Predictive Runtime Code Scheduling for Heterogeneous Architectures

Víctor J. Jiménez[1], Lluís Vilanova[2], Isaac Gelado[2], Marisa Gil[2], Grigori Fursin[3], and Nacho Navarro[2]

[1] Barcelona Supercomputing Center (BSC)
victor.javier@bsc.es
[2] Departament d'Arquitectura de Computadors (UPC)
{igelado,vilanova,marisa,nacho}@ac.upc.edu
[3] ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud University
grigori.fursin@inria.fr

All the authors are members of the HiPEAC European Network of Excellence

Abstract. Heterogeneous architectures are currently widespread. With the advent of easy-to-program general purpose GPUs, virtually every recent desktop computer is a heterogeneous system. Combining the CPU and the GPU brings great amounts of processing power. However, such architectures are often used in a restricted way for domain-specific applications like scientific applications and games, and they tend to be used by a single application at a time. We envision future heterogeneous computing systems where all their heterogeneous resources are continuously utilized by different applications with versioned critical parts to be able to better adapt their behavior and improve execution time, power consumption, response time and other constraints at runtime. Under such a model, adaptive scheduling becomes a critical component.

In this paper, we propose a novel predictive user-level scheduler based on past performance history for heterogeneous systems. We developed several scheduling policies and present the study of their impact on system performance. We demonstrate that such scheduler allows multiple applications to fully utilize all available processing resources in CPU/GPU-like systems and consistently achieve speedups ranging from 30% to 40% compared to just using the GPU in a single application mode.

1 Introduction

The objectives of this work are twofold. On the one hand, fully exploiting the computing power available in current CPU/GPU-like heterogeneous systems and thus, increasing overall system performance is pursued. On the other hand, exploring and understanding the effect of different scheduling algorithms for heterogeneous architectures is intended.

Currently almost every desktop system is an heterogeneous system. They both have a CPU and a GPU, two processing elements (PEs) with different

characteristics but undeniable amounts of processing power. Some time ago programming a GPU for general purpose computations was a major programming challenge. However, with the advent of GPUs designed with general purpose computation in mind it has become simpler. Games still represent the big market for graphical card manufacturers, but thanks to execution models like CUDA [2] now it is possible to use such GPUs as data parallel computing devices.

Applications with great amounts of data parallelism perform considerably better on a GPU than on a CPU. [13] Thus, the GPU is seen as a device destined to run very specific workloads. The current trend to program these systems is as following: (1) Profile the application to be ported to a GPU and detect the most expensive parts in terms of execution time and the most amenable ones to fit the GPU-way of computing (i.e., data parallelism). (2) Port those code fragments

on both CPU and GPU providing explicit data transfer if needed. Considering this study is performed on top of real hardware with a multi-ISA architecture, it does not seem feasible to use a granularity finer than function-level. Indeed, that level seems a good choice for the programmer to provide both versions of the code (CPU and GPU). It it important to mention that the generation of function versions for every PE is orthogonal to this work and it is expected for future compilers to be able to generate multiple versions which can be adaptively used at runtime. [6, 7] Additionally, there are already studies which allow to automatically generate a function version for one PE given the version for another PE. [16]

The code scheduler has been implemented as a dynamic library for the GNU/Linux OS. Being a process-level scheduler, parts of the library must be shared among all the processes which use it (see Figure 1). Specifically the data for the PE management and the task queues for each PE are shared (a task, composed by a function and its arguments, is used as the basic unit of scheduling from now on in this text). Other implementation options such as creating a kernel-level scheduler have been considered. However, it poses many difficulties, involving a longer development cost and the necessity to deal with NVIDIA's proprietary driver. Being simpler and, at the same time, enough to perform this study, the implementation uses the dynamic library approach. The interface to the scheduler is a set of C++ classes.

