Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads

Md Kamruzzaman Steven Swanson Dean M. Tullsen

Computer Science and Engineering
University of California, San Diego
{mkamruzz,swanson,tullsen}@cs.ucsd.edu

Abstract

Multicore processors have become ubiquitous in today's systems, but exploiting the parallelism they offer remains difficult, especially for legacy application and applications with large serial components. The challenge, then, is to develop techniques that allow multiple cores to work in concert to accelerate a single thread. This paper describes inter-core prefetching, a technique to exploit multiple cores to accelerate a single thread. Inter-core prefetching extends existing work on helper threads for SMT machines to multicore machines.

Inter-core prefetching uses one compute thread and one or more prefetching threads. The prefetching threads execute on cores that would otherwise be idle, prefetching the data that the compute thread will need. The compute thread then migrates between cores, following the path of the prefetch threads, and finds the data already waiting for it. Inter-core prefetching works with existing hardware and existing instruction set architectures. Using a range of state-of-the-art multiprocessors, this paper characterizes the potential benefits of the technique with microbenchmarks and then measures its impact on a range of memory intensive applications. The results show that inter-core prefetching improves performance by an average of 31 to 63%, depending on the architecture, and speeds up some applications by as much as 2.8×. It also demonstrates that inter-core prefetching reduces energy consumption by between 11 and 26% on average.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors–Optimization

General Terms Languages, Performance

Keywords chip multiprocessors, helper threads, compilers, single-thread performance

1. Introduction

Although multi-core processors have become ubiquitous over the past decade, leveraging the parallelism they offer to increase application performance remains challenging. There are several reasons for this, including the prevalence of non-parallelized legacy code, the difficulty of parallel programming, and the inherently serial nature of many programs and algorithms. Even in applications that are amenable to parallelization, the applicability of multicore processors has its bounds: as manufacturers supply an increasing number of cores, more and more applications will discover their scalability limits. Furthermore, Amdahl's law dictates that the more hardware parallelism is available, the more critical sequential performance becomes.

For many applications, then, the only way to get parallel speedup is to exploit non-traditional parallelism – to use multiple cores to accelerate the execution of a single thread. For single-threaded legacy codes, this allows the multicore processor to increase performance without altering the program. Non-traditional parallelism allows programs with limited parallelism to profitably use several cores per thread, increasing the application's scalability.

Previous work [5, 8, 9, 17, 18, 20, 22, 33] has demonstrated the use of "helper threads" which run concurrently in separate contexts of a simultaneous multithreading (SMT) processor and significantly improve single thread performance. Helper thread prefetching has advantages over traditional prefetching mechanisms – it can follow more complex patterns than either hardware prefetchers or in-code software prefetchers, and it does not stop when the main thread stalls, since the prefetch threads and the main thread are not coupled together. However, SMT prefetching has its limitations. The helper thread competes with the main thread for pipeline resources, cache bandwidth, TLB entries, and even cache space. The helper thread cannot target more distant levels of the cache hierarchy without thrashing in the L1 cache. In addition, core parallelism is typically more abundant than thread parallelism – even on a 4-core Nehalem, only 1 SMT thread is available for prefetching. Perhaps most importantly, not all multicores support SMT.

This paper describes and evaluates helper threads that run on separate cores of a multicore and/or multi-socket computer system. Multiple threads running on separate cores can significantly improve overall performance by aggressively prefetching data into the cache of one core while the main thread executes on another core. We call this technique *inter-core prefetching*. When the prefetcher has prefetched a cache's worth of data, it moves on to another core and continues prefetching. Meanwhile, when the main thread arrives at the point where it will access the prefetched data, it migrates to the core that the prefetcher recently vacated. It arrives and finds most of the data it will need is waiting for it, and memory accesses that would have been misses to main memory become cache hits.

Inter-core prefetching can target both distant shared caches and caches private to the core. Inter-core prefetching is especially attractive because it works with currently available hardware and system software. Thus, it would be easy to incorporate in a compiler or runtime system. Previous approaches have required fundamental changes to the microarchitecture or were limited to prefetching into shared caches.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

This paper describes inter-core prefetching and our implementation under Linux. We characterize the potential impact of inter-core prefetching on a range of currently-available multicore processors running focused microbenchmarks and then demonstrate its impact on a wide range of memory-intensive applications. Our results show that inter-core prefetching improves performance by an average of 31 to 63%, depending on the architecture, and speeds up some applications by as much as 2.8×. We also demonstrate that inter-core prefetching reduces energy consumption by between 11 and 26% on average. Finally, we show that because of the separation of resources, inter-core prefetching is more effective than SMT thread prefetching.

The remainder of this paper is organized as follows: Section 2 places inter-core prefetching in context with other work on non-traditional parallelism. Section 3 describes the inter-core prefetching technique in detail. Sections 4 and 5 contain our methodology and results, and Section 6 concludes.

2. Related Work

Prefetching is an important technique to speed up memory-intensive applications. There has been significant work on both hardware prefetchers [6, 15] and software controlled prefetchers [2, 7, 25]. The introduction of multithreaded and multicore architectures introduced new opportunities for prefetchers. Multithreaded architectures are a natural target, because threads share the entire cache hierarchy.

Helper thread prefetchers [5, 8, 9, 14, 17, 18, 20–22, 31, 33] accelerate computation by executing prefetch code in another SMT context. Normally, these techniques predict future load addresses by doing some computation in the helper thread and prefetch the corresponding data to the nearest level of shared cache. In many cases, the prefetch code comes from the main thread, allowing those threads to follow arbitrarily complex prefetch patterns.

There are several ways to generate helper threads. The work in [9] uses hand-built helper threads while, Kim and Yeung [18] describe compiler techniques to generate the reduced version statically with the help of profiling. Zhang, et al. [32] and Lu, et al. [21] describe the generation of helper threads in dynamic compilation systems. Collins, et al. [8] generate the helper threads completely in hardware.

Prior research has also targeted prefetching across cores. In Slipstream Processors [14], a reduced version of the program runs ahead of the main program in another core. Multiple specialized hardware pipes transport information (including loaded values) from one to the other. Lu, et al. [21] demonstrate helper thread prefetching on a CMP, but that work is limited to prefetching into a cache that is shared by the main core and the prefetch core. Brown, et al. [3] propose changes to CMP hardware and coherence to enable a thread on one core to effectively prefetch for a thread on a separate core. In contrast to these approaches, inter-core prefetching enables cross-core prefetching into private caches with no new hardware support. Since it requires no special hardware support, inter-core prefetching can work between sockets in a single system. It also works across CMPs from multiple vendors.

Gummaraju, et al. [12] implement a compilation framework that maps a program written for stream processors to general purpose processors. In their system, there are three threads – one for execution, one for data prefetching, and one that manages execution and prefetching. They target SMT threads.

Other research attempts to leverage the hardware parallelism of multicores to accelerate serial execution in other ways. Speculative multithreading [19, 23, 28, 30] uses multiple hardware contexts or cores to execute different parts of the serial execution in parallel and then does runtime verification of correctness. Mitosis [28] additionally uses helper threads to precompute dependent data. Prescient instruction prefetch [1] is another helper thread-based technique that improves performance by prefetching instructions instead of data.

Other work aggregates cache capacity in multi-core/multiprocessor systems to hide memory latency. Cooperative Caching [4] proposes a hardware technique to aggregate private caches. The technique stores cache lines evicted from one private cache to another private cache and ensures maximum utilization of the cache space by keeping a single copy of the same cache line and performing cache-to-cache transfers as needed. Michaud [24] proposes a hardware mechanism to migrate threads and distribute data over the aggregate cache space.

