# Heterogeneous Work-stealing across CPU and DSP cores

Vivek Kumar[†], Alina Sbîrlea[†], Ajay Jayaraj[‡], Zoran Budimlić[†], Deepak Majeti[†] and Vivek Sarkar[†]

[†]Rice University, [‡]Texas Instruments

Due to the increasing power constraints and higher and higher performance demands, many vendors have shifted their focus from designing high-performance computer nodes using powerful multicore general-purpose CPUs, to nodes containing a smaller number of general-purpose CPUs aided by a larger number of more power-efficient special purpose processing units, such as GPUs, FPGAs or DSPs. While offering a lower power-to-performance ratio, unfortunately, such heterogeneous systems are notoriously hard to program, forcing the users to resort to lower-level direct programming of the special purpose processors and manually managing data transfer and synchronization between the parts of the program running on general-purpose CPUs and on special-purpose processors.

In this paper, we present HC-K2H, a programming model and runtime system for the Texas Instruments Keystone II Hawking platform, consisting of 4 ARM CPUs and 8 TI DSP processors. This System-on-a-Chip (SoC) offers high floating-point performance with lower power requirements than other processors with comparable performance. We present the design and implementation of a hybrid programming model and work-stealing runtime that allows tasks to be created and executed on both the ARM and DSP, and enables the seamless execution and synchronization of tasks regardless of whether they are running on the ARM or DSP. The design of our programming model and runtime is based on an extension of the Habanero-C programming system. We evaluate our implementation using task-parallel benchmarks on a Hawking board, and demonstrate excellent scaling compared to sequential implementations on a single ARM processor.

***Keywords—Habanero; Keystone-II; load balancing; scheduling; work-stealing***

## I. Introduction

The road to extreme scale computing has created an unprecedented pressure on creating power-efficient building blocks for supercomputers. Due to the power-hungry nature of general-purpose CPUs, traditional approaches of scaling up and scaling out using more and more nodes with more and more CPU cores on each node are simply not feasible any longer. Many vendors have instead shifted their focus to designing high-performance nodes using a smaller number of general-purpose CPUs aided by a larger number of lower-power special purpose processors (SPPs), such as GPUs [1], FPGAs [2] or DSPs [3]. While offering a much lower potential power-to-performance ratio, such heterogeneous systems are, unfortunately, notoriously hard to program. The users are forced to resort to lower-level direct programming of the SPPs and manually managing data transfer and synchronization between the parts of the program running on CPUs and SPPs. The current state of the art requires the programmer to compose parallel kernels for execution on SPPs, transfer the data necessary for the kernel execution, launch a kernel, wait for its completion and transfer the data back to the CPUs to continue the program execution. The CPUs might be able to perform some useful computation while waiting for the kernel to finish on the SPPs. An alternative to manual resource management that some hardware vendors support is a higher-level model that can do the offloading of tasks and data to the SPPs automatically, but that uses either only the CPUs or SPPs, and is only suited for regular data-parallel computations. Examples of this approach include the OpenMP accelerator model for TI's Keystone II architecture [4]. This approach is unfortunately ill-suited for dynamic, irregular programs, forcing the user to perform the bulk of their computation on either CPUs or SPPs, therefore wasting a large part of the available computation resources, lengthening the total time to solution and total energy spent on the solution.

Task-parallel programming models, such as Chapel [5], Cilk [6], Habanero-C [7], [8], Habanero-Java [9], OpenMP 4.0 [10], and X10 [11]–[13] offer a high-level approach to creating parallelism by creating large numbers of lightweight tasks that can execute in parallel on a fixed and smaller number of worker threads that are bound to processors. While such programming models present a much higher-level and more intuitive methods for designing parallel algorithms, especially for irregular and dynamic problems, they also require a powerful and efficient runtime to manage task creation, scheduling and synchronization.

