Cache optimization for CPU-GPU heterogeneous processors*

Lázár Jani and Zoltán Ádám Mann
Department of Computer Science and Information Theory,
Budapest University of Technology and Economics, Hungary

Abstract

Microprocessors combining CPU and GPU cores using a common last-level cache pose new challenges to cache management algorithms. Since GPU cores feature much higher data access rates than CPU cores, the majority of the available cache space will be used by GPU applications, leaving only very limited cache capacity for CPU applications, which may be disadvantageous for overall system performance. This paper introduces a novel cache management algorithm that aims at determining an optimal split of cache capacity between CPU and GPU applications.

Keywords: Cache management; Cache partitioning; Heterogeneous processors; Multicore processors; CPU cores; GPU cores

1 Introduction

The continuous development of the semiconductor industry has sustained the unbelievable exponential growth rate of the number of transistors on a chip, known as Moore's law, for several decades. For many years, this trend has come along with an increase of clock frequency of digital circuits. However, in the mid 2000s, this trend came to an end: further increasing the clock frequency would have led to intolerable power density and heat dissipation. This phenomenon, called *power wall*, completely changed the industry. Further increasing the performance of computer systems is not possible anymore by increasing the performance of a single thread of execution, but only by parallelization [1]. As a result, processor manufacturers turned their attention to multicore designs, where multiple processing units (processor cores) are integrated in a single chip.

Unlike CPUs (Central Processing Units) that traditionally supported sequential programs, GPUs (Graphical Processing Units), are typically designed to work on multiple data items in parallel, in a single-program-multiple-data fashion. As a result, GPUs offer very high throughput. In the last couple of years, GPUs have been increasingly used for non-graphical computations as well [8].

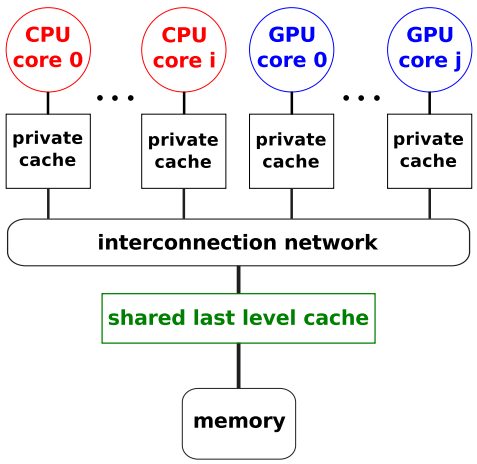


Figure 1: Architecture of a heterogeneous processor with shared LLC

*This paper was published in *American Journal of Algorithms and Computing*, volume 2, number 1, pages 18-31, 2015

A new trend is to combine CPU and GPU cores in the same chip, resulting in a *heterogeneous processor*. Examples of this trend are Intel's Sandy Bridge, Ivy Bridge, and Haswell architectures, just like AMD's Llano, Trinity, and Kaveri. Integrating CPU and GPU in the same chip offers several advantages, especially concerning the streamlined communication between CPU and GPU. Heterogeneous processors also offer the possibility for CPU and GPU to share some resources, e.g. the last-level cache (LLC). A schematic architecture diagram of such a processor is shown in Figure 1.

A shared cache is useful in improving the performance of applications that use both CPU and GPU cores, because it enables the fast sharing of data between CPU and GPU. On the other hand, sharing the cache between CPU and GPU cores also leads to two new challenges. Both are rooted in the much higher levels of parallelism offered by GPU cores compared to CPU cores:

- GPU applications can reach much higher data access rates than CPU applications. As a result, the majority of the available cache space will be used by GPU applications, leaving only very limited cache capacity for CPU applications.
- When a thread in a GPU application must wait for data from the main memory, there are usually many other threads that can execute in the meantime. Thus, cache misses typically have limited impact on the performance of GPU applications. On the other hand, CPU applications usually have few threads, so that the latency of main memory accesses does have significant impact on overall application performance in case of cache misses. As a result, CPU applications are usually more sensitive to the size of the available cache than GPU applications.

Putting together these two aspects, it can be stated that in a heterogeneous processor, CPU applications tend to obtain a relatively small part of the capacity of the shared cache, although they would benefit more from it than GPU applications do.

