Exploring Heterogeneous Scheduling Using the Task-Centric Programming Model

Artur Podobas, Mats Brorsson, and Vladimir Vlassov

Royal Institute of Technology, KTH
{podobas,matsbror,vladv}@kth.se

Abstract. Computer architecture technology is moving towards more heterogeneous solutions, which will contain a number of processing units with different capabilities that may increase the performance of the system as a whole. However, with increased performance comes increased complexity; complexity that is now barely handled in homogeneous multiprocessing systems. The present study tries to solve a small piece of the heterogeneous puzzle; how can we exploit all system resources in a performance-effective and user-friendly way? Our proposed solution includes a run-time system capable of using a variety of different heterogeneous components while providing the user with the already familiar task-centric programming model interface. Furthermore, when dealing with non-uniform workloads, we show that traditional approaches based on centralized or work-stealing queue algorithms do not work well and propose a scheduling algorithm based on trend analysis to distribute work in a performance-effective way across resources.

Keywords: Task-Centric run-time systems, Heterogeneous computing, TilePRO64 tasks, GPU tasks, OmpSs.

1 Introduction

As technology advances computers are becoming more and more difficult to program; especially when aiming to utilize all the chips' features. The current direction concerning processor architecture is to accumulate as many processor cores as possibly on the chip to get so-called multi- or many-core processors. Primarily this direction is driven by the power and thermal limitations of the technology scaling; more transistors on a smaller area create larger power densities putting greater stress on the cooling mechanisms. Furthermore, multi- and many-core processors are slowly evolving to be more heterogeneous in nature. And hardware vendors are already starting to tape-out heterogeneous devices such as the AMD Fusion and ARM bigLITTLE. We cannot say much about the future but it is likely that the current trend of increasing heterogeneity will continue; and software developers are already now struggling keeping the pace with homogeneous multi- and many-core technology. How do we write parallel software that targets several heterogeneous systems so that it is portable, scalable and user-friendly? One solution is to provide the software programmer with a set of libraries that maintains and handles the parallelism that the programmer exposes. The library should provide functions for exposing and synchronizing parallel work. A programming paradigm that

I. Caragiannis et al. (Eds.): Euro-Par 2012 Workshops, LNCS 7640, pp. 133–144, 2013.

has the potential of supporting these features is called the *task-centric* programming paradigm. The task-centric paradigm abstracts the user away from managing threads. Instead, the programmer focuses on exposing parallel work in the form of tasks. Tasks can be of any granularity, ranging from an entire application to a simple calculation. To clarify this further, a task is a sequential part of the application that can be executed concurrently with other tasks. Tasks are dynamic in nature; a task can create several new tasks to further decompose the work and increase the parallelism. There are many benefits of choosing a task-centric over a thread-centric framework, such as improved load-balancing (resource-utilization) , portability and user-friendliness. Notable task-centric libraries include: Cilk+ (based on Cilk-5 [7]) focusing on determinism, Nanos++ on user friendliness and versatility, Threading-Build Block on Object-Oriented-Programming, Wool [6] on fine-grained parallelism, StarPU [1] on GPUs. In the present paper, our primary contributions are:

- A task-centric run-time system capable of handling several types of heterogeneous processing units with distributed memory spaces, clock frequency, core count and ISA.
- A scheduling algorithm that is based around linear regression with user-provided feedback concerning tasks' properties. The regression technique is very dynamic; it is continuously calculated and used on-line while the application is running.

The rest of the paper is organized in the following way: Section 2 describes some related and similar work. Section 3 gives an introduction to the task-centric programming style and section 4 shows how our run-time system deals with heterogeneity. Sections 5 and 6 give information about the scheduling policies we have developed as well as information concerning the benchmark and the system-under-test. We finish with sections 7 and 8 that shows the experimental results and the conclusion.

