Software Data Spreading: Leveraging Distributed Caches to Improve Single Thread Performance

Md Kamruzzaman Steven Swanson Dean M. Tullsen

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

Abstract

Single thread performance remains an important consideration even for multicore, multiprocessor systems. As a result, techniques for improving single thread performance using multiple cores have received considerable attention. This work describes a technique, *software data spreading*, that leverages the cache capacity of extra cores and extra sockets rather than their computational resources. Software data spreading is a software-only technique that uses compiler-directed thread migration to aggregate cache capacity across cores and chips and improve performance. This paper describes an automated scheme that applies data spreading to various types of loops. Experiments with a set of SPEC2000, SPEC2006, NAS, and microbenchmark workloads show that data spreading can provide speedup of over 2, averaging 17% for the SPEC and NAS applications on two systems. In addition, despite using more cores for the same computation, data spreading actually saves power since it reduces access to DRAM.

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

General Terms Languages, Performance

Keywords chip multiprocessors, compilers, single-thread performance

1. Introduction

Hardware parallelism has become pervasive. Every high-performance processor is now multicore, and multi-socket configurations are common even on personal machines. Current mainstream offerings contain 4 to 16 execution cores on each processor chip [1, 27], and there is no sign of the trend toward higher core counts slowing.

This move allows users to benefit from the availability of increasing transistor counts in the presence of thread-level parallelism. However, it would be a mistake to assume that single-thread or few-threads performance no longer matters in this new era. In fact, just the opposite is true, and in some cases its importance will increase: Amdahl's law dictates that as architectures become more parallel, the inherently serial portions of applications will eventually limit the performance of all but the most embarrassingly parallel codes.

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0019/10/06...$10.00

As available hardware parallelism continues to increase, we anticipate the following three phenomena: (1) single thread performance will remain important, (2) in many machines some (if not most) of the cores will be idle, and (3) manual parallelization for most programs will remain difficult. As a result we need automatic, generally applicable techniques for accelerating single threads on many-core systems. Automatic compiler-generated parallelism is one solution, but is still not effective for a large number of applications. Therefore, we also need to pursue non-traditional parallelization techniques – those which provide parallel speedup (many cores run an application faster than a single core) without actually offloading parallel computation to cores – techniques like helper threading, speculative multithreading, etc.

These techniques exploit the computational ability of other cores to accelerate the original thread. In this research, we introduce the concept of *software data spreading* which exploits the capacity of remote caches to accelerate a single thread. By migrating the thread among multiple cores with distinct caches, we can utilize the combined cache space of all of those cores. Aggregating cache capacity is of growing importance: Although total on-chip cache capacity continues to grow with Moore's law, the *per-core* cache capacity is not keeping pace (e.g., 4MB total for the Intel Core 2 Duo at introduction vs. 8MB total for a quad-core Nehalem chip). Previous work [4, 10, 22, 25] has attempted to aggregate cache space through specialized hardware support.

Migrating a thread among multiple cores while it accesses large data structures provides three primary advantages. First, when the thread repeats a memory access pattern (e.g., during multiple instances of a loop), we force the thread to periodically migrate between caches in the same pattern each time. As a result, the thread tends to access the same portion of the data when it is running on a specific core, resulting in lower miss rates. Second, even when the computation moves completely unpredictably through the data structures, periodic migrations result in more of the data structure residing in the combined caches. As a result, many DRAM accesses become faster (and more power efficient) cache-to-cache transfers. Finally, judicious migration while accessing very large data structures (that tend to completely over-write the cache or caches) can, in some cases, shield other data and allow it to remain in another cache.

We have developed a compiler-based, software-only data spreading system that identifies loops which have large data footprints and suitable sharing patterns (e.g., high sharing between instances of the same loop) and spreads those loops and the data they access across multiple cores, both within a chip multiprocessor or across multiple dies or sockets. Data spreading can be applied to any system, multicore or multiprocessor, with private L2 or L3 caches. Our experiments with multiple Intel and AMD multiprocessor systems show that data spreading can speed up a range of applications by an average of 17%. Most impressive, data spreading achieves this speedup without any extra power consumption. In fact, in the best case, it significantly reduces power by avoiding DRAM accesses. Finally, data spreading requires no new hardware support and, since it relies on the system's default caching behavior, does not threaten correctness.

This paper is organized as follows. Section 2 discusses related research. Section 3 describes the motivation and basic data spreading approach. Details of our experimental methodology are presented in Section 4. Section 5 describes the actual data spreading algorithm, evaluating several design options and presenting initial results. Section 6 examines software data spreading results more closely across different systems and working set sizes. It also examines its power efficiency and applicability to multicores. Section 7 concludes.

2. Related Work

A number of compiler and architecture techniques have been proposed to leverage available parallelism to speed up a single thread, targeting either multithreaded processors or chip multiprocessors. This includes speculative multithreading architectures [17, 26, 30, 31], in which separate portions of the serial execution stream are executed (speculatively) in parallel, and it is only discovered later whether the parallel execution was successful (i.e., it did not violate any data or control dependences). More relevant to current multicore and multithreaded processors, however, is the work on helper threading and speculative multithreading.

Helper threads [5–7, 13, 14, 21, 23, 37] execute in parallel with the original thread, enabling that thread to run faster. Typically, those threads precompute load addresses to prefetch data into a shared level of the cache hierarchy before the main thread issues the load instruction. In many cases, the prefetching thread is distilled from instructions in the original thread [7, 37]. Kim and Yeung [14] present algorithms for generating helper thread code statically in the compiler, and Zhang, et al., [36] describe the generation of helper threads in a dynamic compilation system.

Event-driven compilation systems [34–36] use idle cores to invoke a compiler at run-time to perform optimizations in response to events detected by hardware performance monitors. An event-driven compiler could apply data spreading transparently at run time.

Chakraborty, et al., [16] present a technique called Computation Spreading, which also tries to leverage other caches via migration. They migrate threads so that some cores execute exclusively operating system code, and others execute exclusively user code. In this way, they get separation of data that is not typically shared over short time frames, and co-location of data more likely to be shared. However, they do not achieve the spreading effect that is the primary contributor to our speedups.

Cooperative Caching [4] is an architectural technique with the same goal as data spreading – using caches from neighboring cores to support the execution of a single thread. They do this by allowing caches to store data that have been evicted from other private caches. Thus, caches that are lightly used or idle can act as large victim caches [12]. However, this can only transform misses into cache-to-cache transfers. In many cases, data spreading effectively transforms misses into local hits. Other work [10, 22, 25] has suggested blurring the distinction between private and shared caches even further. However, each of these techniques require changes to the architecture. Further, none of these hardware techniques work across multiple chips.

There is a tremendous body of work on detecting parallelism in sequentially-programmed code. In some ways our work is similar in that we also detect the independence or lack of independence of data touched by different segments of code, in the process of applying our optimization. However, because the normal caching system ensures correctness, we can observe this behavior at a much coarser level and make decisions based on trends and tendencies rather than guarantees.