In this paper, we present HC-K2H, a programming model based on the Habanero task-parallel model [9], and a runtime implementation of that model on the Texas Instruments Hawking SoC platform [3], consisting of 4 ARM CPUs and 8 TI's DSP processors. Low-power SoCs with general-purpose processors and on-chip accelerators that share the same address space and physical memory are increasingly being considered as attractive candidates for extreme scale computing. We present the design and implementation of a hybrid work-stealing runtime that allows for creation of tasks that can be executed either on ARM or DSP and enables the seamless execution and synchronization of those tasks regardless of where they run. We evaluate our implementation by executing task-parallel benchmarks on a Hawking board, and show excellent scaling compared to sequential implementations on a single ARM processor.

To appear in IEEE HPEC Conference, September 2015, Waltham, MA USA.

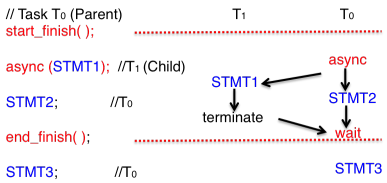



Fig. 1: An example code schema with **async** and **finish**

The next section presents some background on the Habanero task-parallel programming model and the Texas Instruments' Hawking platform. Section III presents the HC-K2H programming model, and Section IV describes the details of our implementation of the HC-K2H runtime, Section V presents an experimental evaluation of HC-K2H runtime, followed by summaries of related work and our conclusions in Section VI and Section VII respectively.

## II. Background

### A. Work-stealing

Work-stealing is a strategy for efficiently distributing work in a parallel system. The runtime maintains a pool of *worker threads*, each of which maintains a local set of *tasks*. When local work runs out, the worker becomes a *thief* and searches for a *victim* thread from which to *steal* work. A steal occurs when a thief takes work from a victim. The runtime provides the thief with the execution context of the stolen work, including the entry point and sufficient program state to initiate the computation. The runtime ensures that work is executed exactly once and that the state of the program reflects the contributions of all workers.

In this paper, we describe our extensions to work-stealing schedulers to execute across ARM and DSP cores, taking into account the limitations of a hybrid CPU+DSP platform.

### B. Habanero-C Library (HClib)

Habanero-C is a C-based task-parallel programming language developed at Rice University. Habanero-C provides a task-parallel programming model based on the **async** and **finish** constructs [9], [11]. In this paper we introduce a library-based implementation of Habanero-C, called HClib. Here we briefly describe the HClib features related to this paper, more details can be found in [7].

Figure 1 shows a sample code written in HClib. The "**async** ⟨*stmt*⟩" clause creates a new task, which will execute ⟨*stmt*⟩ *asynchronously* (i.e., before, after, or in parallel) with the code which follows the **async** block (i.e., the remainder of the parent task). Figure 1 illustrates how the parent task, $T_0$, uses an **async** construct to create a child task $T_1$. Thus, `STMT1` in task $T_1$ and `STMT2` in task $T_0$ can potentially execute in parallel.

**async** is a powerful primitive because it can be used to enable any statement to execute as a parallel task, including statement blocks, for-loop iterations, and function calls.

Finish is a generalized join operation. The statement **start_finish**() starts a finish scope in HClib and the statement **end_finish** terminates this finish scope. Worker starting the finish scope will wait in the **end_finish** scope until all **async** transitively spawned tasks from this scope have completed. Each dynamic instance $T_A$ of an **async** task has a unique *Immediately Enclosing Finish* (IEF) instance $F$ of a finish statement during program execution, where $F$ is the innermost finish containing $T_A$ [14].

