Conserving Memory Bandwidth in Chip Multiprocessors with Runahead Execution

Martin Karlsson and Erik Hagersten

Department of Information Technology, Uppsala University
P.O. Box 337, SE-751 05, Uppsala, Sweden
{martin.karlsson, erik.hagersten}@it.uu.se

Abstract

The introduction of chip multiprocessors (CMPs) presents new challenges and trade-offs to computer architects. Architects must now strike a balance between the number of cores per chip versus the amount of on-chip cache and the cost-efficient amount of pin bandwidth. Technology projections indicate that the cost of pin bandwidth will increase significantly and may therefore inhibit the number of processor cores per CMP.

Runahead execution is a very promising approach to tolerate long memory latencies. In this paper we study the memory access characteristics of runahead execution. We show that temporal and data dependency aspects of runahead execution makes it possible to conserve bandwidth through the use of smaller cache blocks in the cache. We demonstrate, using execution-driven full system simulation, that our method of fine-grained fetching can obtain significant performance speedups in bandwidth constrained systems but also yield stable performance in systems that are not bandwidth limited.

1 Introduction

The continued decrease in transistor size combined with the increasing delay of wires relative to transistor switching speeds has led to the development of chip multiprocessors (CMPs). As more and more transistors become available per chip, it is possible to fit more and more cores per die. However, the number of off-chip signal pins is not growing nearly as fast leading to an increasing disparity between the number of cores that fit on a die and the available chip bandwidth. While chip and memory bandwidth since long have been a performance-sensitive resource, it has now surfaced as one of the major performance limiters.

Large-scale chip multiprocessors are desirable due to their low intra chip communication cost, enabled by shared higher level caches. For many commercial applications with an abundance of thread-level parallelism[25], adding cores or threads yields a very appealing performance improvement per mm^2. The number of cores per chip will likely be scaled up as high as the chip power and bandwidth will support. In this paper we target the CMP chip pin bandwidth by conserving the bandwidth of each thread. By reducing the bandwidth demands of each core, more cores could be supported within the same packaging cost envelope.

Increasing the on-chip cache reduces the bandwidth consumption. However, the performance effect of adding more cache is non-linear and for certain applications does not improve performance at all. Hence increasing on-chip cache can alleviate but not solve the bandwidth bottleneck.

As the memory wall problem has come to overshadow other aspects of processing, various forms of runahead execution have been proposed[21][12][7][3][4]. Runahead execution attempts to reduce the effect of the long memory latencies by increasing the memory-level parallelism. The strength of runahead execution is its ability to prefetch data far ahead of the stall point. We will in this paper analyze runahead execution from a memory system design perspective.

The contribution of this paper is three-fold:

- We highlight the trend of decreasing bandwidth per transistor, which is likely to impact CMP scaling.
- We identify and analyze spatial and temporal aspects of the data fetch pattern of runahead execution.
- We evaluate a scheme to exploit the runahead data fetch characteristics to reduce bandwidth consumption.

1-4244-0910-1/07/$20.00

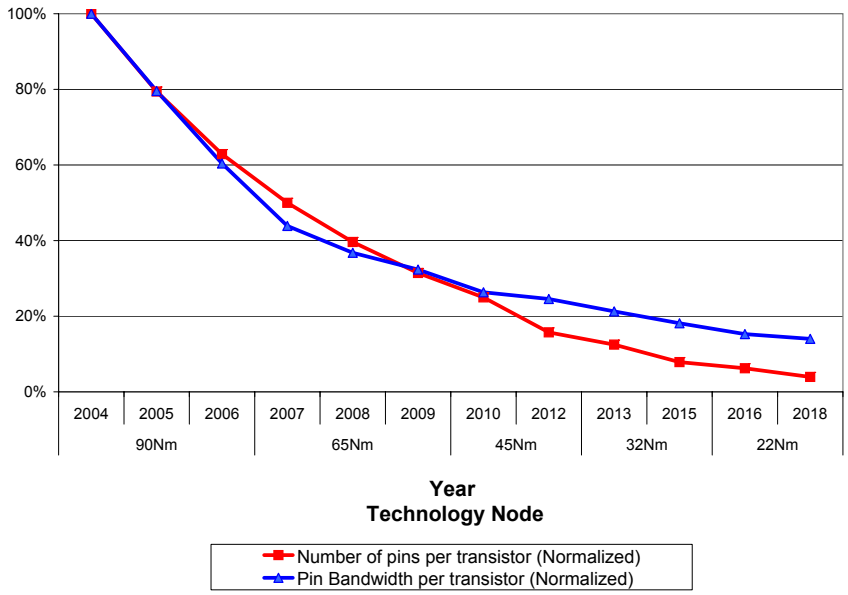

Figure 1. The Semiconductor Industry Association's prediction of bandwidth per transistor.

2 Technology Trends

The off-chip bandwidth of a chip is determined by the number of signal pins and the off-chip frequency. Reliability, power and especially cost restrict the number of pins per chip[5].

The International Technology Roadmap for Semiconductors published by the Semiconductor Industry Association (SIA) contains yearly predictions of the cost-efficient number of transistors and signal pins per chip for microprocessor chip designs [24]. The 2003 edition reported an annual growth in the maximum number of signal pins that is deemed cost-effective per chip of 5% (Avg.) between 2003 and 2018. In the 2004 update, however, the projection now predicts zero growth during the same time span. This indicates a significant challenge ahead to increase the chip bandwidth in a cost efficient way.