Data Spreading [16] is a software-only mechanism to aggregate the space of multiple private caches. The technique transforms a program to leverage the aggregate cache capacity via compiler-directed migrations. They show that data spreading can provide both performance and power benefits, if the working set fits in the aggregate cache space. However, data spreading does not provide benefit if the working set is too large or the program does not have a repetitive access pattern. Inter-core prefetching removes these two limitations by prefetching data into the next cache. Because of this, inter-core prefetching's performance is much less sensitive to the actual cache size or access pattern.

Using separate threads to access memory and perform computation is similar in spirit to decoupled access/execute architectures [29], but inter-core prefetching accomplishes this without specialized hardware. Decoupled access/execute architectures use a queue to pass data from a memory thread to a compute thread. The queue requires that the two threads be tightly coupled, and that the memory thread correctly compute the result of all branches. As a result, there is rarely enough "slack" between the threads to allow the memory thread to issue memory requests far enough in advance to fully hide their latency. The Explicitly-Decoupled architecture [11] uses the queue only for passing branch outcomes and shares the entire cache hierarchy to get the advantage of prefetching. In our scheme, the private cache of a helper core serves as the "queue", but because inter-core prefetching places no ordering constraints on the accesses, it can completely decouple the threads.

Unlike helper thread prefetching, runahead execution [10, 26] does not require extra hardware contexts or cores. During the long latency load operation, instead of blocking, it assumes a dummy value and switches to the runahead mode to continue execution. When the original load operation is satisfied, it switches back to the normal execution. This improves memory level parallelism but still places all of the burden of prefetching, demand misses, and execution on a single core.

3. Inter-core Prefetching

Inter-core prefetching allows a program to use multiple processor cores to accelerate a single thread of execution. The program uses one to perform the computation (i.e., the *main thread*). The other cores run *prefetching threads*, which prefetch data for the main thread. The two types of threads migrate between cores with the prefetching threads leading the way. Thus, when the main thread arrives on a core, much of the data it needs is waiting for it. The result is reduced average memory access time for the main thread and an increase in overall performance. Inter-core prefetching is a software-only technique and requires no special support from the processor's instruction set, the operating system, caches, or coherence.

Inter-core prefetching works by dividing program execution into *chunks*. A chunk often corresponds to a set of loop iterations, but chunk boundaries can occur anywhere in a program's execution. The *chunk size* describes the memory footprint of a chunk. So, a 256 KB chunk might correspond to a set of loop iterations that will

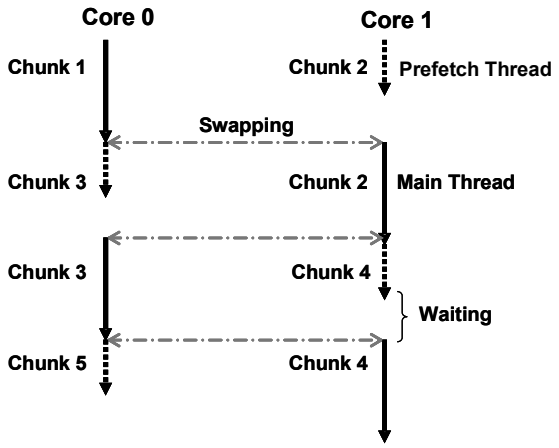


Figure 1. Program execution path for inter-core prefetching. The main thread and the prefetch thread swap cores repeatedly, when the main thread reaches a new chunk of data.

access 256 KB of data. For example, if a loop touches 1 MB of data in 100 iterations, then a 256 KB chunk would consist of 25 iterations, a 512 KB chunk would consist of 50 iterations.

Both the main thread and prefetch thread execute one chunk at a time. In the main thread, the chunked code is very similar to the original code, except for the calls to the run-time that implement migration.

The prefetch thread executes a *distilled* [9, 18, 28, 33] version of the main thread, with just enough code to compute addresses and bring the necessary data into the cache. We will use the term prefetch slice, or *p-slice* (terminology borrowed from prior work) to describe the code that the prefetch thread executes to prefetch the data.

Figure 1 illustrates inter-core prefetching with one main thread (solid line) and one prefetcher thread (dotted line) executing on two cores. Execution begins with the main thread executing chunk 1 and the prefetcher executing chunk 2. Once they complete their chunks, they swap cores and the main thread starts on chunk 2 while the prefetcher moves on to chunk 3.

Sometimes, multiple prefetch threads are useful. In a four-core system, three cores would prefetch chunks 2-4 while the main thread executes chunk 1. When the main thread starts executing chunk 2, the prefetch thread for chunk 2 would move on to chunk 5. In this scenario, each prefetcher thread has 3 chunks worth of time to finish prefetching the chunk.

Our implementation of inter-core prefetching comprises three principal components. First, a program analysis algorithm identifies chunk boundaries in the original program. Second, a p-slice generator creates a distilled version of the code for the chunks. Finally, the inter-core prefetching library provides a mechanism to keep the main thread and the prefetcher threads synchronized as they migrate between cores.

Below we describe the key components of inter-core prefetching in detail and present an example that illustrates inter-thread prefetching in action.

3.1 Identifying chunks

A chunk is a sequence of executed instructions which access a portion of the application's address space that matches (as closely as possible) the target chunk size. We use two different approaches for partitioning the iteration space into chunks – *aligned chunks* and *unaligned chunks*.

Whenever possible, we use aligned chunks. Aligned chunks force chunk boundaries to align with loop iteration boundaries at some level of the loop nest. For instance, if an inner loop accesses 100 KB and the target chunk size is 256 KB, we will likely start and end a chunk every two iterations of the outer loop.

There are some situations where inner loop iterations follow predictable patterns even though the outer loop does not. In that case, it is easier to add a counter and track inner loop iterations, ignoring the structure of the outer loop – that is an unaligned chunk. Unaligned chunks allow us to swap after a predictable number of inner loop iterations, even if we are not sure how many outer loop iterations will have executed. For example, if we are targeting 1024 inner loop iterations, and the inner loop executes 200, 700, then 300 iterations, the first swap would happen in the middle of the outer loop's 3rd iteration. For some loops, it may not be possible to predict the future addresses at all – pointer chasing code being a prime example. We do not prefetch those loops.

We prefer aligned chunks, even when there is not a precise match between inner loop sizes and the desired chunk size. They introduce less overhead because they allow us to use the existing structure of the code to introduce the thread management primitives. In practice, unaligned chunks are rarely needed.

To identify useful chunks, a profiler selects loops that perform at least 0.5% of the total memory accesses. The profiler also creates a dynamic loop nest tree that spans procedure boundaries, and calculates the average memory footprint of an iteration in each loop. While we will invariably suffer from profile mismatch, the information on the data touched in a single iteration is reliable enough to be useful across multiple inputs.

3.2 Distilling code

To build the prefetching thread, we generate p-slices by hand, following the example of much of the original work on helper thread prefetching [9, 22, 33]. Generating aligned chunks is typically simple and the techniques we apply are all implementable in a compiler. Other efforts have used both static [18, 28] and dynamic [32] compilation, as well as specialized hardware [8] to construct p-slices. Those techniques could be applied to inter-core prefetching as well.

For both chunk types (aligned and unaligned), the p-slice contains just the code necessary to compute the prefetch addresses and perform the prefetches. In most cases, the distilled code matches the basic structure of the main thread, but this is not necessary. For instance, if the compiler can determine the range of addresses the code will access, it can easily generate a generic prefetcher to load the data with minimal overhead — unlike SMT prefetchers, the ordering of the accesses within a chunk is unimportant in our scheme.

Our implementation uses normal loads rather than prefetch instructions, because the hardware has the option of ignoring prefetch instructions if there is contention for memory. Using normal loads requires that the prefetcher only issues memory requests to valid addresses. This was not a problem for any of the workloads we examined, but a more aggressive implementation that issued speculative prefetch instructions to potentially invalid addresses could use prefetch instructions to avoid segmentation faults.