2 Related Work

A lot of work have been done to enable heterogeneous multiprocessing. Labarta et al. [3,11,4] introduced Cell/StarSs, a task-centric programming model and run-time system the enables heterogeneous multiprocessing using Cell's seven processors as well as GPUs. Duran et al [5] merged the functionality of StarSs with an OpenMP base to create the OmpSs programming model which contains support for both OpenMP- and StarSs-like syntax. As with StarSs, the OmpSs programming model and its run-time system Nanos++ both support GPUs. Both StarSs and OmpSs are a product of Barcelona Supercomputing Center (BSC). Augonnet et al. introduces the StarPU [1] run-time system that focuses on exploiting multi-heterogeneous systems, primarily GPUs. The programming model concepts are similar to StarSs/OmpSs in that they convey information concerning the tasks memory usage to the run-time system. Augonnet et al.[2] evaluated StarPU using a series of different scheduler with the most popular one being HEFT (Heterogeneous Earliest First) which, similar to our work tries to predict the execution time of tasks. O'Brien et al. [9] evaluated possible support to run OpenMP-annotated programs using the IBM Cell processor. They used similar mechanism as the ones described in this paper, including double buffering and software caching to hide latencies.

Analysis of performance was done using Paraver [10] and speedup figures of up to 16% could be obtained compare to the hand-written benchmark or up to 35x compared to the sequential version when using 8 SPEs. Liu et al. [8] introduces OpenMP extension for heterogeneous SoC architectures. They extended the OpenMP clauses to support their 3SOC architecture thus hiding the complexity of the architecture from the programmer. They evaluated their approach with Matrix Multiplication, SparseLU and FFT and showed linear speedup compared to the sequential version and up-to 30x speedup when using the 3SoC's DSE (Digital Signal Engine). At first glance, the present paper resembles the ideas presented in StarPU [1]. Although there are some small difference, such as the present paper concerns many types of heterogeneous processors perhaps the most notable difference is in the performance model. **In StarPU, the performance model assumes that the execution time is independent from the content of the data** that a task will use. This is not true in a lot of application where the task's data size does not correlate well with the task's complexity (or execution time). Our model uses the information from the user (whatever that may be) to forecast task's with complexity not-seen yet.

3 The Task-Centric Programming Model

The Task-Centric programming model allows a programmer to exploit parallelism within a sequential application by annotating parts of the application source code. Tasks within this programming model should not be confused with other notions of tasks, such as OS tasks or a-prio known tasks or task-graphs. Tasks (and the work within tasks) in the task-centric programming model are dynamic, and they can themselves spawn additional tasks. There is no direct limitation to the complexity of a task; they can range from ten instructions to several millions of instructions. The important property is that tasks can be executed in parallel. Most current task-centric run-time system requires the user to insert task synchronization points in their program. However, there is a way to relax this constraint and let the run-time system itself insert dependencies between tasks; this is done by requiring the programmer to give information about what and how the task will access the memory. Examples of existing run-time systems that supports this implicit way of managing dependencies include the *Ss family (e.g. CellSs[4] and OmpSs[5]). Our run-time system, which is described in section 4 also supports it.

```
1 #pragma omp task input(A) output(D)
          do_work(A,D);
3 #pragma omp task input(D) output(E)
          do_work(D,E);
5 #pragma omp task input(A) output(B)
          do_work(A,B);
```

Fig. 1. OmpSs extended clauses to support automatic dependency insertion transparent to the programmer

Figure 1 shows an OmpSs enabled code which does some work on an array. Tasks are annotated using compiler directive (#pragma omp for C) which follows the OpenMP standard. Functions calls or compound statement annotated with the task directive will become task's able to run in parallel. The work is distributed across three tasks that will consume (input) and produce (output) different memory regions. Unlike traditional task-centric models (Cilk,OpenMP,...), an OmpSs supporting run-time system will execute the third task before the second task since the there is no memory dependency between the first (or second) and the third allowing more parallelism to be exposed. This differ from the classical OpenMP approach where a **#pragma omp taskwait** statement would have to be inserted before the last spawned task to ensure consistency.

4 Integration of Heterogeneity in Task-Centric Programming Models

We created a run-time system called UnMP that is capable of supporting the OmpSs programming model. Memory regions specified to be used as input/output/inout are handled by the run-time system which ensures sequential correctness of the parallelized code by preserving data-dependencies between parallel tasks. The run-time system supports i386/x86-64 as a host-device, and GPU(s) or TilePRO64(s) devices as slaves.

