2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

TFluxSCC: Exploiting Performance on Future Many-core Systems through Data-Flow

Andreas Diavastos, Giannos Stylianou and Pedro Trancoso
Department of Computer Science
University of Cyprus
Nicosia, Cyprus
Emails: {diavastos, gstyli01, pedro}@cs.ucy.ac.cy

***Abstract*—The current trend in processor design is to increase the number of cores as to achieve a desired performance. While having a large number of cores on a chip seems to be feasible in terms of the hardware, the development of the software that is able to exploit that parallelism is one of the biggest challenges.**

In this paper we propose a Data-Flow based system that can be used to exploit the parallelism in large-scale many-core processors in an effective and efficient way. Our proposed system - TFluxSCC - is an extension of the TFlux Data-Driven Multithreading (DDM), which evolved to exploit the parallelism of the 48-core Intel Single-chip Cloud Computing (SCC) processor. With TFluxSCC we achieve scalable performance using a global address space without the need of cache-coherency support. Our scalability study shows that application's performance can scale, with speedup results reaching up to 48x for 48 cores. The findings of this work provide insight towards what a Data-Flow implementation requires and what not from a many-core architecture in order to scale the performance.

***Keywords*-dataflow; many-cores; programming model; Data-Driven Multithreading;**

I. Introduction

Scaling the performance of an application can be achieved by either improving the hardware, or developing more efficient software to solve the particular problem. To retain the power-performance efficiency to an acceptable level we are currently exploring parallel processing as the way to scale performance. Consequently, the trend today is to include more and more cores into the processor resulting in what is known as a multi- and many-core processor. From the hardware design perspective, the more parallel units offered for execution, the higher the performance that can be achieved. From the software design perspective though, this new trend creates new challenges. To achieve scalable performance in these new systems, programmers are required to think parallel. This essentially means learning new programming models and developing new algorithms that exploit parallelism. Therefore, the solution to the scalability of the performance depends on scalable hardware with increasing number of computational units along with efficient programming and parallel execution models that hide the hardware complexity from the programmer.

Data-Flow is a natural paradigm for describing parallelism using directed graphs based on the path of the data [1]. Nevertheless, the original Data-Flow implementations suffered from serious limitations as they required specialized hardware [2]. New hardware designs though, make the Data-Flow model a possible solution for scaling the performance of applications. Consequently, researchers have proposed new alternatives based on this model that are able to exploit the parallelism effectively. One such alternative is the Data-Driven Multithreading (DDM) [3]. Following the footsteps of the Data-Flow model it can exploit the maximum available parallelism of an application by exploiting data dependencies, while at the same time overcomes overheads by increasing the granularity of the execution blocks. In DDM the synchronization part of the program is separated from the communication part allowing it to hide the synchronization and communication delays [4]. Different implementations of the DDM model exist. Some implement the scheduling unit in hardware as to minimize the effects of the overheads and others in software as to guarantee portability of the platform to any new hardware design.

The focus of this paper is to show how a software implementation of the DDM model of execution can offer performance scalability on a many-core architecture by efficiently exploiting the parallelism and at the same time relieve the programmer from the hardware details, such as data communication. The complete software platform we present in this work is based on the TFlux platform [5]. It offers an environment for parallel execution on many-core architectures that is able to scale without major hardware requirements or programming effort. To test our implementation we use the Intel Single-chip Cloud Computer (SCC) [6] that was developed by Intel for many-core software research, as a representative for future many-core processors. Our proposed system includes a source-to-source preprocessor that takes as input programs in C, augmented with directives that specify the threads and their dependencies, as well as, a runtime system to handle the scheduling of the threads in a Data-Flow manner. Our evaluation is performed on a real Intel SCC system and our model is implemented in software as a library that is linked to the application.

1066-6192/15 $31.00 © 2015 IEEE
DOI 10.1109/PDP.2015.69

Table I
TFLUX DDM PRAGMA DIRECTIVES

`#pragma ddm startprogram`	Define the start and the end of a DDM program
`#pragma ddm endprogram`	
`#pragma ddm block ID`	Define the start and the end of a block of threads with identifier *ID*
`#pragma ddm endblock`	
`#pragma ddm thread ID kernel NUMBER`	Define the boundaries of a DDM thread with identifier *ID* and the kernel *NUMBER* to execute on
`#pragma ddm endthread`	
`#pragma ddm for thread ID`	Define the boundaries of a DDM loop thread with identifier *ID*
`#pragma ddm endfor`	
`#pragma ddm kernel NUMBER`	Declare the number of kernels to be used
`#pragma ddm var TYPE NAME`	Declare a shared variable with *NAME* and *TYPE*
`#pragma ddm private var TYPE NAME`	Declare a private variable with *NAME* and *TYPE*

The contributions of this paper are as follows:

- TFluxSCC: The first DDM implementation for a many-core processor;
- Evaluation of TFluxSCC on a real 48-core SCC system, achieving up to 48x speedup;
- Identification of the characteristics that efficiently exploit the parallelism of a scalable architecture.

