P-Ray: A Suite of Micro-benchmarks for Multi-core Architectures [⋆]

Alexandre X. Duchateau, Albert Sidelnik, María Jesús Garzarán, and David Padua

Department of Computer Science
University of Illinois at Urbana-Champaign
{axdn,asideln2,garzaran,padua}@uiuc.edu

Abstract. The increasing complexity of computer architectures has made the approach of automatically generating code that is optimized for the target machine a growing area of interest. Examples of such systems are library generators, such as ATLAS, SPIRAL, or FFTW. To generate optimized code without manual intervention, these systems need to know the values of certain hardware parameters, such as the cache size or the number of registers. Current benchmark suites such as X-Ray or LM-bench can automatically determine some of these parameters for single processor super-scalar machines but cannot determine multi-core specific characteristics.

In this paper, we present P-Ray, a suite of micro-benchmarks that focus on hardware characteristics specific to multi-core architectures. Such characteristics include the number of cores that share the L2 cache, the different processors' interconnection topologies, and the bandwidth-to-memory for multi-cores. Our experiments show that, for several different architectures tested (desktop and server), P-Ray generates accurate results.

1 Introduction

With multi-core processors as the current dominant trend, and architectures more complex and less documented, finding hardware specifications is becoming increasingly difficult. Knowledge of hardware features can be useful in driving program optimization, such as in library generators. ATLAS [9], SPIRAL [5], and FFTW [2] are examples of known library generators. ATLAS generates linear algebra routines (BLAS) with a focus on matrix-matrix multiplication. SPIRAL and FFTW are similar to ATLAS, but generate signal processing libraries. Analytical models have been [10], or are being [1] developed for library generators that use hardware characteristics to reduce the search time. For example, ATLAS will use knowledge of the L2 cache size in order to determine optimal tile sizes for matrix-matrix multiplication.

⋆ This material is based upon work supported by the National Science Foundation under Awards CCF 0702260 and CNS 0509432 and by DARPA under award W911NF0710416

Existing benchmark suites such as X-Ray [11], Saavedra [7], and LMbench [4] try to address the problem of automatically finding machine specifications, but focus on features of uniprocessor super-scalars. To our knowledge, no existing micro-benchmarks measure multi-core characteristics.

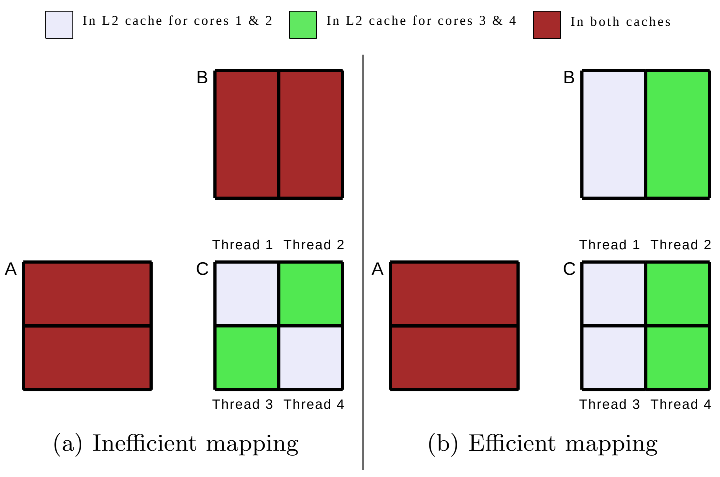
In this paper, we extend the existing sets of micro-benchmarks to multi-cores to find the number of caches shared by the cores, the processors' interconnection topologies, and the effective bandwidth and block size used by the cache coherence mechanism.

Our experimental results for three different platforms show that P-Ray generates accurate results.

The remainder of this paper is organized as follows. Section 2 provides motivating examples for our work. Section 3 presents the different benchmarks implemented. Section 4 describes our implementation requirements and details. Section 5 summarizes the experimental environment and discusses results. Section 6 describes related work. Section 7 proposes future work. Finally in Section 8, we summarize our work and offer concluding remarks.