Our p-slices include several other optimizations as well: We convert stores into loads to prevent misspeculation and avoid invalidations that could delay the main thread. We use profile information to determine the most likely direction for branches within a loop iteration. For highly biased branches, we remove the unlikely path and the instructions that compute the branch condition.

3.3 Thread coordination

For inter-core prefetching to work properly, the main thread and prefetcher threads must work together smoothly. This means that (1) they must take the same path through the program so they execute the same chunks in the same order, (2) the prefetch thread

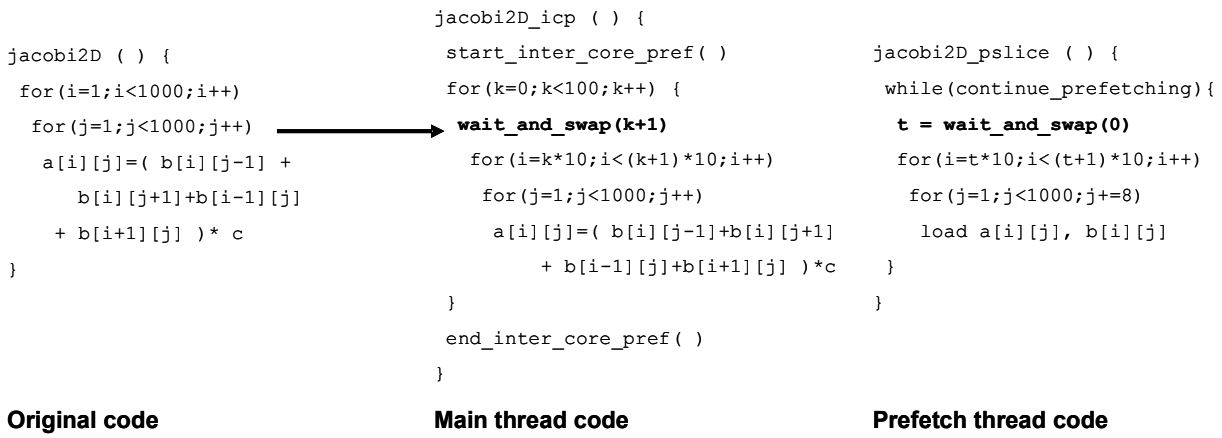


Figure 2. Inter-core prefetching implementation for the 2D Jacobi kernel. Prefetch thread takes an argument that specifies the chunk it should prefetch. The calls to wait_and_swap() synchronizes the two threads before swapping processors.

must stay far enough of the main thread to make the prefetches useful, and (3) they need to synchronize at swap points to exchange cores.

The first requirement is part of creating valid p-slices: The distiller must ensure that both the prefetcher and the main thread take the same route through the program. To handle the second requirement, we always start a prefetch thread at least a full chunk ahead. The prefetch thread can finish its chunk either earlier or later than the main thread does. The prefetcher usually finishes first, in which case one prefetching thread is sufficient (as in Figure 1) and the main thread will rarely stall. In some cases, the p-slice is slower than the main thread, and the main thread must wait for the prefetcher to finish. When that happens, we start prefetching the next chunk of data in the main thread's core to avoid wasting cycles, but we would prefer to never have to stall the main thread. As an alternative, we could stop the prefetching thread prematurely and allow the main thread to execute without any stall. We experiment with this approach but do not observe any advantage in our experiments. The reason is that if the main thread finishes early, eventually either the main thread or the prefetch thread will have to bring in the missing data for the next chunk to execute. Often the prefetcher (because it has more memory level parallelism and less overhead) will do so faster than the main thread, so it is better to just let it finish.

As a third alternative, very effective when we expect the prefetch threads to consistently be slower, we can use multiple prefetching threads to reduce or eliminate the need for the main thread to stall, and increase the effectiveness of prefetching. This exploits additional parallelism to allow the prefetch threads to stay ahead.

To coordinate thread migration we have implemented a simple user-space migration tool that allows threads to migrate between cores without entering the kernel. This saves significant overhead since normal context switches take about $10\mu s$ on our machines. At each wait and swap point, the distiller inserts a call to a library function in both the main thread and the prefetching thread. The function acts as a barrier across the main thread and the prefetching thread working on the next chunk the main thread will execute, so the threads block until they have both completed their chunk. Then the threads swap cores, and the main thread starts executing on the core with a freshly populated cache while the prefetch thread moves on to start prefetching another chunk.

The tool uses standard Linux APIs – *setcontext*, *getcontext*, *swapcontext*, and pthreads to perform the swap. The library reduces the context switch overhead by a factor of 5. The entire thread management substrate requires only about 25 lines of code, not including comments. We use Linux's scheduler affinity interface to "pin" kernel threads to particular cores, while our thread management system moves logical threads between them.

Figure 2 shows the code for a 2D Jacobi implementation and the corresponding main thread and p-slice code. We use aligned chunking and each chunk is a sequence of the outer loop's iterations. To make the p-slice efficient, we remove all the arithmetic operations, and make it touch each cache line only once by incrementing *j* by 8 in the inner loop since each cache line holds 8 elements. This transformation speeds up execution by a factor of 2.2× on a Core2Quad machine using four cores.

3.4 Impact of cache coherence

Cache coherence concerns do not have a significant impact for SMT based helper thread prefetching because both the helper thread and the main thread share all levels of cache. However, on inter-core prefetching, the underlying coherence protocol has a greater impact when chunks share data. This impacts both the generation of p-slices and the optimal chunk size. We will briefly discuss some of these issues here.

Write sharing is particularly problematic for inter-core prefetching. We do not allow helper threads to write, but if they prefetch data that the main thread writes to, it results in a slow upgrade transaction in the main thread and an invalidation in the helper thread rendering the prefetch useless. Read sharing has less of an impact, particularly when there exists a shared inclusive last level cache (like the L3 cache in Nehalem). In that case, both the main thread and the prefetch thread can get data from the shared cache. However, with an exclusive shared cache (L3 cache in Opteron), the prefetch thread will need to get the data from the other core (via cache-to-cache transfer) rather than from the shared cache. This is not a problem except that on this architecture cache-to-cache transfers are slow – they take 4 to 5 times as long as going to the L3 cache [13]. The Core2Quad, with no global shared cache, has similar issues. Adding additional prefetch threads can potentially hide this extra latency and avoid any impact on the main thread.

The degree of sharing between chunks depends significantly on the chunk size. For instance, if neighboring loop iterations share data, a larger chunk size that covers many iterations will cause less inter-chunk sharing than a smaller chunk size. For the 2D Jacobi case in Figure 2, using one iteration per chunk means that the main thread and prefetching threads share 50% of read data, while using 10 iterations per chunk results in less than 10% shared data.

System Information	Intel Core2Quad	Intel Nehalem	AMD Opteron
CPU Model	Harpertown	Gainestown	Opteron 2427
No of Sockets×No of Dies×No of cores	2×2×2	1×1×4	1×1×6
L1 Cache size	32KB	32KB	64KB
L1 hit time	3 cycles	4 cycles	3 cycles
L2 Cache size	6MB (per die)	256KB private	512KB private
L2 hit time	15 cycles	10 cycles	15 cycles
L3 Cache size	None	8MB shared	6MB shared
L3 hit time		38 cycles	36 cycles
Data TLB capacity (Level-1, Level-2)	16, 256	64, 512	48, 512
Memory access latency	300-350 cycles	200-240 cycles	230-260 cycles
Swapping cost	$2\mu s$	$1.7\mu s$	$1.7\mu s$

Table 1. The three processors have very different memory hierarchies that lead to different optimal points of operation for inter-core prefetching.