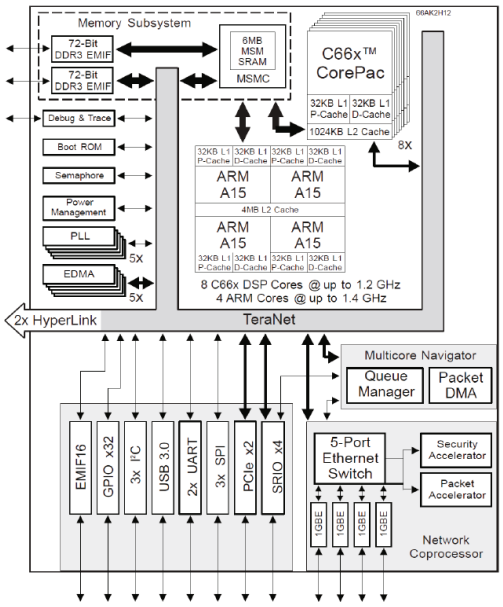



Fig. 2: Keystone-II 66AK2H ARM-DSP SoC

For example, the **start_finish** statement in Figure 1 is used by task $T_0$ to ensure that child task $T_1$ has completed executing `STMT1` before `STMT3` starts executing. If $T_1$ were to create additional transitive tasks, these would become "grandchild" of $T_0$ and $T_0$ will wait for all to complete before executing `STMT3`. The power of these constructs comes from the ability to arbitrarily nest **async** and **finish** constructs.

### C. Keystone II

Texas Instrument's KeyStone platforms provide an innovative platform that integrates RISC and DSP cores with application-specific co-processors and input/output peripherals. This design can achieve excellent performance for embedded DSP applications, but can also be used for general-purpose applications.

The work described in this paper was developed on TI's Keystone II platform (also referred to as Hawking), but the principles should be applicable to other CPU+SPP architectures. The Hawking platform contains 4 ARM Cortex-A15 processors - TI's highest performing ARM processors, each running at up to 1.4 GHz and 8 TI C66x floating-point DSPs each running at up to 9.6GHz, all on a single low-power System-on-chip (SoC). The hardware also provides hardware queues, which can be used to communicate between ARM and DSP cores. There are two queue managers with 8192 queues per queue manager and 64 descriptor memory regions per queue manager.

Figure 2 shows the architecture of the Keystone-II 66AK2H ARM-DSP SoC [3].

The ARM cores have 32KB of L1D (data cache) and L1P cache (program cache) per core and a shared L2 cache of 4MB. The DSP cores also have 32KB of L1D and L1P cache per core, plus a private L2 cache of 1MB cache each. The Multicore Shared Memory Controller (MSMC) provides 6 MB of scratchpad RAM (SRAM) that is on-chip, and shared by all ARM and DSP cores. The additional shared memory between ARM and DSP cores is the off-chip DDR, with 2GB size, managed by two DDR3 controllers. Notably the shared memory controller in Keystone devices neither maintains coherency between two DSP cores nor between ARM and DSP cores. It is the responsibility of the running program to use synchronization mechanisms and cache control operations to maintain coherent views of the memories. The ARM cores are fully cache coherent and have their own MMU (memory management unit), which is not shared with the DSPs.

We propose an approach in which the work-stealing model is extended across ARM and DSP, thus making the DSPs independent and able to request and steal work for themselves. We do this in a high-level programming model (HClib), which abstracts all the above hardware details from the programmer.

## III. Programming Model

In this section, we use a parallel array addition example to introduce the HC-K2H programming model. Parallel programming constructs in HC-K2H are extended from HClib. As we target HC-K2H for hybrid work-stealing across ARM and DSP cores, its parallel programming constructs slightly differs from HClib.

### A. ARM and DSP shared buffers

Figure 3 shows a simple parallel array addition code using HC-K2H programming model. For using the HC-K2H library, a 'C' program has to include the HC-K2H header file (line 2). The variables declared on line 5 are not shared by default between the ARM and DSP. Buffers 'a', 'b' and 'c' can only be shared between the ARM and DSP if they are heap allocated by contiguous memory allocator (CMEM) [15]. CMEM can allocate buffers either on MSMC or on DDR (Section II-C). This is done by using the **ws_malloc** function for memory allocations (lines 10-12). Currently HC-K2H allows CMEM allocations only by the ARM but not by the DSP. The array initialization done at line 14 are visible only on the ARM, unless the ARM explicitly writes back and invalidate its cache. This is done by using the function **ws_cacheWbInv** (lines 16-18). It accepts the CMEM allocated buffer pointer and buffer size as parameters. This CMEM allocated pointer is valid only on the ARM. The conversion of this ARM pointer to DSP pointer is performed by using the function **ws_dspPtr** (line 51). This function accepts an ARM pointer as a function parameter and returns the corresponding DSP pointer as **uint32_t** type.