During the same period (2003-2018) SIA predicts the number of transistors per chip to grow annually by 26% (Avg.). Therefore the the number of signal pins per transistor will decrease considerably. As can be seen in Figure 1 the transistor count per signal pin ratio is projected to increase 20 fold to the year 2018. When taking relative on/off chip frequency development into account the increase is reduced to a factor of 7. This is due to the fact that signal pin frequencies are predicted to grow faster than on-chip frequencies. Several promising approaches, including optics and proximity communication, lie on the pin technology horizon. However in both cases such paradigm shifts are likely to provide a one-time bandwidth increase, not fundamentally change the bandwidth growth rate. Therefore in the long-term perspective, the off-chip bandwidth will decrease in relation to the number of transistors per chip. If the number of processor cores and cache per chip is going to be scaled linearly with the transistors made available by future process generations, it will lead to bandwidth requirements that will be hard to satisfy.

3 A Trend of Growing Cache blocks

The lack of growth in off-chip bandwidth will force architects to reevaluate memory system design from a bandwidth-centric view. This contradicts another trend we have observed in processor design. A trend towards larger and larger cache block sizes in the highest level on-chip caches. Examples of this is observed in the Pentium, Itanium and SPARC64 processor families. The L2 cache block size increased from 32 byte to 128 between the Pentium III and 4 processors. In Itanium it increased from 64 bytes per on-chip L3 block to 128 byte per block between Merced and McKinley. SPARC64 showed the largest increase between version V and VI, which went from 64 bytes to 256 bytes per cache block. These increased L2 cache block sizes were undoubtedly chosen to amortize the cost of longer and longer memory latencies. However, unless all of the fetched data are used, larger cache blocks leads to some data being unnecessarily brought on chip consuming the scarce bandwidth.

4 Runahead Execution and Hardware Scout

Runahead execution has been proposed as a viable path to tolerating the ever increasing memory latencies. A runahead processor enters speculative runahead mode upon encountering a stall condition. Before runahead mode is entered, a checkpoint of the architected register state is taken. The execution then continues in the hope of finding independent memory operations further down the instruction stream. A memory operation is considered independent if its address computation is not dependent of the destination register of the load causing the launch into runahead mode or a previous cache miss. Throughout this paper we refer to the memory operation causing runahead execution as the *launch point*. Once the data of the launch point load miss has arrived, execution is resumed from the checkpointed register state. We will use the term re-execution to refer to normal mode execution of instructions that previously have been executed in runahead mode.

In runahead mode each register that is dependent on the destination register of the launch point load is marked by a Not-A-Thing bit. The effective address of memory operations whose source registers are not marked by the corresponding Not-A-Thing bit, can be computed and forwarded to the memory system as non binding prefetches. If such a prefetch misses in the cache, the destination register is marked by a Not-A-Thing bit. Similarly for stores where the effective address is known, a prefetch is forwarded to the memory system. Runahead can therefore be viewed as an intelligent prefetcher that operates when the processor would otherwise have been stalled.

Hardware Scouting, described by Chaudhry et al.[7], is an extension of runahead execution, that includes several optimizations to previous runahead proposals. In hardware scouting, launching and exiting out of runahead is a zero-latency operation and runahead mode is also entered on low latency misses (L2 hits). This can be contrasted to the proposal by Mutlu et al. where runahead mode is entered first when an L2 miss has been detected[21]. In addition to entering runahead mode on a load miss, the hardware scouting proposal also identifies the case of a pending store when the store buffer is full as a stall event on which to launch into runahead mode. To execute a dependent instruction, i.e. an instruction with a source register marked as Not-A-Thing, simply requires an *OR* operation of the Not-A-Thing bits associated with the source registers. Therefore, by providing specialized Not-A-Thing functional units, runahead mode enables a faster execution than normal mode since the instructions producing a Not-A-Thing value will not consume regular execution unit resources. In runahead mode the serialization enforced by certain instructions (e.g. CASA) can also be ignored, further speeding up the runahead mode execution.

The ability of runahead execution to continue past stall events and not being restricted by an Out-of-order window size allows data to be prefetched further from the stall point and also in more of the demand miss cases, than in conventional out-of-order processors. Although runahead execution leads to a quite different data fetch pattern than exhibited by in-order and ooo processors, this have not been analyzed in depth in the literature. In the following section we will start by characterizing this fetch pattern from a spatial locality point of view.

5 Independent Spatial Cache Misses

Throughout this paper we use the term *spatial miss*$_{(S,C)}$ to refer to a cache miss in a system with a fetch block size of *S* that would have been avoided if the larger cache block size *C* would have been used. To isolate from replacement algorithm effects we use the metric in the context of sub-blocked caches[23], where *C* bytes of data map to the same cache tag and each sub-block is of size *S*. An example of a *spatial miss*$_{(16,64)}$ is an access that leads to a cache miss in a cache with 64 byte cache blocks divided into 16 byte sub-blocks but that would hit if the cache was not sub-blocked.