2 Motivation

Multi-threaded matrix-matrix multiplication



(a) Inefficient mapping (b) Efficient mapping

Fig. 1. Data Locality depending on thread to core affinity

Library generators need a detailed knowledge of the architectural features of the machine to generate high-performance code. To show that this is the case, we ran an implementation of matrix-matrix multiplication ($C = A * B$) using POSIX threads on an Intel Core 2 Quad desktop that has four cores and two L2 caches, each cache shared by two cores. Figure 1 shows two different possible mappings for the matrices depending on thread affinity. For this experiment, matrix C is split into four sub-matrices, and each thread is assigned to one quadrant. Matrices are of size 800×800 each, so that they fit in memory but not in the L2 cache. With the mapping in Figure 1(a), both matrices A and B need to be loaded in both L2 caches. Using the mapping of Figure 1(b), matrix B can be split so that one half goes to one L2 cache and the other half goes to the second. Our experimental results show that an inefficient mapping can run up to 32% slower than an efficient one. To correctly map the threads to cores as in Figure 1(b), it was necessary to use P-Ray to obtain the ID of the cores that share the L2 cache. Thread affinity has been used in the past to pin a thread to a core in order to avoid its mapping to a different core (and subsequent cache trashing) after a context switch [8]. However, in current architectures where several cores could share a cache, thread affinity can be used to place in L2 the data shared by two threads. In most cases, this use of thread affinity can only be done if the programmer has the information provided by a tool such as P-Ray, by exhaustive search of all the possibilities, or if additional operating support is provided.

3 Benchmarks

In this section, we provide a high level description of each benchmark implemented in P-Ray. Implementation specifics and detailed interpretation of the produced results will be discussed in Sections 4 and 5.

3.1 Cache Coherence Protocol Block Size

Knowing the block size used by the coherence protocol can aid the programmer in reducing false sharing misses. Other benchmarks already exist to measure cache line size, but are relatively slow. By exploiting false sharing our benchmark infers the block size in a fast and simple way.

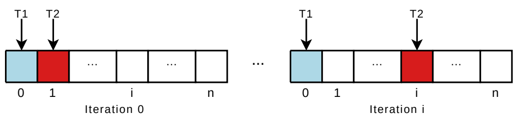


Fig. 2. Coherence Block Size Benchmark

Figure 2 illustrates Algorithm 1, that is used to compute the block size.

Two threads are spawned to work on a shared array of characters[1]. Both threads modify the shared data in order to induce coherence traffic. The data is

[1] In most architectures, a character is a basic type of size 1 byte.

Algorithm 1 Calculate block size

```
measure-size($core_1$,$core_2$) {
$char\ data[MAXLSIZES]$
$i \leftarrow 1$
while $i \leq MAXLSIZE$ do
    Start timing
    Spawn thread-work($core_1$,0)
    Spawn thread-work($core_2$,$i$)
    Wait for threads to complete
    Stop timing
    Print $i$, $timing$
    $i \leftarrow 2 * i$
end while
}
```

```
thread-work($core$,$index$) {
Set-thread-affinity($core$)
$i \leftarrow 0$
while $i \leq SAMPLES$ do
    $data[index] \leftarrow data[index] + 1$
    $i \leftarrow i + 1$
end while
}
```

also read to ensure it resides in L1: some architectures implement write-through write-no-allocate caches[2].

Thread one will always access the first element of the array. Thread two starts accessing the second element of the array; however, with each iteration, it accesses an element which is further apart from the first one.

At first, both threads will access the same cache line and have poor performance due to false sharing; as the spacing between accesses increases, the performance stays poor until both accessed values are on two separate cache lines. At this point, execution time decreases drastically and we automatically infer the coherence block size.

This algorithm can tell us the block size of different levels of cache. When the threads are mapped to cores that share a L2, this algorithm measures the block size of L1. However, when threads are mapped to cores that do not share a L2, this algorithm measures the block size of L2. When we do not have information about the mapping of a core to the caches, the second thread can be mapped to different cores in the system. By comparing the execution times of the different mappings, P-Ray can determine whether the block size corresponds to the L1 or L2 cache. When there is no coherency between the caches, this mechanism cannot determine the block size.

3.2 Cache Mapping

This benchmark is used to find the number of caches at a given level on the system and cores that share them.