II. DATA-DRIVEN MULTITHREADING

Recently we have seen an increasing interest in the Data-Flow model as a way to efficiently exploit the maximum available parallelism of applications. DDM is a Data-Flow model where the granularity of the Data-Flow code is a thread and the synchronization part of the program is separated from the communication part as to overcome the synchronization and communication overheads imposed by the dynamic scheduling process [3], [4]. DDM programs are composed of Data-Driven Threads (DThreads) that contain an arbitrary number of instructions. Within a DThread the instruction execution follows the classic control-flow model, thus allowing any other runtime or compile-time optimizations to be performed. The programming of the DDM model is done explicitly by the programmer by defining the DThreads in a program and the dependencies amongst them, either by declaring a direct dependence on other DThreads or by declaring the inputs and outputs of the DThreads.

This work is based on the TFlux implementation [5] of the DDM model. We chose TFlux as it is a complete platform that includes a programming environment for DDM applications using compiler directives, a source-to-source preprocessor that translates the application augmented with the directives into DDM parallel code and a Thread Scheduling Unit (TSU) that handles the Data-Flow scheduling of the DThreads at runtime. An important advantage of TFlux is that it is not built for a specific machine but rather works as a virtualization platform for DDM program execution on a variety of computing systems. Another reason we chose the TFlux implementation is that the produced parallel code from the platform is in ANSI-C which complies with the supported programming languages for most systems, such as the Intel SCC programming API used in this work.

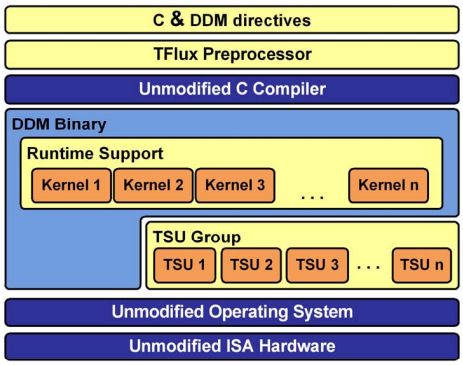


Figure 1. The layered design of the TFlux system [5]

Using the TFlux directives, the programmer can define the DThreads that form a DDM program along with their dependencies. Figure 1 shows the layered design of the TFlux system. The programmer uses the top layer to develop DDM applications. This abstracts the details of the underlying hardware. DDM applications are developed using ANSI-C with DDM directives [7] as the ones shown in Table I. The directives are used to define the code of the DThreads and to express the dependencies between them. In Figure 2 we show an example of how to program a sequential application using TFlux directives. The code in the figure is part of an actual DDM program that was used for the evaluation in this work. The DDM source-to-source preprocessor is used to parse the C+DDM directives code and produce a C program that can be compiled with any commodity C compiler. The executable produced invokes the *TFlux Runtime Support* operations that allow the application to execute as a DDM program.

TFlux also requires a TSU to enforce the Data-Flow execution of the DThreads. The TSU's task is to load the synchronization graph of a DDM application and according to the dependencies, initiate and schedule the execution of all DThreads in a Data-Flow manner. In the first implementation of DDM, the D^2NOW [3], each processor needed to have its own private TSU since the execution nodes were indepen-

```
#pragma ddm startprogram
...
#pragma ddm block 1
...
#pragma ddm for thread 4 unroll 1

    for (j = 0; j < 512; j++)
    {
      for (i = 0; i < 512; i++)
        yp[j][i]+= c[i][j]*(y[j]+k3[j]);
    }

#pragma ddm endfor

#pragma ddm for thread 5
      \ reduction sum + double total depends(4)

   for (i = 0; i < 512; i++)
   {
     if(!initFlag02)
     {
       initFlag02=1;
       for (j = 0; j < 512; j++)
       {
         k4P[j]=h*(pow[j]-(yp[0][j]+yp[1][j]));
       }
     }
    yout[i] = y[i] + (2*k3[i]+k4P[i])/6.0;
    sum+=yout[i];
   }

#pragma ddm endfor

#pragma ddm endblock
...
#pragma ddm endprogram
```

Figure 2. Code example of how to use TFlux directives in a program.

dent machines. In the TFlux implementation the TSUs were unified in a single unit named the *TSU Group*. This unit is logically split in *n+1* parts. One part per core (totalling *n*) for the core's own TSU operations and one common part which is located on a dedicated core and manages the common operations of the TSU for all cores. In Section IV we show how our implementation of the TSU combines ideas from both the D^2NOW and the original TFlux implementations.

III. Intel SCC Architecture

The need to optimize performance per watt has resulted in the increase of the number of cores in processor chips [8]. Many light-weight cores will be replacing the few sophisticated cores we have currently in multi-core chips, creating the many-core chips with hundreds or more cores. In our work we examine the Intel SCC experimental processor as a 48-core 'concept vehicle' created by Intel Labs as a platform for many-core software research [6].