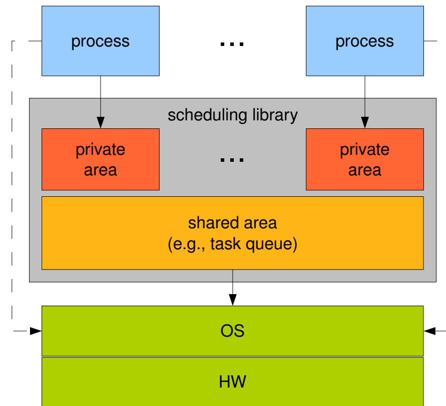

Fig. 1. Call scheduler implementation overview.

2.2 Usage

A matrix multiplication is used as an example of the scheduler usage. A typical implementation would provide a function, matrix_mul, which implements the matrix multiply operation. This function would be called with two input matrices and one output matrix. Additionally, the matrix size would be also provided:

```
matrix_mul(A,B,C,N);
```

If this function must run on several PEs, multiple implementations are necessary. In the case of a CPU and a GPU they could be named matrix_mul_cpu and matrix_mul_gpu. Considering this is already done, the interaction with the scheduler is quite simple. The user would get an instance of the main scheduler class (CallScheduler). Then the user constructs a call (Func, Args) and executes the schedule method. This creates a Task which is added to the queue for the PE selected by the scheduling algorithm. Different algorithms can be plugged-in in the scheduling system, thus making the system very flexible for trying new scheduling algorithms. Following is the extra code necessary to be able to let the scheduler select at runtime which PE will be used to perform the operation:

```
CallScheduler* cs = CallScheduler::getInstance();
MatrixMulFunc* f = new MatrixMulFunc();
MatrixMulArgs* a = new MatrixMulArgs(A,B,C,N);
cs->schedule(f,a);
```

where class MatrixMulArgs is a very simple class which just stores the values for the arguments to the function and MatrixMulFunc is a wrapper which allows to select the right function version to execute in a given PE. This is easily doable because CUDA stores both versions in the same executable file. In our implementation just a call to either matrix_mul_cpu or matrix_mul_gpu is necessary. Although it may seem a considerable amount of code, it is possible to use some "syntactic sugar" which would allow, for instance, a source-to-source compiler to generate all that code from a line similar to:

```
#pragma cs matrix_mul(A,B,C,N) matrix_mul_cpu matrix_mul_gpu
```

2.3 Scheduling Algorithms

In a heterogeneous scheduling process the following two steps can be distinguished: *PE selection* and *task selection*. The former is the process to decide on which PE a new task should be executed. It does not mean the execution is going to start at that time. The latter it is the mechanism to choose which task must be executed next in a given PE. It typically takes place just after another task finishes and its PE becomes free.

Several options have been tested for the first step. All the algorithms basically follow a variant of the *first-free* (FF) design, meaning that tasks are first tried to be scheduled in a PE which is not being used. As the results will show, this approach does not work consistently good all the time and thus, new algorithms based on *performance-prediction* have been developed.

For the second step, all the algorithms implemented in this work follow a *first-come, first-served* (FCFS) design. It could be also possible to implement some more advanced techniques such as work stealing in order to increase the load balance of the different PEs. However, the main goal of the study was to find algorithms which led to a good scheduling depending on the code to be

executed and the characteristics of the PEs present in the system. Thus, the study for load balancing techniques is left for future work.

Several variants of different families of algorithms have been developed. The following description gives the general scheme for these families. The specific parts for every variant are abstracted as calls to functions (g and h).

Algorithm 1 shows the general scheme for a FF design. It traverses the PE list in search for a not busy one. As soon as one is found it is selected as the target PE. If none is idle the algorithm must decide which PE to use. Several variants have been tried and thus the algorithm contains a call to a function g which will be responsible to select somehow a PE in case all of them are busy.