Benchmark name	Suite, Input	Memory footprint	LLC misses / 1000 inst	No of loops icp applied	Chunking technique
BT	NAS, B	1200 MB	19.96	20	Aligned
CG	NAS, B	399 MB	20.86	5	Aligned
LU	NAS, B	173 MB	17.42	14	Aligned
MG	NAS, B	437 MB	11.74	6	Unaligned
SP	NAS, B	314 MB	19.61	35	Unaligned
Applu	Spec2000, Ref	180 MB	13.09	8	Aligned
Equake	Spec2000, Ref	49 MB	28.53	4	Aligned
Swim	Spec2000, Ref	191 MB	25.16	4	Aligned
Lbm	Spec2006, Ref	409 MB	19.95	1	Aligned
Libquantum	Spec2006, Ref	64 MB	24.45	6	Aligned
Mcf	Spec2006, Ref	1700 MB	47.57	6	Aligned, Unaligned
Milc	Spec2006, Ref	679 MB	27.34	20	Aligned
Svm-rfe	Minebench	61 MB	16.97	1	Aligned

Table 2. Our benchmarks, with memory footprint, last level cache misses per 1000 instructions on Opteron, the number of loops where inter-core prefetching is applied, and the chunking technique used.

4. Methodology

This section describes the processors and applications we use to experimentally evaluate inter-core prefetching. The next section presents the results of those experiments.

4.1 Processors

Inter-core prefetching depends on the details of the processor's memory hierarchy and interconnect. Currently-available processors vary widely in the number of caches they provide on chip, the geometry of those caches, the interconnect between them, and the policies used to manage them. Table 1 summarizes the three processors we use to evaluate inter-core prefetching. On the Core2Quad, we use only one core per die, so that the L2 acts like a private cache.

All of the processors have hardware prefetching enabled. This allows us to evaluate whether inter-core prefetching effectively targets cache misses traditional prefetchers cannot handle.

4.2 Applications

To evaluate inter-core prefetching, we use a collection of applications from Spec2000, Spec2006, the NAS benchmark suite, and MineBench [27]. Inter-core prefetching is only of interest for applications that are at least partially memory bound. For applications that are not memory bound, the technique will have little effect. To identify memory bound applications, we used performance counters to count last-level cache misses. If the application incurred more than 10 cache misses per thousand instructions on the Opteron, we included it in our test suite. Table 2 provides details about the workloads and inputs.

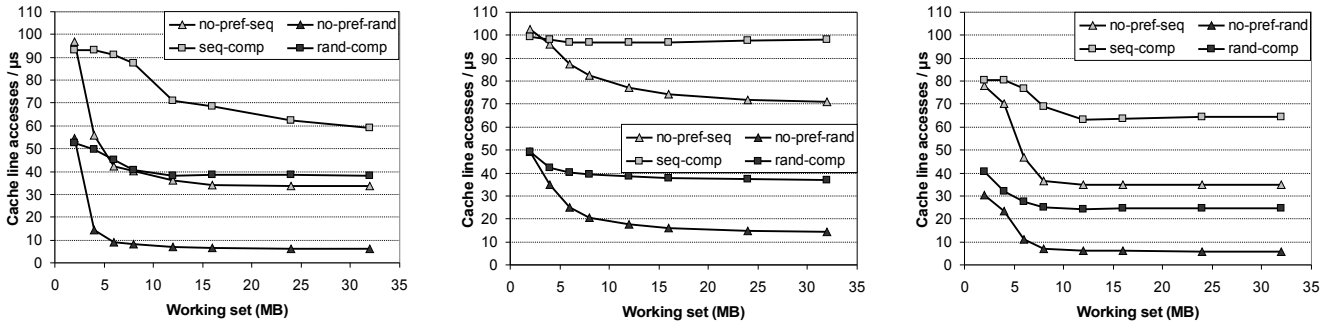
The table also lists how many loops we applied inter-core prefetching to and what type of chunks we used. For the NAS benchmarks we used the "W" inputs for profiling. For Spec2000 and Spec2006 we used the train inputs. Svm-rfe does not include multiple inputs, so we profiled on the same input we used to collect performance results.

5. Results

We evaluate inter-core prefetching in four stages. First, we use simple microbenchmarks to measure its potential and understand some of its tradeoffs. Then, we evaluate the technique's application-level impact using the workloads described in Section 4. Next, we evaluate inter-core prefetching's effect on power and energy consumption. Finally, we compare inter-core prefetching's performance across cores to a version of the technique that targets SMT contexts, and also with data spreading, another software-only cache optimization.

5.1 Microbenchmarks

To establish inter-core prefetching's potential, we use a simple microbenchmark that allows us to study the interplay between the chunk size, the number of prefetching threads, and the ratio of computation to memory access. Our microbenchmark accesses cache lines either sequentially (in virtual address space) or pseudo-randomly – in that case accessing each cache line in a region of



(a) Core2Quad (b) Nehalem (c) Opteron

Figure 3. As working set size increases to overflow one processor's cache capacity, inter-core prefetching prevents performance from dropping by using a cache on another processor to stage data.

memory exactly once, but in random order. The order is deterministic, though, allowing the prefetch thread to predict it accurately (this mimics dereferencing an array of pointers, for example). After accessing a cache line, the microbenchmark does a configurable number of arithmetic operations. We consider two cases: The *base* configuration executes four arithmetic operations per cache line access, and the *comp* configuration issues 32 operations. This gives four different microbenchmark configurations: rand-base, rand-comp, seq-base, and seq-comp.

5.1.1 The potential of inter-core prefetching

We first use the microbenchmarks to establish the overall potential for inter-core prefetching as the working set size changes. Figure 3 shows the throughput (measured as cache lines accessed per μs) of seq-comp and rand-comp for the three machines. It compares performance with and without inter-core prefetching over a range of working set sizes. The chunk size is 256 KB, and it uses one prefetch thread (two cores total).

For the microbenchmark without prefetching, throughput is high when the working set fits in local cache, but drops off (sometimes precipitously) when it no longer fits. Inter-core prefetching significantly (and, in one case, completely) mitigates that effect. For working sets that fit within the processor's cache, inter-core prefetching adds a small amount of overhead, reducing performance by between 3 and 4%. As the working set size grows, however, the benefits of inter-core prefetching are potentially large. For instance, for sequential access, throughput improves 1.76×, 1.38× and 1.85× for Core2Quad, Nehalem, and Opteron, respectively. These gains are in addition to the performance that the hardware prefetcher provides. The gains for random accesses are even larger: 6×, 2.5× and 4×, respectively. This is primarily due to the fact that the hardware prefetcher is ineffective for these access patterns.

The performance gains for random accesses are particularly interesting, since inter-core prefetching delivers between 2.5 and 6× improvements in performance using just two threads. It may seem counterintuitive, since the number of cores only doubled relative to running without prefetching. This is an important result – inter-core prefetching is not bound by the limits of traditional parallelism (i.e., linear speedup). Instead, the only limit on speedup it offers is the ratio of memory stall time to compute time in the pipeline. When that ratio is large (i.e., the processor stalls for memory frequently), and the stalls can be removed by a small number of helper cores, we can achieve speedups well above linear. Therefore, inter-core prefetching is more than a fall-back technique to use when thread-level parallelism is not available. For some applications inter-core prefetching will be more effective than conventional parallelism.