### B. Asynchronous tasks

HC-K2H provides three different types of asynchronous tasks. They are **async**, **asyncDSP** and **forasync**. The use

```
1  #include <stdio.h>
2  #include "hc-k2h.h"
3
4  #define INT uint32_t
5  INT size, *a *b, *c;
6
7  void arm_init(INT n) {
8    int i;
9    size = n;
10   a=(INT*)ws_malloc(n * sizeof(INT));
11   b=(INT*)ws_malloc(n * sizeof(INT));
12   c=(INT*)ws_malloc(n * sizeof(utin32_t));
13   for(i=0; i<n; i++) {
14     a[i] = 0; b[i] = 100; c[i] = 1;
15   }
16   ws_cacheWbInv(a, n * sizeof(INT));
17   ws_cacheWbInv(b, n * sizeof(INT));
18   ws_cacheWbInv(c, n * sizeof(INT));
19 }
20
21 void dsp_init(void* args) {
22   INT* in = (INT*) args;
23   size = in[0];
24   a = (INT*) in[1];
25   b = (INT*) in[2];
26   c = (INT*) in[3];
27 }
28
29 void cleanup() {
30   ws_free(a); ws_free(b); ws_free(c);
31 }
32
33 void kernel(void* args, int i) {
34   a[i] = b[i] + c[i];
35 }
36
37 void par_sum() {
38   loop_domain_t loop = {0 /*lowbound*/,
39     size /*highBound*/, 1 /*stride*/, 32 /*tile size*/};
40   ws_args_t t1 = {a, size*sizeof(INT)};
41   start_finish(1, &t1);
42   forasync(kernel, NULL, 0, 1, &loop, WS_RECURSION);
43   end_finish();
44 }
45
46 int main(int argc, char ** argv) {
47   INT n = 1048576;
48   // initialization at ARM
49   arm_init(n);
50   // initialization at DSP
51   INT in[]={n, ws_dspPtr(a), ws_dspPtr(b),ws_dspPtr(c)};
52   start_finish(0);
53   asyncDSP(dsp_init, in, sizeof(in));
54   end_finish();
55   // launch parallel sum
56   par_sum();
57   cleanup();
58   return 0;
59 }
```

Fig. 3: Parallel array addition in HC-K2H

of **asyncDSP** and **forasync** is shown in our array addition example. The only difference between **async** and **asyncDSP** is their place of execution. **async** tasks are free to execute either on the ARM or on the DSP, whereas **asyncDSP** can execute only on the DSP. Synchronization on these asynchronous tasks is done by wrapping them within a finish scope. A finish scope is started by using **start_finish** function and closed by using **end_finish** function (Section II-B).

The CMEM allocations by the ARM (lines 10-12) are still not visible to DSP. They are informed about the corresponding DSP equivalent pointers by using **asyncDSP** (line 53). The first parameter is the function pointer, the second parameter

contains arguments to this function, and the third parameter contains the size of the arguments. The **start_finish** is a variable argument function (line 52). Writes performed by the DSP on shared buffers are visible to ARM only after ARM invalidates its cache. HC-K2H runtime can do this only if its informed about the pointer to CMEM buffer (write access types) and corresponding buffer size. **ws_args_t** structure is provided by the runtime to pack the CMEM buffer pointer and buffer size. However, as the **asyncDSP** task at line 53 will not write to any shared variable, **start_finish** at line 52 is passed with zero as parameter. Parallel addition of arrays using **forasync** is launched at line 56 The **forasync** function (line 42) has the first parameter as function pointer to the user function, second parameter is arguments to the user function, third parameter is the size of user function arguments, fourth parameter is the dimension of the **for** loop (1 for single dimension), fifth parameter as pointer to the loop domain containing information on **for** loop parameters (lines 38-39), and the last parameter as scheduling mode (recursive or chunked scheduling). **forasync** will write to CMEM allocated buffer 'a' (line 34), hence **ws_args_t** is used at line 40 to pack this information and is passed as parameter to **start_finish** at line 41. The HClib APIs has been designed to be accessible to C programs. Higher-level APIs can instead be used in C++ programs by using lambdas and related constructs, as was done in Habanero-C++ [16].

