Characterizing Memory Hierarchies of Multicore Processors

May 9, 2011

Christopher Celio
University of California Berkeley
586F Soda Hall
Berkeley, California 94704
celio@eecs.berkeley.edu

Aleah Caulin
University of California San Francisco
2340 Sutter St, Room N231
San Francisco, CA 94109
foxaleah@gmail.com

ABSTRACT

We have designed and implemented a series of micro-benchmarks that help to empirically characterize a given hardware platform's memory hierarchy. These small micro-benchmarks are able to deduce a variety of characteristics, including cache-to-cache transfer latency, inter-core bandwidth, and memory access latency. We have tested these micro-benchmarks on a variety of processors, and provide results from an i7 640M Arrandale notebook processor, an i7 975 EE Bloomfield processor high-end desktop processor, and a TILEPro64 embedded 64-core processor.

1. INTRODUCTION

1.1 Motivation

This section describes the main motivations for creating a set of micro-benchmarks that can characterize memory hierarchies.

1.1.1 Simulator Verification

One of the difficulties in implementing a new simulator is confirming its accuracy. A set of micro-benchmarks that empirically characterize a simulator's target memory hierarchy would help confirm that the stated parameters of the target platform are correct. It would also be useful if a simulator is attempting to target an existing platform. The micro-benchmarks could be run on both the simulated target and the physical machine the simulator is attempting to reproduce. If the performance results provided by the suite of micro-benchmarks are identical, then one can have more confidence in the simulator.

1.1.2 Published Hardware Specifications

A couple problems arise with using the published hardware specifications to characterize a given machine.

First, the specifications are often incomplete and do not provide all of the details an efficiency programmer or compiler may need to know to better tune the code. Such underspecified characteristics include details of the cache coherency protocol, including cache-to-cache (c2c) transfer latencies.

Second, the specifications may be incorrect or misleading. For example, the stated off-chip bandwidth value may be considered "unhelpful" if it is impossible for the caches to generate enough traffic to saturate the off-chip memory. Thus, an empirically deduced value for off-chip bandwidth would prove more useful in tuning algorithms.

1.1.3 Auto-tuning Awareness

It may be possible for just-in-time compilers and auto-tuners to produce better performing code if they has access to machine-specific characteristics provided by a suite of micro-benchmarks. It is also likely that future platforms will involve partitioning of resources, meaning that having the full specifications of the machine may not provide enough information to accurately predict code performance if the code is only run on a subset of the hardware.

In this same vein, it may also be possible to predict performance of a piece of software given a set of known machine characteristics provided by this set of micro-benchmarks.

1.2 Goals

The goals of this project were to 1) come up with a list of interesting and fundamental machine characteristics that affect code design and performance, 2) implement micro-benchmarks that are capable of empirically deducing these characteristics, and 3) gather results on a set of different machines to measure the accuracy of the implemented micro-benchmarks.

1.2.1 Parameters

We have explored the following parameters:

- number of cache levels
- data cache sizes at each level of the cache hierarchy
- access time at each level of the cache hierarchy
- off-chip bandwidth for unit-stride accesses

- cache-to-cache (c2c) transfer latency
- cache-to-cache (c2c) transfer bandwidth
- request bandwidth

These parameters provide an excellent first-order understanding of the memory hierarchy and its capabilities.

1.3 Challenges

Isolating a fundamental parameter of the machine can be difficult.

Virtual memory is a big obstacle. The translation lookaside buffer (TLB) reach provided by many machines can actually be smaller than the on-chip memory size. A typical processor uses 4 KB pages, thus a TLB that can hold only a few hundred entries provides a TLB reach on the order of a megabyte. Therefore, isolating TLB misses will be difficult as they can be expected to occur frequently for benchmarks with large working sets and irregular access patterns.

The operating system and its scheduling behavior can also obfuscate results. This includes process migration and scheduling distinct threads on the same core through mechanisms such as multithreading. Multithreading causes a contention for pipeline resources and cache space. While this is an issue for any threads running on the system, multithreading creates an asymmetry between two threads scheduled on the same core and two threads running on distinct cores.

Memory prefetching is a technique used to hide the latency of otherwise onerous off-chip accesses. Hardware prefetchers can be excellent at recognizing regular, strided memory accesses and prefetching future accesses to provide significant performance gains. However, memory prefetching can dramatically obscure results by hiding the latencies we are trying to measure, and requires memory accesses to be randomized to defeat prefetching.

Cache coherency protocols and memory controller behavior can also make isolating the parameters under study difficult. For example, the memory access penalties can depend on whether the page is currently open in the off-chip DRAM memory.

Turbo boost is a feature in which the processor can run at a higher clock frequency to increase performance for short periods time at the expense of increased thermal output. Unfortunately, turbo boosting can make it difficult to compare results from multiple runs or monitor trends if some data points were "boosted." Turbo boosting is also invisible to the software, so it is not possible to know which data was "boosted."

Measuring time is also a surprisingly difficult prospect. The accuracy and precision of counters differs for each platform, and it may be difficult to provide code that can study multiple platforms and accurately measure time on each. For this report, we implemented the micro-benchmarks under the assumption that the timers have precision on the order of a millisecond.

2. METHODOLOGY

We have developed a variety of micro-benchmarks in order to reverse engineer the hardware of a given processor. Each program is written in C and compiled using `gcc` for the Intel processors and `tile-cc` for the Tilera processor.

We used `gettimeofday()` to time the relevant sections as this helps maintain the portability of the code across all platforms, as not all systems provide the programmer with easy access to a cycle-accurate timer. Each micro-benchmark ran the critical section for at least a hundred milliseconds to provide a significant amount of time to measure.

We used the interface provided by `sched.h` to direct the scheduling of individual threads on specific cores. In some instances we also used the Tilera `iLib` API to provide direct control of the scheduling of our code.

2.1 Cache Size and Access Latency

We have designed and implemented a micro-benchmark called `caches` to determine the number of levels in the cache hierarchy, the capacity of each cache, and the access latency at every level of the cache hierarchy.

The micro-benchmark `caches` runs a pointer chase on an array of a given size. Each element in the array provides the index to the next element in the array. This prevents the processor from speculatively executing ahead, thus isolating the memory access time required to read the element in the array (example of code below). If the array fits entirely within the cache, the pointer chase will execute very quickly. As the array grows larger however, more misses occur and the program runs slower. For the most accurate results possible, both spatial locality and memory prefetching must be defeated. This can be accomplished by striding the indices to point to different cache lines (removing spatial locality), and by randomly sorting the indices in the array (to prevent intelligent prefetchers from guessing the access pattern).

