cycles paid for executing prefetching instructions, and increased register usage.

Early work on cache prefetching includes the one-block-lookahead (OBL) scheme by Smith [22]. This approach initiates a prefetch for $(i+1)$-th block into the data cache when the i-th block is accessed into the cache. Jouppi [14] extends this idea by introducing stream buffers external to the cache to keep prefetched data. Each referenced buffer entry is loaded into the cache while the remaining prefetched blocks are kept in the external buffers.

Palacharla and Kessler [18] propose several improvements to the stream buffers. They have limited the number of unnecessary prefetches by using a history buffer to record the most recent primary cache misses and detect streams. Prefetching is done only for detected streams thus reducing bandwidth requirements at the expense of the reduced stream buffer hit rates.

Hardware-based prefetching techniques require no changes to existing executables and can be implemented with relatively simple hardware. However, compared to software prefetching techniques, sequential hardware prefetching performs poorly when non-sequential memory access patterns are encountered.

Gschwind and Pietsch [11] 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.* [24] 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 chasing, where the code follows a serial chain of loads. The approach described in [9] uses a pointer cache to assist prefetching for pointer load sequences.

In prior work, Gibson *et al.* [10] have evaluated the effectiveness of the FLASH simulation environment, and correlated hardware and simulation results.

Puzak *et al.* [19] discuss prefetching metrics, and analyze the potential for prefetching in SPECcpu and OLTP workloads.

In this work, we concentrate on workloads with established regular access patterns in compute intensive applications with regular memory access patterns.

7 Conclusion

Large capacity eDRAM caches make high bandwidth access to high capacity on-chip storage a reality by offering both wide data paths, and high on-chip transfer speeds. In conjunction with chip-multiprocessor solutions, it is possible to deliver increased performance at low power and with reduced bandwidth requirements to off-chip memory.

eDRAM is characterized by low power, high density, high bandwidth but also high latency. Prefetching fundamentally trades off the bandwidth to hide this access latency.

This study presents an exhaustive analysis of design options for a prefetch cache, designed specifically to interface to a large L3 cache implemented with embedded DRAM. This study concentrates on the performance of supercomputer class applications, and considers operating system impact by comparing different operating systems (a full-featured Linux and a custom-tailored single-threaded lightweight kernel specially designed for the Blue Gene/L supercomputer). 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.

In architectures where prefetching is implemented within a cache, careful prefetching is important so as not to pollute the cache. This equation has changed for the architectures where prefetch cache lines are outside of the L1 cache. Using wide prefetch cache lines captures spatial locality present in many applications (and in particular many HPC workloads), thus exploiting the wide L3 lines of the eDRAM cache efficiently. The high bandwidth of the shared eDRAM cache is capable of sustaining both processors' memory requests. This enables efficient data stream prefetching, thereby reducing execution time.

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. While for Blue Gene/L architecture the prefetch cache's size is only 2kB per processor, it reduces execution time across a wide range of workloads by 10% on average.

Our simulation environment could be used to influence as yet undisclosed future Blue Gene processor designs.

8 Acknowledgments

The Blue Gene/L system is the result of the dedicated work of a large team, and the authors would like to thank all members of the Blue Gene team. The authors would like to acknowledge in particular the contributions of Dirk Hoenicke, who was responsible for the design of the stream prefetching logic.

The authors would like to thank John-David Wellman and Michael Gschwind for many useful discussions, and for their help in the preparation of this paper, and Thomas Puzak and Ruud Haring for their suggestions during the preparation of this manuscript.

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.

References

[1] L. R. Bachega, J. R. Brunheroto, L. DeRose, P. Mindlin, and J. E. Moreira. The BlueGene/L pseudo cycle-accurate simulator. In *Proceedings of the 2004 IEEE International Symposium on Perfomance Analysis of Systems and Software*, Austin, TX, March 2004.

[2] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-929, NASA Ames Research Center, December 1995.

[3] M. Blumrich, D. Chen, G. Chiu, T. M. Cipolla, P. Coteus, P. Crumley, A. Gara, M. E. Giampapa, S. Hall, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. Liebsch, L. S. Mok, M. Ohmacht, V. Salapura, R. A. Swetz, T. Takken, and P. Vranas. A holistic approach to system reliability in Blue Gene. In *Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*. IEEE Computer Society Press, 2006.

[4] P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo – a full system simulator for the powerpc architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4), March 2004.

[5] A. A. Bright, M. R. Ellavsky, A. Gara, R. A. Haring, G. V. Kopcsay, R. F. Lembach, J. A. Marcella, M. Ohmacht, and V. Salapura. Creating the BlueGene/L supercomputer from low power SoC ASICs. In *Digest of Technical Papers, 2005 IEEE International Solid-State Circuits Conference*, pages 188–189, 2005.