Care should be taken in deciding the total bytes of shared buffer written by tasks generated by **forasync** and **async** constructs. The DSP cache line size is 128 bytes, whereas it is 64 bytes for the ARM. The ARM can read a buffer written by the DSP only after it invalidates its own cache. If the size of the buffer accessed in an asynchronous task is not a multiple of 128 bytes, a cache flush by the ARM can overwrite the DSP results. To avoid this situation, we have chosen a tile size as 32 in line 39. This will ensure that 128 bytes of contiguous data are written to shared buffer 'a' by each task. Another (less efficient) way to achieve this is to pack each element of 'a' in a structure with a padding to ensure each array element is 128 bytes (thereby allowing for tile sizes as small as 1).

A HC-K2H program is compiled with both the ARM C++ compiler and the DSP C compiler so as to obtain executables for both classes of processors.

## IV. Implementation

In this section, we describe the design of the HC-K2H work-stealing runtime.

### A. Data-structures for task management

In traditional work-stealing runtimes, each worker thread maintains a software-implemented *deque*. Thieves can steal task from this deque using atomic operations. However, the DSP C platform supports neither atomic operations nor the pthread library. Though it can support hardware queues and up to 32 hardware semaphores (not accessible from ARM). Hardware queue accesses are always thread safe and can be performed both from the ARM and the DSP. Hence, in HC-K2H the DSP workers use hardware queues for task management. These hardware queues allow `push` operations to be performed at either end, but `pop` operations can only be performed from the tail. To decide the work-stealing data-structure

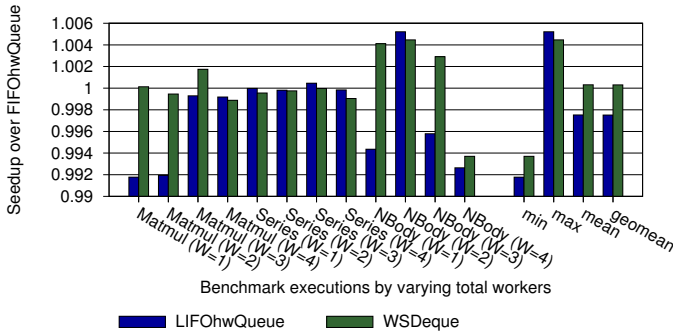



Fig. 4: Work-stealing performance at ARM using software deques and hardware queues

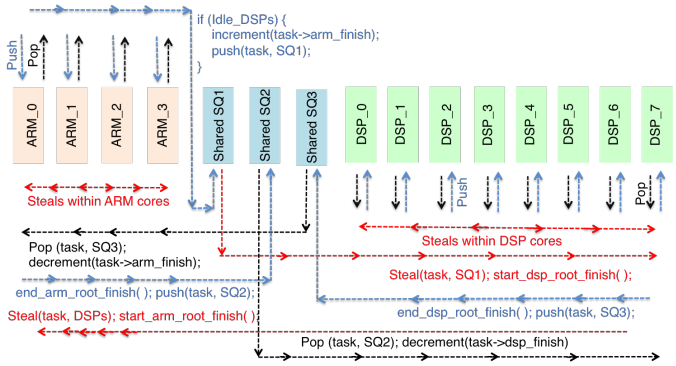



Fig. 5: HC-K2H work-stealing runtime

for ARM, we performed the experiment shown in Figure 4 (experimental methodology described later in Section V-A). We implemented three different versions of Habanero-C library and executed each of our benchmarks using them on ARM processor only. Each version of the runtime differs from each other in the way the tasks are managed by the workers. The first version (WSDeque) uses the traditional work-stealing deques, second version (FIFOhwQueue) uses a FIFO hardware queue, and the third version (LIFOhwQueue) uses LIFO hardware queues at each worker for task management. We observed that all the three versions performed roughly the same. However, HC-K2H uses WSDeque for ARM workers rather than hardware queues so that they don't have to rely on hardware queue descriptors (shared across all hardware queues in HC-K2H) for managing every asynchronous task.