Algorithm 1 First-Free algorithm family.

```
for all pe ∈ PElist do
  if pe is not busy then
    return pe
return g(PElist)
```

As the CPU and the GPU present different characteristics, the same function may perform differently in both PEs. It could be the case that one of them performs better for some kind of tasks. Therefore a modification to the previous algorithm is introduced, allowing to queue more elements into one PE, thus introducing a bias in the scheduling system. This can also be seen as a simple load balancing mechanism. Algorithm 2 is still *first-free*-based, but in case all the PEs are busy it will assign tasks to PEs following a distribution given by a parameter $k = (k_1, \ldots, k_n)$. Given two PEs, a and b, the ratio k_a/k_b determines the amount of work which will be given to them. For instance, with $k = (1, 4)$ the number of tasks given to the second PE will be four times bigger.

Algorithm 2 First-Free Round Robin (k).

```
for all pe ∈ PElist do
  if pe is not busy then
    return pe
if k[pe] = 0 ∀pe ∈ PElist then
  set k with initial values
for all pe ∈ PElist do
  if k[pe] ≥ 0 then
    k[pe] = k[pe] − 1
    return pe
```

The idea behind this algorithm is that if a set of applications is biased towards one of the PEs, consistently obtaining better performance on it, the scheduler may address the load imbalance by biasing the assignment towards the other

PE. However it may also happen that performance for an application drastically differs depending on the PE where it is run. In those cases the previous algorithm may not perform well. This observation motivated the introduction of a *performance history*-based scheduler (algorithm 3). Basically a performance history is kept for every pair of PE and task. During the initial phase, the performance history is built by forcing the first n calls to the same function to execute on the different n PEs. In the next phase every time a call to that function is made, the scheduler looks for any big unbalance between the performance on the different PEs. Thus, a list ($allowed_{PE}$) is built where only the PEs without such a big unbalance are kept. If that list came to be empty, g would determine which PE to select. h performs the corresponding action when there is more than one possibility to schedule the task.

Algorithm 3 Performance History Scheduling.

```
if ∃pe ∈ PElist : history[pe, f] = null then
    return pe
allowedPE = {pe| ∄pe' : history[pe, f]/history[pe', f] > θ}
if ∃pe ∈ allowedPE : pe is not busy then
    return pe
if allowedPE = ∅ then
    return g(PElist)
else
    return h(allowedPE)
```

Relying on this performance prediction mechanism, a variant of algorithm 3 has been developed. It uses the performance history to predict the waiting time for every PE. This version aims at a better load balancing among the PEs.

3 Experimental Methodology

The runtime CPU/GPU scheduler has been evaluated on a real machine with a set of benchmarks. In the following subsections the benchmarks will be described in detail as well as the experimental setup.

3.1 Workload

A mix of synthetic and real benchmarks have been used in order to evaluate the performance speedup obtained with the use of the runtime code scheduler for the CPU/GPU system. The benchmarks used are: matmul, ftdock, cp and sad. Their performance characterization can be seen in table 1.

matmul is a synthetic benchmark which performs multiple square-matrix multiplications using either the ATLAS library [21] for the CPU and the CUBLAS library [2] for the GPU. As can be seen in table 1, performance on GPU does not extremely differ from performance on CPU. As the input size is increased, the

Benchmark	CPU	GPU	Speedup	TX time	Comp Time	Ratio
cp	28.79s	0.39s	74X	0.13s	0.14s	1.08
sad	0.79s	0.87s	$\sim$ 0.9X	0.11s	0.04s	0.36
FTDock	38.77s	19.99s	$\sim$ 1.9X	$\sim$ 0.03s	0.34s	11.75
matmul	38.52s	12.65s	3X	0.01s	0.04s	3.89

Table 1. Benchmark list characterization.

GPU can better amortize the cost of bringing data in and out to main memory. The matrices used are considerable large (1024×1024).

FTDock [8] is a real application which computes the interaction between two molecules (docking). FFTW [5] is used in order to speedup this process. A hybrid version has been developed allowing to execute any of the rotations either on the CPU or the GPU. NVIDIA's CUFFT library [2] is used for the GPU. The changes introduced in the program are minimal since both libraries have almost the same interface. Although the GPU version runs twice as fast, the difference with the CPU is not big.