One of the most common examples of spatial misses occur when traversing an array, which often demonstrates a high degree of spatial locality. The address computation is often based on a base register holding the base address of the array and an offset based on a loop variable that is not dependent on a memory access. Similarly for more complex data structures, compilers often use fixed offsets to access member variables. Since these accesses often are based on address computations that are not dependent on previous memory accesses, it leads us to the hypothesis explored in this paper:

Spatial misses encountered in runahead mode are based on address computations of independent registers.

Independent registers refers to source registers of a runahead mode memory operation that is not, directly or indirectly, dependent on any previous cache miss, including the launch point memory access. If we assume that a large fraction of the cache misses are encountered in runahead mode and that address computations of spatial misses are independent, most spatial misses would lead to prefetches in a runahead processor. If the assumption that many misses occur in runahead does not hold, it would imply that cache misses are so far apart that the runahead thread launched on a miss does not cover the next miss. If that is the case it is unlikely that the performance of such applications are limited by the memory system, and are therefore less interesting for this kind of study.

Sometimes in runahead mode the effective address of a memory operation can not be computed due to one of the

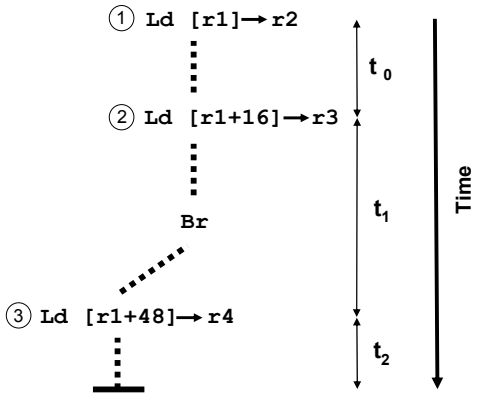

Figure 2. Temporal Aspects of Runahead Prefetching

source registers being marked as Not-A-Thing by a previous instruction. One example of this behavior is a linked-object list traversal. If the load of a list-item pointer misses in the cache, runahead execution will not be able to prefetch any of the subsequent list items since all items are dependent on the launch-point load. From a spatial access point of view, this case is not affected by the use of smaller cache blocks unless multiple objects fit in the same cache block. The reason for this is that the compiler usually uses register plus offset based addressing for this kind of accesses and if the register is Not-A-Thing, the accesses can not be sent out in either way. If we assume that all address computations of spatial misses are independent, all spatial misses will be found except for accesses of memory operations not covered by the runahead thread. One such example is if the runahead thread is terminated when only part of a cache block's spatial misses have been discovered.

It is worth noting that while runahead execution can be very effective at prefetching instructions into the instruction cache, runahead can not explore spatial instruction misses. The reason is that once an instruction miss occurs, the runahead thread can not continue and is therefore not able to find the spatial misses. Instruction misses are therefore costlier than data misses and presents a challenge since a runahead launch only can prefetch one cache block of instructions.

6 Temporal Aspects

In a system where spatial misses lead to separate data fetches, there is a temporal aspect. Spatial misses occurring in runahead mode are requested later than they would be in a system with a larger cache block size. If we assume that re-execution in normal mode is equally fast as runahead execution, then due to the runahead way of returning to a checkpoint and re-executing instructions, the delayed fetch of spatial misses should not incur any extra delay. The reason for this is that by the time they are re-executed the same amount of time has passed as the latency of the launch point miss.

This is illustrated in Figure 2 where the first load into r2 occurs in normal mode and triggers runahead execution. Then the second load into r3 is a spatial $\text{miss}_{(64,16)}$ with respect to the first load, i.e. the launch point load. The second load is forwarded to the memory system t_0 cycles after the launch point load access is sent out. However the re-execution will not occur until $t_1 + t_2 + t_0$ cycles after it was fetched, which is equal to the latency of the launch point load access. The same reasoning holds if the first load is not a launch point miss, but instead a regular cache miss occurring in runahead mode. As mentioned in Section 4 runahead execution can be faster than normal mode execution since additional specialized functional units can execute instructions marked as Not-A-Thing and serialization can be ignored. Therefore, despite the delayed request, the data have a high probability of being available in a timely fashion.

7 Architectural Proposal

The implications of our observations from a memory system design point of view is that spatial misses in runahead mode to a much higher degree than in conventional architectures can be identified and sent out as prefetches. Therefore memory systems do not need to rely on the prefetching effect of large cache blocks and instead allow smaller pieces of data to be requested separately. To exploit this observation we propose *fine-grained fetching*, which saves bandwidth by avoiding fetching of unnecessary data. Architecturally, this could be implemented by using a small cache block size in the highest level cache. To avoid the increase in tag overhead associated with small cache blocks, we evaluate our system using a sub-blocked cache organization.

Fine-grained fetching will lead to an increased on-chip address bandwidth. While on-chip bandwidth also is a restricted resource we believe it to be a less constrained resource than off-chip bandwidth. The off-chip address bandwidth consumption will also increase and we will discuss in Section 10.1 how this can be mitigated.

Reducing the instruction fetch block size would not have the same effect as for the data side, since runahead can not find spatial instruction misses beyond an instruction miss. Instructions often exhibit a high degree of spatial locality, suggesting the use of larger cache blocks for instruction caches. For unified second level caches, this leads to a complication due to the complexity involved in maintain-

ing coherence between a L1 cache with a larger cache block size than the L2 cache. The fine-grained fetch solution proposed in this paper is to use a small L1 cache block size for both the L1 instruction and data cache, and automatically prefetch multiple continuous cache blocks for each instruction miss into the L1 instruction cache.