The results produced by the `caches` micro-benchmark are presented in Section 3.2 and Figure 1.

```
// 'stride' is an input parameter.
// The 'multiply' instruction is removed
// by enumerating each case.
// Instead, a 'shift' is synthesized.

if (stride == 16)
{
   uint32_t const stride_16 = 16;
   start_time = mysecond();

   for (uint32_t k = 0; k < g_num_iterations; k++)
   {
      idx = g_arr_n_ptr[idx*stride_16];
   }

   stop_time = mysecond();
}
```

Our Related Work section (Section 5.3) describes a strided memory hierarchy micro-benchmark that was discussed by Professor Yelick in the UC Berkeley class CS267. Like `caches`, this micro-benchmark also fetches data from an array of a given size, but it does this using regular, strided accesses. Both the array size and the size of the stride are changed. The output of this micro-benchmark ideally gives the cache sizes at each level as well as the access time, line size, and page size. However, we hypothesized that memory prefetching would make it difficult, if not impossible, to use this micro-benchmark to accurately evaluate the memory hierarchy.

We chose to re-implemented this strided memory hierarchy micro-benchmark, to augment our own data, to provide a comparison to our own micro-benchmark `caches`, and to highlight the importance of randomizing the memory access stream. The results are presented in Figure 2.

2.2 Request Bandwidth

We define request bandwidth as the number of memory requests issued and fulfilled per second. The micro-benchmark called `band_req` discovers both the request bandwidth of the processor, and the maximum amount of outstanding requests that can be serviced simultaneously at each level of the cache hierarchy.

This benchmark requires a small change to the `caches` micro-benchmark. The `band_req` micro-benchmark still performs a pointer chase on an array; however, this version allows for the core to issue multiple, independent streams simultaneously. The number of independent request streams is set at run-time. The code is shown below:

```
#define STRIDE 16 // 64B, i.e., a cache-line stride
uint32_t *idx = malloc(num_requests * sizeof(uint32_t));

for (int i=0; i < num_requests; i++)
   idx[i] = (rand() % g_arr_size);

start_time = mysecond();

for (uint32_t k = 0; k < g_num_iterations; k++)
{
   for (uint32_t i=0; i < num_requests; i++)
      idx[i]  = g_arr_n_ptr[idx[i]*STRIDE];
}

stop_time = mysecond();
```

2.3 Cache-to-Cache Transfer Latency

We have developed a micro-benchmark called `cache2cache` to accurately measure the cache-to-cache transfer latency between two cores. This is done by running two threads on two separate cores of a processor. One thread initializes an array and writes to each element while the other thread waits on a spin lock until the first thread is done. The second thread then gets exclusive rights to the array and writes to each element while the first thread is now waiting on a spin lock. This is repeated for many iterations and for various array lengths in order to get a long enough runtime to determine the time it takes each thread to access one element coming from the cache on the other core. Unfortunately, this does measure the locking overhead. To counter this issue, we measure the cache-to-cache latency by sending an entire array of data to amortize the locking overhead.

The index to the next element in the array depends on the previous element in the array. This is done to force a serialization of the memory requests and to prevent processors from overlapping memory requests. In doing so, we can accurately measure the *latency* of each individual request, by tracking the total time of sending the entire array and dividing by the number of elements in the array.

Also, we wanted to guarantee that there was no benefit of spatial locality, so the array access was strided such that no sequential elements written by the thread were on the same cache line. The main piece of code is provided below:

```
//initialization code...
for(i==0; i<num_elements-1; i++)
   array[i] = i+STRIDE;
array[num_elements -1] = 0;

//after initialization...

tid = pthread_self();
double start_time = mysecond();

while(count < num_iters){
   if(tid == waiting_thread)
      while(*mylock == 0);

   count++;
   int idx = 0;
   int i=0;

   do{
      #ifdef BANDWIDTH
      array[i] = count;
      i++;
      #endif

      #ifdef LATENCY
      idx = array[idx];
      i+=STRIDE;
      array[idx] =  i;
      #endif
   }while(i<num_elements-STRIDE);

   waiting_thread = tid;
   *mylock = 0;
   *otherLock = 1;
}

double stop_time = mysecond();
```

2.4 Cache-to-Cache Bandwidth

The same micro-benchmark, `cache2cache`, is used to measure the bandwidth between caches (as shown above).

Each thread waits on a spin lock for its turn to write to the array, and the exclusive read/write privileges are ping-ponged back and forth between the threads for many iterations. From this we are able to calculate the MB/s that are transferred between the caches.

The bandwidth version, to be consistent with the STREAM benchmark (described in Section 5), accesses the array in a unit-stride pattern and does not serialize the access to each element. This allows the processor to overlap accessing each element for maximum bandwidth.

2.5 Cache-to-Cache using Direct Messages

The TILEPro64 supports direct access to the underlying mesh network through the use of their direct messaging APIs. This allows the programmer to avoid the cache coherence system by invoking separate *processes* and communicating between processes using direct messages.

The iLib API provides a variety of messaging primitives, many of which trade off programability for performance. For our `cache2cache` micro-benchmark, we used *Raw Channels* for our implementation, which provides the highest level of performance.

The main loop is shown below. Each process has a private array, called `localarray`. The `sender_id` process sends its copy of `localarray` to the other process which writes the incoming data to its own copy of `localarray`. The receiving process then increments each element in the array to precisely mimic the behavior required in the cache coherence version of `cache2cache`.

Notice that there is no explicit barrier in the direct messaging version of `cache2cache`. This is because the receiving messages is a naturally blocking operation, and implicitly synchronizes both processes.

```
while(count < num_iters)
{
   count++;
   if (tid == sender_id)
   {
      ilib_rawchan_send_buffer(port_send, localarray,
           buf_size);
   }
   else
   {
      ilib_rawchan_receive_buffer(port_send, localarray,
            buf_size);
      for (int i=0; i < num_elements; i++)
         localarray[i] = localarray[i] + 1;
   }

   if (sender_id == 0)
      sender_id = 1;
   else
      sender_id = 0;
}
```

It may be possible that a more optimal implementation would overlap the execution of sending data with incrementing the `localarray` data structure.

3. RESULTS

3.1 Experimental Setup

Results were gathered from the following processors: an Intel Core i7 640M (Arrandale), an Intel Core i7 975 EE (Bloomfield), and a Tilera TILEPro64 embedded processor. A subset of their relevant features are provided in Table 1.