The Parboil Benchmark Suite [1] is a set of benchmarks designed to measure the performance of a GPU platform. They are available from the Impact Research Group at University of Illinois (UIUC). The benchmarks used here are cp and sad. cp computes the coulombic potential at each grid point over on plane in a 3D grid in which point charges have been randomly distributed. sad is used in MPEG video encoders in order to perform a sum of absolute differences between frames. While sad performs almost equal on both PEs, cp does so much more efficiently on the GPU. The application speedup is really large, thus, it is really crucial to schedule the application to the right PE (the GPU in this case). Due to some constraints, such as a big memory footprint for some of the others benchmarks included in the suite, only those two benchmarks have been used to evaluate the scheduling system. The GPU has a limited amount of memory, but a recent work proposes an architecture which would remove this constraint allowing to use virtual memory from the coprocessor (the GPU in this case). [9]

3.2 Experimental Setup

All the experiments have been run on real hardware. A machine with an Intel Core 2 E6600 processor running at 2.40GHz and 2GB of RAM has been used. The GPU is an NVIDIA 8600 GTS with 512MB of memory. The operating system is Red Hat Enterprise Linux 5.

The execution of the benchmarks is organized as combinations of N benchmarks running in parallel. In order to evaluate the scheduling system, different values of N have been tried. The amount of memory on the GPU limits how many processes can be concurrently run on it. Therefore, the values selected for N are $N = \{4, 6\}$. In order to keep experimenting time under reasonable values, fifteen randomly selected combinations have been chosen from all the possible permutations. In early tests performed with all the executions it was not possible to observe any significant change in the results, so the number selected seems a good compromise between results accuracy and experimenting running time.

As a way to improve results accuracy, experiments are repeated a small number of times and the results are averaged. The results obtained show that variance is not very high between executions ($< 6\%$ for the prediction-based algorithms).

4 Evaluation

Fig. 2. CPU versus GPU performance.

Every experiment is composed of the execution of several combinations of benchmarks. Short names for the benchmark combinations are in the X-axis for some of the following figures (first letter for every benchmark name is used).

Figure 2 shows the performance obtained when running the combinations of benchmarks on just the CPU or the GPU. The GPU performs much better in around half of the cases and slightly better in the rest, with just one case being the other way around. Considering most of the benchmarks fit quite well the data-level parallelism paradigm, it is not surprising to see those results. Although it would be interesting to study the effect of a more balanced set of benchmarks, where some of the benchmarks performed better in the CPU than the GPU, this situation is still very interesting for the study. As some of the benchmarks perform much better on a specific PE, the scheduler must be very careful about its decisions in order not to damage performance. It will be shown that by using performance prediction algorithms the risk of scheduling a task on a very poor performing PE can be greatly reduced.

For the rest of the performance graphs no bar for the CPU is plotted since its performance is consistently worse than the one for the GPU. Therefore, the baseline for speedup graphs is the performance of the GPU alone executing the benchmarks serially (Figure 2). A trivial algorithm like a plain first-free one could

also be used as the baseline as well. However, current systems do not allow the user to schedule tasks either on the CPU or the GPU, and thus it would not reflect the real benefit against current ways of CPU/GPU-systems utilization.

In general the benchmarks run using the heterogeneous scheduler achieve a considerable speedup compared to running them on a single PE. As can be seen in Figures 3 and 4, the average speedup obtained compared to the baseline is between 30% and 40% (individual speedups for each combination are averaged using the harmonic mean). This is obviously a noticeable speedup which confirms to be worth to consider all the PEs in the system as resources where code can be scheduled for execution.