8 Methodology

Our evaluation of runahead fine-grained fetch is based on full system simulation using both commercial and scientific workloads.

8.1 Simulation Setup

Simics is an execution-driven simulator that models a SPARC V9 system accurately enough to run Solaris unmodified[19]. To model the timing of a runahead processor we have extended Simics with a processor model, *Headrunner*, and a detailed memory hierarchy simulator VASA[10]. While equally relevant for single thread performance, we have opted to evaluate fine-grained fetching in chip multiprocessor context due to the increased bandwidth requirement.

8.1.1 Benchmarks

We use five multithreaded benchmarks. Two commercial applications SPECJBB 2000 and APACHE. SPECJBB 2000 (JBB2000) is a commercial JAVA-based middleware benchmark which evaluates the performance of server-side JAVA [26]. APACHE is a benchmark modeling the Apache open source Web server to which URL-requests are sent by a client [1]. SPECJBB 2000 is run for 2000 transactions after a warm-up period of 100 000 transactions. APACHE was run for 800 transactions with a 1000 transactions warm-up period.

In addition we used three randomly chosen applications from the SPLASH2 benchmark suite, LU_NC, WATER_S and WATER_N. We ran the SPLASH2 benchmarks with the default input sets specified in the SPLASH2 code and warmed the caches according to Woo et al [27].

8.1.2 Processor and Memory System Configuration

We model a highly simplified non-pipelined, in-order and single-issue processor model. Hence, no wrong path effects are modeled. We believe that the exclusion of wrong-path effects in this study is likely to lead to an underestimation of the benefit of fine-grained fetch, since fetching smaller blocks of data along the wrong path is likely to have positive performance effects. The simplified and fast processor model allowed us to make longer and more realistic runs of the commercial workloads, something the speed of a detailed processor model with full-system simulation would not have allowed.

Processor	16 cores per chip
Frequency	4 GHz
L1 data cache	64 KB, 8-way, 1 cycles
L1 block size	16 B
Shared L2 cache	2 MB, 8-way, 24 cycles
L2 block size	64 B sub-blocked 1, 2, 4 times
Memory Latency	200 cycles

Table 1. Simulated Target System Parameters

The CMP we are simulating contains 16 single-threaded cores each capable of invoking a runahead thread. Each core have separate L1 instruction and data caches, and a L2 cache which is shared among all the cores. We evaluate fine-grained fetch with a sub-blocked L2 cache configuration. To reduce interaction effects we have opted to simplify our memory system by assuming an infinite storebuffer and a perfect instruction cache.

We model separate unidirectional address and data buses of varying bandwidth. We assume an address packet size of 8 bytes, write packet header of 4 bytes and data packet header of 4 bytes, where each bus packet includes control state and ECC. We do not model any additional bus latency instead only the queueing delay is added. Further details of the simulated configuration can be found in Table 1.

8.1.3 The Headrunner Simulator

The Headrunner simulator is a simplified and idealized model of an in-order runahead processor. Runahead mode is entered on a data cache miss. We have assumed that a checkpoint can be made without incurring any stall time. The execution returns to normal mode when the cache block associated with the launch point load is filled. If the runahead thread comes across a memory access with side-effects, an asynchronous trap or an external interrupt, the execution stalls until relaunch into normal mode. Furthermore, if a miss is encountered in normal execution mode and the application is in kernel mode, the simulator does not enter runahead mode. The reason for this is simulator simplicity. Many simulation-wise complicated corner cases tend to happen when launching into runahead in kernel mode.

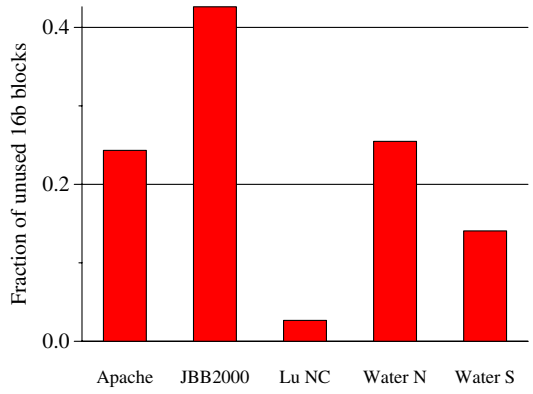

Figure 3. The fraction of unused 16 byte sub-blocks at eviction with a 64B cache block.