There have also been several projects that attempt to share resources across multiple cores in a single CMP to speed up a single thread. The work in [29] exploits migration to aggregate cache space, but again relies on hardware mechanisms. Our approach is software only and works on current systems. Core fusion processors [11] take a different approach and allow multiple cores to be dynamically combined into a single larger core. Conjoined core designs [18] allow two cores to share even the lowest-level (L1) caches. These approaches require significant changes to the hardware and do not involve thread migration for sharing.

Cache blocking [20, 28] is a compilation technique that reorders computation on a single core to increase locality and reduce cache misses. Data spreading could actually be complementary to techniques such as this, because they can reduce the reuse distance for only a subset of the accesses, and must still incur some capacity misses on any structure that does not fit in the cache.

3. Software data spreading

Software data spreading allows a single thread of computation to benefit from the private cache capacity of idle cores in the system, whether on the same processor or on other, idle sockets. As the thread executes, it moves from core to core, spreading its accesses across the caches. If we time the migrations correctly, the thread can either avoid misses in the private cache it happens to be using or have its misses serviced out of another core rather than from main memory. This results in reduced execution time and energy consumption, since accesses to main memory are both slow and power-hungry.

Data spreading works best in systems with large private caches (typically L2 or L3), spread across multiple cores, dies, or sockets. As shared caches face scaling limitations, we expect private caches (or caches shared among a subset of cores) to become more common. It also applies to any machine with multiple processors on separate dies, since each die includes a private (relative to the other dies or sockets) cache.

In the next three subsections, we give some examples of how software data spreading applies in different scenarios, describe our implementation of the data spreading mechanism, and then discuss its potential impacts on performance and power efficiency.

3.1 Examples

Algorithm 1 contains a pair of loops that are good candidates for spreading. It accesses two arrays, a and b, and we assume that the combined size of both arrays is roughly eight times the capacity of a single cache. To illustrate data spreading, we will assume (to keep the example simple) a CMP with only private caches.

Algorithm 1 – Simple example code Data spreading can accelerate this code if the working set does not fit in a single L2 cache.

```
for $i = 1$ to 100 do // Loop 0
  for $j = 1$ to 1000 do // Loop 1
    $a_j = a_{j-1} + a_{j+1}$
  end for
  for $j = 1$ to 2000 do // Loop 2
    $b_j = b_{j-1} + b_{j+1}$
  end for
end for
```

If this code executes on a single core, the cache miss rate will be very high, since Loops 1 and 2 will destructively interfere.

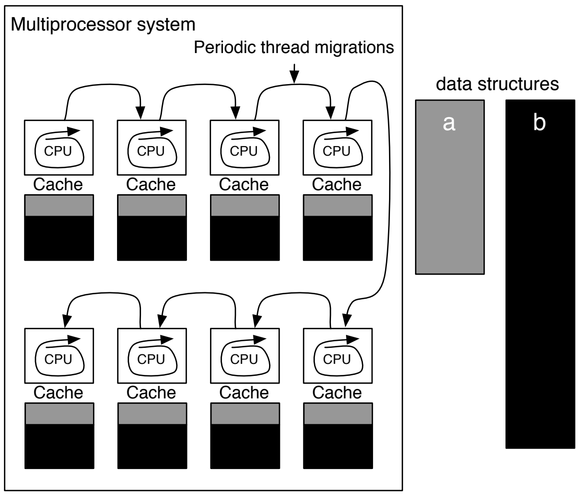


Figure 1. One iteration of the outer loop in Algorithm 1 with data spreading across 8 cores.

However, if we have an 8-core CMP, the aggregate capacity of the cores' private caches is enough to hold both arrays, and there will be no capacity misses. Our spreading technique allows us to perform this distribution without any hardware support. Figure 1 illustrates the distribution of data across the private caches in the system.

Data spreading may provide benefits even if the data structures are too large to fit in the entire on-chip cache space. It will still collect a larger portion of the data into the private caches. Alternately, we could spread as much of b as will fit across all but one of the caches, and isolate the rest of it to a single cache. Accesses to the spread out portion will be fast, while the remainder will be slower. If b is very large, then we can isolate execution of Loop 2 in a single core, while spreading Loop 1 to take advantage of the remaining caches. Loop 2 will "thrash" in its cache (this is unavoidable, since b is large), but Loop 1 will remain largely unaffected. Our compilation system does not currently support this last option.

Algorithm 2 – Data spreading can reduce the cost of misses for irregular access patterns.

```
for $i = 1$ to $100$ do
    $p = list$
    while $p \neq null$ do
        $p = p \rightarrow next$
    end while
    Shuffle(list)
end for
```

Software data spreading can also speed up irregular access patterns. Algorithm 2 traverses a linked list that is too large to fit in a single cache, then a second function shuffles the list. On a single core, most of the accesses would miss in the cache. With spreading, the number of misses to private cache remains mostly unchanged, but nearly all of them (assuming the working set fits in the combined caches) will be satisfied via cache-to-cache transfers, saving an expensive off-chip access. Current multicores do not typically support fast cache-to-cache transfers, so our experiments do not show large gains in this case; however, we expect that to change in future chips. Still, if the "shuffle" does not completely randomize the ordering, we will see gains even without fast cache-to-cache transfers.

3.2 Implementing data spreading

The only support our implementation requires is from the operating system: The OS must provide the means to "pin" a thread to a particular (new) core, and a mechanism to determine how many cores are available to the application. With this support, migrating from one core to another requires a single system call. This support already exists in most operating systems running on multicores or multiprocessors.

The main challenge in software data spreading is determining when to migrate. Our compiler profiles applications to identify the data-intensive loops it will spread. Then it adds code to count loop iterations and call the migration function periodically. Algorithm 3 shows the code from Algorithm 1 with the extra code for spreading across eight caches. We discuss the loop selection process and spreading policies in Section 5.

Choosing the loop's period requires balancing two opposing forces: Spreading data across as many cores as possible is desirable, since it will spread cache pressure out evenly and avoid spurious cache conflicts. However, a shorter period means additional thread migrations, which can be expensive.

Algorithm 3 – **Data spreading in action** The code in Algorithm 1 after the data spreading transformation.

```
for $i = 1$ to $100$ do
    for $cpu = 0$ to $7$ do
        MigrateTo(cpu)
        for $j = 125 \times cpu$ to $125 \times (cpu + 1)$ do
            $a_j = a_{j-1} + a_{j+1}$
        end for
    end for
    for $cpu = 0$ to $7$ do
        MigrateTo(cpu)
        for $j = 250 \times cpu$ to $250 \times (cpu + 1)$ do
            $b_j = b_{j-1} + b_{j+1}$
        end for
    end for
end for
```

3.3 The cost of data spreading

Like most optimizations, data spreading is not free. There are three potential costs that we must manage to make the technique profitable: its impact on the availability of other cores, the cost of thread migration, and its impact on power and energy consumption.

Performance impact on other threads Since data spreading increases the number of cores a thread is using, it could potentially interfere with other threads' performance. However, when idle cores are unavailable, or better used for other purposes, we can forgo data spreading – so the opportunity cost of using other cores is very low. We assume the main thread queries the OS to find the number of available cores. If it returns 0, the code runs without spreading enabled, and the only sources of overhead are the quick (it simply returns null in this case) and infrequent calls to the *Migrate_To* function.