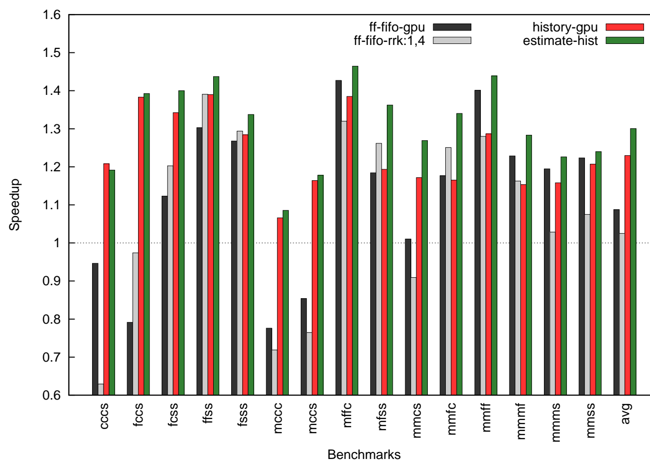

Fig. 3. Performance speedup for $N = 4$.

4.1 First-free Algorithms

Figures 3 and 4 show the relative speedup compared to the GPU for two first-free algorithm variants (ff-fifo-gpu and ff-fifo-rrk:1,4). They work as described in algorithms 1 and 2 in section 2. In case both PEs are busy, the first algorithm chooses the GPU, while the second schedules four tasks to the GPU for each one scheduled to the CPU in a round-robin way.

Despite being considerably simple, these algorithms perform well enough for some cases, reaching up to a 60% speedup over the baseline for a specific case (fffffc in Figure 4). However, they are quite sensible to heavily-biased tasks. If a task performs much better on one PE than in the other ones, scheduling it on the wrong PE will considerably degrade overall system performance. This can be seen for instance in Figure 4 for the benchmark combination mmcccs. That combination contains three times the benchmark cp, which is strongly biased towards the GPU.

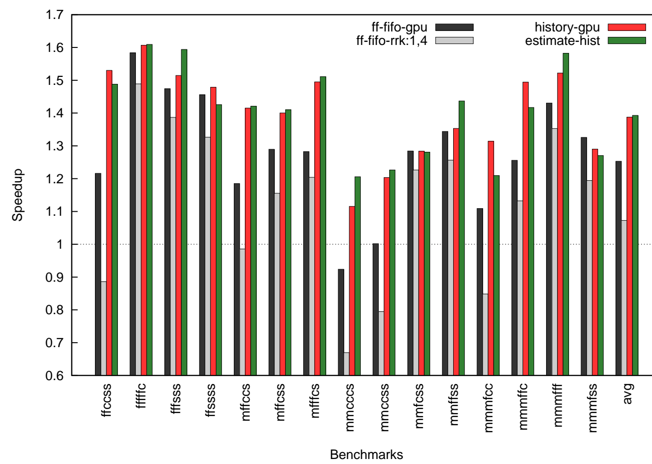

Fig. 4. Performance speedup for $N = 6$.

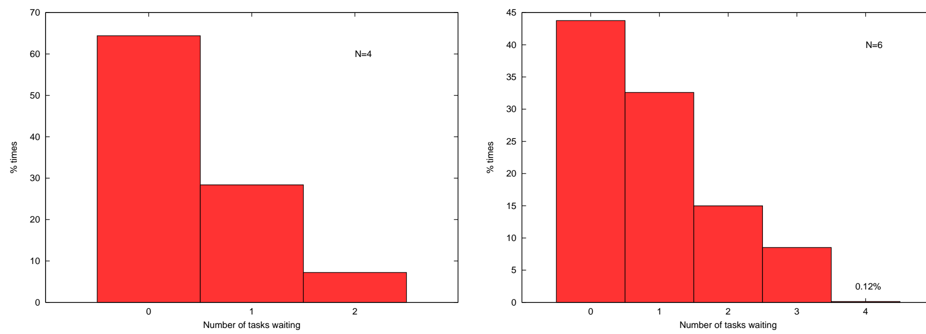

Fig. 5. Task queue usage histogram.