Here the benchmark needs to know the size of the cache at the level in which we are interested. For completeness, P-Ray includes a benchmark to approximate it; however cache size can also be measured with other benchmarks [11, 4]. Algorithm 2 calculates the number of caches. Each thread accesses an array

[2] Sun Niagara T1: `http://opensparc-t1.sunsource.net/specs/OpenSPARCT1_Micro_Arch.pdf`

Algorithm 2 Calculate cache mapping

```
cache-mapping($core_1$, $core_2$) {
$i \leftarrow 1$
while $i \leq SAMPLES$ do
    Spawn thread-work($core_1$,1)
    Spawn thread-work($core_2$,2)
    Wait on thread barrier
    Wait for threads to complete
    $i \leftarrow i + 1$
end while
}
```

```
thread-work($core$, $id$) {
Set-thread-affinity ($core$)
Pointer p $\leftarrow$ Initialize local data
Wait on thread barrier
Start timing
for $i \leftarrow 0$ to SIZE do
    $p \leftarrow *p$
end for
Stop timing
if $id = 1$ then
    Print core pair, timing
end if
}
```

approximately sized to L2 in order to cause misses between cores that share the same cache. This array is initialized as described in Algorithm 5 below. The first step of the algorithm is to measure the time it took for one thread to read the elements of this array when running in isolation. This time will be used as a reference to interpret the results.

Then, we run a similar test with two threads. Each thread sets its affinity to a different core, initializes its workset, and waits on a barrier for the other thread. Once both threads leave the barrier, we measure the time it takes for the threads to read their arrays while running simultaneously. If the measured execution time is higher than the reference time, we conclude that both threads ran on cores that share a cache, and that performance degraded due to the worksets of the two threads competing for cache space. If it is the same, we instead infer that both threads ran on separate caches.

This test is run for all pairs of cores on the system. After gathering all the results, P-Ray determines the number of caches on the system and the ID of the cores that map to them.

3.3 Processor Mapping

This benchmark is used to determine the processors' interconnection topology.

Algorithm 3 uses two threads sharing a workset the size of the L1 cache, but running on separate cores. This algorithm functions by having one thread that reads and modifies its workset (i.e. brings it into L1) and measuring the time it takes for the second thread to read the data. By comparing the different access times of all possible pairs of cores, this benchmark can determine the different relative distances between all cores.

3.4 Effective Bandwidth

This benchmark is used to measure the available bandwidth for one core to memory by saturating it from one or several threads. In the following description

Algorithm 3 Calculate processor mapping

```
thread-work1(core_id$_2$) {
Set-thread-affinity (core_id$_2$)
p ← InitData($data$,$size$,$stride$)
Wait on thread barrier
}
```

```
Require: Pointer p is global
  thread-work2(core_id$_1$) {
  Set-thread-affinity (core_id$_1$)
  Wait on thread barrier
  Start timing
  for $i \leftarrow 0$ to L1SIZE do
      $p \leftarrow *p$
  end for
  Stop timing
  Print (core_id$_1$,core_id$_2$), timing
  }
```

the term "memory" will be used both for memory and caches unless otherwise specified.

Algorithm 4 Calculate bandwidth

```
Iteration1() {
Start timing
for $i \leftarrow 0$ to N_ITER do
    $p \leftarrow *p$
end for
Stop timing
Print timing
}
```

```
Iteration2() {
Start timing
for $i \leftarrow 0$ to N_ITER/2 do
    $p1 \leftarrow *p1$
    $p2 \leftarrow *p2$
end for
Stop timing
Print timing
}
```

We use an array that does pointer chaining with multiple entry points (this data structure and its initialization are described in Algorithm 5 and Figure 3 below). The offset between two entry points is here set to the size of a memory page. The stride between accesses is set to the smallest multiple of the page size that avoids overlap between chains.

To target a specific level in the memory hierarchy, we control the number of elements in the pointer chain before the loop back. We ensure that any reuse would happen after the data was displaced from levels closer to the core. Moreover, when measuring memory bandwidth, L2 is flushed after initialization.

Single-threaded bandwidth The first step is to measure the bandwidth to memory for an isolated thread.