Thread migration cost This is the primary cost for implementing data spreading in the systems we tested. GNU/Linux (starting with Linux kernel 2.6) provides an API to pin threads to processors, but it is an expensive operation. Linux 2.6.18 on an Intel Nehalem processor takes about 14μs to perform one migration. If the other core is in a sleep state, the cost could be even higher. The high cost of migration in current systems restricts our ability to employ data spreading successfully on a single CMP, as explored further in Section 6.6. Proposals for hardware migration support such

System Information	Intel Pentium-4	Intel Core2Quad	Intel Nehalem	AMD Opteron
CPU Model	Northwood	Harpertown	Gainestown	Opteron 2427
# of Socket×# of Die×# of core	4×1×1	2×2×2	2×1×4	2×1×6
Last Level Cache	2M	6M (per die)	8M	6M
Cache to cache transfer latency	300-500 cycles	150-250 cycles	120-170 cycles	210-215 cycles
Memory access latency	400-500 cycles	300-350 cycles	200-240 cycles	230-260 cycles
Migration cost	14μs	10μs	14μs	9μs
Linux Kernel	2.6.9	2.6.28	2.6.18	2.6.29

Table 1. The multiprocessor system configurations used to test data spreading.

Benchmark	Resident Memory	Benchmark	Resident Memory
Art_T	3 MB	BT_A	298 MB
Applu_T	20 MB	CG_A	55 MB
Equake_T	12 MB	LU_A	45 MB
Mcf_T	45 MB	MG_A	437 MB
Swim_T	56 MB	SP_A	79 MB
Art_R	4 MB	BT_B	1200 MB
Applu_R	180 MB	CG_B	399 MB
Equake_R	49 MB	LU_B	173 MB
Mcf_R	154 MB	MG_B	437 MB
Swim_R	191 MB	SP_B	314 MB
Libq_R	64 MB		

Table 2. Resident memory requirements for benchmarks.

as [3] could reduce this significantly. A migration that requires OS intervention can be made to cost on the order of 2μs on a 3 GHz processor with OS changes but no hardware support [32].

The other cost of migration, besides the overhead of transferring the thread context itself, is cold start effects – the cost of moving frequently accessed data into the new cache, the loss of branch predictor state, BTB state, etc. Typically, cache state is the most expensive to move. Data spreading, when done correctly, minimizes this cost by moving a thread to a location where future accesses are already present in the cache (and away from a core where they are not present).

Power cost In contrast to traditional parallelization and most of the non-traditional parallelization techniques described in Section 2, data spreading can have a positive effect on both power and energy consumption. Other techniques achieve speedups by executing instructions on otherwise unused cores, and those instructions can consume extra power and energy. The only extra instructions that data spreading executes are in the migration function that moves threads between cores. More importantly, only one core is actively executing at any time.

Most current multi-core processors lack the ability to power-gate or even voltage-scale individual cores. In that case, one core being active implies they are all powered (and therefore dissipating leakage power). An idle core uses less power than a running core, but that is true whether the same core is always idle or whether activity (and, therefore, inactivity) shifts from core to core. Recent processors, such as Nehalem [1], are able to voltage-scale individual cores, or even power-gate cores. This will lower the power consumption of idle cores even further, strengthening the power argument for data spreading. Power gating will increase the migration latency somewhat, but when data spreading is effective, the idle periods for each core are far longer than the time required to wake the core from sleep [15, 19]. We can reduce this cost further by predictively waking the core several loop iterations before we plan to migrate.

Section 6.4 quantifies the power benefits of data spreading.

3.4 Which cores to use

The largest gains from data spreading typically come from aggregating the largest caches. For example, on a multi-socket Nehalem architecture, we gain more from aggregating L3 caches across sockets than from aggregating L2 caches on-chip. This will depend on the application – if the working set fits in four L2 caches, but not one, it will only gain from spreading at that level. Because of the large working sets of the applications we study, we focus on spreading at the socket level, except for Section 6.6. We also find we typically gain from using the minimum number of cores to aggregate the caches, because this reduces the frequency of migration. So for example, with two Nehalem sockets, our best gains usually come from data spreading across two cores (one on each socket), rather than across all eight cores. For the Core2Quad, we use one core per die (two per socket), and on the Opteron, we use one per socket. Tuning data spreading for individual loops and to work across multiple levels of the memory hierarchy is the subject of future work.

4. Methodology

To evaluate our approach to data spreading, we implement it under Linux 2.6 on several real systems with Intel and AMD IA32 processors. The system configurations are given in Table 1. We compute the latency information for our machines by running microbenchmarks. The migration cost shown here is the latency to call the function *sched_setaffinity* that changes the CPU affinity mask, and causes thread migration. All experiments run under Linux 2.6. We use gcc 4.1.2 with optimization flag -O3 for all of our compilations, PIN 2.6 for profiling and analysis, and hardware performance counters to measure cache miss rates.

Our benchmark applications are a set of memory intensive applications from Spec2000 [8], and the serial version of NAS [2]. We also pick one Spec2006 [9] integer benchmark *Libquantum*, since it is easy to vary its working set size. To identify the memory intensive benchmarks we use the cycle accurate simulator, SMTSIM [33] (configured to roughly match one of our real experimental machines) to identify those workloads that achieve at least 75% speedup with a perfect L1 data cache. Our rationale for selecting this set of workloads is that if a workload shows little benefit from a perfect memory hierarchy, there is no reason to expect data spreading to offer any benefit. Furthermore, since it is a software-only technique, there is no danger of it penalizing workloads – if it is not useful for a particular application it should not be applied. Our system currently works on C code. We were able to convert Fortran77 code using an f2c converter, but we were not able to convert Fortran90 code, which excludes some of the SPEC benchmarks.

For the SPEC benchmarks, we profile using the train input, and experiment with the reference input – for some experiments we also run the train inputs, just to get more variation in working set size. When train performs better than ref, it is typically because the working set falls in the optimal range, rather than due to increased profile accuracy. For NAS, we profile using the *W* input, and experiment with both the *A* and *B* inputs. We do all the experiments

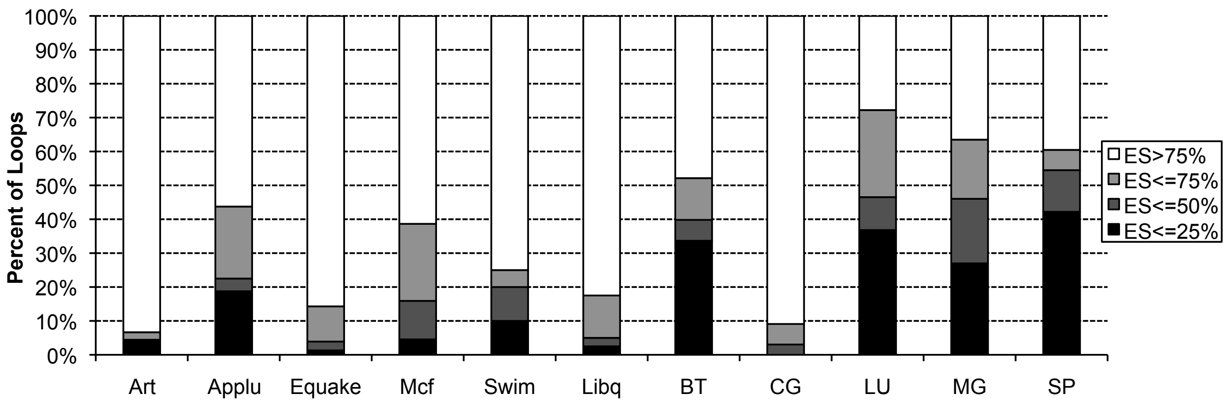


Figure 2. Epoch sharing breakdown of data intensive applications.