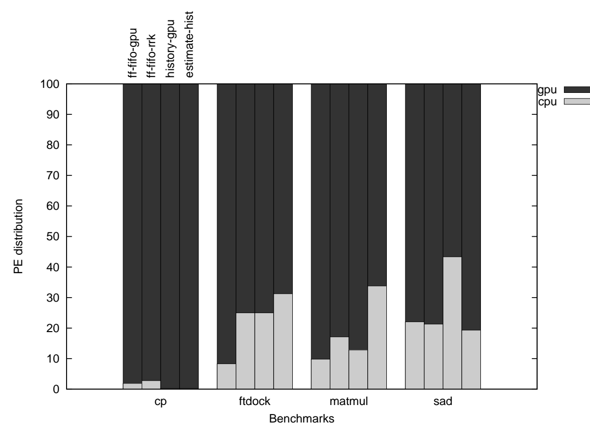

Fig. 6. PE distribution for different benchmarks and scheduling algorithms.

As first-free algorithms blindly select a PE without taking into account the characteristics of the PEs and the task to be scheduled, they eventually schedule cp to the CPU, resulting in a dramatic loss of performance. This behavior can be observed in Figure 6, where the distribution of PEs is shown for every benchmark and scheduling algorithm. The figure shows how few executions of cp are placed on the CPU. Even such a small percentage can greatly reduce performance, and if cp is not scheduled to the CPU more times is because by the time cp executions on the CPU finish, the rest of the benchmarks have already finished, leaving the GPU free and allowing cp to be scheduled on the GPU the rest of the time.

Nonetheless, despite being ill-suited for workloads with highly-biased tasks, these algorithms perform quite acceptable in average, reaching around 10% speedup for $N = 4$ and 20% for $N = 6$. Other FF variants have been evaluated but they are not listed here since their performance was typically worse.

4.2 Predictive Algorithms

The algorithms studied here are two: history-gpu and estimate-hist. Both are based on algorithm 3 described in section 2. They keep the performance history for every pair (task, PE) in such a way that it is not allowed to schedule a task on a PE if the performance ratio to all the other PEs is worse than a threshold θ. Experimentally, the value $\theta = 5$ has been selected.

Both algorithms determine the set of PEs which can be used to schedule a task, $allowed_{PE}$. After that, history-gpu looks for the first idle PE in that set and if there is not such a PE, it selects the GPU as the target. A more fair version which schedules a task randomly to any of the PEs in $allowed_{PE}$ has also been evaluated, but the performance was not as good and results are not shown here.

The behavior of estimate-hist after $allowed_{PE}$ has been computed is quite different. It basically estimates the waiting time in each queue and schedules the task to the queue with the smaller waiting time (in case both queues are empty it chooses the GPU). For that purpose the scheduler uses the performance history for every pair (task, PE) to predict how long is going to take until all the tasks in a queue complete their execution.

Both algorithms achieve significant speedups compared to the baseline. For $N = 4$ estimate-hist has around a 30% speedup and history-gpu obtains a 20% speedup. Those speedups become bigger when $N = 6$, reaching almost a 40% in both cases. As can be seen in the Figures 3 and 4 both algorithms perform consistently well across all benchmark combinations compared to first-free algorithms. The main reason for that is the proper scheduling of cp to the GPU. However there seem to be other factors as well, as for some combinations of benchmarks where cp is not appearing (for instance, mmmfff in Figure 4) these algorithms, and especially estimate-hist, perform noticeably better than first-free based ones. The reason for that is that estimate-hist manages to better balance CPU and GPU task queues. This observation can be seen in Figure 6 where both predicting algorithms tend to schedule a higher percentage of the total number of tasks executed on the system to the CPU. Obviously, this is done without falling into scheduling a strongly-biased benchmark, as cp, to the CPU.

One interesting thing to note is the relatively poor results of estimate-hist for $N = 6$, not being able to improve history-gpu performance as much as in the $N = 4$ case. Due to the non-deterministic nature of scheduling there may not be just a single explanation for this effect. However, it has been observed how the prediction accuracy decreases by more than a 10% when the number of concurrent tasks increases from $N = 4$ to $N = 6$. estimate-hist, having two levels of predictions, is more affected by this loss of accuracy than history-gpu. In order to improve the analysis this effect it would be interesting to conduct new tests on, for instance, quad-core machines where one or two cores can be freed from executing tasks, thus reducing possible interferences.