Heterogeneity Support

Most of the heterogeneity support within the run-time system is contain within the internal threading system. At the lowest abstraction layers, we are using POSIX-threads (we call them PHY*ysical*-threads) to control each of the processing units available at the host system. However, since there is no native or OS control over the external heterogeneous devices, we decided to add another abstract layer called LOG*ical*-threads. A LOG-thread is a thread that represents a processor from the run-time systems point of view. For each type of LOG-thread, there is set of API functions that control the selected type. For example, a LOG-thread that represents a GPU will have API functions that starts, finished and prepares a task for the GPU. Using this methodology, we can essentially hide the complexity of using the different units since they look the same from a scheduling point of view. Furthermore, we have support for mapping a LOG-thread to several PHY-threads. This means that we can literally have two different host processors (PHY-threads) controlling the same GPU; should a PHY-thread be busy performing work, another PHY-thread can start-up any work on the GPU improving the resource utilization of the system. The difference between previous work, such as OmpSs[5] and StarPU [1] is also that we focus on adding a third heterogeneous system: TilePRO64. At first glance, it may seem that we use the TilePRO64 as a GPU accelerator. They do bear a resemblance, however, while using a GPU allows access to a primitive bare-metal interface, where the performance relies on the programmer's skill to produce well-behaved code, our implementation open up several **user-transparent optimizations** on the TilePRO64. For example, branch-divergence does not have as big negative impact on performance on the TilePRO64 as it would have on the GPU due to how our TilePRO64 scheduler works. Furthermore, this opens up for a treasure of future work

where we can adopt several locality or energy related techniques on the TilePRO64 side; something that cannot be done on the GPU due to its bare-metal interface.

Amortizing Transfer Costs

One bottleneck of using current heterogeneous devices is that they usually sit on the PCI-express bus which, compared to RAM, has a relatively low bandwidth.To amortize (hide) transfer costs, the UnMP run-time system uses techniques that either reduces or removes overheads related to memory transfers. **Double buffering** enables the run-time system to transfer data associated with the next task to execute while the current task is running. This enables the run-time system to hide the latency associated with upcoming tasks. Another method is to emply a **software cache** that monitors where valid copies exist in our distributed heterogeneous system, as well as properly invalidating the copies when they are overwritten, we can exploit temporal locality found in the application and reduce the memory transfer overhead. The software cache behaves as a SI-cache, with each memory region being either Shared or Invalid, similar to [3,11,4]. Other variations of the protocol includes the MSI protocol employed by StarPU[1].

Writing Heterogeneous Code

The application developer is responsible for creating the different versions of each of the task if heterogeneity support is to be used as well as specifying the task's data usage. Spawning a task with dependencies and with heterogeneous support is conceptually shown in figure 2 using the *device* and *implements* proposed by [5]. The *device* clause specifies the different architectures that the task have been implemented for and *implements* specifies which function this implements. In this example, the task has 3 versions: a host version, a TilePRO64 version and a GPU version. Creating task for the TilePRO64 processor is conceptually very similar to using OpenMP **#pragma omp parallel** directives, enabling SPMD execution. A GPU task should invoke the kernel that implements that particular function. For both the TilePRO64 and GPU case, the memories are maintained and allocated by the run-time system, relieving the programmer from managing them. Figure 2 also show the code-transformation of when a task is spawned. The (#pragma omp task) statement is transformed into a series of library function calls that create a task, sets the memory dependencies and arguments and finally submits the task to be scheduled onto the system resources. We did also include functionality for hinting the run-time system about a task's *complexity*. The *complexity* is a single-value number that can be anything as long as it reflects the parameter that has a profound impact on the task's execution time. Examples of such parameters would be the block-size in matrix multiplication or the amount of rays to cast in a ray tracing program. The complexity clause is used as shown below:

```
#pragma omp task complexity(N)
        matmul_block(A,B,N);
```