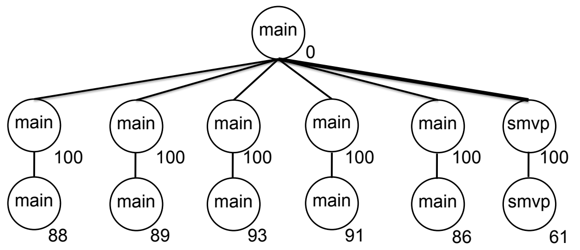


Figure 3. Loop nest tree annotated with epoch sharing values for the main kernel of *equake*.

multiple times and use the average execution time as our result. The results presented here are all normalized with respect to unmodified code.

5. Data Spreading in the Compiler

Our software data spreading compilation system involves a profiling step, a loop identification and selection stage, and a loop transformation step. This section describes and evaluates options for all three steps.

5.1 Profile collection and program structure analysis

Our system uses PIN [24] to statically identify loops, profile the program, and collect information about all the loops that make up a program's execution. Each time a loop executes, an event we term a loop *epoch*, we count the number of iterations it executes, the set of cache lines it accesses, and the number of memory accesses it makes each iteration. The profiling part takes 50 times that of normal execution time for a program.

We profile unoptimized code to simplify the process of identifying loops in the binary and connecting them with the source code. This is necessary as the compiler does optimizations like loop unrolling and function inlining. However, with some compiler support, profiling on optimized code would be possible. After we transform loops based on this analysis, full optimization is applied to generate code for our measurements.

For each loop, ℓ, we calculate its *total memory footprint* as the set of cache lines ℓ touches across all its epochs and its *epoch footprint* as the lines ℓ touches during the execution of a single epoch. We also define M_ℓ to be the maximum footprint size for any epoch of ℓ and N_l to be the number of iterations in that epoch.

We use this data to compute the *epoch sharing ratio* (ES) for each loop. Intuitively, epoch sharing is a measure of the degree to which multiple epochs of the same loop touch the same data. Loops with large ES values are good candidates for spreading, because once the first epoch fetches the epoch-shared data into the caches, the following epochs will likely reap significant benefit when they access the same data.

To compute the ES for a loop, we start by calculating, for each loop epoch, the fraction of that epoch's footprint that overlaps with the union of the footprints of all previous epochs of the same loop. The final ES is the average of this value over all the epochs of the loop. Formally, if ℓ has k epochs and the corresponding epoch footprints are $e_0, e_1, \ldots, e_{k-1}$, the epoch sharing of ℓ is

$$ES(\ell) = \frac{\sum_{i=0}^{k-1} S(\bigcup_{j<i} e_j, e_i)}{k-1}.$$

where

$$S(e_i, e_j) = \frac{|e_a \cap e_b| \times 100}{|e_a \cup e_b|}$$

Here $S(e_a, e_b)$ denotes the sharing between two epochs a and b with footprints e_a and e_b.

For example, in Algorithm 1, loops 1 and 2 each have epoch sharing of 100, since they always touch the same data. Loops with only one epoch have an epoch sharing of zero in our analysis. We find epoch sharing to be a very common characteristic of loops in data intensive applications. In Figure 2, we compute the epoch sharing breakdown of loops with more than 1 epoch. It clearly shows that loops reuse data heavily. Even for an irregular benchmark like *mcf*, we see that 61% of loops have epoch sharing of more than 75%.

We use the profile data to create a dynamic loop nesting tree for the application. Each node in the tree represents a single loop, and its children are the loops nested within it. We ignore function call boundaries when creating the tree (i.e., if a loop calls a function that contains loops, those loops are directly nested within the calling loop). Our analysis does not currently handle recursion, so we ignore back edges in the loop nest tree. Figure 3 shows the loop nest tree of *equake* from Spec2000 [8], annotated with epoch sharing values.

Clearly, data spreading is most effective on loops with large memory footprints and high epoch sharing. If an inner loop (child loop) has high epoch sharing, it is likely that the outer (parent) loop also exhibits high epoch sharing. However, it does not work to spread both loops, as the migrations in the inner loop will just override the outer loop migration commands. For example, the outer loop in Algorithm 1 inherits all of its epoch sharing from the inner loops, and spreading the outer loop would actually hurt performance. Even when both parent and child loop have significant epoch sharing, and the parent does not inherit the sharing from the child, the child is often a better candidate to spread (assuming its

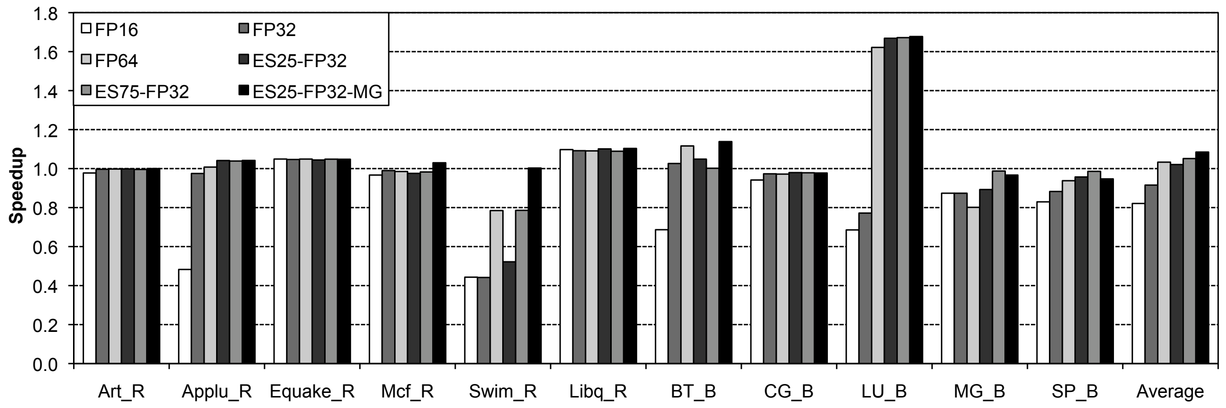


Figure 4. Wall clock speedup for all policies for the Core2Quad system

Policy	Description
FP16	Epoch foot print $\geq$ 16KB
FP32	Epoch foot print $\geq$ 32KB
FP64	Epoch foot print $\geq$ 64KB
ES25-FP32	Epoch foot print $\geq$ 32KB and ES $\geq$25%
ES75-FP32	Epoch foot print $\geq$ 32KB and ES $\geq$75%
ES25-FP32-MG	Epoch foot print $\geq$ 32KB and ES $\geq$25% and Epochs/sec $\leq$ 1000

Table 3. Loop selection policies.