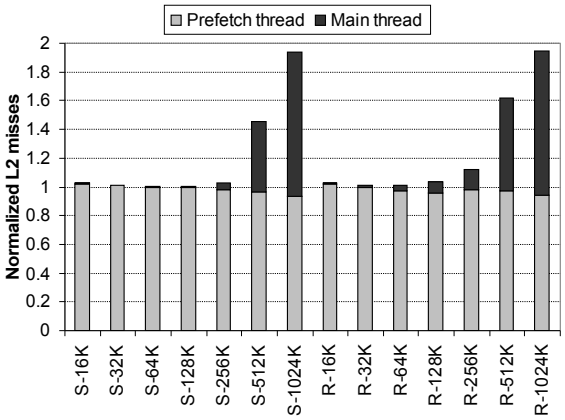


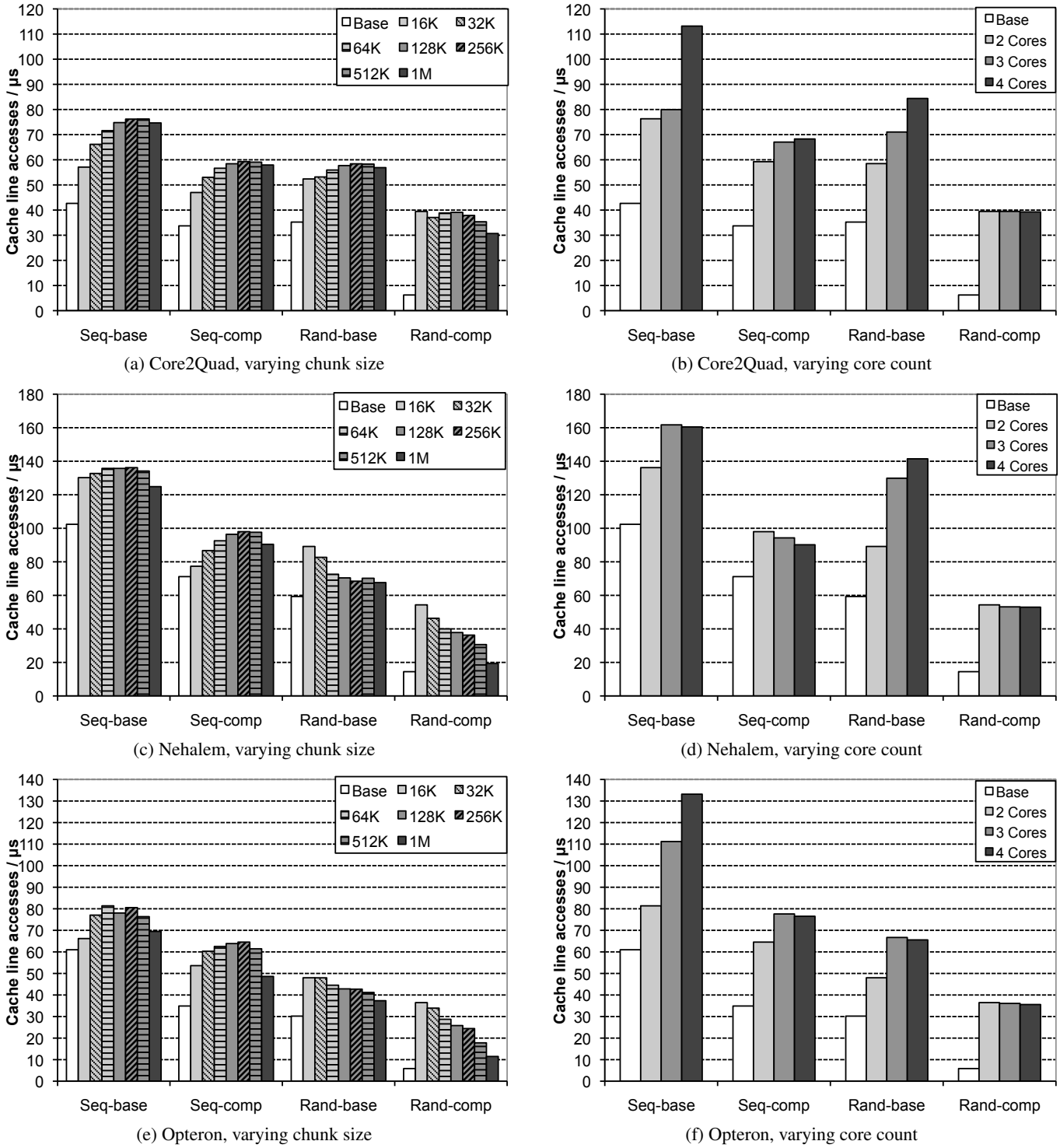
Figure 4. Misses incurred by the main thread and prefetch thread for different chunk sizes (R-16K means random access, 16 KB chunk), relative to the number of misses incurred in the baseline (no inter-core prefetching).

Of course, in those cases, a combination of traditional parallelism and inter-core prefetching is likely even better.

Looked at another way, inter-core prefetching achieves large speedups because it adds memory-level parallelism to applications that otherwise lack it. This is not obvious in the 2-core case, because we just move the data accesses from one core to the other. However, not only do the p-slices sometimes remove data dependences between loads that may have been in the code, they also minimize the number of instructions executed between loads and thus maximize the number of loads in the instruction window at one time.

Figure 4 demonstrates that inter-core prefetching moves almost all of the misses from the main thread to the prefetch thread. As long as the chunk fits in the L2 cache (for Opteron in this case), inter-core prefetching neither increases nor decreases L2 cache misses, it just performs them earlier and off the critical path. The speedup comes from the fact that we can access the data more quickly, and with more parallelism, than with a single thread.

In effect, inter-core prefetching partitions the work the application is doing into two, time-intensive parts: Executing non-memory instructions and waiting for cache misses. In this respect, it is similar to a decoupled access/execute [29] approach to computation. If a single thread must handle both tasks, it performs poorly at both: Memory stalls inhibit instruction level parallelism, and dependent



(a) Core2Quad, varying chunk size

(b) Core2Quad, varying core count

(c) Nehalem, varying chunk size

(d) Nehalem, varying core count

(e) Opteron, varying chunk size

(f) Opteron, varying core count

Figure 5. The graphs at left show the impact of chunk size on performance for our microbenchmark. The right hand figures measure the effect of varying the number of prefetching threads. Each pair of figures is for a different processor. Inter-core prefetching is especially effective for random access patterns that the hardware prefetcher cannot handle. For threads with less computation per memory access, more than one prefetching thread is necessary for optimal performance.

// non-memory operations retard memory level parallelism by filling the instruction window. Allocating a core to each task removes these counterproductive interactions and increases the performance of both. Additionally, it separates prefetch accesses and what demand misses the main thread still experiences onto separate cores, potentially increasing total memory bandwidth.

5.1.2 Chunk size and the number of prefetch threads

Once the p-slice is generated, two parameters remain to be set – the chunk size and the number of prefetching threads (i.e., the number of cores to use). Changing the chunk size changes the level of the cache hierarchy prefetching will target and how completely we try to fill it, but it also determines how much swapping overhead occurs. For small chunk sizes, the frequent swaps might dominate the gains from prefetching.

Increasing the number of prefetching threads can improve performance if the prefetching threads have difficulty keeping up with the main thread, since using additional cores to prefetch increases the amount of memory level parallelism the system can utilize. However, allocating more cores to prefetching increases power consumption and also means more cores are unavailable to run other threads.

To understand the interplay of chunk size, compute intensity, and number of prefetch threads, we measure throughput for our microbenchmark with a 32 MB working set while varying the chunk size and the number of prefetch threads.

Figure 5 shows the results for our three processors. The figures on the left measure the impact of chunk size, while those on the right measure the effect of total core count. For the Core2Quad, a chunk size of 128 or 256 KB gives the best performance regardless of access pattern, but for other processors sequential accesses favor larger chunks between 64 and 256 KB, while 16 KB chunks perform best for random accesses.

The performance gains are again largest for the random access patterns, where there is limited benefit from hardware prefetching. For 16 KB chunks, inter-core prefetching improves Nehalem throughput by 3.75× and Opteron's performance by 6×.