To overcome this problem, previous research suggested to partition the cache between the CPU and GPU cores [6, 9]. This way, it can be guaranteed that also CPU applications get a fair share of the cache. Technically, this is accomplished by partitioning the number of cache ways between the CPU and GPU. The previous works considered two approaches to determine the share of the CPU and GPU, respectively, in the cache. The first approach is *static partitioning*, in which a constant percentage (specifically, 50%) of the cache is reserved for the CPU, the rest for the GPU. The other, more sophisticated approach is *dynamic online partitioning*, in which the behavior of the CPU and GPU applications is analyzed at runtime to determine how sensitive they are to cache size, and the partitioning is adjusted to reflect this.

Both approaches were shown to lead to some improvements over standard cache management algorithms that are not aware of the heterogeneity of the cores. Nevertheless, both approaches have serious drawbacks. Static partitioning does not take into account the characteristics of the applications; since the cache sensitivity of both CPU and GPU applications can vary significantly, static partitioning will deliver suboptimal results in many cases, leading to poor usage of the available cache capacity. Dynamic online partitioning largely eliminates this problem by adapting the partitioning to the characteristics of the given applications. However, this approach is associated with considerable hardware overhead. Moreover, measuring application cache sensitivity may also temporarily degrade the performance of the application.

In this paper, we propose a new approach to strike a balance between the ability to adapt to the applications' characteristics and the method's overhead. Our approach is *dynamic offline partitioning*: it relies on historical data on the applications' cache sensitivity to determine an optimal partitioning when the applications start. In most computer systems – whether in an embedded, desktop, or server environment – the same applications are run again and again. Therefore, information on the applications' performance with different cache settings is piling up and can be used for future decisions on cache settings. Our algorithm makes use of this information to estimate how much each application would benefit from different cache sizes, and determines the partition that is likely to be the overall optimum based on these estimates. This way, our algorithm is run only when the applications start, thereby eliminating any interference with the applications during their run and minimizing overhead.

The rest of this paper is organized as follows. Section 2 presents an overview of previous work. Section 3 shows an analysis of CPU and GPU applications' cache sensitivity, followed by the description of our cache management algorithm for heterogeneous processors in Section 4. Empirical results are presented in Section 5, while Section 6 concludes the paper.

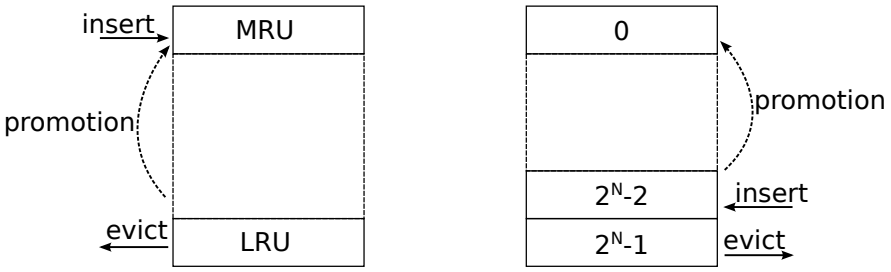
2 Previous work

The problem that different applications can have different cache sensitivity existed also before the advent of heterogeneous processors (although heterogeneous processors considerably aggravate the problem). Traditional solutions

can be grouped into two categories: cache partitioning techniques and special replacement policies.

Cache partitioning techniques were pioneered by [13] and later extended by [12, 10, 15]. These are dynamic online approaches that monitor application performance during runtime and adapt the partitioning of the cache between the applications during runtime. Their objective is to maximize the number of cache hits. Partitioning is carried out by splitting the cache ways among the applications.

Traditionally, cache replacement policies are based on the LRU (Least Recently Used) principle: when a new piece of data enters the cache and a cache line needs to be freed to accommodate the new data, then the least recently used data block is sacrificed [3]. Technically, this can be realized with a stack of height 2^N, in which new data are entered in position 0, the MRU (Most Recently Used) position, pushing down all other items by one position, and the data item that was in the LRU position with index $2^N - 1$ is removed. When a data item that is in the cache is accessed again, it is promoted to the MRU position (see Figure 2(a)).



(a) LRU policy (b) RRIP policy

Figure 2: Comparison of different cache replacement policies

The LRU policy performs poorly for applications that have either a working set that is larger than the cache or that exhibit streaming behavior, i.e., no reuse of data. In such cases, data items enter the MRU position, then move down towards the LRU position one by one, until they drop off the LRU position. Hence, data blocks occupy the cache for a long time, without any benefit. In order to reduce the negative impact of such behavior, several *alternative replacement policies* have been suggested in the literature [11, 14, 4]. In particular, the RRIP (Re-Reference Interval Prediction) policy enforces a shorter lifetime for data items that are not reused, by inserting them near the LRU position. If a data item is reused, then it is promoted to MRU, but otherwise it is quickly evicted (see Figure 2(b)). RRIP also has some variants, based on where exactly new items are inserted and how much they are promoted in case of reuse.