3.1.1 The Arrandale & Bloomfield Processors

Arrandale, an implementation of the Westmere micro-architecture, is a hyperthreaded, dual-core processor used in notebook computers. Bloomfield, an implementation of the Nehalem micro-architecture, is a hyperthreaded quad-core processor aimed at the high-end desktop/single-socket server market. Both processors utilize aggressively speculating, super-scalar, out-of-order pipelines. They also make use of hardware prefetching to analyze memory streams and predict future memory requests to hide memory latencies. Both processors can utilize Turbo Boost to increase the clock frequency for short periods of time.

[1]The TILEPro64 is able to aggregate a subset of the private L2s to build a shared, distributed L3.

Table 1: Processor Specifications. The provided cache sizes refer only to the data caches.

Processor	i7 Arrandale	i7 Bloomfield	TILEPro64
Cores	2	4	64
Threads	4	8	64
L1	2 x 32 KB	4 x 32 KB	64 x 8 KB
L2	2 x 256 KB	4 x 256 KB	64 x 64 KB
L3	4 MB	8 MB	4 MB *[1]
Frequency	2.8 GHz	3.33 GHz	700 MHz
Technology	32nm	45nm	90nm
Max Bandw	17.1 GB/s [1]	25.6 GB/s [2]	25 GB/s [5]

3.1.2 The TILEPro64 Processor

The TILEPro64 is a 64-core embedded processor from Tilera. It aims to be a high compute, low power processor that can be used in networking, video, and cloud computing applications. Each core is a three-way VLIW engine. The processor has no hardware floating point units, but it can invoke software routines when running floating point code. Each core is located on a *tile*, which also contains a L1 cache, a L2 cache, and a network switch. The tiles are arranged in a two-dimensional grid, and the network switches are connected to its four neighbors to create a two-dimensional, bi-directional mesh network. The L2 caches can be combined to provide a "virtual", distributed L3 cache, at the programmer's request.

The TILEPro64 has full cache coherence support, allowing it to run "off-the-shelf" `pthead` applications. Tilera also provides a set of direct messaging APIs that give access to the network for inter-core communication.

3.2 Cache Size and Access Latency

The results from the `caches` micro-benchmark are shown in Figure 1.

The access time per memory request is graphed for a unit-stride access pattern, cache-line-stride access pattern, and random access pattern. The random access pattern randomly chooses amongst a set of cache-line addresses, eliminating all spatial locality.

Figure 1 demonstrates the dramatic benefits of spacial locality and pre-fetching. Unit-strided takes advantage of both of these while the cache-line-strided version removes spatial locality since each new element access will be on a different cache line, but the pre-fetcher can still look ahead since the stride is constant. When pointer chasing is done on a randomized array with a cache-line-stride all of these optimizations are removed. By removing these optimizations, we are allowed a more accurate glimpse of the underlying hardware.

The `caches` micro-benchmark assumes that an array will try to "sit" in the memory that is closest to the processor pipeline. As the array size increases the array will overflow into the next level of the cache hierarchy. The dramatic increases in the access time in the graphs shown in Figure 1 correspond to the moments when the array begins to overflow its current cache in the hierarchy. Each plateau on the graphs can be read as the access time for that given cache level. Figure 1 shows that it is necessary to use the randomized, cache-line-strided implementation to arrive at the most accurate results. The published specifications of each processor are shown in Table 1, while our derived specifications are shown in Table 4.

Two subfigures (1(c) and 1(d)) are provided for the TILEPro64 processor; Figure 1(c) shows the default malloc behavior in which L2 caches are treated as private, while Figure 1(d) shows fifty-six L2 caches (3.5 MB) being aggregated together to enable a virtual L3 cache by using Tilera's "Hash-for-home" technology[7].

For comparison, the results from the strided memory hierarchy micro-benchmark, described by Professor Yelick in the UC Berkeley class CS267 and reimplemented by us for this project, are shown in Figure 2.

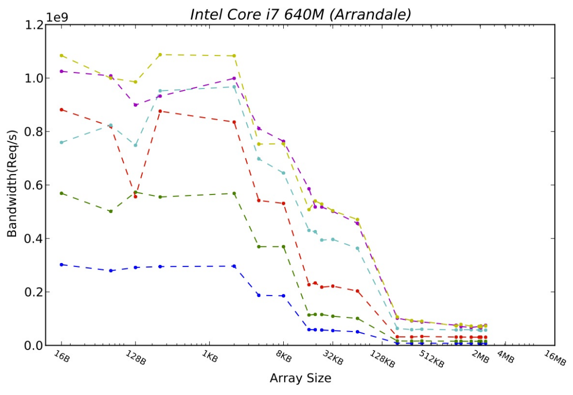
The strided memory hierarchy program gives results that are a bit convoluted due to the effect of prefetching (Figures 2(a) and 2(b)). However, this does allow one to distinguish processors that lack prefetchers, as shown by Figures 2(c) and 2(d) with the TilePro64 processor. With a processor that does not have a prefetcher, the different cache sizes are determined by the array size at which there is an increase in the time per iteration. However, a prefetcher can hide this latency and for processors with such abilities (Figures 2(a) and 2(b))) all of the arrays that can fit in the on-chip cluster together, making it difficult to accurately distinguish the different levels in the memory hierarchy.

3.3 Requests Bandwidth

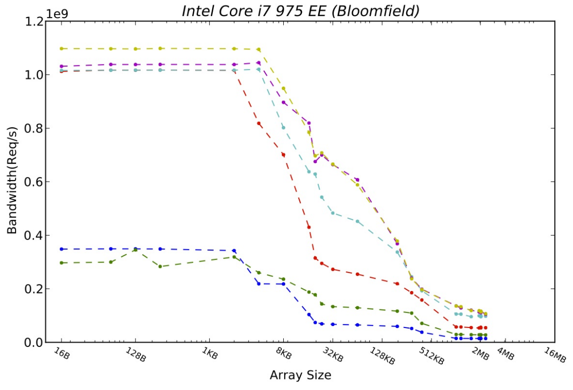
We implemented the `band_req` micro-benchmark to determine the request bandwidth on all three processors. This is a value which is not published, but it may be useful for programmers to know both at what point the memory system becomes saturated with requests, and also how many requests need to be executed in order to hide memory access latency. Figure 3 shows our results.

There are two important parts of the graph to analyze. The first is for small working sets that fit entirely within the L1 cache. Here, the processor does not stall waiting for memory, and only needs to hide the few cycles of latency required for L1 accesses. The bandwidth requests will only be limited by pipeline resources of the processor (i.e., the size of the reorder buffer, the issue queues, and the load/store queues). However, `band_req` will highlight how many requests the processor must execute before completing hiding the L1 access latency.