The right hand graphs in Figure 5 show the impact of using multiple prefetching cores for the optimal chunk size from the left hand graphs. Adding additional prefetchers improves performance almost linearly up to four cores (three prefetchers) for seq-base on the Core2Quad and Opteron machines. Nehelam sees benefits with up to three cores. For seq-comp, conversely, one or two prefetchers give the best results. This is an expected result: When the main thread does little computation for each memory access, it is difficult for a single helper thread to execute as quickly as the main thread. For the random access patterns, results are similar.

It is not surprising that the optimal number of cores varies by memory access pattern and computation/memory ratio. However, the optimal chunk size also varies, especially by access pattern. This implies that the optimal chunk size depends not only on a particular cache size, but also at which level of the memory hierarchy the most misses are occurring, the effectiveness of the prefetcher, and the relative cost of migration/synchronization. For example, with random access, the program runs more slowly. This reduces the relative cost of migration, so smaller chunks are desirable. In fact, the migration overhead is amortized to the point that targeting the L1 cache is the best strategy for Nehalem and Opteron. For sequential access (because it is harder to amortize the swap cost and because the hardware prefetcher is addressing the L1 cache fairly well), the sweet spot seems to be targeting the L2 cache. Less obvious is why the Core2Quad, with its huge L2 cache, tails off at 1 MB chunk size. However, inter-core prefetching allows us to not only pre-fill caches, but also TLB entries. The Core2Quad has 256 TLB entries per core, just enough to hold 1 MB if there are absolutely

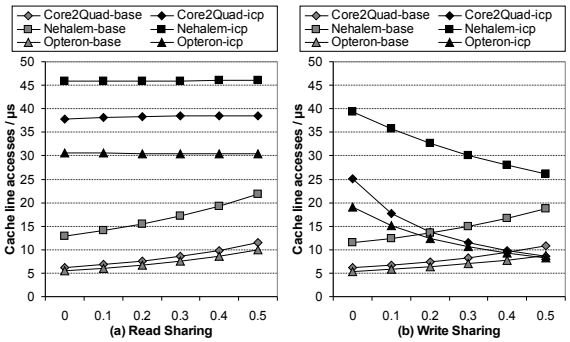


Figure 6. Performance improvement for different degree of read/write sharing using inter-core prefetching. Sharing factor of 0.3 means adjacent chunks share 30% of data.

no conflicts in the TLB. Therefore, the optimal chunk size is just a bit smaller than the maximum space supported by the TLB.

We have also found that prefetching each chunk in reverse is helpful in some cases. By prefetching the same chunk data, but in the opposite order that the main thread will access it, inter-core prefetching can target multiple levels of the cache hierarchy at once – even if a chunk is too big for the L1 cache. In this case, the last data prefetched will be in the L1 when the main thread arrives and tries to access it. In general, though, this technique is more difficult to apply, so we did not include it in our experimental results shown.

5.1.3 Inter-chunk sharing

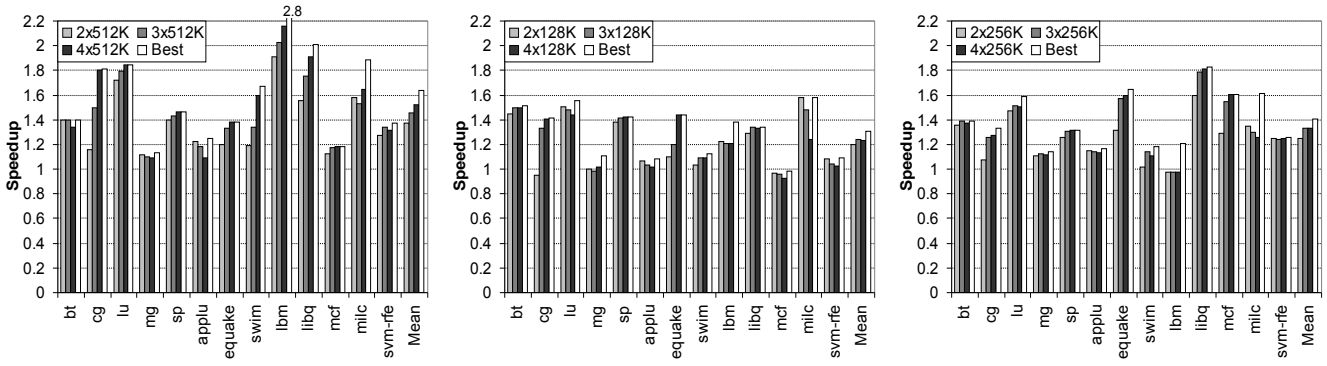
To quantify the effect of inter-chunk data sharing, we modify our microbenchmark by adding a configurable sharing factor. Figure 6 shows the effect of inter-chunk sharing across different machines in the rand-comp case for both read and write sharing. We use 16 KB chunks with one prefetch thread and vary the fraction of shared data from 0 to 50%. As expected, without inter-core prefetching, throughput improves with the sharing because of the increase in locality. With inter-core prefetching, throughput stays the same for read sharing, but degrades rapidly for write sharing. Cache to cache transfer latency is the critical factor for write sharing. Nehalem, having the best cache to cache transfer latency among the three machines, is the most tolerant of write sharing and sees improvement even when the main thread and prefetcher share half their data. Core2Quad, with a private cache design, has expensive cache to cache transfer operations, so inter-core prefetching hurts performance when more than 40% of data is shared.

5.2 Application Performance

This section examines the effectiveness of inter-core prefetching on real applications, with the p-slices generated as described in Section 3. Figure 7 displays inter-core prefetching's impact on our benchmark suite. The first three bars represent results for a single chunk size (that gave good performance across the entire benchmark suite) per architecture, varying the number of cores. The last bar gives the result that uses the best chunk size and core count per application (out of the 12 combinations– 2, 3, 4 cores and 128, 256, 512, 1024 KB chunks).

Overall, prefetching improves performance by between 20 and 50% on average without tuning the chunk size or core count for each benchmark. Per-benchmark tuning raises performance gains to between 31 and 63%. A further step would be to tune the parameters on a loop-by-loop basis.

Performance gains are largest for Core2Quad, because it has the largest cache miss penalty. For some applications, the gains are far above the average: Prefetching speeds up LBM by 2.8× (3 cores and 1024 KB chunks) on the Core2Quad and MILC sees



(a) Core2Quad (b) Nehalem (c) Opteron

Figure 7. Inter-core prefetching provides good speedups (between 20 and 50% on average) without tuning the chunk size and thread count on a per-application basis. With that level of tuning, performance rises to between 31 and 63% (the "best" bars).

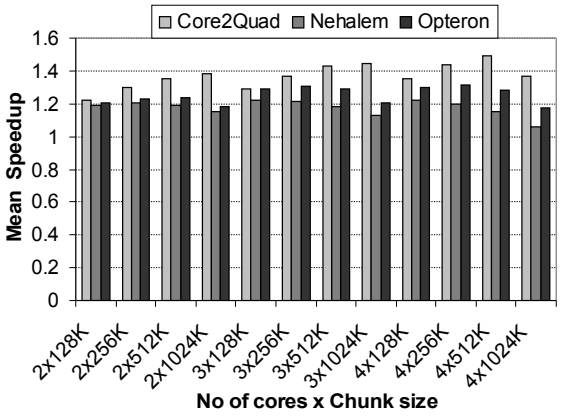


Figure 8. Mean speedup across all benchmarks for different combinations of chunk size and number of cores

improvements of nearly 1.6× for Nehalem. Even a complex integer benchmark like MCF gets 1.6× on Opteron. MCF also shows the impact of microarchitectural differences across machines on performance improvements. Unlike Opteron, Nehalem does not see any improvements and Core2Quad sees moderate speedup of 20%. This variation is due to the difference in cache hierarchy, window size, hardware prefetcher implementation, coherence protocol, etc. across these machines. To understand these differences, we measure the percentage of execution time covered by loops selected for inter-core prefetching (measured in the baseline case). Nehalem and Core2Quad spend between 44 and 47% of execution in those loops. Opteron spends 64%, which accounts for the larger overall impact on performance.