Specifically the problem of shared LLC in a heterogeneous processor was addressed by two previous papers [6, 9]; these are the closest to our work.

The approach of Lee and Kim, named TAP (thread-level parallelism aware cache management policy) consists of two techniques: core sampling and cache block lifetime normalization. Core sampling aims at determining what policy is the most advantageous for the given applications. To that end, two cores are selected and two very different policies are applied to them. If the application is cache-sensitive, the performance of the two cores will likely differ significantly, otherwise it will not. Of course, the implicit assumption behind this idea is that threads belonging to the same application but running on different cores are homogeneous in terms of performance and cache-sensitivity. Cache block lifetime normalization detects differences in the rate of cache accesses and uses this information to enforce similar cache residential time for CPU and GPU applications. TAP has been implemented both as an extension to existing cache partitioning techniques (TAP-UCP) and as an extension to existing alternative replacement strategies (TAP-RRIP). The authors reported speedups of up to 12% over LRU.

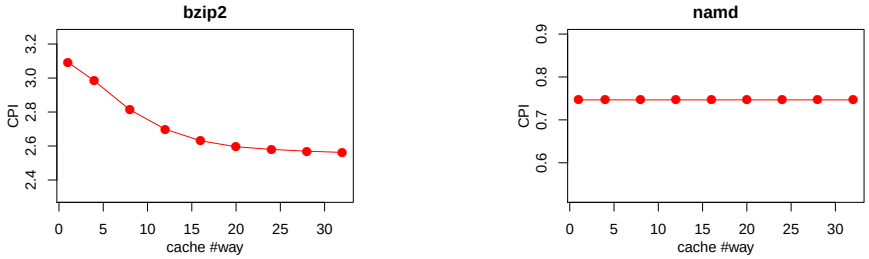
The work of Mekkat et al., termed HeLM (heterogeneous LLC management), goes one step further. It detects the level of thread-level parallelism (TLP) available in GPU applications; if the TLP is high, then the GPU application can likely tolerate cache misses. In this case, HeLM will let the GPU's data accesses selectively bypass the LLC and direct them straight to the main memory. This way, more cache space remains for CPU applications that usually cannot tolerate memory access latencies. To achieve this behavior, HeLM also uses core sampling to continuously measure both GPU and CPU cache sensitivity, and LLC bypassing is activated if the cache sensitivities are over given thresholds. The necessary threshold values are determined dynamically in order to adapt to the applications' characteristics. The authors reported speedups of 12.5% over LRU.

Our approach is conceptually different from the above approaches in that we make partitioning decisions offline, based on historical data, instead of at runtime. This way, we can avoid both the negative effects on performance caused by online monitoring (e.g., core sampling) and the special hardware requirements of the above approaches.

It is also worth mentioning that Lee and Kim also experimented with static cache partitioning, but only in its simplest form, where 50% of the cache is reserved for CPU applications and the other 50% is reserved for GPU applications. They found that this simple static partitioning slightly improves average performance compared to LRU, but it actually performs worse than LRU on several benchmarks [6].

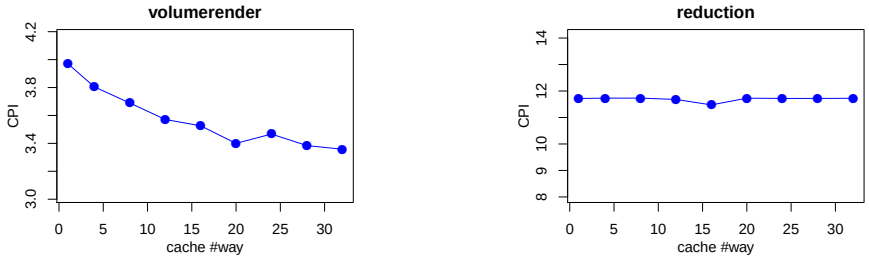
Our approach differs from static partitioning as it adapts to the applications' characteristics.