### B. Hybrid work-stealing

Figure 5 shows the design of HC-K2H's hybrid work-stealing runtime. During the ARM runtime initialization phase, the message descriptor queue and the hardware queues are heap-allocated by the CMEM allocator. This heap resides on the on-chip MSMC memory. Next, the ARM launches the DSP executable and passes the information on queues to DSP core 0, which then writes back the information in DSP shared variables. Once all the queues are initialized by DSP core 0, it unblocks other DSP cores from a barrier. Every DSP core now becomes a thief and start hunting for tasks. ARM core 0 executes the user main and launches the computation.

To manage hybrid steals, three extra hardware queues (SQ1, SQ2, and SQ3) are used in HC-K2H runtime. Both ARM and DSP cores first try a local steal. After failing they try to steal from each other. ARM can access DSP queues but DSP cannot access ARM's deque. Also the finish scope cannot be shared between DSP and ARM. If there is at least one task on ARM deque, it will check if any of the DSP cores are free (CMEM allocated flag accessed by ARM and DSP). If yes, then ARM will pop a message descriptor from descriptor queue and copy the task information into it. It also stores its current finish scope inside this descriptor. ARM increments its local finish counter and then pushes this descriptor to SQ1 (accessible to DSP). After a hybrid steal, both ARM and DSP wraps the task execution within a new finish scope. Once the finish scope is terminated, ARM pushes a descriptor in SQ2 whereas DSP pushes to SQ3. This descriptor contains the original finish scope of the root task. During steal cycles, ARM try pop from SQ3 and DSP try from SQ2. If they succeed they will decrement the finish scope embedded in this descriptor.

### C. Cache updates

ARM cores are cache coherent but DSP cores are not. DSP cache settings allow two write modes: a) write-back; and b) write-through. In write-back mode, DSP core has to explicitly call the cache write back and invalidate method after writing to shared data. In write-through the cache contents are written back to memory by default. However, in either case a cache invalidation method is always called before executing any task. DSP cores allocate memory for finish object on non-cached memory. Any writes to finish (counter increment and decrement) are never cached and visible to all DSP cores.

CMEM allocated buffers (writable by ARM and DSP) are packed in **`ws_args_t`** and passed as parameters to variable argument function **`start_finish`** (Figure 3). ARM stores this information in every subsequent finish scopes. It calls **`ws_cacheWbInv`** on all these CMEM buffers at the outermost finish scope. When exiting **`end_finish`**, ARM cores call **`ws_cacheWbInv`** on CMEM buffers such that ARM neither looses the data in its cache, nor the results from DSP execution. It performs similar action at all **`end_finish`** for the tasks stolen from DSP (Section IV-B).

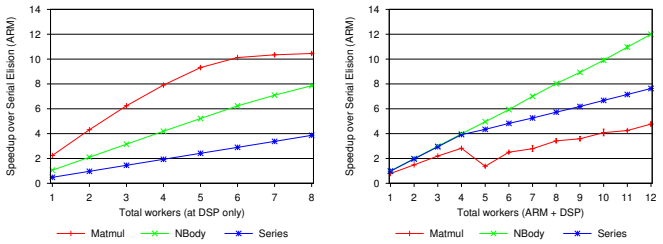
## V. Experimental results

### A. Experimental setup

We have used three different benchmarks: a) NBody problem [7] (predicting the individual motions of a group of celestial objects interacting with each other gravitationally), b) Series test [17] (calculates first n Fourier coefficients of the function $(x+1)^x$ defined on the interval 0,2), and c) Matmul [6] (2D matrix multiplication). NBody and Series benchmarks are non-recursive and use **`forasync`** inside a flat finish scope. Matmul is recursive **`async-finish`** type generating nested finish scopes. We run each experiment six times, reporting the mean along with a 95% confidence interval based on a Student t-test.