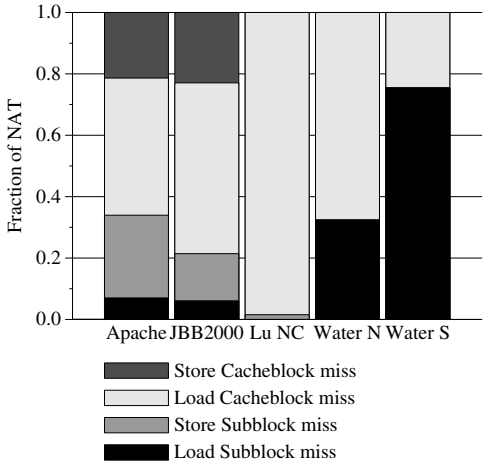

Figure 4. Distribution of memory accesses with Not-A-Thing marked source registers (64 B cache block 4 sub-blocks)

9 Results

9.1 Spatial Usage

To measure the potential bandwidth savings achievable by using a smaller cache block size, we have measured the number of untouched 16 byte sub-blocks upon eviction, assuming a 64 byte cache block size. As can be seen in Figure 3 the untouched sub-blocks constitutes up to 40% of the allocated sub-blocks, showing a very low spatial utilization. Previous studies have reported that less than half of the allocated data gets used before eviction even for a small cache block size of 32 byte[18]. These unused pieces of data consume bandwidth without contributing to performance. In a bandwidth constrained system these extra sub-blocks lead to increased queuing delay and therefore increases the latency of other memory requests. If these blocks could be identified ahead of time, the amount of data transferred and therefore the memory access time, could be reduced.

	NaT miss per 1000 instrs	NaT miss per L2 miss	NaT miss per L2 Access
Apache	2.04	0.07	0.01
JBB2K	13.52	0.73	0.14
Lu NC	0.00	0.00	0.00
Water N	0.06	0.13	0.00
Water S	0.00	0.00	0.00

Table 2. Number of Not-A-Thing misses.

We start our evaluating by using a sub-blocked L2 cache. We use sub-blocking instead of just reducing the cache block size to isolate the miss ratio effect from replacement effects. In a sub-blocked cache the same replacements will be made regardless of sub-blocking degree. Throughout this paper we refer to the number of sub-blocks mapping to the same tag as degree of sub-blocking, where a sub-blocking degree of one corresponds to a non-sub-blocked cache.

One of the potential obstacles towards successful runahead fine-grained fetching is if spatial misses can not be forwarded to the memory system due to source registers being dependent (marked as Not-A-Thing). Due to the way the headrunner simulator is implemented we can obtain the effective address also of memory operations where the source register is Not-A-Thing. We are therefore able to send non-intrusive probes to the memory hierarchy for Not-A-Thing memory operations and count the number of accesses that would have lead to a hit, block miss or a sub-block miss. A sub-block miss is an access that leads to a tag match but where the valid bit is not set for the sub-block. Table 2 shows the number of Not-A-Thing misses in relation to (normal mode) instructions, L2 misses and L2 accesses.

Across all benchmarks the total number of Not-A-Thing misses are relatively few compared to the number of normal mode instructions. JBB2000 shows the highest amount which could be due to the object-oriented nature of the workload with pointer-indirections etc. JBB2000 also shows a very high number of Not-A-Thing misses per L2 miss fraction, indicating that the number of memory accesses that could not be forwarded to the memory system due to dependencies are high. From the perspective of fine-grained fetching, the question is however how many of these misses are spatial misses.

Figure 4 shows a breakdown of the Not-A-Thing misses into cache misses and sub-block misses for both loads and stores. We find the fraction of load sub-block misses to be low for both JBB2000 and APACHE. The store sub-block

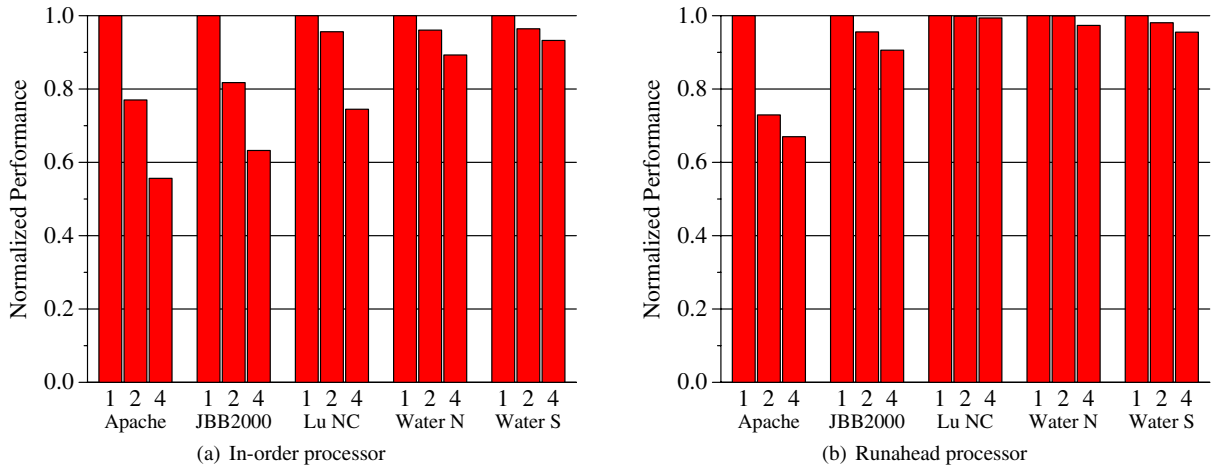

Figure 5. Performance effect of reducing fetch block size in a system with infinite bandwidth. The 64B cache block is divided into 1, 2 and 4 sub-blocks

misses are also fairly low although significantly higher than the load sub-block miss fraction. This leads us to conclude that the high number of Not-A-Thing misses observed for JBB2000 in Table 2 includes relatively few spatial misses that would penalize fine-grained fetching. The breakdowns are less interesting for the SPLASH applications since they barely have any Not-A-Thing misses.