Bench mark	Total Loops	FP16	FP32	FP64	ES25 FP32	ES75 FP32	ES25 FP32 MG
Art	84	32	32	31	20	20	20
Applu	199	23	23	19	12	11	12
Equake	121	27	27	27	6	6	6
Mcf	70	22	19	19	11	7	9
Swim	69	16	14	12	9	9	9
Libq	63	16	16	16	14	9	14
BT	243	56	52	50	43	43	43
CG	63	30	26	20	14	14	14
LU	190	41	35	27	15	15	15
MG	82	15	15	15	8	5	6
SP	333	82	82	76	72	72	72

Table 4. Number of selected loops for different selection policies.

working set exceeds the cache size). Currently, we always spread the child in this situation.

5.2 Baseline spreading slgorithm

Our baseline algorithm takes three parameters to do loop selection – an epoch sharing threshold, $E_{\min}$, a minimum footprint size, $F_{\min}$, and a maximum migration frequency. We examined other inputs, such as sharing with siblings, etc., but found that our best algorithms used just these three. The algorithm examines the nodes of the loop nest graph in reverse topological order, starting at the leaves and working toward the root. The algorithm selects a loop for spreading if a) none of its descendants have been selected and b) its epoch sharing and epoch footprints are larger than $E_{\min}$ and $F_{\min}$, respectively. Typically, $F_{\min}$ would be set to a value smaller than the cache, given that the profiled working set is not necessarily indicative of the working set size of future runs.

Once the candidate loops are finalized, we do simple source to source transformations. For loops with known iteration bounds before entering the loop, we spread it using Algorithm 3. For loops with unknown iteration bounds, we need to know the expected iteration count – N_l from the profile.

5.3 Candidate loop selection

The basic algorithm described above provides three parameters that we can tune to improve performance. Table 3 lists the settings we evaluate. Table 4 gives the number of selected loops for each policy on our benchmarks, and shows how loops are filtered across different policies. For instance, FP16 admits just 24% of all loops, indicating that there are a significant number of loops with very small footprints. Policy ES25-FP32-MG is more restrictive, and includes only 15% of all static loops.

Figure 4 shows the performance improvement for all six policies shown in Table 3, run on a 2-socket (4 die) Core2Quad machine. The first three policies filter loops based solely on footprint size. Allowing loops with very small foot prints (<16KB) degrades performance by 18%, but performance improves by 3% on average if we require footprints to be at least 64KB.

The next three policies use the epoch sharing threshold to filter loops that do not benefit from data spreading because they touch different data during each epoch. Although without ES considerations, the FP64 limit outperformed FP32, we find that using a more liberal policy (FP32) and then letting the ES restriction pare down the list was preferable. Limiting epoch sharing to at least 25% works well, and the ES25-FP32 policy provides a small speedup (compared to a slowdown for FP32). Increasing the ES limit to 75% reduces the slowdown of two benchmarks– *Swim* and *MG* by filtering some loops that cause too frequent migrations. The last policy, which also guards against too-frequent migrations, improves performance further (up to an 8% speedup), in large part by eliminating slowdowns where data spreading does not work well. This policy eliminates loops whose epochs occur more than once per milisecond. This caps migration overhead at 5% of execution time. Ultimately, we see that a combination of large footprint, high ES sharing and avoiding too-frequent migrations yields the best candidates for spreading.

LU clearly gains the most from software data spreading. As we explore in Section 6, performance is highly sensitive to working set size. Unfortunately, *LU* is the only one of these applications with a significant data structure in the sweet spot – between the size of one and four caches. But we also get reasonable gains on *BT* and *libquantum*.

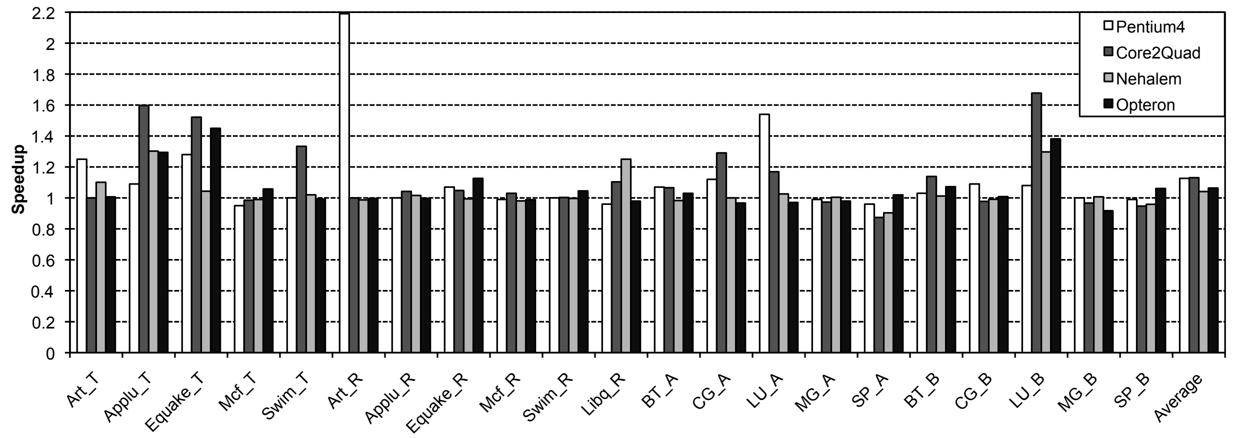


Figure 5. Speedup across different machines using the ES25-FP32-MG policy.

Swim and *Mcf* each see little or no benefit for any of the policies. *Swim* has a very large working set of around 191MB, so even using four caches (combined cache space 24MB) data spreading fails to keep sufficient useful data. In the case of *Mcf*, most of the data accesses are irregular, and we do not see benefit because of expensive cache-to-cache transfer in our real experimental system.

For current systems, migration overhead has a significant impact on the performance of data spreading. In our measurements, the split of user vs. kernel time provides insight into these costs. For policy FP16, the most aggressive, *LU* sees a 14% user time speedup, but the increase in kernel time due to OS migration code resulted in a wallclock slowdown of 31%. This implies techniques that reduce migration costs [3, 32] will increase the benefits of data spreading and compilers can apply the technique more aggressively. Section 6.5 explores this further.

5.4 Loop spreading policies

Once we have selected a loop for spreading, we must determine how and when to migrate. We considered several different policies, but the results in this paper only reflect one. That policy is also our simplest.