In the first iteration, the benchmark traverses the array through a single entry pointer, as shown by Iteration1 in Algorithm 4. The code in Iteration1 serializes array accesses, as the access to the next element of the array cannot be issued until the pointer load returns. The second iteration of this benchmark traverses the array through two entry pointers, as shown by Iteration2 in Algorithm 4.

This loop has two independent accesses that can be sent simultaneously to the memory. However, accesses in an iteration depend on the accesses of the previous iteration for the same pointer chain due to the loop-carried dependences for all pointers. The benchmark proceeds by increasing the number of independent requests. By measuring the execution time of these loops, P-Ray can determine the number of requests that a core can have in-flight as well as its saturation point.

To calculate effective bandwidth, we use the following equation: Effective Bandwidth for program = $\frac{ClockFreq*ReadSize}{CyclesPerRead}$, where $CyclesPerRead$ is obtained by dividing the Execution Cycles at any of the saturation points by the number of iterations in our access loop. $ClockFreq$ is the clock rate for the given core. $ReadSize$ is the size of the cache line being read.

Multi-threaded bandwidth We then look at the memory bandwidth when multiple cores are sending requests simultaneously. For that, we use Algorithm 4 in parallel over multiple threads. When considering a cache, we limit ourselves to running the benchmark with threads on the cores that share that cache. When considering memory, we run the benchmark with any number of cores in the system.

To better understand the impact of concurrent access on the bandwidth for the targeted memory, we test different numbers of threads: we test anywhere between two and the number of cores sharing the targeted memory.

4 Implementation

4.1 Requirements

Our benchmarks have two major requirements: i) a high resolution wall timer (e.g. on Intel machines we use the RDTSC instruction to get timing in clock cycles [3]) and ii) library and operating system support to set thread to core affinity.

4.2 Implementation details

Pointer chaining The data structure used by most of our benchmarks is an array of pointers where each element of the array contains the address to the next element to access when traversing the structure. A similar data structure has been used by X-ray [11] and LMbench [4], but we initialize it using our own techniques.

A picture of the data structure is shown in Figure 3, and the algorithm for its initialization is shown in Algorithm 5. The initialization algorithm takes four arguments: i) *data* is a pointer to the allocated memory, ii) *size* is the size in memory of the data structure, iii) *stride* is the distance between two consecutive accesses, iv) $offset$ is the distance between two entry points, and v) *entries* is the number of entry pointers.

Our initialization routine uses a stride larger than page size to circumvent the hardware prefetcher and offset larger than the cache line size to prevent consecutive entry pointers from sharing a cache line. Since some experiments need to run for a large number of iterations, we limit the size of the array by having the last element of the chain point back to the first element (bottom lines of Algorithm 5).

Algorithm 5 Pointer Chaining

```
Init-data($data$,$size$,$stride$,$offset$,$entries$)
$i \leftarrow 0$
while $i < entries$ do
    $uoffset \leftarrow i * offset$
    Init-entry($data$,$size$,$stride$,$uoffset$)
    $i \leftarrow i + 1$
end while

Init-entry($data$,$size$,$stride$,$offset$)
$i \leftarrow offset$
while $i \leq size - stride$ do
    $data[i] \leftarrow \&data[i + stride]$
    $i \leftarrow i + stride$
end while
$data[i] \leftarrow \&data[offset]$
```

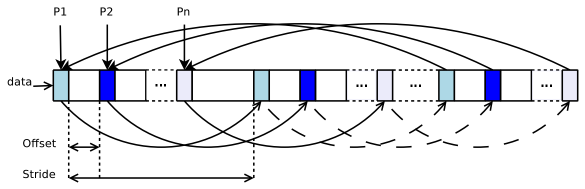


Fig. 3. Pointer chaining: General case

This structure has many advantages: i) it minimizes overhead, as no address has to be computed, ii) it allows for easy ways to experiment with different access patterns by tuning the initialization parameters, and iii) it prevents compiler optimizations that could interfere with performance measurements.

Loop Overhead The loop overhead should not be considered in the timing. Thus, in order to minimize the control overhead, the main data access loops are unrolled by a factor of 512. This value was chosen because it reduces loop overhead without adding substantial instruction cache pressure. Additionally, we time an empty loop to remove the control overhead from our timing. This way, the final time reflects the actual access times.