4.3 Effect of the Number of Tasks on the Scheduler

The number of simultaneous benchmarks run in an experiment is denoted by the value of N. As N increases, the number of tasks which compete for execution raises as well. This effect can be seen in Figure 5, which shows the number of waiting tasks that are in the queue every time a new task is being scheduled on a PE. Left graph is for the case where there are four benchmarks running at the same time, whereas right one depicts the case for six. The number of tasks waiting at the queues substantially increases from one case to the other because more processes are simultaneously using the scheduler.

If the number of tasks to be scheduled increases means it is possible to get closer to fully use all the PEs in the system. Thus, the number of times that the PEs are idle is reduced. Theoretically this must improve the throughput, as can be seen in Figures 3 and 4. When running four benchmarks a speedup of around 30% is achieved, whereas for the case of six the speedup is around 40%.

However, increasing the number of tasks increases the pressure on all the PEs. While on the GPU only tasks are being run, on the CPU parts of the benchmarks not corresponding to tasks and other processes such as Linux system processes are run at the same time as well. Increasing the number of tasks running on the CPU leaves less processing power for non-task parts and in some cases the overall performance may degrade.

In the future, and especially considering new processors with 4 and 8 cores, it may be worth reserving some cores for non-task computations, in order to decrease the interferences between task execution and other processes.

5 Related Work

Job and resource scheduling is a vastly explored topic. However, it seems just a few studies target scheduling for heterogeneous architectures. It is true that heterogeneity has been present in many scheduling studies, but it was mainly from the perspective of distributed and grid systems. The area of scheduling for heterogeneous architectures (i.e., within a single machine) is relatively new and has not been studied in detail yet. Some of the few papers on this topic [12, 19] also agree on this lack of studies.

Scheduling for heterogeneous distributed systems is somehow similar to the problem being dealt here. PEs across different machines or within a single machine are heterogeneous and thus, they present different performance characteristics. However, in the case of distributed systems, the scheduling is burdened with many more complexities such as interprocessor communication (typically done across some kind of external network), distributed management and variant amount of computing resources (some resources may disappear suddenly, whereas others can turn up). [18, 3]

Most of the studies related to heterogeneous distributed systems propose new scheduling algorithms in order to improve the performance through job and resource allocation. They mathematically represent a program as a graph where every node is a task which can be mapped on a PE. Tasks which depend on each other are connected in the graph with the edge weight meaning the communication cost. The list of nodes is generated at compile time and the cost of running tasks on every PE is known in advance. Some relevant works on this area are [20, 11, 14, 15]. However, communication cost in heterogeneous multicore systems is several orders of magnitude smaller than in distributed systems.

A few papers [12, 19] do study the effect of scheduling for heterogeneous architectures. Similarly to the schedulers for distributed systems, programs are also represented as graphs with nodes corresponding to tasks in the program using information known a priori. In order to conduct performance measurements they mainly use random graphs as the input for the scheduler. This is one of the main differences compared to this work, where all the measurements are conducted on real hardware with a real software implementation and real applications.

In [4] a runtime scheduling system for the IBM Cell processor which eases application partitioning among different PEs is presented. However, in that work the heterogeneous architecture is viewed in the common way where the coprocessors are responsible for executing the computationally expensive parts while the main PE is just used to control the coprocessors.

As far as the authors know, this is the first work which deals with scheduling for a heterogeneous architecture using a real implementation and considering the system as a pool of PEs, being able to schedule tasks to any PE.

Initially a set of "classical" scheduling algorithms have been used. FF (first-free), FCFS (first come, first served), SJF (shortest job first) and RR (round-robin) are very well-known algorithms. Because of its simplicity and its relatively good performance, several variations of the FF algorithm have been tested in this study. It is difficult to track down the origin of those algorithms, so the reader is referred to Tanenbaum's work [17] for their description.

6 Conclusions and Future Work

This work shows how using a predictive scheduler for a CPU/GPU-like heterogeneous architectures can improve overall system performance. By adaptively scheduling versioned functions at run-time we can obtain speedups as high as 40% on average compared to perform the computation serially on the GPU.