10 Isolating Sub-block Miss Penalty

To isolate the miss penalty effect we start off by ignoring bandwidth effects, i.e. a data fetch always takes a constant number of cycles regardless if 64 bytes or 16 bytes are fetched. This is a very conservative assumption, penalizing higher degrees of sub-blocking, since smaller blocks often can be supplied faster than larger blocks. Figure 5 a, shows the performance effect of increasing the degree of sub-blocking, i.e. using smaller and smaller sub-blocks, in a conventional in-order processor. The performance cost for fetching smaller sub-blocks is significant. Three out of the studied applications suffer an overall performance degradation of 25% or more. Especially the commercial workloads, APACHE and JBB2000 are negatively affected with up to 45% and 35% reduction respectively. Such a considerable slowdown is unlikely to be acceptable in any design paying attention to single-thread performance and workloads that are not bandwidth bound. Since the performance effects of scaling cacheline size and sub-blocking degree for in-order processors have been thoroughly studied elsewhere, we will focus on runahead enabled processors for the remainder of this paper.

In an runahead processor, a cache miss can be overlapped by additional fetches in runahead mode. Due to the overlapping effect (memory level parallelism), the number of cache misses can not entirely indicate overall performance. As can be seen in Figure 5 b the isolated miss penalty is significantly lower than in the in-order case. Except for APACHE none of the applications suffer more than 10% performance degradation. In the case of APACHE the result is likely affected by the kernel mode limitation of the headrunner simulator. Since we have observed that 70% of L2 cache misses occur in kernel mode in APACHE, the simulator limitation of not launching into runahead mode when encountering a kernel mode stall condition, is likely to have a significant effect.

10.1 Taking bandwidth effects into account

Figure 5 b showed that from a miss ratio perspective the performance cost of decreasing fetch block size is limited for four out of the five studied applications. However reducing fetch block size also affects the bandwidth consumption. In Figure 6 we show the normalized data bandwidth when reducing the fetch block size. In the case of JBB2000 the bandwidth is reduced by 51%, while APACHE show a 21% reduction. The data bandwidth consumption of LU NC is increased by 15%, which can be explained by very high spatial locality in combination with additional data packet header overhead. The data packet header size of 4 byte which we have assumed leads to a four times as high overhead for 16 byte fetches compared to 64 byte fetches. This is somewhat exaggerated since the ECC does scale with the data size. We find that the bandwidth savings correlates well with Figure 3, since we see the largest effect in JBB2000 and the smallest effect for LU NC. Note that the bandwidth

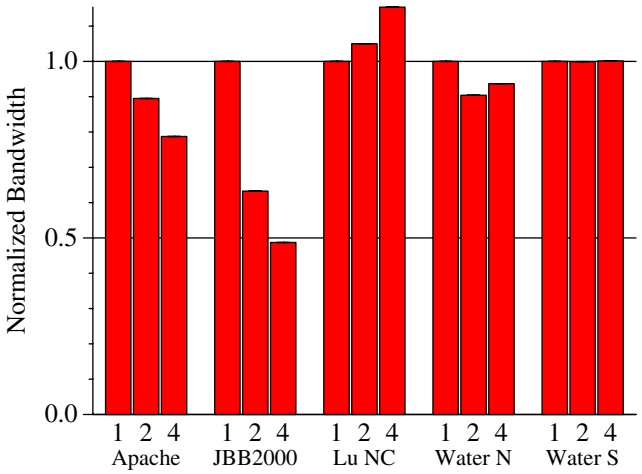

Figure 6. Normalized data bandwidth when scaling the degree of sub-blocking for a cache with 64 B cache blocks

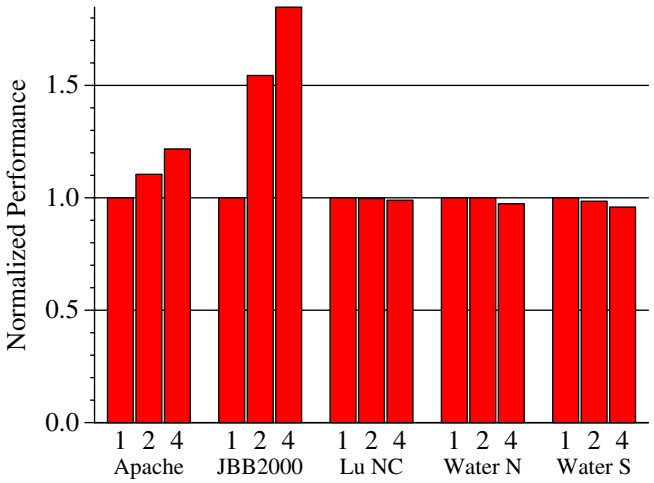

Figure 7. Performance Effect of increasing the degree of sub-blocking in a system with 8 GB/s data bandwidth

savings shown in Figure 6 includes all fetched data, while the Figure 3 only measured spatial utilization on the evicted cache blocks.