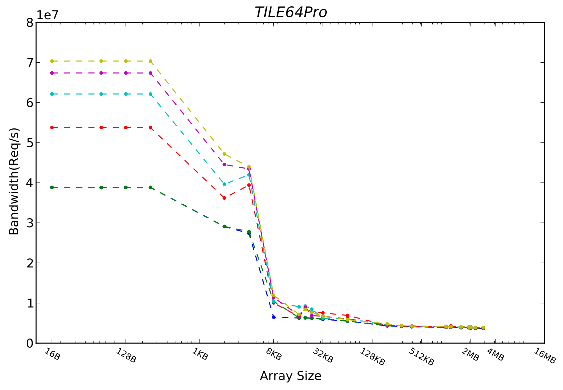
If the working set is larger than the L1 cache capacity, the `band_req` micro-benchmark begins testing the number of outstanding requests that can be supported by the caches. The data in Figure 3 suggests that both of the Intel processors are saturated at up to sixteen requests in flight at a time. Likewise, the TILEPro64 processor appears to saturate at four requests.



(a)



(b)

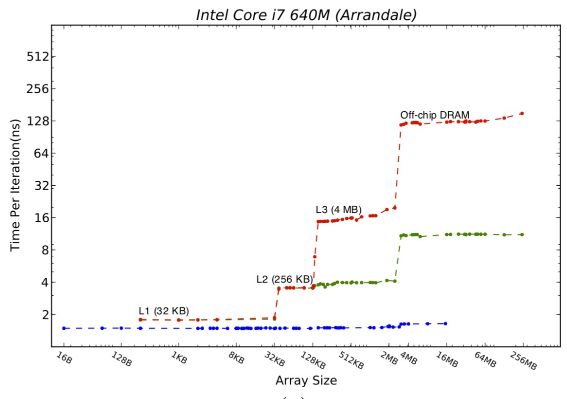


(c)

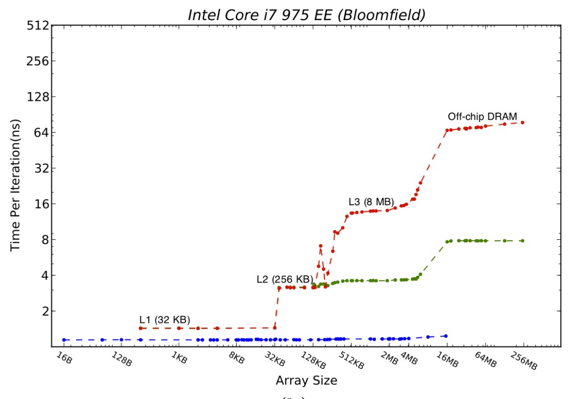
Figure 3: Request Bandwidth. Each line shows the results for a different number of parallel requests by a thread. From bottom to top they are blue=1, green=2, red=4, cyan=8, magenta=16 and yellow=32 requests.

3.4 Cache-to-Cache Transfer Latency

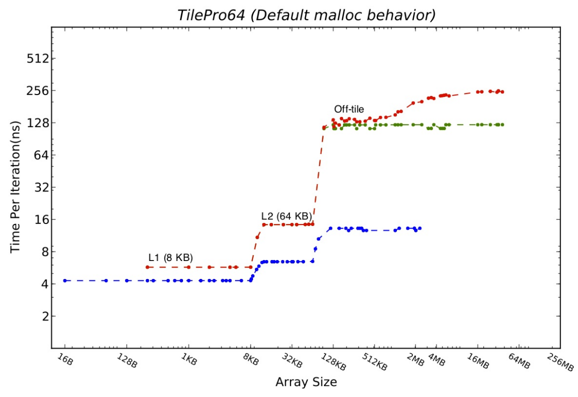
In an attempt to determine additional unpublished specs we implemented the `cache2cache` micro-benchmark to measure



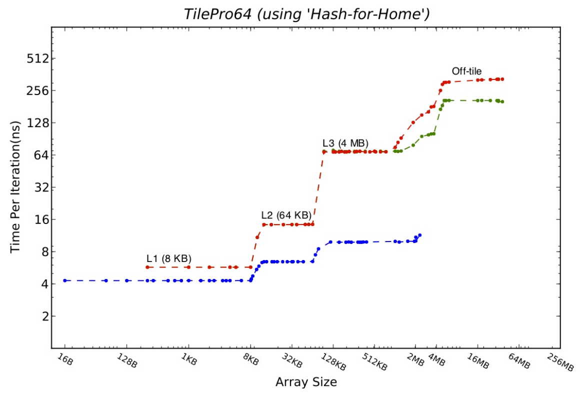
(a)



(b)

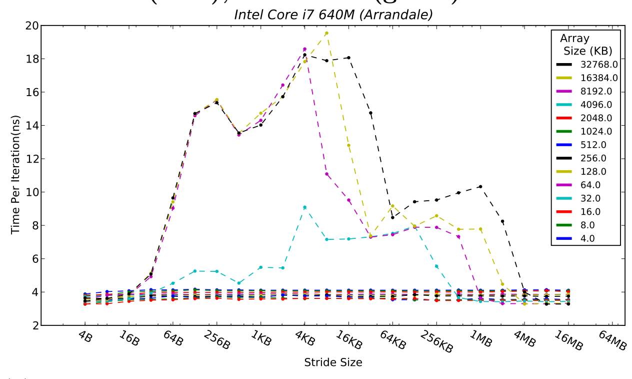


(c)

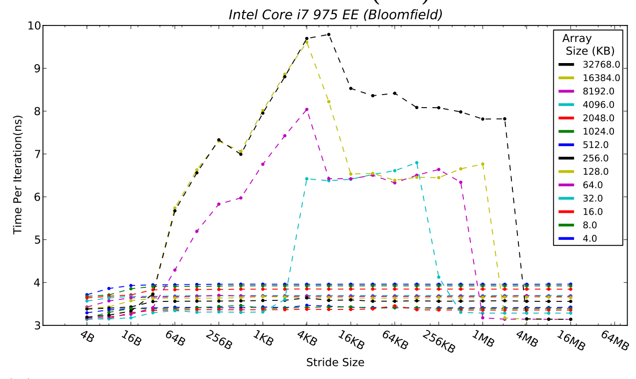


(d)

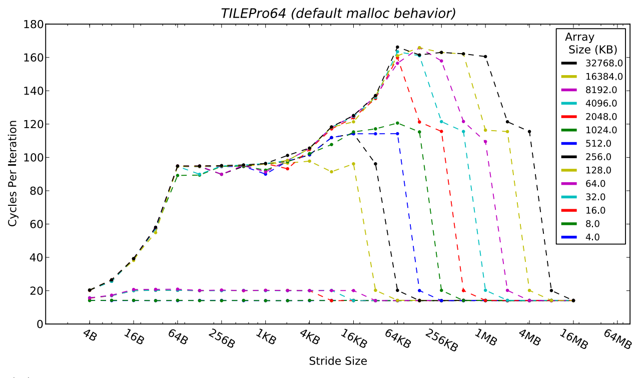
Figure 1: Cache Size and Access Latency. The pointer chase micro-benchmark was run three different ways: unit-strided (blue), 16-strided (green) and randomized with a stride of 16 elements (red).