Code Reordering In order to prevent the compiler from performing any reordering of instructions within the timed kernel, *volatile* data modifiers are used. We were careful to not excessively mark every variable *volatile* when used outside of timed kernels, as this can hurt performance substantially.

System Noise To deal with the problem of system noise from the operating system and other user applications, we take numerous timing samples and use the minimum timed value as our result. This compensates for other programs and daemons running on the system.

5 Evaluation

5.1 Experimental Environment

We tested on three different architectures described in Table 1.

	X86-64 Intel Xeon Harpertown	X86-64 Intel Core 2 Quad Kentsfield	Sun UltraSPARC T1 Niagara
Num cores	8[a]	4	8 (32 threads)
Clock Rate (GHz)	2.0	2.4	1.2
L2 Cache size (MB)	6	4	3
OS (Kernel)	Fedora 8 (2.6.24)	Fedora 8 (2.6.24)	Solaris 10
Compiler	GCC 4.1.2	GCC 4.1.2	GCC 3.4.3

[a] Composed of two four-core chips

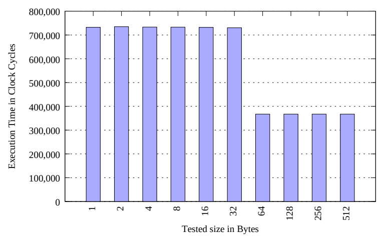
Table 1. Architectures tested

5.2 Experimental results

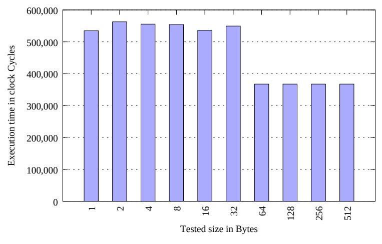
Coherence Block Size Figure 4 illustrates the results for Algorithm 1. On the Intel machines (Figure 4(a) and 4(b)), the threads were mapped to cores not sharing the L2 cache. The results show that, on the Intel machines, there is a notable time decrease as soon as the two accesses are 64-bytes apart. From this, we conclude that the coherence protocol on these machines uses 64-byte blocks.

On the Sun UltraSPARC T1 (Figure 4(c)), we observe that the largest performance difference occurs at the 16-byte block size, which corresponds to the size of the L1 data cache block.

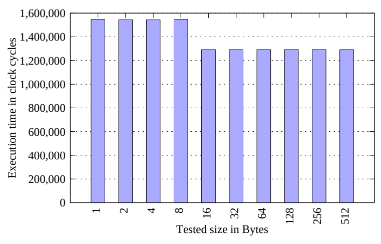
To show block sizes at the different cache levels and communication latencies, we evaluated different mappings of threads to cores. Figure 5 shows the results for the Intel Harpertown architecture, which has eight cores. We first mapped the threads to the two cores on the same dual-core. We then mapped the threads to cores on the same chip but not the same cache. Finally we mapped the threads to cores on different chips. We see that for this machine, while the block size is always 64-bytes, the different values of execution time show the different communication latencies among the different cores. There is a higher communication latency when the communicating cores are on different chips, and the communication cost is lower when the cores are closer together.



(a) Harpertown



(b) Kentsfield



(c) Niagara

Fig. 4. Coherence Block Size Results

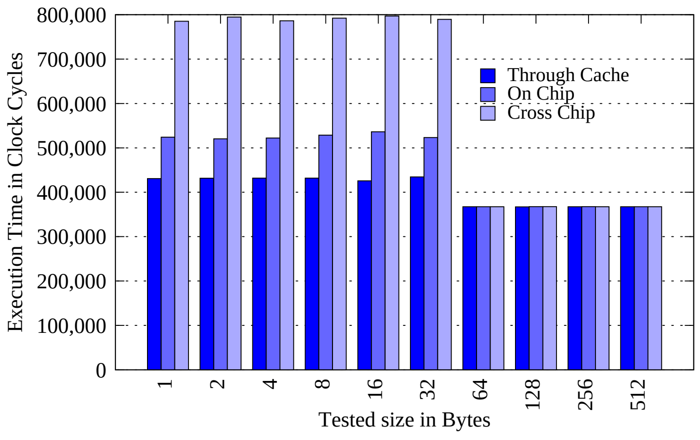
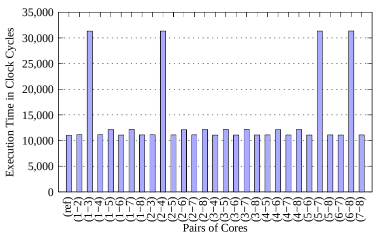