The effectiveness of increased prefetching threads varies widely among applications. For some (e.g., CG) additional prefetching threads are useful on all three architectures. For CG running on Nehalem with one prefetch thread, the main thread has to wait for the prefetch thread 89% of the time. With two and three prefetch threads, that number goes down to 25% and 0.4%, respectively. We see similar data for CG on Core2Quad and Opteron. For others (e.g., applu), adding threads hurts performance. If thread count has a strong impact on performance, the trend tends to be consistent across all three processors. The optimal number of cores is a function of the ratio of memory stalls to computation, which tends not to change across architectures.

Figure 8 shows results averaged across all benchmarks, for a wider range of chunk sizes for the three machines. We see again from these results that speedups are somewhat tolerant of chunk size, although there are some poorer choices to be avoided.

5.3 Energy Considerations

Inter-core prefetching has two competing effects on application energy consumption. The technique increases performance, which would reduce energy consumption if power remained constant. However, using multiple cores for execution increases power consumption both because multiple cores are active and because the throughput of the main thread increases.

To measure the total impact on energy, we measure total system power and energy with a power meter. The measurement includes everything in the system, including the power supply and its inefficiency, so the results are not directly comparable with simulation studies that focus only on processor power.

Figures 9 and 10 show results for Nehalem and Core2Quad. We were unable to measure results for the Opteron because of system administration issues. Our measurements show that the applications running without prefetching require 282 W (Nehalem) and 381 W (Core2Quad) on average (i.e., total system power increased by 42 W and 87 W while running the application compared to an idle system). Inter-core prefetching with a single prefetcher increases power consumption by 14 W (Nehalem) and 19 W (Core2Quad), and adding another prefetching thread requires an additional 6 W.

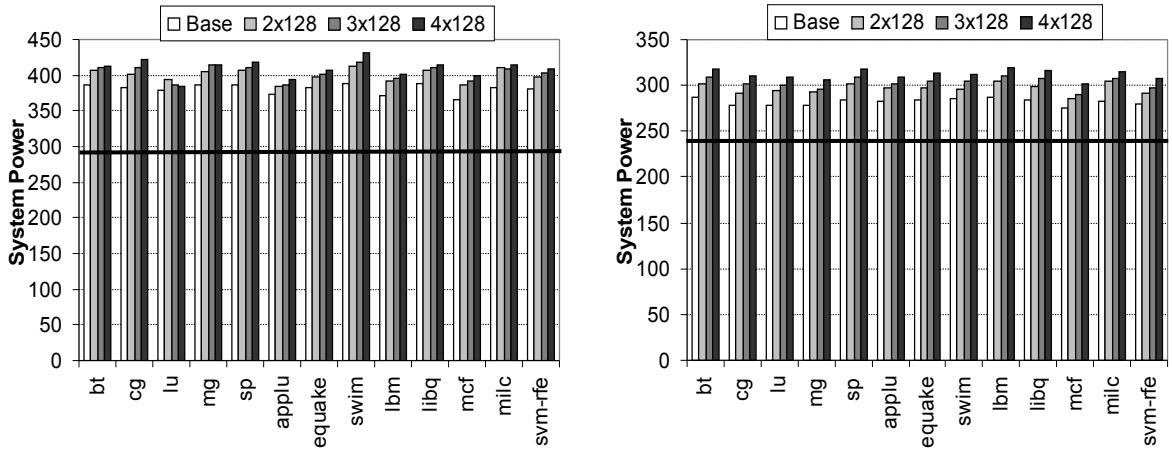
In terms of energy, the performance gains that inter-core prefetching delivers more than compensates for the increased power consumption. Per-application energy drops by 11 and 26% on average for the two architectures, and as much as 50% for some applications.

5.4 Comparison with SMT Prefetching

Previous work on helper-thread prefetching focused on SMT machines. This section compares that approach with inter-core prefetching.

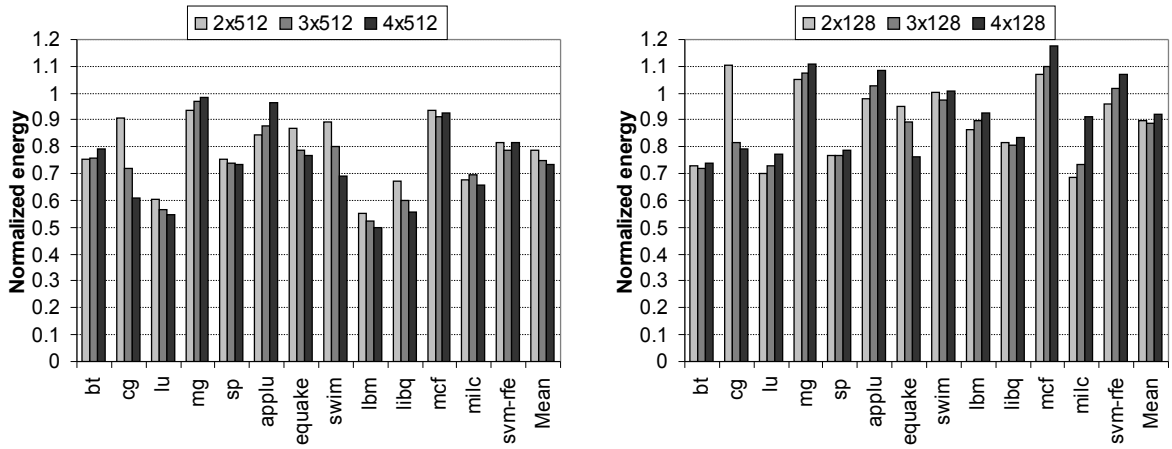
To provide a reasonable comparison, we use the same chunk and p-slice generation techniques that we use for inter-core prefetching, but we run the p-slice on the second context of one of Nehalem's SMT cores. Since the main thread and prefetching thread will run concurrently on the same core, there is no need for context switches or the associated overhead.

Figure 11(a) compares the performance of applying prefetching across SMT contexts and applying it across cores for Nehalem using our microbenchmark. We see from these results that SMT



(a) Core2Quad power (b) Nehalem power

Figure 9. Inter-core prefetching increases power consumption slightly compared to the non-prefetching version of the application, but the prefetching threads consume much less power than the main thread. The horizontal line denotes idle power on each system.



(a) Core2Quad energy (b) Nehalem energy

Figure 10. Inter-core prefetching's performance gains counteract the increase in power consumption it causes, resulting in a net reduction in energy of between 11 and 26%. Measurements in the graph are normalized to the application without inter-core prefetching.

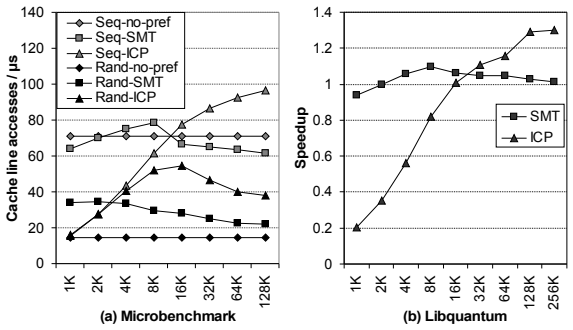


Figure 11. Comparing inter-core prefetching and SMT prefetching for different chunk sizes in Nehalem. On SMT, smaller chunk sizes give better performance because migration is not necessary and the prefetcher can target the L1 cache. However, using multiple cores still results in better overall performance.

prefetching favors smaller chunk sizes since they minimize interference between the helper thread and the main thread in the L1 cache. With very small chunks, of course, CMP prefetching is not effective because of the cost of swapping. However, with large chunks inter-core prefetching easily outperforms the best SMT result. Figure 11(b) shows the same comparison for one of our Spec2006 benchmark, *libquantum*. This application has regular memory access pattern, and so it follows the trend of sequential access in Figure 11(a).