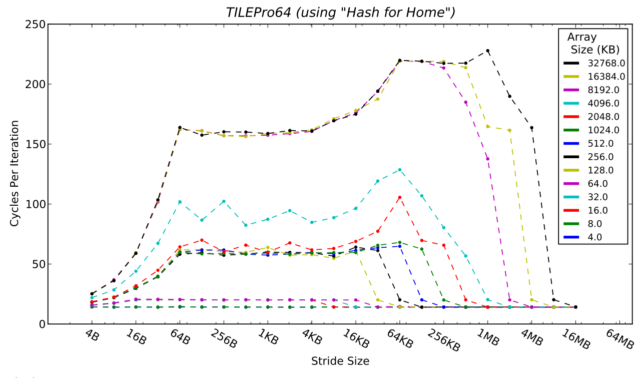
(a)



(b)



(c)



(d)

Figure 2: Memory hierarchies described by strided memory accesses. Provided for comparison to results presented in Figure 1, these graphs are produced from a reimplementation from an algorithm described by Professor Yelick. Prefetching in the Intel processors greatly obfuscate the design of the memory hierarchy. Note that data from the Tilera processors is presented in *cycles* per iteration for this figure.

the time it takes to move data from one L1 cache to another L1 cache on separate cores. For small arrays we see the unavoidable locking overhead, but this is absolved for larger data sets at which point the transfer latency is constant regardless of the array size (Figure 4).

This time is approximately equal to the access time of the highest level cache, which we found surprising but believe that this reflects an accurate picture of the processor's organization. This result suggests that the directory information is stored in the last level cache (LLC), therefore requiring that all data requests go through that level in order to change the exclusive rights from the thread on one core to the thread on another (Figure 5). Then, the current data holder is able fulfill the request by directly transferring the data to the requesting L1 cache.

For a processor with more than two cores it may be useful to know if the cache-to-cache transfer latencies differ between pairs of cores. We tested this on the TilePro64 processor by determining the time per memory access for neighboring tiles compared to tiles that were on opposite sides of the chip (Figure 4(c)). There is an observed increase of roughly 20 nanoseconds in the time it takes to move data from one L1 to an L1 on a tile that is far away compared to the transfer time for neighboring tiles, confirming the NUMA characteristics created by Tilera's mesh interconnect.

We also experimented with the direct messaging library for the TilePro64, shown in Figure 4(c).

3.5 Cache-to-Cache Bandwidth

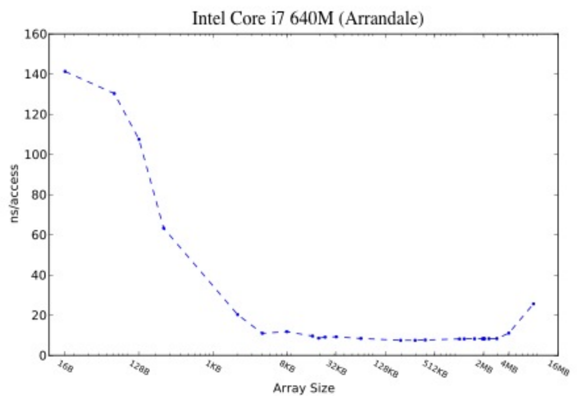
The cache-to-cache bandwidth results from the `caches` micro-benchmark are reported in Table 3. For comparison, we provide off-chip bandwidth reported by the STREAM benchmark in Table 6. Note that the c2c bandwidth is generally much lower than the off-chip bandwidth.

Table 2: Sustainable off-chip bandwidth (MB/s) obtained by the STREAM benchmark, described in Section 5. `Copy` measure the bandwidth to copy an array, `Scale` involves calling an array, `Add` adds two arrays together, and `Triad` scales and then adds two arrays together. TILEPro64 results are given for both the default `Copy` implementation, and a `memcopy()` implementation.

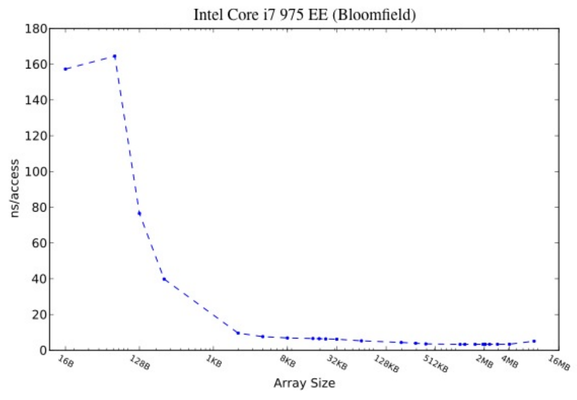
Processor	# Threads	Copy	Scale	Add	Triad
i7 Arrandale	4	7292	7011	7511	7536
i7 Bloomfield	8	12470	12130	12431	12641
TILEPro64	1	423	99	151	81
TILEPro64	56	403	100	148	80
memcpy()	1	1516	-	-	-
memcpy()	56	1189	-	-	-

4. DISCUSSION

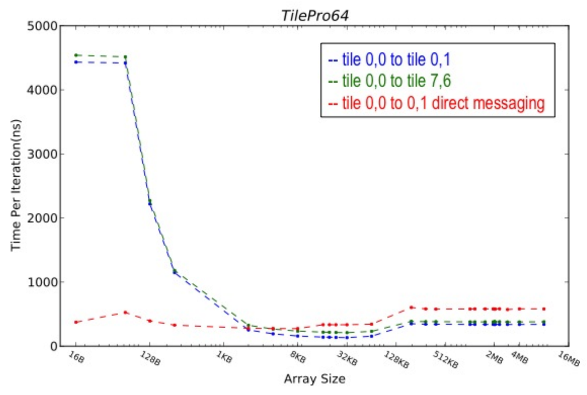
This section will attempt to use the data provided in Section 3 to derive some of the important memory hierarchy parameters, and discuss some of the implications from some of the micro-benchmarks. In particular, some of the micro-benchmarks provide further insight into where performance



(a)



(b)



(c)

Figure 4: Cache-to-Cache Transfer Latency

Table 3: Maximum cache-to-cache bandwidth (MB/s) obtained by our `cache2cache` micro-benchmark.