The *complexity*, although relatively primitive in its current state have a lot of potential. A smart compiler could theoretically derive the *complexity* itself by analyzing the in-

```
GPU)
#pragma omp task device(gpu) implements(inc_arr)
void cuda_inc_arr(int *A, int *B)
{
        cuda_inc_array_kernel <<<4,256>>>(A,B);
}
```

```
TilePRO64)
#pragma omp task device(tilera) implements(inc_arr)
void tilera_inc_arr(int *A,int *B)
{
#pragma omp parallel
        {
                int i = omp_get_thread_num();
                B[i] += A[i];
        }
}
```

```
Task Spawn)
#pragma omp task input(A) output(B) target(tilera,gpu,host)
        inc_arr(&A[0], &B[0]);
```

```
Code-Transformation)

_unmp_task *tsk = _unmp_create_task();
_tsk_arg *arg = (_tsk_arg *) malloc(...);
arg->A = &A[0];
arg->B = &B[0];
_unmp_fan_input (tsk , &A[0] , ...);
_unmp_fan_output (tsk , &B[0] , ...);
_unmp_set_task_host(tsk , &x86_inc_arr , arg);
_unmp_set_task_gpu(tsk , &cuda_inc_arr , arg);
_unmp_set_task_tilera(tsk , &tilera_inc_arr , arg);
_unmp_submit_task(tsk);
```

Fig. 2. Three different tasks

termediate representation of the task's them self. We consider it future work to improve upon this metric.

5 Heterogeneous Task Scheduling

We developed and implemented the **FCRM (ForeCast-RegressionModel)** scheduler, which is a new scheduling algorithm presented in this paper. We also implemented

three well-known scheduling algorithms in our run-time system to evaluate and compare our FCRM scheduler against: Random, Work-Steal, Weighted-Random. All the four scheduling policies have per-core private task-queues and the major main difference is in the work-dealing strategy.

Random. The *Random* scheduling policy will send task at random to different processing units. It is an architecture- and workload oblivious *work-dealing* algorithm which we use as a baseline due to its familiarity and simplicity.

Work-Steal. The *Work-Steal* is a greedy policy that creates and puts work into the creator's private queue. Load-balancing is achieved by randomly selecting victims to steal work from when the own task-queue is empty. Work-Steal scheduling is a popular policy used in run-time systems such as Wool, Cilk and TBB as well as several OpenMP implementations.

Weighted-Random. The *Weighted-Random* is a work-dealing that distributes work according to each processors weight. The weights were estimated offline using test-runs of the benchmarks on different processing units running in isolation.

FCRM. The FCRM scheduler uses segmented regression to estimate trends concerning the execution time of task on different heterogeneous processors. The idea is to estimate the time a certain task takes on all available processing units and use that information to derive weights according to the estimation. More specifically, we adopt linear regression to fit our data-points (observed task execution time). Should the linear regression fail to fit the entire data-point set to a single linear function, it will segment the data-points so that several function cover the entire fit:

$$f_T(U) = \begin{cases} a_1 + b_1 * U & \text{when } U < BP_1 \\ a_2 + b_2 * U & \text{when } U > BP_1 \text{ and } U < BP_2 \\ \ldots & \\ a_n + b_n * U & \text{when } U > BP_{n-1} \end{cases}$$

Where BP_n are the calculated break-points for the n segments and U is a task's complexity. For each task, and every processor, we calculate the regressed segmented function. We denote it: $f_{P_T}(U)$ where P is the processor, and T is the task and U is the complexity. To estimate the execution of a task whose complexity has not yet been seen, we have to take where the data is into account:

$$t_{P_{T_U}} = f_{P_T}(U) + g_P(T)$$

where:

$$g_P(T) = BW_{P_{to}} * T_{data-use} + BW_{P_{from}} * T_{data-produce}$$

$BW_{P_{to}}$ = Measured bandwidth to device for processor P

$BW_{P_{from}}$ = Measured bandwidth from device for processor P

$T_{data-use}$ = Data **needed** by task T, taking the software cache into account.