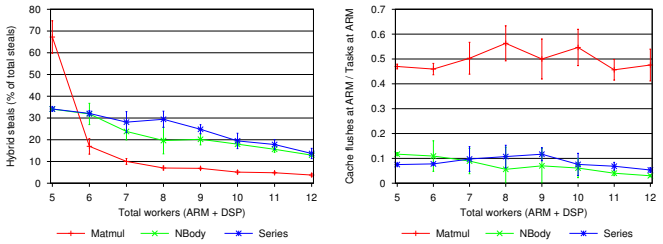
### B. Work-stealing performance

We execute each benchmark using both hybrid work-stealing and DSP only work-stealing. In hybrid work-stealing




(a) DSP work-stealing performance (b) Hybrid work-stealing performance

Fig. 6: Work-stealing performance




(a) Total percentage of hybrid steals (b) Ratio of total cache flushes at ARM to total asynchronous tasks generated at ARM

Fig. 7: Hybrid work-stealing statistics

results we very the total number worker threads from 1 to 12. Using up to 4 worker threads results in ARM only work-stealing. Increasing worker threads beyond 4 involve DSP cores, resulting in hybrid work-stealing. In DSP only work-stealing, one of the ARM cores offloads the entire computation to DSP and waits for the completion. Figure 6 shows the performance of both hybrid work-stealing and DSP only work-stealing for all the benchmarks. The speedup at each worker thread is calculated against the execution of serial elision version of benchmarks at ARM. Removing all HC-K2H programming constructs from a benchmark creates serial elision version. Increasing worker threads in hybrid work-stealing increases the speedup of all the benchmarks. NBody and Series show excellent speedup with 12 worker thread using hybrid work-stealing significantly outperforming the 8 core DSP only work-stealing. However, Matmul using hybrid work-stealing does not outperform DSP-only work-stealing.

The main reason for this anomaly is the nature of the Matmul benchmark. Matmul is a recursive benchmark generating several nested finishes, and relatively fine-grained tasks, the consequences of which can be observed on Figure 7. Figure 7(a) shows the total number of hybrid steals in each benchmark as a percentage of total steals. When a single DSP worker is added to the execution (resulting in 5 worker threads), hybrid steals in Matmul increase to 67%. Hybrid steals are always much more costly than local steals, resulting in worse performance for 5 worker threads than for 4 worker threads. Figure 7(b) shows the ratio of total number of **`ws_cacheWbInv`** called from **`end_finish`** at ARM to the total number of asynchronous tasks generated at ARM. We can observe that for Matmul, the ARM side of the machine has to perform a cache flush approximately once for every 2 tasks

generated, a much more frequent occurrence than for NBody and Series. Large number of **`end_finish`** implies frequent calls to **`ws_cacheWbInv`** at ARM (Section IV-C), resulting in a significant loss of performance.

We would like to note that increasing the granularity of Matmul tasks (an obvious approach to reducing the total number of steals and thus the total number of hybrid steals) does eventually result in a scenario where 4 ARM cores and 8 DSP cores outperform 8 DSP cores. We chose not to include those results since such large task granularity results in a very poor sequential performance of each task due to cache issues, negating any advantage of using additional cores. We chose instead to present the results with the best-performing tile size. This point to an obvious topic for future research in devising a runtime strategy for minimizing hybrid steals without affecting the sequential performance or parallel slackness.

## VI. Related Work

Previous work has looked into providing programming models for architectures with heterogeneous cores. Models such as CUDA and [18] and OpenCL [19] can target GPU accelerators using a restricted programming model that also includes a considerable amount of hardware detail. Models such as StarSs offer a pragma-based approach for expressing data parallel computations and its instantiations target lower level architectures such as IBM's Cell [20] and GPUs [21]. Another model, which looks at creating a high level model for embedded systems is Lime [22] - an enhanced Java language which can generate Verilog code for FPGAs and OpenCL for GPUs. Closer related to our work is the CnC-HC programming model [23] which relies on a runtime that offers work-stealing across CPUs, GPUs and FPGAs. Previous work [4], has demonstrated an OpenMP 4.0 implementation on the Keystone-II 66AK2H ARM-DSP SoC. In addition to this, TI also offers an OpenCL programming models for their Keystone II platform. Both these models use the DSP cores as accelerators, with the ARM cores offloading the work in parallel regions to them. To our knowledge however, HC-K2H is the first system to provide a work-stealing runtime for ARM and DSP cores, along with abstracting away all the hardware details.