Some specific applications achieve a large speedup when executed on a data-processing architectures such as a GPU. For these applications, with speedups over $100X$, it may not be worth to execute computationally expensive parts of them on the CPU. However, as this study demonstrates, there are other applications which can greatly increase performance by using a system like the one presented here.

Different kinds of scheduling algorithms have been tried. *first-free*-based ones perform noticeably well for some cases; however they fail to do so for biased computations where one PE performs much better than the others. Performance predicting algorithms, being able to avoid these cases and better balancing the system load, perform consistently better. We intend to study new algorithms in order to further improve overall system performance. Additionally, other benchmarks with different characteristics will be also tried. We expect that with a more realistic set of benchmarks (not only GPU-biased) the benefits of our system would be increased.

The results show how the tasks can receive interferences from other computation occurring in the system. Exploring how the number of cores present in the CPU affect that interference is an interesting future work. Additionally, a way to couple (or even merge) the scheduler presented here with the OS scheduler can greatly help to increase performance.

Finally, we plan to consider different program inputs and analyze their influence on predictive scheduling and run-time adaptation. We plan to use and extend techniques such as clustering [10], code versioning and program phase runtime adaptation [6, 7] to improve the utilization and adaptation of all available resources in the future heterogeneous computing systems.

Acknowledgements

This work has been supported by the European Commission in the context of the SARC Integrated Project (EU contract 27648-FP6), the HiPEAC European Network of Excellence and the Ministry of Science and Technology of Spain and the European Union (FEDER) under contract TIN2007-60625.

References

1. Parboil benchmark suite. `http://www.crhc.uiuc.edu/impact/parboil.php`.
2. CUDA Programming Guide 1.1. NVIDIA's website, 2007.
3. Rosa M. Badia, Jesús Labarta, Raúl Sirvent, Josep M. Pérez, José M. Cela, and Rogeli Grima. Programming grid applications with grid superscalar. *J. Grid Comput.*, 1(2):151–170, 2003.
4. Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006. ACM.
5. Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

6. Grigori Fursin, Albert Cohen, Michael O'Boyle, and Oliver Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, number 3793 in LNCS, pages 29–46. Springer Verlag, November 2005.
7. Grigori Fursin, Cupertino Miranda, Sebastian Pop, Albert Cohen, and Olivier Temam. Practical run-time adaptation with procedure cloning to enable continuous collective compilation. In *Proceedings of the GCC Developers' Summit*, July 2007.
8. Henry A. Gabb, Richard M. Jackson, and Michael J. Sternberg. Modelling protein docking using shape complementarity, electrostatics and biochemical information. *Journal of Molecular Biology*, 272(1):106–120, September 1997.
9. Isaac Gelado, John H. Kelm, Shane Ryoo, Steven S. Lumetta, Nacho Navarro, and Wen mei W. Hwu. Cuba: an architecture for efficient cpu/co-processor data communication. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 299–308, New York, NY, USA, 2008. ACM.
10. David J. C. Mackay. *Information Theory, Inference & Learning Algorithms.* Cambridge University Press, June 2002.
11. Muthucumaru Maheswaran and Howard Jay Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *HCW '98: Proceedings of the Seventh Heterogeneous Computing Workshop*, page 57, Washington, DC, USA, 1998. IEEE Computer Society.
12. Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In *Euro-Par, Vol. II*, pages 573–577, 1996.
13. Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.
14. G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):175–187, 1993.
15. H.S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *Software Engineering, IEEE Transactions on*, SE-3(1):85–93, Jan. 1977.
16. John Stratton, Sam Stone, and Wen mei Hwu. Mcuda: An efficient implementation of cuda kernels on multi-cores. Technical Report IMPACT-08-01, University of Illinois at Urbana-Champaign, March 2008.
17. Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
18. Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.
19. H. Topcuoglu, S. Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, pages 3–14, 1999.
20. H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, Mar 2002.
21. R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.