$T_{data-produce}$ = Data **produced** by task T

The FCRM scheduler extends the Weighted-Random scheduling policy by calculating the weight on-line, adapting the any anomalities that were found by the regression. It does also takes the information in the software cache into account when deriving the weights. The entire forecast is recalculated on a miss-prediction (large deviation between predicted and observed time) known after a task have been executed on a particular device; this dynamic re-calculation also allows for adaptation when a certain processing units becomes overloading due to external interference, such as other thermal properties(performance throttling) and other processes sharing that also uses the processor.

6 Experimental Setup

System Specification

We performed the experimental evaluation using a single socket, Quad-Core Intel processor. The processor is connected to a TilePRO64, which is a 64-core chip multiprocessor targeting the embedded market. The system do also contain nVidia Quadro FX 570 CUDA enabled GPU. Both Quadro FX and TilePRO64 sit on the PCI slots.

Benchmark Selection

We selected benchmarks that are well-known and very parallel. This was done intentionally to show that even if the benchmarks themselves are very parallel, when homogeneity stop being a property, even these benchmarks will fail to scale well using well-known scheduling strategies; especially if the tasks are non-uniform concerning the amount of work they perform. In their original form, for each benchmark, the work is divided into segments that can be executed in parallel.

- **Multiple-List-Sort** - *Synthetic* benchmark that simulates incoming packets which contains several lists that needs to be sorted. In the present examples, we assume that the packets are already there, but must be processed one at the time. Each task contains a number of lists, and the lists themselves are of various sizes. The lists are sorted using the Odd-Even sorting mechanism for all architectures. The Odd-Even sort is of $O(n^2)$ complexity and while we understand that the algorithm is not the optimal one for the individual platforms, it is meant to show the benefits and efficiency of various schedulers on this type of application.
- **Matrix Multiplication** - Multiplies two input matrixes A and B to produce matrix C. Matrix multiplication is best suited for GPUs as a long as the matrix block size is enough to keep as many of the GPUs threads busy.
- **n-Body** - Simulation of celestial bodies based on classical Newtonian physics. Our implementation is based on the detailed model which *O(n^2)* in complexity (not the Barnes-Hut algorithm). Tasks are generated to work on subsets of all the celestial bodies; these subsets are of varying length to pronounce the different complexities of tasks.

7 Experimental Results

7.1 Experimental Results

We evaluated our run-time system together with difference scheduling approach under a configuration consisting of three x86-64 processors, one CUDA-enabled GPU and one TilePRO64. For the speedup comparison, the execution time for the schedulers running a certain benchmark were normalized to the serial execution time running on one x86-64 processor. We also included the processor with the best individual performance as a reference point. For each scheduler, and for different parameters, we executed the benchmark ten times taking the median to represent the performance. For the Weighted-Random scheduler, the weights were calculated according to the different processors execution time when running in isolation for one particular set of parameters. For the FCRM scheduler we evaluated both when the scheduler is used without any pre-loaded trend estimation function and when a trend estimation function (FCRM-PRELOAD) have been established from a previous run. Figure 3 shows the performance when executing the MLS (Multiple-List-Sort). We evaluated two cases where the total amount of sorted elements varied. For both the cases, we notice that the FCRM-PRELOAD scheduler that uses a trend-regression function from a previous runs performs much better than the other schedulers. The Weighted-Random scheduler performs slightly worse than the Work-Steal in this case. The reason Weight-Random scheduler and the Random scheduler do not perform well is due to the pushing of work to the GPU, which is the slow processor in benchmark.

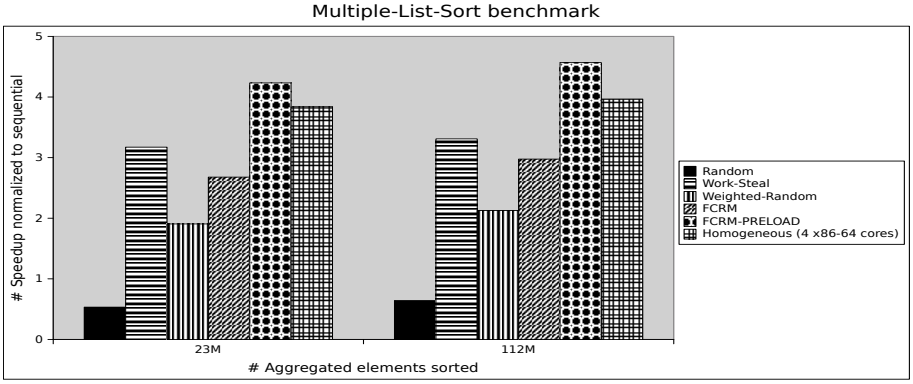

Fig. 3. Multiple-List-Sort benchmark performance and speedup relative sequential execution time. Four x86-64 cores showed as reference point.