3 Cache sensitivity



(a) Cache-sensitive application (b) Cache-insensitive application

Figure 3: Examples for the cache-sensitivity of different CPU applications



(a) Cache-sensitive application (b) Cache-insensitive application

Figure 4: Examples for the cache-sensitivity of different GPU applications

We started by performing some experiments to assess the cache sensitivity of different CPU and GPU applications. We varied the number of cache ways, thus investigating different cache sizes (all other parameters equal, the cache size is proportional to the number of cache ways). We measured application performance by the average number of cycles per instruction (CPI). The CPI of an application can be determined by measuring the number of clock cycles the execution of the application takes and counting the number of instructions that it executes; CPI can then be calculated as

$$CPI = \frac{\#cycles}{\#instructions},$$

where $\#cycles$ is the number of clock cycles and $\#instructions$ is the number of instructions executed. The lower the CPI value, the faster is the execution of the application.

Some results are shown in Figures 3-4. As can be seen, there are applications – both CPU (Figure 3(a)) and GPU (Figure 4(a)) applications – where increasing the cache size does lead to improved performance. On the other hand, there are also applications the performance of which is practically independent on the cache size; this is possible both on the CPU (Figure 3(b)) and the GPU (Figure 4(b)). We also found that the majority of the investigated CPU applications is cache-sensitive and the majority of the investigated GPU applications is cache-insensitive, but all four combinations occur. These findings are in line with previous results in the literature.

We observed also another phenomenon that was previously not reported in the literature. While previous papers [6, 9] speak about cache-sensitive and cache-insensitive applications, the cache-sensitivity is actually not a characteristic of the application alone. We found that cache-sensitivity can also depend heavily on the size of input data. This phenomenon is exemplified in Figure 5, where we see the results of running the same application with

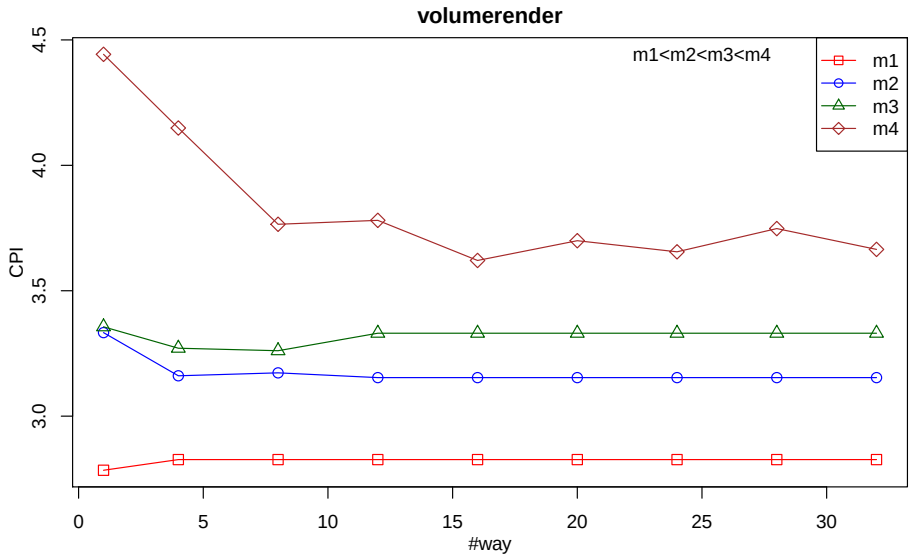


Figure 5: Dependence of the cache-sensitivity on input data size

inputs of different size. As can be seen, the cache sensitivity is very low for small inputs (inputs of size m1, m2, and m3), but significantly higher for a bigger input (input of size m4). A possible explanation of this phenomenon can be that for small inputs, the working set of the application is small enough to fit into the smallest investigated cache, corresponding to a single cache way, so that more cache space does not improve application performance, but for a bigger input, the working set may also be bigger, so that more cache space is helpful in this case. As a consequence, if we want to predict application performance depending on available cache size, we also need to take the size of the input into account.

A related aspect is the number of threads the applications are using. The considered CPU benchmarks are single-threaded, regardless of the size of the input. However, the GPU applications may exhibit different threading behavior depending on the size of the input.