Processor	cache-to-cache write (MB/s)
i7 Arrandale	1789
i7 Bloomfield	2150
TILEPro64	246

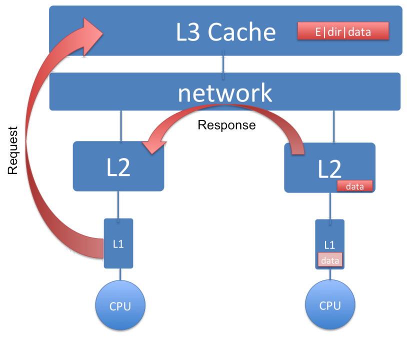


Figure 5: Proposed Model to Explain the Cache-to-Cache Transfer Latency Observations.

can be lost from typical coding patterns, and to help aid in the decision to program applications using direct messages or shared memory constructs.

4.1 Derived Cache Specifications

Table 4 shows the cache parameters as derived from the data presented in Figure 1.

4.1.1 *Cache Sizes*

The L1 data cache size can be determined with great accuracy; if the working set for a memory-bound application fits within the L1, the processor will run at constant throughput. Otherwise, if the working set fails to fit within the L1, the processor must occasionally stall to wait on resolving L1 misses.

Figuring out the number of caches in the hierarchy, and the boundaries between levels, is more difficult. Strided accesses remove all spatial locality and help to accentuate the boundaries between different cache levels. However, outer level caches may be distributed across the chip and exhibit non-uniform memory access times (NUMA), making it more difficult to differentiate between a new cache level and accessing a different cache slice. For the processors explored in this report, the data from Figure 1 clearly shows that each processor exhibits three levels of caching and the boundaries between each.

We do note that the L2 and L3 cache capacities found for the i7 Arrandale processor by the `caches` micro-benchmark (Table 4) do not seem to agree with the published specifications[1], provided in Table 1.

The boundary between the last level cache (LLC) and off-chip DRAM is even more difficult to discern. TLB reach starts to become a contributing factor once the working set grows to about 2 MB. Larger page sizes were not utilized, as this requires rebooting the machine and reserving the pages ahead of time. Also, not all operating systems support huge pages.

Tilera's virtual L3 cache, as described in Table 4, comes out to less than the reported 4 MB in Table 1. This is largely due to the fact that eight tiles are reserved by the hypervisor, presenting at most 3.5 MB of distributed L3 cache. The L3 cache is also distributed across the entire chip with varying access times. Coupled with TLB reach and the variability in access times to DRAM, we see that the the effective L3 cache size appears to between 2 and 3.5 MB.

4.1.2 *Cache Access Latencies*

Cache access latencies are more challenging to compute. By running the benchmark for a significant length of time, the "time per loop iteration" can be calculated. Ideally, this corresponds directly to the memory access time. However, loop overheads and memory accesses that hit in multiple parts of the cache hierarchy can pollute this value.

The L1 cache access time can be calculated by looking at the data points in which the working set fits entirely within the L1 cache. At this point in the graph, time per iteration is dominated by overheads involved with calculating addresses from the index, incrementing the loop counter, and branching. Indeed, the unit-stride and strided versions in Figure 1 differ entirely due to the strided address calculation: a *shift* instruction must be executed in the strided version that is not present in the unit-stride implementation. The L1 access time reported in Table 4 assumes zero overhead, which overestimates the true L1 access time.

The L2 and L3 access times are less susceptible to the differences in overhead, however, Figure 1 demonstrates the power of spatial locality, given the significant difference between the unit-strided and cacheline-strided data sets.

The random data set shows the power of the prefecther: Intel processors show roughly 10x improvement over cacheline-strided accesses. The TILEPro64 data shows that the processor does not use a prefetcher. The discrepancy in the TILEPro64 data at larger working sets between the random pointer chase and the strided pointer chase is probably due to TLB misses and a lack of locality for the DRAM and memory controllers to exploit.

In terms of measuring how accurate the results are, the *Tile Processor Architecture Overview*[6] user guide specifies the pipeline latencies for the TILE64 processor (an older iteration of the processor used here, but it is fair to assume the pipeline latencies have not changed significantly). The user guide reports the following load use latencies, shown in Table 5.

Table 5: TILE64 pipeline latencies provided by the Tilera architecture manual[6].

Operation (LD to use)	Latency
L1 hit	2 cycles
L1 miss, L2 hit	8 cycles
L1/L2 miss, adjacent $ hit	35 cycles
L1/L2 miss, DDR2 page open (typical)	69 cycles
L1/L2 miss, DDR2 page miss (typical)	88 cycles

Comparing these values to the empirical values in Table 4 suggest that the loop overhead involved makes up about one or two cycles of each iteration, as expected. The derived L3 access time appears to be about thirteen cycles slower, or

Table 4: Derived processor specifications provided by the `caches` micro-benchmark.

	data cache size			access latency		
Processor	L1	L2	L3	L1	L2	L3
i7 Arrandale	32 KB	128 KB	3 MB	1.8ns 5.0 cycles	3.4ns 9.5 cycles	14.8ns 41.4 cycles
i7 Bloomfield	32 KB	256 KB	8 MB	1.4ns 4.7 cycles	3.1ns 10.3 cycles	15.5ns 51.6 cycles
TILEPro64	8 KB	64 KB	2-3.5 MB	4.2 ns 2.9 cycles	14.3 ns 10.0 cycles	69.2 ns 48.4 cycles

30%, but this is most likely because the specifications only provides the L3 access time for adjacent cache hits, whereas the distributed L3 can hold data as far away as 15 hops in the network. The TILE disassembly for the main loop for the `caches` benchmark can be found in the Appendix.

4.2 Memory Request Bandwidth

The results for memory request bandwidth for the Intel processors shows that the requests per second is the same for 16 independent request streams and for 32 independent request streams (see Figure 3). This suggests that both Arrandale and Bloomfield cannot service more than at most 16 independent requests simultaneously out of the L1 cache.