Figure 4 shows the performance of the schedulers when running a blocked matrix multiplication. There is only a slight difference between Weighted-Random scheduling policy and the FCRM scheduler that is using an already established estimation function and the different is primarily in that the FCRM also takes the locality of the tasks into account when deciding upon the weight; something that the Weighted-Random scheduling policy does not. Also notice that even for a uniform benchmark such as matrix multiplication neither the Random nor Work-Steal perform any good. The reason

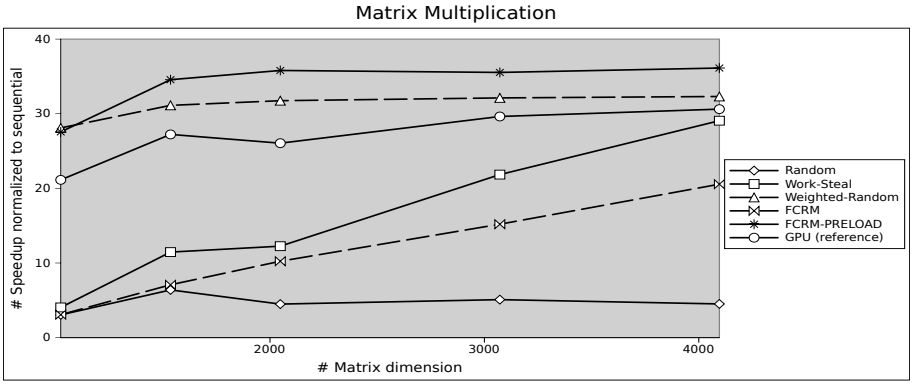

Fig. 4. Matrix multiplication speedup amongst schedulers with the GPU as a reference point

for this due to the work being stolen by the TilePRO64 thread as soon as it is idle, and the TilePRO64 is the slowest processor to execute the matrix multiplication code in this benchmark.

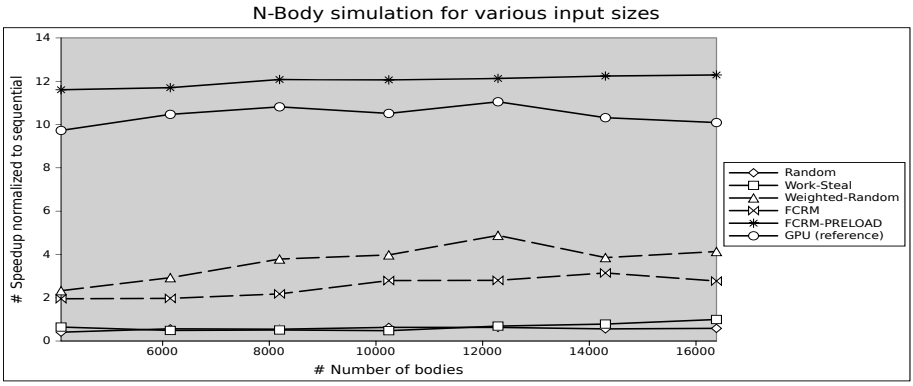

Fig. 5. N-body speed-up for 30 time-step iterations with GPU as a reference point

Figure 5 shows the performance of the schedulers when running the simulation of N-Body. Our implementation of N-Body is highly non-uniform, making the weights calculated for the Weight-Random scheduler near useless as it does not take the complexity of tasks into account. For example, a task that calculates the gravitational force of 10 bodies have equally high chance of being scheduled on the TilePRO64 as a task that computes 1000 bodies; even though the latter should instead be placed onto the GPU to reduce the critical path. The FCRM scheduler without an initial estimation function performs slightly worse than Weighted-Random. This is one of the drawbacks on using the FCRM scheduler without an initial estimate function on small applications (with few iterations).