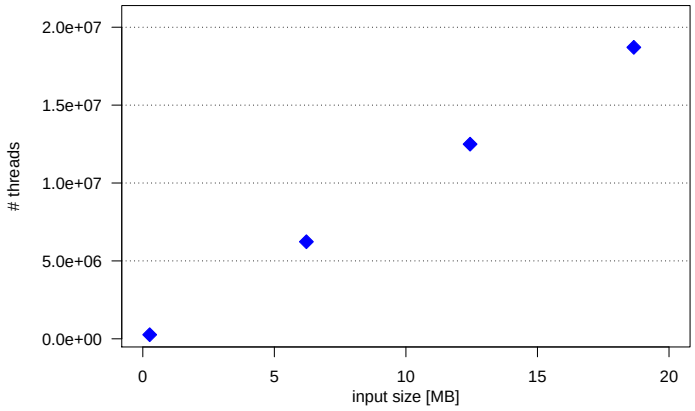
- Some GPU applications use constant number of threads. For example, the blackscholes benchmark application uses 61,440 threads, regardless of the size of the data to be processed.
- Some other GPU applications increase the number of threads if the input size increases, but their execution remains structurally the same. For example, in the bicubic benchmark application, the number of threads scales linearly with the size of the input, see Figure 6(a).
- Other applications change their behavior more fundamentally. For example, the montecarlo benchmark application starts one more kernel with a high number of threads if the input size surpasses a given threshold, see Figure 6(b).

For our purposes, the applications can be considered as black box, without explicitly taking into account how the number of threads is set internally by the application. Rather, the number of threads impacts the CPI and the cache sensitivity of the application, and so it will be implicitly taken into account through the CPI's sensitivity to input size and cache size.

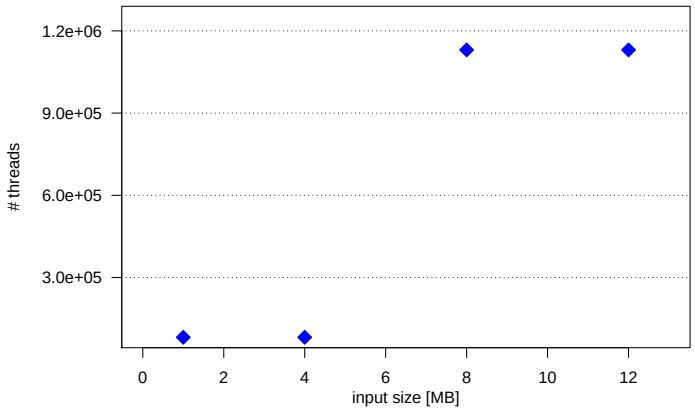
4 Dynamic offline partitioning

In this section, we present our algorithm LP4HP (LLC Partitioning for Heterogeneous Processors). Its aim is to determine the optimal partitioning of the cache between a CPU and a GPU application. Our approach is offline in the sense that it partitions the cache before the applications start, thus avoiding any runtime interference with the applications. On the other hand, our approach is dynamic in the sense that it takes the cache sensitivity of the applications into account and partitions the cache accordingly.

For this purpose, we consider the dependence of application performance on cache size and input data size, denoted as $CPI(s, w)$, where s is the cache size and w the input data size. We assume that some discrete points of this 3-dimensional surface are available, see Figure 7 for an example. In other words, we assume that the CPI of the applications has already been measured for some specific values of s and w – each such measurement results



(a) bicubic



(b) montecarlo

Figure 6: Number of threads, depending on input size, for different GPU applications

in a new point in the $(s, w, CPI(s, w))$ space. The justification of this assumption is that in a typical computer system – be it in an embedded, desktop, or enterprise environment – the same application is typically run many times, with different inputs and different available cache capacities (the latter depending on the other applications that run at the same time). Thus, over time, an ever increasing number of discrete $(s, w, CPI(s, w))$ tuples is available, leading to a good approximation of the real $CPI(s, w)$ surface. In the very beginning, of course, no cache sensitivity information is available. In this case, a default partition can be used, for example, a random partition. Later on, more and more experience is gathered with every run of the application, resulting in an ever improving approximation of the $CPI(s, w)$ function.

Our algorithm is invoked when the applications are about to be started. At this moment, the actual input data size $\bar{w}$ of the application is known. In order to determine the best cache partitioning, we need the $s \mapsto CPI(s, \bar{w})$ function in order to know how beneficial a possible cache size is for the given application. Therefore, the LP4HP algorithm first interpolates the $CPI(s, \bar{w})$ values based on the nearest available $(s, w, CPI(s, w))$ tuples.