[6] J. Brunheroto, V. Salapura, F. Redígolo, D. Hoenicke, and A. Gara. Data cache prefetching design space exploration for Blue Gene/L supercomputer. In *Proc. of SBAC-PAD*, October 2005.

[7] L. Ceze, K. Strauss, G. Almasi, P. J. Bohrer, J. R. Brunheroto, C. Cascaval, J. G. Castanos, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and E. Schenfeld. Full circle: Simulating Linux clusters on Linux clusters. In *Proceedings of the Fourth LCI International Conference on Linux Clusters: The HPC Revolution 2003*, San Jose, CA, June 2003.

[8] L. Chang, D. Fried, J. Hergenrother, J. Sleight, R. Dennard, R. Montoye, L. Sekaric, S. McNab, A. Topol, C. Adams, K. Guarini, and W. Haensch. Stable SRAM cell design for the 32 nm node and beyond. In *VLSI Symposium on Technology*, Kyoto, Japan, June 2005.

[9] J. Collins, S. Sair, B. Calder, and D. Tullsen. Pointer cache assisted prefetching. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture MICRO-35*, 2002.

[10] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich. FLASH vs. (simulated) FLASH: closing the simulation loop. In *9th International conference on Architectural support for programming languages and operating systems*, Cambridge, MA, 2000. ACM Press.

[11] M. Gschwind and T. Pietsch. Vector prefetching. *ACM Computer Architecture News*, 23(5):1–7, December 1995.

[12] M. Gschwind and J.-D. Wellman. Exploiting fine-grained memory locality with predictive dispatch. IBM Research Report RC23633, IBM TJ Watson Research Center, Yorktown Heights, NY, 2004.

[13] S. S. Iyer, J. E. Barth, P. C. Parries, J. P. Norum, J. P. Rice, L. R. Logan, and D. Hoyniak. Embedded DRAM: Technology platform for the Blue Gene/L chip. *IBM Journal of Research and Development*, 49(2/3), 2005.

[14] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of 17st International symposium on Computer Architecture*, pages 364–373, May 1990.

[15] P. Mindlin, J. R. Brunheroto, L. DeRose, and J. E. Moreira. Obtaining hardware performance metrics for the BlueGene/L supercomputer. In *Proceedings of Euro-Par*, Klagenfurt, Austria, August 2003.

[16] J. E. Moreira, G. Almsi, C. Archer, R. Bellofatto, P. Bergner, J. R. Brunheroto, M. Brutman, J. G. Castanos, P. G. Crumley, M. Gupta, T. Inglett, D. Lieber, D. Limpert, P. McCarthy, M. Megerian, M. Mendell, M. Mundy, D. Reed, R. K. Sahoo, A. Sanomiya, R. Shok, B. Smith, and G. G. Stewart. BlueGene/L programming and operating environment. *IBM Journal of Research and Development*, 49(2/3), 2005.

[17] M. Ohmacht, R. A. Bergamaschi, S. Bhattacharya, A. Gara, M. E. Giampapa, B. Gopalsamy, R. A. Haring, D. Hoenicke, D. J. Krolak, J. A. Marcella, B. J. Nathanson, V. Salapura, and M. E. Wazlowski. Blue Gene/L compute chip: Memory and Ethernet subsystem. *IBM Journal of Research and Development*, 49(2/3), 2005.

[18] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of 21st International Symposium on Computer Architecture*, April 1994.

[19] T. Puzak, A. Hartstein, P. Emma, and V. Srinivasan. When prefetching improves/degrades performance. In *ACM Computing Frontiers 2005*, Ischia, Italy, 2005. ACM Press.

[20] V. Salapura, R. Bickford, M. Blumrich, A. A. Bright, D. Chen, P. Coteus, A. Gara, M. Giampapa, M. Gschwind, M. Gupta, S. Hall, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, M. Ohmacht, R. A. Rand, T. Takken, and P. Vranas. Power and performance optimization at the system level. In *ACM Computing Frontiers 2005*, Ischia, Italy, May 2005. ACM Press.

[21] V. Salapura, R. Walkup, and A. Gara. Exploiting workload parallelism for performance and power optimization in Blue Gene. *IEEE Micro*, 26(5), September 2006.

[22] A. Smith. Cache memories. *ACM Computer Surveys*, 14:473–530, September 1982.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[24] D. F. Zucker, R. B. Lee, and M. J. Flynn. Hardware and software cache prefetching techinques for MPEG benchmarks. *IEEE Transactions for Circuits and Systems for Video Technology*, 10(5):782–796, August 2000.