Fig. 5. Coherence block size and communication latency

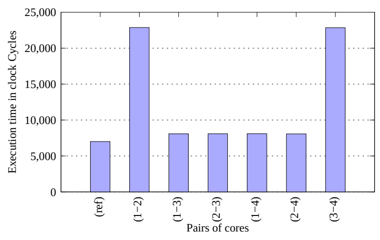
Cache Mapping Figure 6 illustrates the results for Algorithm 2. The results clearly show which of the cores share a cache.

By looking at the pairs of cores with the highest access times, we see that for the Harpertown (Figure 6(a)), core pairs of ID $(1-3)$,$(2-4)$,$(5-7)$, and $(6-8)$ share a cache, and for the Kentsfield (Figure 6(b)), the core pairs ID $(1-2)$ and $(3-4)$ share a cache.

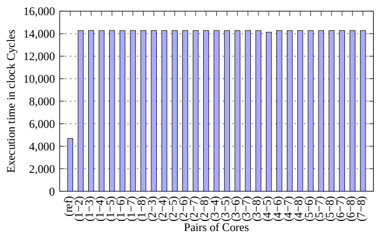
For the Sun UltraSPARC (Figure 6(c)), we see that all core pairs have the same performance. When comparing this performance with the single thread



(a) Harpertown



(b) Kentsfield



(c) Niagara

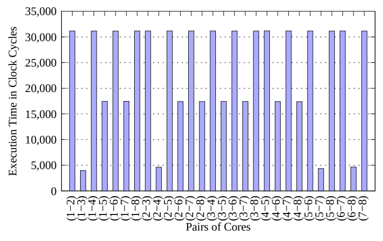
Fig. 6. Cache Mapping

reference time, we realize that all core pairs perform poorly and, thus, we infer that all cores share the same cache.

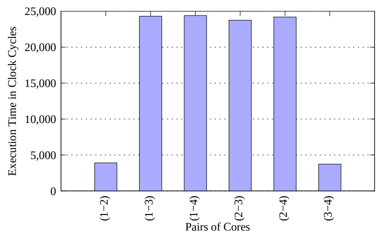
Processor Mapping Figure 7 illustrates the results for Algorithm 3. For the Harpertown (Figure 7(a)), we see three different distances. First, we have the pairs of cores that are the closest. These pairs correspond to those that share a cache in Figure 6(a): $(1-3)$,$(2-4)$,$(5-7)$, and $(6-8)$. Then we have two groups of four cores, where communicating between pairs in a group is faster than communicating between cores in different groups: $((1-3)(5-7))$ and $((2-4)(6-8))$. Those results confirm what is found on the design of the two architectures: the machine is composed of two four-core chips, with each four-core chip composed of two combined dual-cores. For the Kentsfield (Figure 7(b)), we confirm that pairs of cores that share a cache communicate faster.

For the Niagara (figure 7(c)), results show that all cores are equidistant, which confirms the results obtained for the cache mapping.

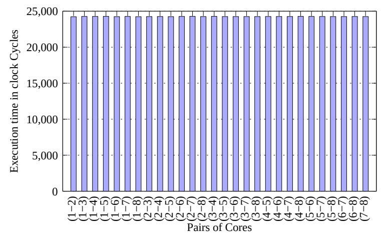
Effective L2 Bandwidth Figure 8 illustrates the results for Algorithm 4. We show data for each platform when a thread is run in isolation and when two or more threads run concurrently. By looking at results for one thread, we observe that, for all Intel machines, the execution time decreases as the number of independent accesses that can be issued simultaneously increases. There is a saturation point where the execution time remains constant. The smallest number of independent accesses where the saturation point is reached tell us the number of requests that can be served in parallel. When two or more threads run