When the approximate $s \mapsto CPI(s, \bar{w})$ function of each application has been generated, the optimal cache partitioning can be determined. This is done by considering each possible partition. If there are m cache ways, then those cache ways can be partitioned between CPU and GPU in $m + 1$ possible ways: $(0, m), (1, m - 1), (2, m - 2), \ldots, (m - 1, 1), (m, 0)$. From the $m + 1$ theoretical possibilities, we exclude the two extremes in order to avoid the pathologic situation where an application does not receive any cache space at all. It should be noted that, for practical settings, the number of possible cache partitions is not high; e.g., there are 31 possibilities to partition a 32-way cache between two applications such that both get at least one cache way. Thus, we can simply calculate the estimated CPI values of the applications for all possible partitions, and based on this, select the best partition.

There is one more detail to be clarified: the exact objective function. This determines how, for a given cache partitioning, the CPI values of the applications should be combined to a single value characterizing the performance of the given partitioning. We experimented with two different objective functions. The first one is the overall throughput of the system, i.e. the total number of instructions performed per unit time (IPS, Instructions

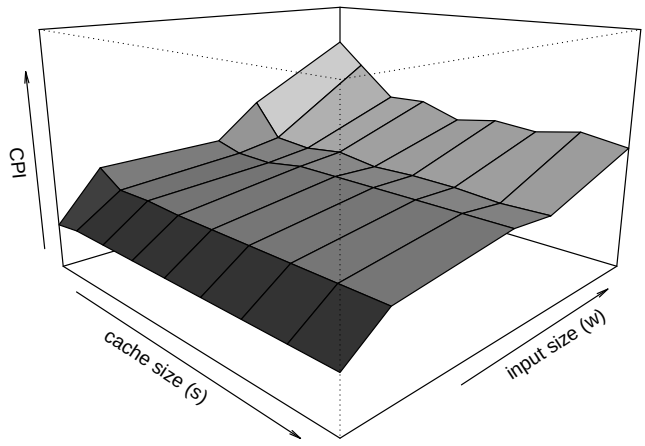


Figure 7: $CPI(s, w)$ function of a given application

Per Second):

$$IPS_{total} = IPS_{CPU} + IPS_{GPU} = \frac{\#instructions_{CPU}}{t_{CPU}} + \frac{\#instructions_{GPU}}{t_{GPU}} = \frac{f_{CPU}}{CPI_{CPU}} + \frac{f_{GPU}}{CPI_{GPU}},$$

where t_{CPU} and t_{GPU} denote CPU/GPU time and f_{CPU} and f_{GPU} denote the clock frequency of the CPU and GPU, respectively.

The second objective function that we used is the geometric mean of the speedups of the CPU and GPU applications. The speedup of an application, for a given cache size $\bar{s}$, is defined as

$$speedup(\bar{s}) = \frac{IPS(\bar{s})}{\min(IPS(s))}.$$

To sum up, the high-level steps of the LP4HP algorithm are as follows:

1. For each application to be started, and for its actual input size $\bar{w}$, interpolate the $s \mapsto CPI(s, \bar{w})$ function based on the nearest available $(s, w, CPI(s, w))$ tuples.
2. For each possible cache partitioning, calculate its value using the objective function (either total IPS or geometric mean of speedups).
3. From the values calculated in Step 2, select the best one, thus determining the best partition.

The time complexity of the algorithm is linear with respect to the number of cache ways. For practical scenarios, the algorithm is very fast, resulting in negligible overhead.

It is worth noting that the algorithm ensures that both the CPU and GPU applications will always be assigned at least one cache way.

5 Empirical results

In order to evaluate our algorithm, we implemented it and tested it using MacSim, one of the few simulators that support the modeling of heterogeneous processors [5]. MacSim is a trace-based simulator; the traces of the CPU applications can be created using Pin [7], whereas the traces of GPU applications with GPUOcelot [2]. The overall simulation flow is depicted in Figure 8.

The details of the simulated heterogeneous processor are shown in Table 1.

For our tests, we used SPEC CPU 2006 benchmarks as CPU applications. For the GPU, we used sample applications from the CUDA SDK. For all applications, we considered 4 different input sizes, and measured their performance for 9 different cache sizes, by varying the number of available cache ways between 1 and 32 with steps of 4. Therefore, the database of 'historic' $(s, w, CPI(s, w))$ tuples consists of 36 items for each application.

As described in Section 4, when the LP4HP algorithm faces an input size $\bar{w}$ for which it has no data, it first estimates the $s \mapsto CPI(s, \bar{w})$ function with linear interpolation from the nearest available data points. In our experiments we found that the difference between the CPI estimated this way and the real CPI (that we measured using MacSim) is 8% on average.