The Tilera results suggest that the L1 can only service up to four misses at a time. However, because the TILEPro64 is an in-order VLIW machine, it may be possible that the bottleneck resides in the code itself, and not in the L1. For comparison, out-of-order processors can completely unroll the request loop, as shown in Section 2.2, until they are limited by the re-order buffer size, the load/store queue size, or the number of misses the L1 cache can handle.

4.3 Cache to Cache Memory Traffic

4.3.1 Intel Transfer Latency

Figure 4 shows the latency of transferring write permissions from one thread to another, which is around 15 ns for both the Arrandale and the Bloomfield processors. For working sets less than the L1 cache size, this can be construed as the cache to cache transfer latency. However, if the working set is too small, the timing is dominated by the locking overhead to synchronize the two threads. For Intel machines, this locking overhead for a spinning lock is on the order of 150 ns, according to our results in Figure 4.

An interesting result from Figure 4 is that the L1 cache to L1 cache transfer latency is about the same as the last-level cache (LLC) access time. Figure 5 shows the hypothesized coherence traffic: a miss in the L1 on core A generates a request that is broadcast across the network to both the LLC and core B's L2 cache. The L2 cache for Core B, which is snooping requests that go across the network, can then service the request by sending the data over the network to core A. This path will look identical to the LLC servicing the request itself.

It is not obvious whether the LLC must be contacted for directory information, whether the L2 is write-through to the L3 and thus allowing the L3 to directly service the request, or whether the L2s can snoop the network to service the request directly by themselves. A different micro-benchmark will be required to differentiate between inclusive LLCs and victim-cache LLCs.

4.3.2 Tilera Cache Coherence Transfer Latency

Data from the Tilera processor regarding cache to cache latencies are surprisingly high, at best down to 150 ns.

A Tilera processor uses a two-dimensional mesh, with a core at each node. Each hop is traversed in a single cycle, however, many hops can be required to travel between cores far apart. This non-uniform access can be seen in Figure 4. However, the difference in travel times between cores appears to be dominated by some other penalty. It should also be noted that the clock frequency is much slower than Intel processors, at 700 MHz. The Tilera chip also uses a directory protocol. Thus a cache to cache transfer will require a request to the directory from core A, an invalidation request from the directory to the current exclusive user core B, a flush-back response from core B back to the directory, and a response to the requester core A from the directory with the data (four messages total, although it is possible to cut this down to three messages on the critical path). If this entire process stays on-chip, it could at worst take 16 hops for each trip (consider the requester being located at (0,0) in the mesh, the current owner being located at (0,1), and the directory located at node (7,7)). This could lead to roughly 64 hops, which at 700 MHz is equal to 90 ns just in the network travel time.

Further analysis will be required to confirm that spurious instructions in the main loop are not adding additional, unnecessary costs to the cache to cache transfer code. It is also not clear if the pipeline latency cited in the Tilera architecture manual[6] for an "adjacent cache hit" is referring to a *read* request to a *shared* line, which would require no invalidation traffic, or to a more expensive request.

The synchronization overhead is also incredibly high on the TILEPro64: this is possibly due to the fact that a `ilib_msg_barrier()` call was used instead of the spin-lock used on the Intel processors.

4.3.3 Tilera Direct Messaging Transfer Latency

Perhaps a more surprising result was the timing of the Tilera direct messaging latency. The direct messaging code does not require a barrier, as the blocking message itself acts as a natural synchronization point. However, cache coherence ends up demonstrating superior performance with larger arrays. This is reasonable for arrays that do *not* fit within the L1: in order to send data in a packet from one core to the next, the sender must first fire a request across the memory hierarchy to bring the data into its L1!

However, this may not explain why direct messages are on the order of 150 ns per word for working sets of 4 KB.

4.3.4 Cache to Cache Bandwidth

Cache to cache bandwidth results are shown in Table 3. The inner loop to this code is inspired by the STREAM benchmark, thus the "sustainable" bandwidths from both the STREAM benchmark (which provides the off-chip bandwidth) and the cache to cache benchmark (which provides the L1 to L1 transfer bandwidth) should be comparable.

Given this, it is surprising that the cache-to-cache bandwidth is a factor of 2-6x worse. This may be explainable by the fact that processors are likely heavily optimized for the off-chip access path, which is certainly the most dominate use case.

4.4 Using Results to Drive Better Software

One of the surprising results from these benchmarks arose when the processors performed dramatically worse than expected. For example, preliminary tests using our `caches` micro-benchmark suggested that the "distributed L3" in the TILEPro64 was not being utilized. This issue was recognized and quickly fixed with the help of the `caches` benchmark.

4.4.1 The TILEPro64's Virtual L3

Tilera manuals advertises that threads can utilize up to 4 MB of on-chip cache by way of aggregating L2 caches from other tiles to create a "distributed, shared L3" cache. Preliminary results using unmodified C code showed no existence of this virtual L3 cache (see Figures 1(c) and 2(c)).

In studying Tilera's *Multicore Development Environment Optimization Guide*[7], we learned that the default behavior for malloc allocates memory to reside *only* on-tile. However, it is possible, within the application, to specify that malloc instead use the "hash-for-home" behavior. In this scenario, allocated memory is striped "evenly and randomly"[7] across the other tiles' L2 caches. However, which tiles are used is specified by settings in the hypervisor at boot time. "Reserved" tiles cannot be used, so our applications can only use up to 56 tiles, or 3.5MB.

The boost to performance for the cache sizing benchmarks can be seen in Figures 1(d) and 2(d), as the working set increases from 64KB to 3.5MB. Without the use of our `caches` micro-benchmark, the realization that we were not utilizing the full power of the TILEPro64's cache hierarchy would have escaped us.

4.4.2 Tilera's STREAM Results

The `STREAM` results for the TILEPro64 were also surprisingly low. Online literature denotes that the memory bandwidth is 200 Gbps (i.e., 25600 MB/s) with four DDR2 memory controllers [5]. Our results, shown in Table 2 are about 25x smaller.

These results pushed us to explore The TILE's DMA functions and `memcpy()` implementations, which showed improvements over the baseline STREAM implementation. Using the `mcstat` profiling tool, we still only see the memory controllers being utilized up to 10% of their peak throughput. Finer grained profiling will be required to find the bottleneck behind these results.

5. RELATED WORK

There are a variety of micro-benchmarks that are currently available that can provide some information about a processor's memory system.

5.1 STREAM

The STREAM benchmark, developed by Dr. John McCalpin in 1995, measures the sustainable memory bandwidth (MB/s) for a given processor [4]. More specifically, STREAM measures the unit-stride access bandwidth from off-chip memory by running a series of tests which copy, scale, sum and perform a triad (multiply and sum) on the data (Table 6).