The *balanced* policy spreads loop iterations evenly across the available cores. Our results for more complex policies failed to show significant or consistent gains. For example, policies that considered the size of the cache more explicitly tended to reduce sharing between sibling loops. For example, the first one quarter of loop A tended to touch the same data as the first one quarter of loop B. With *balanced*, those data all go into the same cache. With other policies, we might migrate at different points in the two loops depending on how much other data was being touched. The results shown in this paper all use the balanced policy.

6. Understanding Data Spreading

This section strives to acquire a deeper understanding of data spreading, particularly in light of the uneven performance gains demonstrated in the previous section. It uses microbenchmarks, kernels, and one full benchmark running on a whole suite of real machines to examine the interplay of working set size and data spreading effectiveness. It also measures data spreading's impact on power and energy. Finally, it examines data spreading's sensitivity to migration overhead, especially for spreading within a CMP.

6.1 Diverse memory hierarchies

Results for our benchmark suite on four different machine architectures are shown in Figure 5. Here we see that each machine gets speedup from data spreading, but the speedups are not very predictable. In most cases, speedup is simply a case of whether or not the key data structures fit in the aggregated caches. Since each of these architectures has a different cache hierarchy, the results varied. We see, though, that across a diverse range of working sets, performance improved overall. Both the Pentium 4 and the Core2Quad systems achieve an average speedup of 13%.

6.2 Microbenchmarks

To further understand the sensitivity of these techniques to working set size, we focus on a more restrictive set of benchmarks that give us the ability to modify the working set size continuously. First, we create two microbenchmarks – one accesses data sequentially, the other chases pointers through memory at random. Both perform the same set of accesses on each iteration through an outer loop. We ran these benchmarks on four different machines, as shown in Figure 6. The Core2Quad results (Figure 6b) illustrate the phenomenon well. Without data spreading, when the working set fits in the cache, throughput is high, but it degrades quickly when the working set overflows the cache. With data spreading, there is a cost when the data fit in a single cache, and it asymptotically matches the baseline when the working set is very large. But in between, data spreading *significantly* extends the region where we maintain close to full throughput. This is true both for sequential access (where the hardware prefetcher is actively assisting performance) and random access (where it is not). All four machines show a similar effect.

Also notice that the data spreading curves do not drop as sharply as the baseline – we continue to get some performance even after the working set no longer fits fully in the aggregated caches.

6.3 Applications

We can perform similar experiments for two full applications with easily configurable working sets – the 2D Jacobi kernel and the SPEC2006 libquantum application. For these, we show speedup of data spreading over the baseline for a given working set size.

The Jacobi results are shown in Figure 7. The graph shows speedup for the region between the size of a single cache and the size of the aggregated caches that data spreading operates on. On either side of this "hump" data spreading offers no benefits. All the machines achieve speedups between 1.25 (Nehalem) and 3.5 (Pentium 4). The Pentium 4 does especially well because it has the highest latency to main memory. The Core2Quad delivers improvements over the widest range – from 5 to 35 MB.

Results for *libquantum* from the SPEC2006 Integer benchmark suite are in Figure 8. Libquantum is a more complex computation with multiple, and varied, spread loops. Although the complexity

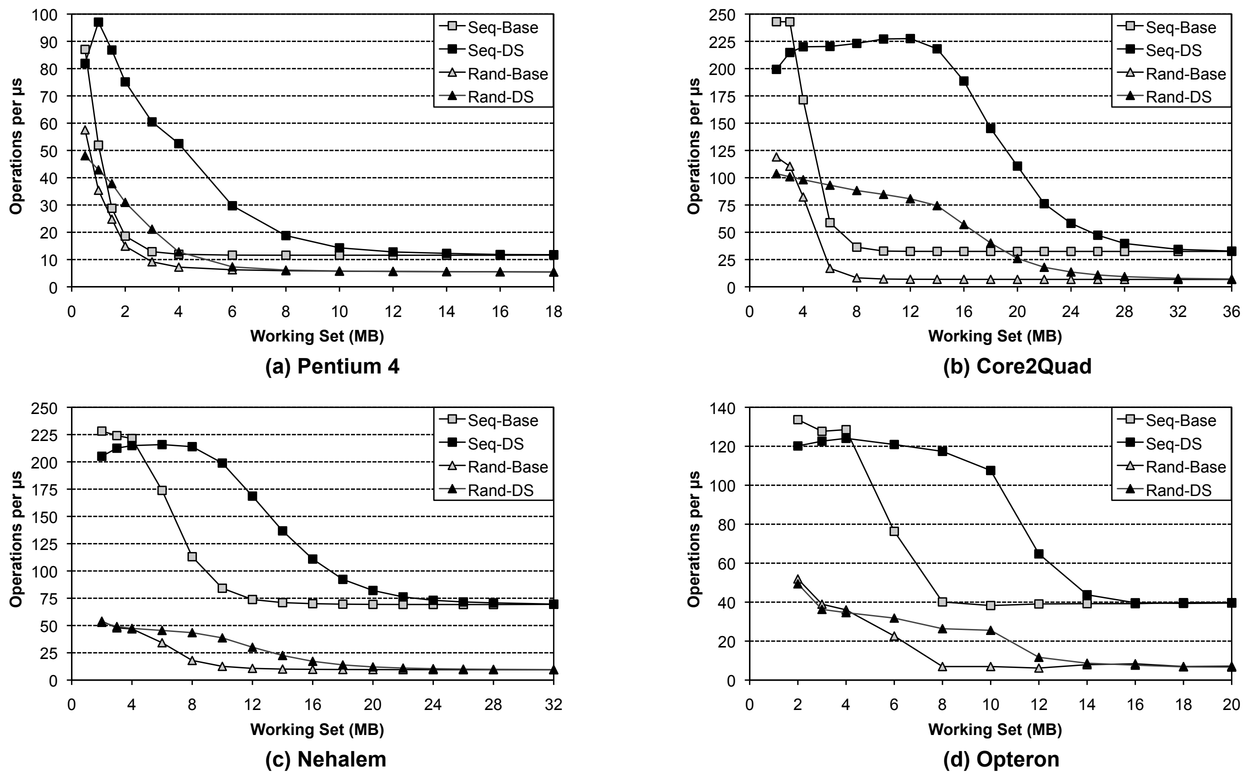


Figure 6. Data spreading throughput for sequential and random access for different machines. The '-Base' data are for code that does not perform data spreading.

of the application adds some noise to the results, the same trends emerge: Data spreading provides speedup after the local cache is exhausted, again extending the high-IPC range of the benchmark across a wider range of input sizes. The Core2Quad achieves over 3× speedup for this application.

These results all point to the same conclusion. While data spreading does not always provide speedups over non-spread code, it consistently makes the application's performance *significantly more robust* in the face of varying working set size. In the best case, data spreading achieves dramatic speedups – it achieves parallel-type speedups on parallel machines, without ever requiring parallel execution. This last point makes the optimization particularly attractive from an energy efficiency standpoint, which is explored further in the next section.

6.4 Power

On the surface, using multiple cores to execute a single thread does not appear to be a power-conserving optimization. However, just the opposite is true, since only one core is ever active at any time. Indeed, data spreading can *save* power (and energy) by eliminating accesses to DRAM.