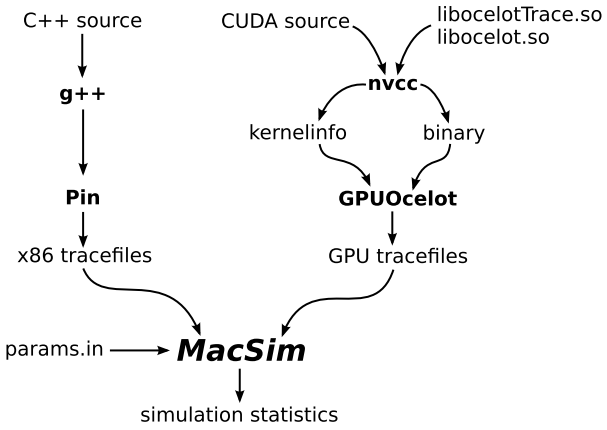


Figure 8: Simulation environment

Table 1: Simulated heterogeneous processor configuration

CPU	L1I cache: 32 KB, 2-way set-associative
	L1D cache: 16 KB, 4-way set-associative, line size 64 byte, 3 cycles latency
	L2 cache: 256 KB, 8-way set-associative, line size 64 byte, 8 cycles latency
	4 cores, 3 GHz
	4-wide superscalar, out-of-order instruction scheduler
	gshare branch predictor
GPU	6 cores, 1.5 GHz
	no branch predictor
	L1I cache: 4 KB, 2-way set-associative, line size 64 byte, 2 cycles latency
	L1D cache: 16 KB, 4-way set-associative, line size 64 byte
LLC	1 GHz, 30 cycles latency, 8 bank, 1 cycle latency
	8 MB, 32 associative, line size 64 byte
Memory	dual channel, 1.6 GHz

We tested 14 combinations of one CPU and one GPU application. In each case, when one of the applications finishes while the other is still in its first run, then the finished application is restarted, so that both applications are active throughout the whole simulation. (This way, a simulation run took up to 10 hours.)

In line with previous research results and with our expectations, the presented algorithm can lead to significant improvements over the standard LRU method only if the CPU application is cache sensitive and the GPU application is cache insensitive. In other cases, LP4HP hardly affects the overall system performance. However, in the interesting case it leads indeed to considerable performance improvement, as shown in Figures 9-10. Specifically, Figure 9 shows the results of optimizing the overall system throughput (IPS). As can be seen, overall IPS improves by 6-13% compared to LRU. It should be noted that in order to improve overall system throughput, it is necessary that the CPU and GPU applications have comparable throughput. In several cases, the throughput of the GPU application is much higher, so that improving the performance of the CPU application has hardly any effect on overall IPS.

Figure 10 shows the results attained when optimizing the geometric mean of the CPU's and GPU' speedup. This metric allows a more differentiated optimization in the cases where the throughput of the two applications are very different. In this figure, the speedup of both the CPU and GPU applications are shown: for each pair of applications, the first bar shows the speedup of the CPU application, whereas the second bar shows the speedup of the GPU application. As can be seen, there is some modest speedup (0-4%) on the GPU side, but the CPU's performance is improved by up to 42%.

Overall, the saving in runtime achievable with our method is comparable to what has been reported previously for dynamic online approaches. On the other hand, we avoid the overhead associated with online approaches. More specifically, the overhead of previous online approaches consists of

- Counters and registers, requiring extra chip area. The method of [6] requires altogether 120 bits, the method of [9] requires 166 bits of extra counters and registers.