(a) Harpertown

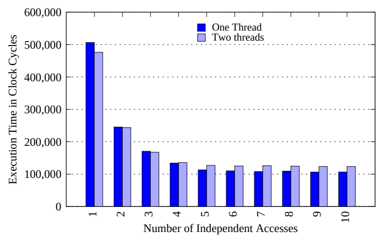


(b) Kentsfield

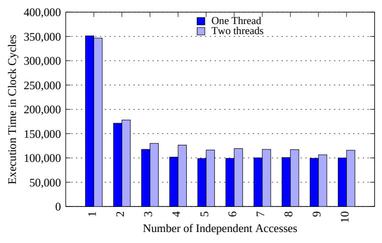


(c) Niagara

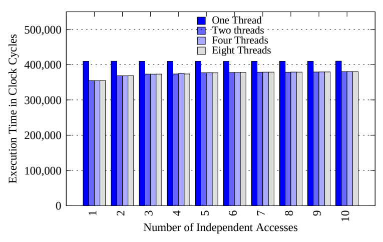
Fig. 7. Processor Mapping



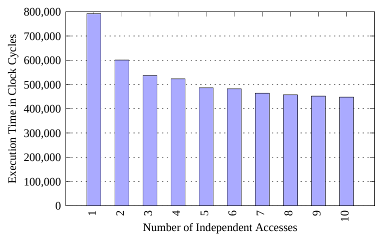
(a) Harpertown



(b) Kentsfield



(c) Niagara



(d) Niagara with branch penalty

Fig. 8. Effective Bandwidth to L2 Results

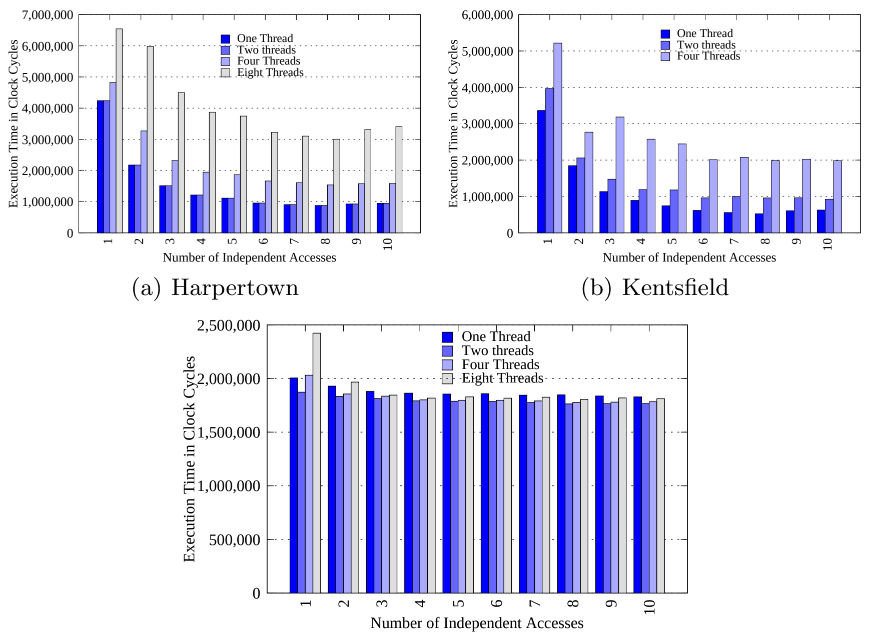
concurrently, the bars have a similar trend but have a slightly higher execution time.

Figure 8(d) shows the execution time for the benchmark run without removing the loop overhead. The improvement in execution time looks like it comes from more parallelism between memory requests. In fact, the Niagara does not have branch prediction; the reduction in execution time is only due to the decrease in number of iterations (i.e. number of branches per access).

Finally Table 5.2 shows the bandwidth computed using the formula shown in Section 3.4 for the different number of threads.

Machine	L1 Block	Cycles per Access (concurrent accesses)				Effective Bandwidth GB/s			
# threads		1	2	4	8	1	2	4	8
Harpertown	64B	6.18 (9)	7.54 (9)	-	-	19.29	15.81	-	-
Kentsfield	64B	6.05 (6)	6.50 (9)	-	-	23.65	22.01	-	-
Niagara	16B	25.00 (6)	21.66 (1)	21.64 (1)	21.66 (1)	0.71	0.83	0.83	0.83

Table 2. Effective Bandwidth to L2



(a) Harpertown

(b) Kentsfield

(c) Niagara

Fig. 9. Effective Bandwidth to Memory Results

Effective Memory Bandwidth Figure 9 illustrates the results for Algorithm 4. For the Intel processors (Figure 9(a) and 9(b)), as the number of cores accessing memory increases, we observe a substantial decrease in bandwidth.