Fine-grained fetching results in an address bandwidth increase in applications with a high spatial utilization. We have observed that the increase in address bandwidth, sometimes erases the savings made in data bandwidth. We have found that for the SPLASH applications, the high spatial utilization leaves few sub-blocks to be saved and can not offset the increase in address bandwidth. Therefore, in terms of total bandwidth, we only see a reduction for the commercial workloads, APACHE and JBB2000.

While beyond the scope of this paper, we believe multiple techniques can be applied to reduce the address overhead. We have observed that spatial misses often are encountered sequentially, indicating that bus encoding schemes may be very successful at reducing the address overhead. Another possibility is to collapse multiple sub-block requests into larger requests when requests are queued up on-chip.

By adding a bus model to our experimental setup we can evaluate the performance effect of conserving bandwidth through reduced fetch block size. Our model includes both an address and data bus, each providing the same bandwidth. Figure 7 shows the normalized performance in a system with 8 GB/s of dedicated data bandwidth (16 GB/s in total). This can be compared with the total (address and data) memory bandwidth of the Itanium 2 and Power4 processors, which are 6.4 and 12.8 GB/s respectively[22]. For the Power4 the bandwidth is divided equally in address and data bandwidth, leading to a data bandwidth of 6.4 GB/s.

We can observe that for APACHE and JBB2000, the miss ratio penalty is outweighed by the bandwidth effect and reducing the fetch block size results in a speedup. For APACHE the speedup is 22% and for JBB2000 the speedup is 85%. The reason why we see such a large performance increase, despite a fairly modest bandwidth reduction, is likely due to burstiness in the memory requests. Since our bandwidth measurements are made for the entire run it does not reveal how much of the bandwidth is saved during the performance critical periods of contention.

For the SPLASH applications we note a similar behavior as observed in Figure 5 b, indicating that the SPLASH applications are insensitive to bandwidth constraints. The reason is that the working sets of our benchmark setups are too small and fits in the relatively large L2 cache of 2 MB. We have nevertheless chosen to include them in this study in order to demonstrate the performance stability of fine-grained fetching.

We have also simulated a system with 16 byte cache blocks, in order to estimate the performance increase possible from using small cache blocks instead of sub-blocking. We ignore the increase in tag overhead and simulate a 2MB L2 cache with 16 byte cache block and 8 GB/s of data bandwidth. For Apache and the SPLASH applications, the use of a smaller cache block size does not yield any significant performance improvement over the use of sub-blocking. However, for JBB2000 the speedup went from 85% with sub-blocking to 115% with small cache blocks.

In Figure 8 we show how performance is affected when scaling up the data bandwidth. As expected when we increase the bandwidth the positive performance effect decreases. But also if we decrease the bandwidth to 4 GB/s the speedup of APACHE and JBB2000 increases to 32% and

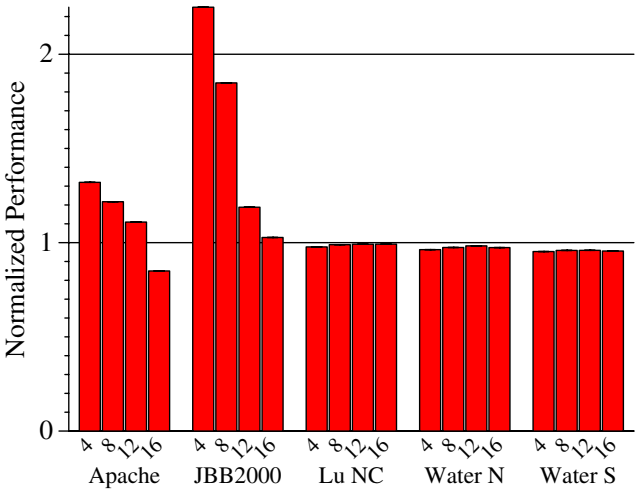

Figure 8. Normalized Performance when using a sub-block size of 16 B (compared to 64 B) and scaling up the data bandwidth (4, 8, 12, 16 GB/s)

125% respectively. Since we are assuming a fixed DRAM access latency regardless of fetch size, the only performance contribution from smaller cache blocks are gained from a reduction in bus queuing delay. Therefore when we scale up the bandwidth the performance approaches the fixed latency without bandwidth modeling result from Figure 5 b. The interesting thing to note here is that we need to scale up the bandwidth fairly high in order for the performance improvement to diminish.

10.2 Power and Latency Implications of fine-grained fetching

Chip power is one of the most constrained resources in processor design today. A significant source of power consumption are the inter-chip buses. Signal pin drivers can consume 15 to 20 percent of the total chip budget [20]. Therefore by reducing the amount of data transferred by using fine-grained fetching, substantial overall chip power savings can be made.

In narrow channel memory technologies like RAMBUS and DDR, a cache block request is divided into multiple reads (bursts) of e.g. 8 bytes as in DDR. If burst-scheduling is used, the bursts that make up a cache block, are read sequentially from a DRAM bank. In such systems the latency of reading a cache block is linear with the number of bursts. Therefore fine-grained fetching could also result in latency reductions, further reducing the memory access cost.

11 Related Work

In a patent by Burger and Wood[6] they propose an dynamic number of sub-block fetching where the number of sub-blocks fetched is determined by inspecting the usage of previously allocated blocks. They also propose fetching a pattern of discontinuous sub-blocks if an historical pattern is discovered. Kumar and Wilkerson proposes the spatial footprint predictor[18], that predicts how much data to fetch for each cache miss. Other proposals on predicting or adapting cache block size have been made by Gonzalez et al.[14], Johnson et al.[17] and Chen et al.[8].