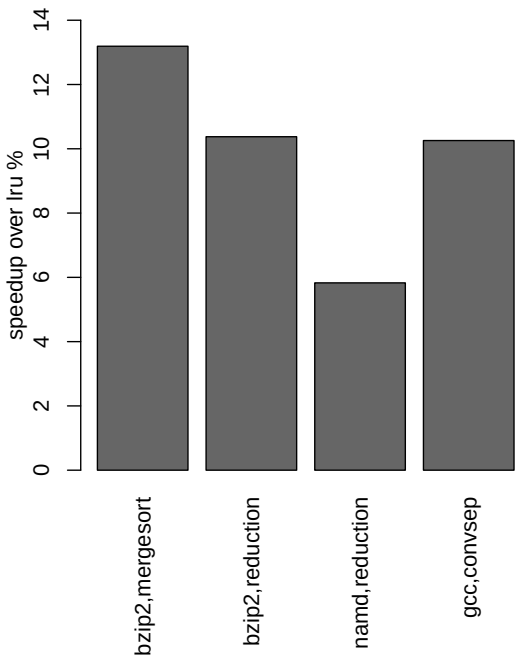


Figure 9: Overall speedup

- Resources for algorithm execution. Although the previously proposed online methods are not very complicated, but their hardware implementation requires significant additional chip area because of the use of division operations. Alternatively, the cache partitioning algorithms can be implemented in software, taking away some CPU time from the applications.
- Suboptimal cache parameters for measurement purposes. Core sampling uses different cache parameters for two cores in order to find out whether performance is influenced. Thus a deliberate performance deterioration is used in order to determine cache sensitivity.

6 Conclusions and future research

In this paper, we introduced LP4HP, an algorithm for determining the optimal partitioning of the last-level cache jointly used by CPU and GPU cores in a heterogeneous processor. The main novelty of this algorithm is its dynamic offline nature, which lets it avoid any interference with application performance at runtime, and at the same time also enabling the adaptation to application characteristics. The empirical results have shown that, for cache-sensitive CPU applications and cache-insensitive GPU applications, LP4HP achieves significant speedup over LRU. The results are similar to the ones previously reported for dynamic online approaches, but without the overhead of online methods.

As a future research direction, more complex interaction scenarios with more than two concurrent applications should be investigated; we expect that LP4HP can be extended in a straight-forward way, although at the price of increased running time. Another possible extension of LP4HP would allow it to not only partition the cache but also specify – based on historic data – the desired lifetime of cache lines for each application. This information can then be used by a RRIP-style cache replacement policy to determine where to insert new lines in the cache.

Acknowledgement

This work was partially supported by the Hungarian Scientific Research Fund (Grant Nr. OTKA 108947).

References

[1] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: Recording microprocessor history. *Communications of the ACM*, 55(4):55–63, 2012.

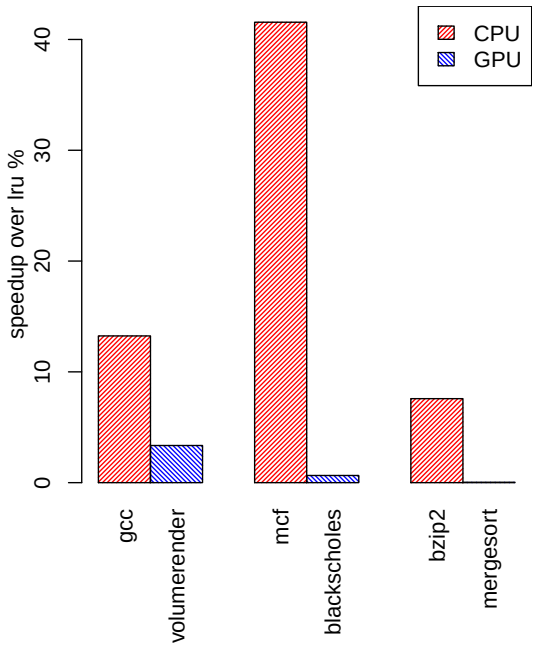


Figure 10: Speedup of CPU and GPU

[2] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*, pages 353–364, 2010.

[3] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.

[4] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*, pages 60–71, 2010.

[5] Hyesoon Kim, Jaekyu Lee, Nagesh B. Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. MacSim: A CPU-GPU heterogeneous simulation framework. `http://comparch.gatech.edu/hparch/macsim/macsim.pdf`, 2012.

[6] Jaekyu Lee and Hyesoon Kim. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *Proceedings of the 18th IEEE International Symposium on High-Performance Computer Architecture (HPCA '12)*, pages 1–12, 2012.

[7] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 190–200, 2005.

[8] Zoltán Á. Mann. GPGPU: Hardware/software co-design for the masses. *Computing and Informatics*, 30(6):1247–1257, 2011.

[9] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. Managing shared last-level cache in a heterogeneous multicore processor. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*, pages 225–234, 2013.

[10] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. MLP-aware dynamic cache partitioning. In *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers (HiPEAC '08)*, pages 337–352, 2008.

[11] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*, pages 381–391, 2007.

[12] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*, pages 423–432, 2006.

[13] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[14] Yuejian Xie and Gabriel H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, pages 174–183, 2009.

[15] Yuejian Xie and Gabriel H. Loh. Scalable shared-cache management by containing thrashing workloads. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC '10)*, pages 262–276, 2010.