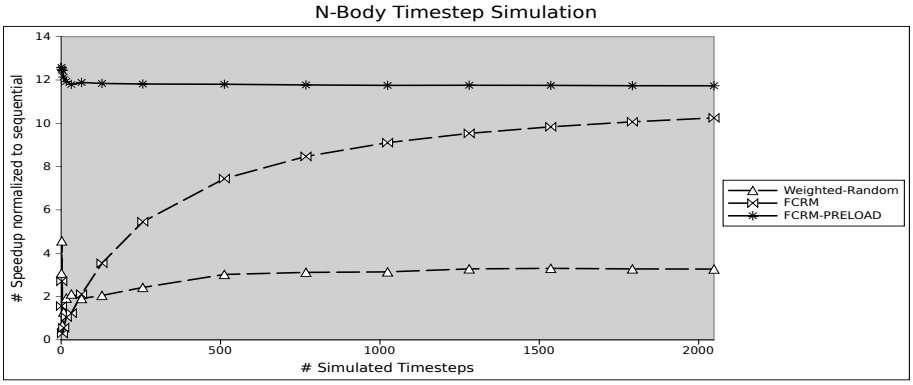

Fig. 6. Impact of application length on FCRM with and without preloaded regression function

However, when we increase the duration (figure 6) of the application, we see that the FCRM scheduler without the initial estimate function slowly converges to the result of the scheduler with an initial estimate function.

8 Conclusions

In the present paper, we showed how to utilize heterogeneity that differs in several aspects: frequency, core count and core type. We presented a task-centric run-time system capable of handling the OmpSs programming model as well as extensions for improving the transfers by utilizing software caching and overlapped memories. We further gave example of a scheduler capable of both adapting-to and improving the execution time of a series of benchmark; benchmarks that are highly dynamic in nature. This dynamic nature makes scheduling a large workload on a bad processor very inefficient, which we have captured by extending the Weighted-Random scheduling approach with trend analysis for weight estimation. We have shown that our scheduler performs significantly better than the original Weighted-Random, and far better than Random Work-Stealing approach which is a well-known and embraced scheduling policy for homogeneous cores and workloads.

Acknowledgements. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (www.encore-project.eu), grant agreement nr. 248647. The authors are members of the HiPEAC European network of Excellence (http://www.hipeac.net). We would like to thank Alejandro Rico and the team at BSC for reference designs that uses the data-driven task-extensions of OmpSs.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.: Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23(2), 187–198 (2011)
3. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 851–862. Springer, Heidelberg (2009)
4. Bellens, P., Perez, J., Badia, R., Labarta, J.: Cellss: a programming model for the cell be architecture. In: SC 2006 Conference, Proceedings of the ACM/IEEE, p. 5. IEEE (2006)
5. Duran, A., Ayguadé, E., Badia, R., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: A proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters 21(2), 173–193 (2011)
6. Faxén, K.: Wool-a work stealing library. ACM SIGARCH Computer Architecture News 36(5), 93–100 (2009)
7. Frigo, M., Leiserson, C., Randall, K.: The implementation of the cilk-5 multithreaded language. ACM Sigplan Notices 33(5), 212–223 (1998)
8. Liu, F., Chaudhary, V.: Extending openmp for heterogeneous chip multiprocessors. In: Proceedings of the 2003 International Conference on Parallel Processing, pp. 161–168. IEEE (2003)
9. O'Brien, K., O'Brien, K., Sura, Z., Chen, T., Zhang, T.: Supporting openmp on cell. International Journal of Parallel Programming 36(3), 289–311 (2008)
10. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualise and analyze parallel code. In: Proceedings of WoTUG-18: Transputer and occam Developments, vol. 44, pp. 17–31 (1995)
11. Planas, J., Badia, R., Ayguadé, E., Labarta, J.: Hierarchical task-based programming with starss. International Journal of High Performance Computing Applications 23(3), 284–299 (2009)