Inter-core prefetching, even on architectures where SMT threads are available, benefits from the absence of contention for instruction execution bandwidth, private cache space, TLB entries, cache bandwidth, etc. This lack of contention allows the prefetcher threads to run very far ahead of the main thread and makes it easier to fully cover the latency of cache misses. As a result, inter-core prefetching is vulnerable to neither useless late prefetches nor early prefetches whose data are evicted before the main thread accesses it. This also explains why the hardware prefetcher cannot

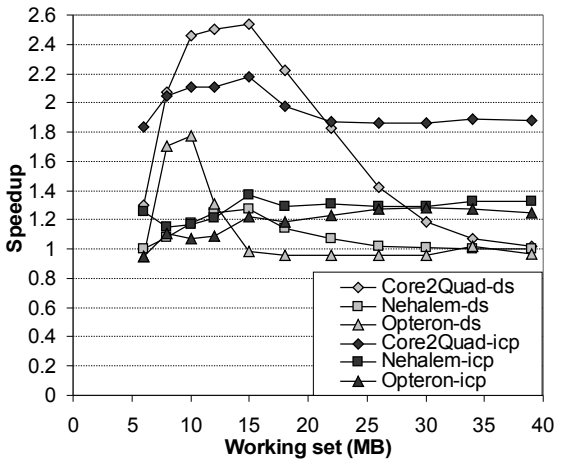


Figure 12. Inter-core prefetching and data spreading are complimentary techniques. Data spreading delivers better performance and power savings for small working sets, but for larger working sets, inter-core prefetching is superior. Speedups are relative to a conventional implementation.

completely eliminate the need for inter-core prefetching in case of the fully predictable (e.g. sequential) access pattern. In addition, the hardware prefetcher will not cross page boundaries, while inter-core prefetching ignores them.

5.5 Comparison to data spreading

Inter-core prefetching shares some traits with previous work on data spreading [16], which also uses migration to eliminate cache misses. Under that technique a single thread periodically migrates between cores, spreading its working set across several caches. For applications with regular access patterns and working sets that fit in the private caches, data spreading converts main memory accesses into local cache hits. For more irregular access patterns, it converts them into cache-to-cache transfers. The results in [16] show that it can improve performance by up to 70%.

Inter-core prefetching and data spreading are complimentary, but they differ in several ways. First, data spreading is only useful when the working set of the application fits within the aggregated caches. Inter-core prefetching works with working sets of any size. Second, data spreading uses a single thread, and, as a result, does not incur any power overheads: only one core is actively executing at any time. Inter-core prefetching actively utilizes multiple cores. Finally, as datasets grow, data spreading requires more and more cores. In many cases, inter-core prefetching gets full performance with 2 cores, even with very large data sets.

Figure 12 compares inter-core prefetching and data spreading on Jacobi. The figure measures speedup relative to a conventional implementation on the same architecture. The data show that data spreading provides better performance when Jacobi's data set fits within the aggregate cache capacity of the system. Inter-core prefetching offers better performance for larger data sets. The technique also does not require larger private caches. This helps to effectively apply inter-core prefetching in state of the art multicores with moderately sized private L2 caches.

6. Conclusion

This paper describes inter-core prefetching, a technique that allows multiple cores to cooperatively execute a single thread. Inter-core prefetching distills key portions of the original program into prefetching threads that run concurrently with the main thread but on a different core. Via lightweight migration, the main thread follows the prefetching threads from core to core and finds prefetched data waiting for it in its caches. Our results show that inter-core prefetching can speed up applications by between 31 and 63%, depending on the architecture, and that it works across several existing multi-core architectures. Our results also show that, although it uses multiple cores, it can reduce energy consumption by up to 50%. Inter-core prefetching demonstrates that it is possible to effectively apply helper thread-based techniques developed for multithreaded processors to multi-core processors, and even overcome several limitations of multithreaded prefetchers.

Acknowledgments

The authors would like to thank the anonymous reviewers and James Laudon for many useful suggestions. They would also like to thank Sajia Akhter for help with some of the graphics. This work was supported by NSF grants CCF-0702349 and NSF-0643880.

References

[1] T. M. Aamodt, P. Chow, P. Hammarlund, H. Wang, and J. P. Shen. Hardware support for prescient instruction prefetch. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004.

[2] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th annual international symposium on Computer architecture*, 2001.

[3] J. A. Brown, H. Wang, G. Chrysos, P. H. Wang, and J. P. Shen. Speculative precomputation on chip multiprocessors. In *In Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation*, 2001.

[4] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd annual International Symposium on Computer Architecture*, June 2006.

[5] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the international symposium on Computer Architecture*, May 1999.

[6] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, (5), May 1995.

[7] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[8] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic speculative precompuation. In *Proceedings of the International Symposium on Microarchitecture*, December 2001.

[9] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the International Symposium on Computer Architecture*, July 2001.

[10] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing*, 1997.

[11] A. Garg and M. C. Huang. A performance-correctness explicitly-decoupled architecture. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[12] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, 2005.

[13] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherence protocols on x86-64 multicore smp systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[14] K. Z. Ibrahim, G. T. Byrd, and E. Rotenberg. Slipstream execution mode for cmp-based multiprocessors. In *Proceedings of the*

9th International Symposium on High-Performance Computer Architecture, 2003.

[15] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the international symposium on Computer Architecture*, June 1990.

[16] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Software data spreading: leveraging distributed caches to improve single thread performance. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, 2010.

[17] D. Kim, S. Liao, P. Wang, J. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. Shen. Physical experiment with prefetching helper threads on Intel's hyper-threaded processors. In *International Symposium on Code Generation and Optimization*, March 2004.

[18] D. Kim and D. Yeung. Design and evaluation of compiler algorithm for pre-execution. In *Proceedings of the international conference on Architectural support for programming languages and operating systems*, October 2002.

[19] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading". *IEEE Transactions on Computers*, September 1999.

[20] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *Proceedings of the conference on Programming Language Design and Implementation*, October 2002.

[21] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, 2005.

[22] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th annual international symposium on Computer architecture*, July 2001.

[23] P. Marcuello, A. González, and J. Tubella. Speculative multithreaded processors. In *12th International Conference on Supercomputing*, November 1998.

[24] P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, February 2004.

[25] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, 1992.

[26] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. 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.

[27] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. Nu-minebench 2.0. technical report. Technical Report CUCIS-2005-08-01, Center for Ultra-Scale Computing and Information Security, Northwestern University, August 2006. URL http://cucis.ece.northwestern.edu/techreports/pdf/CUCIS-2004-08-001.pdf%.

[28] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[29] J. E. Smith. Decoupled access/execute computer architectures. In *ISCA '82: Proceedings of the 9th annual symposium on Computer Architecture*, 1982.

[30] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the International Symposium on Computer Architecture*, June 1995.

[31] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: improving both performance and fault tolerance. *SIGPLAN Not.*, 35(11), 2000.

[32] W. Zhang, D. Tullsen, and B. Calder. Accelerating and adapting precomputation threads for efficient prefetching. In *Proceedings of the International Symposium on High Performance Computer Architecture*, January 2007.

[33] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the International Symposium on Computer Architecture*, July 2001.