We use a power meter on two of our experimental systems to quantify this effect for our random access microbenchmark. We measure total power of the entire system. Figure 11 shows the results. Where data spreading is effective, the power savings is dramatic. Spreading on the Core2quad saves up to 51W, or 81% of non-idle system power. For the Opteron, the savings are smaller – 13W, or 72% of non-idle power. The results also show the cost of the unnecessary migrations: At 2MB, migrations increase power consumption by up to 7W and 2W on the Core2Quad and Opteron, respectively.

From this graph, we see the dramatic impact on power when DRAM accesses are made unnecessary. This bodes for our real applications, where DRAM accesses are consistently reduced, in some cases by an order of magnitude. Figure 12 shows the normalized (to no data spreading) last level cache (LLC) misses for the Core2Quad system. These results correlate well with the speedup results in Figure 5. Note that spreading-induced coherence (cache-to-cache) transfers show up as misses, even though in some machines they will be faster (and lower power) than memory misses. Data spreading converts 80 and 90% of misses into local hits for *LU* (B input) and *applu* (train input), respectively. For 11 of our 21 benchmarks data spreading eliminates 20% of misses or more.

6.5 Migration overhead

Data spreading's primary overhead comes from the migrations it requires, especially the software overhead. Triggering these switches with the *sched_setaffinity* system call incurs a fixed overhead of 9 to 14 μs (Table 1), since it requires a trap to the operating system. If context switches occur too frequently, this overhead will eliminate the benefits of data spreading. Increasing the number of cores exacerbates this problems, since the frequency of context switching increases linearly with the core count. This is unfortunate, since spreading across more cores also increases the amount of cache capacity that data spreading can exploit.

To reduce the migration overhead, we developed a userspace context switch mechanism (*User-CS* in the figures, as opposed to *OS-CS*) that uses user space migrations (via `setcontext()` and `getcontext()`) and spin locks to reduce migration costs to just

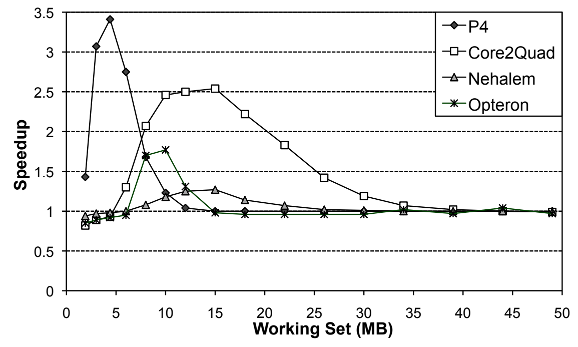


Figure 7. Speedup across different machines for 2D Jacobi with data spreading.

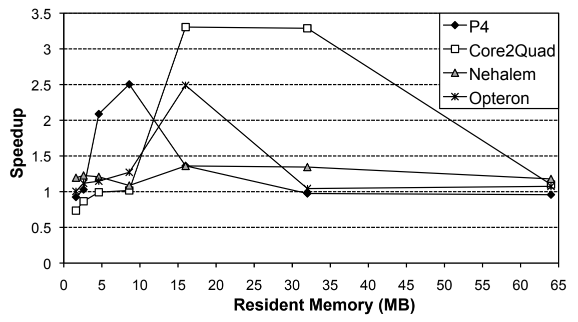


Figure 8. Speedup across different machines for libquantum.

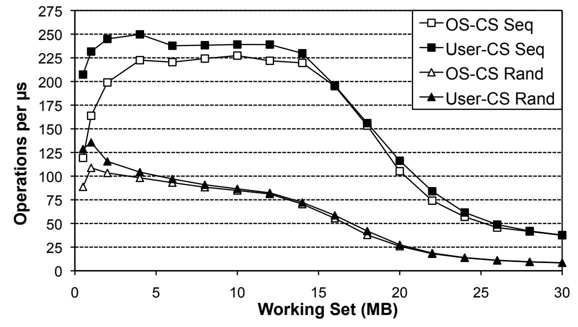


Figure 9. Userspace and OS directed thread migration in Core2Quad for the microbenchmarks.

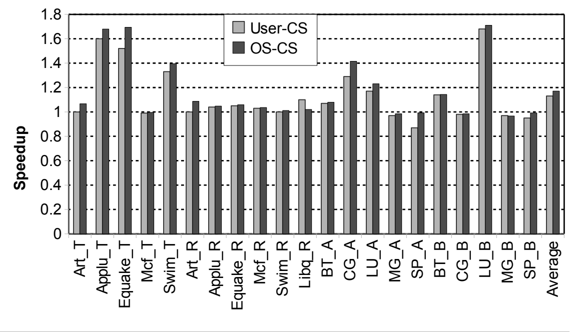


Figure 10. Speedup in Core2Quad using user space thread migration.

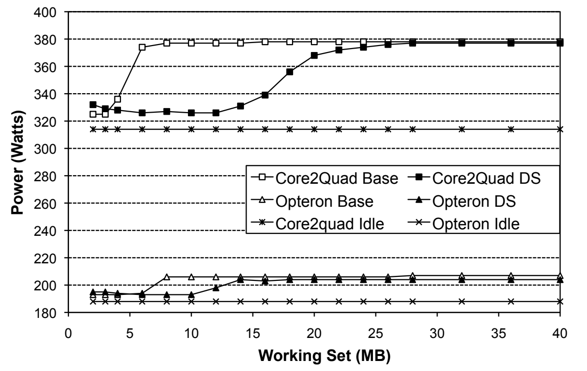


Figure 11. Power requirements for random access in Core2Quad and Opteron

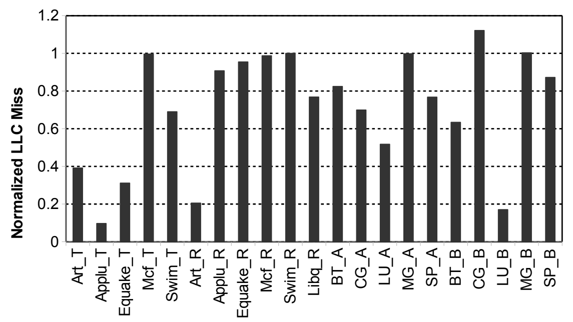


Figure 12. Reduction of last-level cache misses for Core2Quad.

1-3μs. Idle cores spin until called upon to wake up and acquire the context of the running thread.

Figure 9 compares the performance of both migration schemes for our microbenchmarks on the Core2Quad. The User-CS scheme increases performance by 5% for working set sizes up to 10 MB (12% for 4 MB or less) for sequential accesses and by up to 2% for working set sizes under 12 MB (6% 4 MB or less) for random accesses. The gain is larger for smaller working sets because migration occurs more frequently in this case.