On the Niagara (Figure 9(c)), we observe similar results as for the L2 cache; the bandwidth available to the cores stays the same regardless of the number of concurrent requests. Finally, Table 3 shows the values for the effective bandwidth.

Machine	L2 Block	Cycles per Access (concurrent accesses)				Effective Bandwidth GB/s			
# threads		1	2	4	8	1	2	4	8
Harpertown	64B	52.09 (8)	53.60 (8)	93.92 (9)	183.30 (9)	2.29	2.22	1.29	0.65
Kentsfield	64B	32.03 (8)	56.49 (10)	121.02 (10)	-	4.47	2.53	1.18	-
Niagara	64B	111.62 (10)	107.61 (8)	108.45 (8)	110.14 (8)	0.64	0.67	0.66	0.65

Table 3. Effective Bandwidth to Memory

6 Related Work

As discussed in the introduction, there are other micro-benchmarks such as LM-Bench[4], Saavedra[6], and X-Ray[11] that measure architectural characteristics and, while these benchmarks focus on single core features, P-Ray focuses on multi-core specific features. Other benchmark suites like SPEC OMP[3] are designed for shared memory multiprocessors. Such benchmarks only give a relative performance scale, but do not give any information about the characteristics of the targeted system.

7 Future Work

We are looking into additional benchmarks to add to this suite. While synchronization contention is mostly a library/OS issue, it could be a useful feature. It would be also be beneficial to have a similar suite of benchmarks for heterogeneous systems such as the Cell or GPU-based architectures.

From feedback received, there is a need for these multi-core benchmarks to be completed online as well as offline. This would be beneficial for projects such as adaptable systems with dynamic hardware features[5]. If we are to do online processing of certain benchmarks, execution time will have to be addressed to minimize the overhead of using our framework.

It would be interesting to run our benchmark suite on additional hardware architectures, such as IBM Power6 and Intel Itanium 2.

8 Conclusion

We have developed a suite of conceptually simple micro-benchmarks that focuses on multi-core characteristics. Our suite returns results that are in accordance with vendor specifications when available and coherent when they are not.

[3] Standard Performance Evaluation Corporation. `http://www.spec.org/omp`

The main difference between P-Ray and existing benchmarking suites is that P-Ray offers a unique view of the system design, showing the position of the different cache levels and relative distances between (virtual) cores in the system. With this information at hand, a programmer has the ability to use more efficient hardware-aware optimizations in their applications. In addition, we provide a faster means to calculate a cache block size by exploiting false sharing. Finally, the execution and analysis framework is extensible, allowing for the addition of benchmarks.

Acknowledgments We would like to thank Dario Suarez Garcia (University of Zaragoza, Spain), Basilio Fraguela (University of Coruña, Spain), and James Brodman (University of Illinois at Urbana Champaign) for their advice and comments.

References

1. In personal communication with Basilio B. Fraguela, Universidade da Coruña.
2. M. Frigo. A Fast Fourier Transform Compiler. In *PLDI'99 — Conference on Programming Language Design and Implementation*, 1999.
3. V. U. Instruction. Intel architecture software developer's manual.
4. L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *ATEC '96: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 23–23. USENIX Association, 1996.
5. M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.
6. R. Saavedra. Characterizing the performance space of shared memory computers using micro-benchmarks, 1993.
7. R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst.*, 14(4):344–384, 1996.
8. J. Torrellas, A. Tucker, and A. Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 24(2):139–151, 1995.
9. R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Sofware and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.
10. K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, 93(2):358–386, Feb. 2005.
11. K. Yotov, K. Pingali, and P. Stodghill. X-ray: A tool for automatic measurement of hardware parameters. In *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems on The Quantitative Evaluation of Systems*, page 168. IEEE Computer Society, 2005.