Table 6: STREAM kernels used to measure sustainable off-chip bandwidth

name	kernel	bytes/iter	FLOPS/iter
COPY	a(i) = b(i)	16	0
SCALE	a(i) = q*b(i)	16	1
SUM	a(i) = b(i) + c(i)	24	1
TRIAD	a(i) = b(i) + q*c(i)	24	2

STREAM uses 64bit floating point numbers, which heavily penalizes the TILEPro64, as it lacks floating point hardware and is a 32bit machine. STREAM is multithreaded using OpenMP, however, the Tilera toolchain does not currently support OpenMP. For this reason, we modified STREAM to use a pthreads implementation and used this version for all of the data collected. All data was collected using the default function implementations provided in STREAM, except where stated in Table 6.

We also implemented *tuned* implementations of the `Copy` function for the TILEPro64 to display the difference between naive array code, DMA accesses, and `memcpy()`. These results from the default and `memcpy()` implementations are shown in Table 6. The DMA implementation runs even faster than `memcpy()` at 2.0GB/s, but currently fails to pass verification for some array elements.

STREAM can provide a useful upper-bound on the sustainable bandwidth for when applications exhibit unit-stride access patterns. Spatial locality provide huge gains, in regards to locality in the cache line, in the virtual memory page, and in the DRAM. Prefetchers can also hide most of the remaining latency in fetching data from off-chip, as shown in Figure 1. However, application developers may be more interested in knowing the substainable bandwidth for strided accesses and scatter/gather accesses.

5.2 Roofline

The Roofline model gives a visual representation of realistic performance and productivity expectations [8]. By running a series of micro-benchmarks, including the STREAM benchmark[4], the Roofline model can determine the hardware limitations on a given platform for a specific application kernel. For example, the application developer can determine if his program is memory-bound or compute-bound. The Roofline model then aids in the prioritization of additional optimizations that can be implemented.

5.3 Strided Access Memory Hierarchy Graphs

Professor Yelick described in the UC Berkeley class CS267 a micro-benchmark that measures access time to an array of a given size using a strided access. However, complicated memory hierarchies, in particular prefetchers, complicate the results and make it difficult to discern the true access latencies and cache sizes.

We chose to reimplement this micro-benchmark to provide a comparison to our `caches` micro-benchmark. The results of the strided memory access micro-benchmark are shown in Figure 2.

5.4 MEMBENCH

MEMBENCH is a collection of micro-benchmarks that measure 33 different transfer types [3]. This includes measurements of main memory read/write, L2 cache read/write, video memory write, main memory transfer, L2 cache transfer and video memory transfer. Though this provides many valuable numbers, the program is single-core focused and therefore lacks the ability to derive some of the parameters that our micro-benchmarks are able to obtain (e.g. cache-to-cache transfer latency) which may be crucial in final tuning and optimization.

6. CONCLUSION

We have designed and implemented a series of micro-benchmarks that empirically characterize a given hardware platform's memory hierarchy. These small micro-benchmarks are able to deduce a variety of characteristics, including cache to cache transfer latency and cache to cache transfer bandwidth. We have tested our micro-benchmarks on an Intel notebook processor, an Intel high-end desktop processor, and a Tilera 64-core embedded processor.

Our results on cache size and cache access latency are generally close to the published specifications, with the exception of the measured L2 and L3 cache capacities for the i7 Arrandale processor. Also, the maximum inflight requests, the cache-to-cache transfer latency, and the cache-to-cache transfer bandwidth are not published parameters, making it more difficult to ascertain the accuracy of our micro-benchmarks. However, it seems surprising that the cache-to-cache transfer bandwidth for all processors is lower than the off-chip bandwidth.

Nonetheless, the results gained from these micro-benchmarks provide an interesting first look into the design and behavior of different memory hierarchies.

Memory hierarchies are becoming more complicated, both to address the growing gap between memory and processor speeds, but also to address the increasing request bandwidth needed to feed the growing numbers of cores. Features such as non-uniform memory access times, complicated cache coherence protocols, dynamic cache policies, and direct access to the raw network for inter-core communication can create confusion for programmers attempting to design optimal algorithms, and for hardware researchers attempting to model and debug accurate simulators. Micro-benchmarks can play an important role in empirically characterizing these memory hierarchies and in providing a cost-benefit analysis of the different tools available to software programmers and auto-tuners.

7. REFERENCES

[1] Intel. Intel®Core^{TM}i7-640M Processor (4M Cache, 2.80 GHz). http://ark.intel.com/Product.aspx?id=49666.

[2] Intel. Intel®Core^{TM}i7-975 Processor Extreme Edition (8M Cache, 3.33 GHz, 6.40 GT/s Intel®QPI). http://ark.intel.com/Product.aspx?id=37153.

[3] M. Krech. Membench.

[4] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[5] Tilera Corporation. TILEPro64 Processor Product Brief. http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf.

[6] Tilera Corporation. TILE PROCESSOR ARCHITECTURE OVERVIEW, 2007. REL. 1.1.

[7] Tilera Corporation. MULTICORE DEVELOPMENT ENVIRONMENT OPTIMIZATION GUIDE, 2009. MDE RELEASE 2.0 DOC. RELEASE NO. 2.1 DOC. NO. UG105.

[8] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

8. APPENDIX

Below is the TILE disassemble from the `caches` benchmark described in Section 2.1. The C code this is disassembled from can be found in Section 2.1. Only the contents of the loop are shown (i.e., none of the initialization is shown).

The TILE ISA describes a 3-way VLIW machine. Each instruction bundle is 64-bits. The code below has unrolled the loop twice, allowing two loads to occur before branching. Register `r7` holds the address to the beginning of the array (*g_arr_n_ptr*), register `r0` holds the address to be loaded from memory (*g_arr_n_ptr[idx]*), register `r1` tracks the loop iteration count (variable *k*), and register `r9` checks for the loop end condition.

```
loop :
110b8:   { shli r0, r0, 6 ; addi r1, r1, 2}
110c0:   { add r0, r7, r0 ; sne r9, r1, r6}
110c8:   { lw r0, r0 }
110d0:   { shli r0, r0, 6 }
110d8:   { add r0, r7, r0 }
110e0:   { lw r0, r0 }
110e8:   { bnzt r9, 110b8 /*br to loop*/}
```

The important thing to notice is that the loop overhead for each iteration entails a shift instruction, an address calculation instruction, a loop increment instruction, and a branch. It is difficult to eliminate this overhead entirely to calculate the L1 access latency, especially for an in-order processor that cannot help hide these overheads.