Several proposals have been made suggesting memory compression to save bandwidth[2][15][13]. We deem these approaches orthogonal to fine-grained fetch since compression of sub-blocks could further alleviate the bandwidth problem. Proximity communication[11] appears to be a very promising new technology. If proven successful it could very well disrupt current bandwidth trends.

Spracklen et al. [25] presented latency and bandwidth implications of Hardware Scouting and describes several challenges involved in the design of chip-Multithreaded (CMT) processors. Other work targeted at understanding, exploiting or optimizing runahead execution have been presented by Sorin et al.[16] and Chou et al.[9].

12 Conclusions

Runahead execution is a very promising approach to tolerate long memory latencies. In this paper we have identified spatial and temporal aspects of the data fetch patterns of runahead execution. We have also demonstrated how these observations can in a very simple manor be exploited to alleviate one of the most pressing memory system design challenges, the pin bandwidth of a chip. Our proposal of reducing bandwidth consumption by employing smaller fetch blocks have been shown to lead to execution time speedups of up to 125%.

Smaller cacheline sizes have in many studies been shown to increase performance for certain workloads and under certain bandwidth constraints. In this study we show that with fine-grained fetching and runahead execution, stable performance can be obtained across a wide range of bandwidth constraints and workloads.

This is the first step towards exploiting the characteristic of independent spatial misses in runahead execution through fine-grained fetching. A first step leading to more efficient bandwidth utilization, since the amount of unnecessary data brought on-chip is decreased. We believe that future proposals can extend the method to reduce the increase in address bandwidth and to reduce the performance penalty for workloads with a high degree of spatial utilization.

13 Acknowledgments

We would like to thank Mark Hill and Kevin Moore for valuable comments on this work and Håkan Zeffer for valuable suggestions on the Headrunner simulator implementation.

References

[1] A. R. Alameldeen, M. Martin, C. Mauer, K. Moore, X. Min, M. Hill, D. Wood, and D. Sorin. Simulating a $2m Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, 2003.

[2] A. R. Alameldeen and D. A. Wood. Adaptive Cache Compression for High-Performance Processors. In *Proceedings of the 31th Annual International Symposium on Computer Architecture (ISCA'04)*, page 212, Washington, DC, USA, 2004. IEEE Computer Society.

[3] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically allocating processor resources between nearby and distant ilp. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*, pages 26–37, 2001.

[4] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. mei W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 387, 2003.

[5] D. Burger, J. R. Goodman, and A. Kgi. Limited bandwidth to affect processor design. *IEEE Micro*, 17(6), 1997.

[6] D. C. Burger and D. A. Wood. Cache with dynamic control of sub-block fetching. U.S. Patent No 6,557,080 issued April 29, 2003.

[7] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-Performance Throughput Computing. *IEEE Micro*, 25(3):32–45, 2005.

[8] C. Chen, S. Yang, B. Falsafi, and A. Moshovos. Accurate and complexity-effective spatial pattern prediction. volume 00, 2004.

[9] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31th Annual International Symposium on Computer Architecture (ISCA'04)*, page 76, 2004.

[10] Dan Wallin and Hkan Zeffer and Martin Karlsson and Erik Hagersten. VASA: A Simulator Infrastructure with Adjustable Fidelity. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems*, November 2005.

[11] R. Drost, R. Hopkins, R. Ho, and I. Sutherland. Proximity communication. *IEEE Journal of Solid-State Circuits*, 39(9):1529– 1535, Sept 2004.

[12] J. Dundas and T. N. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *International Conference on Supercomputing*, pages 68–75, 1997.

[13] M. Ekman and P. Stenström. A robust main-memory compression scheme. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*, pages 74–85, 2005.

[14] A. Gonzalez, C. Aliagas, and M. Valero. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. In *Proceedings of International Conference of Supercomputing*, pages 338–347, 1995.

[15] E. G. Hallnor and S. K. Reinhardt. A unified compressed memory hierarchy. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 201–212, 2005.

[16] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 1–11, 2004.

[17] T. Johnson, M. Merten, and W. Hwu. Run-time spatial locality detection and optimization. In *MICRO*, pages 57–64, 1997.

[18] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98)*, pages 357–368, 1998.

[19] P. S. Magnusson, M. Christensson, D. F. J. Eskilson, G. Hllberg, J. Hgberg, A. M. F. Larsson, and B. Werner". Simics: A Full System Simulation Platform. *IEEE Computer*, pages 50–58, February 2002.

[20] T. Mudge. Power: A First-Class Architectural Design Constraint. *IEEE Computer*, 34(4), Apr. 2001.

[21] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[22] M. R. Newsletter. Power5 Tops On Bandwidth, December 2003.

[23] J. B. Rothman and A. J. Smith. The pool of subsectors cache design. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 31–42, 1999.

[24] Semiconductor Industry Association (SIA). International Technology Roadmap for Semiconductors 2004 Update.

[25] L. Spracklen and S. G. Abraham. Chip multithreading: Opportunities and challenges. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[26] http://www.spec.org/osg/jbb2000/.

[27] 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)*, 1995.