## VII. Conclusion

In this paper, we presented a high-level task-parallel programming model based on Habanero and an implementation of that model on the heterogeneous Texas Instruments KeyStone II platform that combines general-purpose ARM cores and special-purpose DSP cores. Our model allows creation and synchronization of parallel tasks without any indication on where they should execute. Our implementation allows task execution on either ARM or DSP cores and load balancing through work-stealing amongst heterogeneous cores, while using the mechanisms available on the KeyStone II platform for synchronizing operations within the runtime.

Our experimental evaluation of task-parallel benchmarks shows that our system can effectively use all the available resources, achieving optimal load balance and even outperforming DSP-only executions. The experiments also show that future research in runtime heuristics for heterogeneous systems is necessary since hybrid execution does not always outperform DSP-only execution, due to the fact that overhead of heterogeneous work-stealing and increased number of steal attempts might outweigh the benefits of complete load balance.

## Acknowledgment

We would like to thank Vincent Cavé for implementing the library version of Habanero-C language.

## References

[1] M. Feldman, "Titan sets high water mark for GPU supercomputing," *HPCWire*, 2012.

[2] W. Vanderbauwhede and K. Benkrid, *High-Performance Computing Using FPGAs*. Springer, 2013.

[3] "Texas Instruments Literature: SPRS866: 66AK2H12/06 Multicore DSP+ARM Keystone II System-on-Chip (SoC)," Texas Instruments, Tech. Rep., November 2012.

[4] G. Mitra, E. Stotzer, A. Jayaraj, and A. P. Rendell, "Implementation and optimization of the openmp accelerator model for the TI keystone II architecture," in *IWOMP'14*, 2014.

[5] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *PLDI '98*. ACM, 1998.

[7] "Habanero-C Overview," https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C, Rice University, 2013.

[8] V. Cavé, "HClib: a library implementation of the Habanero-C language," http://habanero-rice.github.io/hclib/, 2013.

[9] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: the new adventures of old X10," in *PPPJ*, 2011, pp. 51–61.

[10] O. A. R. Board, *OpenMP Application Program Interface, Version 4.0*, July 2013, http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[11] P. Charles *et al.*, "X10: An object-oriented approach to non-uniform cluster computing," in *OOPSLA*, 2005, pp. 519–538.

[12] O. Tardieu, H. Wang, and H. Lin, "A work-stealing scheduler for X10's task parallelism with suspension," in *PPoPP*, 2012, pp. 267–276.

[13] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu, "Work-stealing without the baggage," in *OOPSLA*, 2012, pp. 297–314.

[14] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *ICS '08*. ACM, 2008, pp. 277–288.

[15] T. Instruments, "CMEM overview," http://processors.wiki.ti.com/index.php/CMEM_Overview.

[16] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar, "HabaneroUPC++: A compiler-free PGAS library," in *PGAS*, 2014, pp. 5:1–5:10.

[17] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey, "A benchmark suite for high performance java," *Concurrency - Practice and Experience*, vol. 12, no. 6, pp. 375–388, 2000.

[18] J. Nickolls, I. Buck, M. Garland, Nvidia, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[19] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[20] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a programming model for the Cell BE architecture," in *SC '06*, 2006.

[21] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the starss programming model for platforms with multiple GPUs," in *Euro-Par*, 2009.

[22] J. S. Auerbach, D. F. Bacon, P. Cheng, and R. M. Rabbah, "Lime: a java-compatible and synthesizable language for heterogeneous architectures." in *OOPSLA'10*, 2010.

[23] A. Sbîrlea, Y. Zou, Z. Budimlić, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms," in *LCTES '12*, 2012.