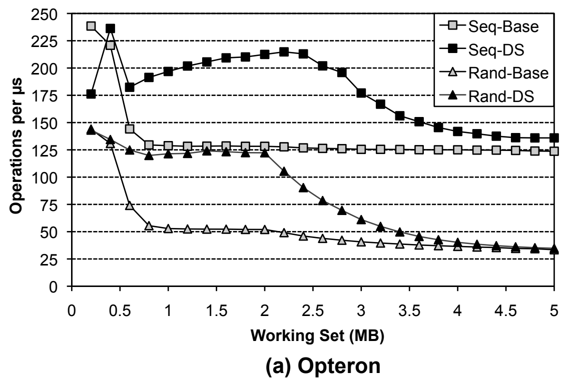
Figure 10 shows the comparison between User-CS and OS-CS for the full benchmark suite on the Core2Quad system. Overall, User-CS gives us a 17% speedup on average (compared to 13% average speedup by OS-CS). Again, user-CS provides the greatest new boost for small working sets (like the train input sets); it also reduces or eliminates performance degradation we previously observed when we were experiencing frequent migrations (e.g., SP).

The P4 machine also sees 4% additional speedup when we use user-level migration. However, we do not notice significant changes for Opteron or Nehalem. In these machines, we use two cores (vs. 4 cores used in Core2Quad and P4) which reduces the frequency of migrations by half.

User-level migration, as demonstrated here, is a reasonable approach when performance is the highest goal. However, the idle, spinning threads may reduce the power savings that data spreading can deliver.

6.6 Data spreading in CMPs

So far we have applied data spreading across sockets and dies, but data spreading can also be applied in multicore architectures. However the smaller private caches in these architectures mean that working sets small enough to fit into the aggregate on-chip private



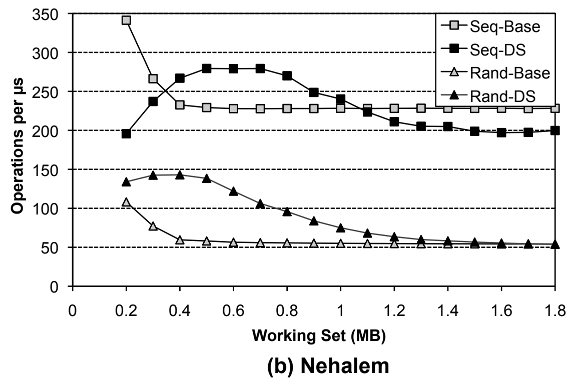


Figure 13. CMP data spreading throughput for sequential and random access on a 6-core Opteron and 4-core Nehalem

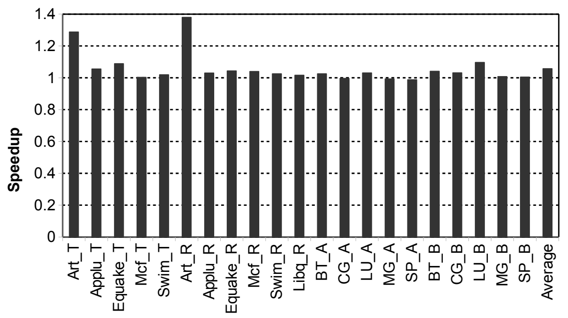


Figure 14. Data spreading performance on a single 6-core Opteron CMP.

caches will induce frequent migrations. Therefore, data spreading requires the use of a fast migration mechanism (like User-CS) to realize performance gains.

To evaluate CMP data spreading we ran experiments on the Nehalem (4 core) and Opteron (6 core) machines. Nehalem has 256K private L2 caches per core whereas the Opteron has 512K private L2 caches. Figure 13 shows the results for the microbenchmarks on both machines. The combined cache space in Nehalem is 1M, so the benefit comes when the working set lies between 300K and 1M. The combined cache space in Opteron is 3M and so data spreading improves performance over a broader region.

Figure 14 shows the performance improvement for different benchmarks on Opteron while spreading is deployed within the single socket. Overall we see 6% average performance improvement and, as expected, the speedup mainly occurs for smaller working sets.

7. Conclusion

This paper presents a new compiler optimization, software data spreading, targeted at multiprocessors and multicore processors. It uses thread migration to allow a single thread to utilize the space of multiple private caches. This allows the program to transform off-chip accesses into local cache hits when the data access pattern is highly repetitious. In the case where it is not predictably repeatable, it still turns DRAM accesses into cache-to-cache transfers. Using an approach that relies on profiling to identify loops with large data footprints and to characterize their sharing patterns, we identify for each application a small set of loops that are spread across multiple caches via migration. We achieve average speedups of 17% using four processors. Speedups on the SPEC2006 libquantum benchmark are as high as 3.3x, depending on the input size. Data spreading can also provide significant power and energy savings since it actively uses only one core at a time and can dramatically reduce memory accesses.

Acknowledgments

The authors would like to thank the anonymous reviewers for many useful suggestions. They would also like to thank Jeff Brown for frequent help with the simulation tools, Sajia Akhter for help with some of the graphics, and Nathan Goulding with the writing. This work was supported by NSF grants CCF-0702349 and NSF-0643880.

References

[1] First the tick, now the tock: Next generation Intel microarchitecture (Nehalem). 2008. Intel White paper.

[2] D. H. Bailey, E. Barzcz, L. Dagum, and H. D. Simon. NAS parallel benchmark results. *IEEE Concurrency*, February 1993.

[3] J. A. Brown and D. M. Tullsen. The Shared-Thread Multiprocessor. In *International Conference on Supercomputing*, June 2008.

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

[7] 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.

[8] J. L. Henning. SPEC CPU2000: Measuring cpu performance in the new millennium. *Computer*, July 2000.

[9] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, September 2006.

[10] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. Keckler. A NUCA substrate for flexible CMP cache sharing. In *International Conference on Supercomputing*, June 2005.

[11] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. *SIGARCH Comput. Archit. News*, May 2007.

[12] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Pro-*

ceedings of the international symposium on Computer Architecture, June 1990.

[13] 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.

[14] 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.

[15] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, February 2008.

[16] Koushik Chakraborty and Philip M. Wells and Gurindar S. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the international conference on Architectural support for programming languages and operating systems*, November 2006.

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

[18] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-core chip multiprocessing. In *Proceedings of the International Symposium on Microarchitecture*, December 2004.

[19] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[20] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimization of blocked algorithms. In *Proceedings of the international conference on Architectural support for programming languages and operating systems*, April 1991.

[21] 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.

[22] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, February 2004.

[23] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the International Symposium on Computer Architecture*, July 2001.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 conference on Programming Language Design and Implementation*, June 2005.

[25] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proceedings of the 32nd annual International Symposium on Computer Architecture*, June 2005.

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

[27] H. McGhan. Niagara 2 opens the floodgates. *Microprocessor Reports*, November 2006.

[28] A. McKeller and E. Coffman. The organization of matrices and matrix operations in a paged multiprogramming environment. *Communications of the ACM*, Mar. 1969.

[29] 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.

[30] 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.

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

[32] R. Strong, J. Mudigonda, J. C. Mogul, N. Binkert, and D. Tullsen. Fast switching of threads between cores. *SIGOPS Oper. Syst. Rev.*, April 2009.

[33] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference*, December 1996.

[34] W. Zhang, B. Calder, and D. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *International Symposium on Code Generation and Optimization*, March 2006.

[35] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, September 2005.

[36] 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.

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