The Intel SCC processor consists of 24 dual-core tiles interconnected by a 2D-grid network as illustrated in Figure 3. These tiles are organized in a 6x4 mesh with each one containing two cores with dedicated L1 and L2 caches of 16KB and 256KB respectively, 16KB Message Passing Buffer (MPB) for message storing travelling through the network, a Traffic Generator (TG) for testing the performance of the on-chip network, a Mesh Interface Unit (MIU) connecting the tile to its' network router and two test-and-set registers.

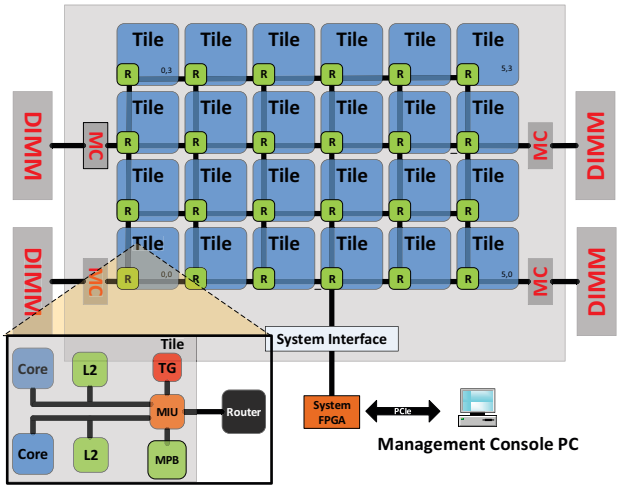


Figure 3. Intel SCC top-level and tile top-level architecture

The maximum main memory the current system can support is 64GB and the 32-bit memory addresses of the cores are translated into system addresses by the MIU through a lookup table (LUT). The main memory of the system is located outside the chip and the access to it is achieved through four on-chip DDR3 Memory Controllers (MC). The SCC supports both distributed and shared memory models. The system memory is composed of four regions. Each cores' private main memory (*Private off-chip memory*), the systems' global address space (*Shared off-chip memory*), the Message Passing Buffer (MPB) used to store messages to be sent through the network (*Shared on-chip memory*) and the *L2 cache* of each core.

The Intel SCC is equipped with a large number of light-weight processing units with low power consumption and with multiple memory controllers serving them. Based on the characteristics of the Intel SCC we understand that future many-core processors will focus on energy-efficient designs. Expensive, in matters of design- and operating-cost, hardware support units will be avoided and other solutions must be exploited. One example in the Intel SCC is the cache coherency mechanism [9], where caching is only supported for data allocated on the private address space. Techniques like this though, have an impact on the performance and the programming of new architectures.

Being an experimental processor, the SCC comes with

a concept Programming API called RCCE that supports the Single Program Multiple Data (SPMD) model of execution. It supports C/C++ programming languages with RCCE API extensions to implement the message passing communication of the system. More information on the RCCE communication environment and all the operations supported can be found in [8].

IV. TFluxSCC Implementation

Our goal in this work is to achieve performance scalability of the DDM model using the TFlux Platform on the Intel SCC processor. We also want to show that using the DDM model of execution on the Intel SCC we are able to exploit the performance even for simple scalable hardware. TFlux uses compiler directives as to define DThreads and the dependencies amongst them. Given that TFlux is an existing platform, we maintained the programming style and syntax for TFluxSCC. We used the global address space of the SCC for storing application data. Thus, no communication mechanism is necessary to exchange data between cores as the hardware system itself will take care of this. This way we manage to retain the programming directives of TFlux unchanged as we don't need to add extra information. Therefore, this allows the portability of the already existing applications.

Although the Intel SCC supports global address space, there is no cache-coherency protocol. In order to ensure correctness of the data coming from the global address space, the SCC does not allow the use of the cache for storing application data coming from the global address space. The DDM model though, and consequently our TFluxSCC implementation, doesn't require cache coherency as it doesn't allow simultaneous access on shared data. In a DDM program, data is shared through the input and output of the threads as they are defined by the dependencies declared by using the pragma directives. Consequently, at each moment only one thread can access a specific data structure on the global address space. Thus, caching global address space data is possible when using TFluxSCC. To ensure correctness, we flush the cache after the thread completes its execution, as to guarantee that the shared data is saved back to memory.

We were able to modify the SCC configuration (using a modified Linux image, provided by the SCC platform) as to allow the caching of data coming from the global address space. We also modified the TFluxSCC runtime as to flush the cores caches after writing data to the global address space. To measure the impact on the performance imposed by this technique we tested both scenarios and in Figure 5 we present the speedup results for 48-cores. In the first scenario (*Uncacheable Shared Memory*) we test all our applications implemented in TFluxSCC without caching application data. The second scenario (*Cacheable Shared Memory*) shows the

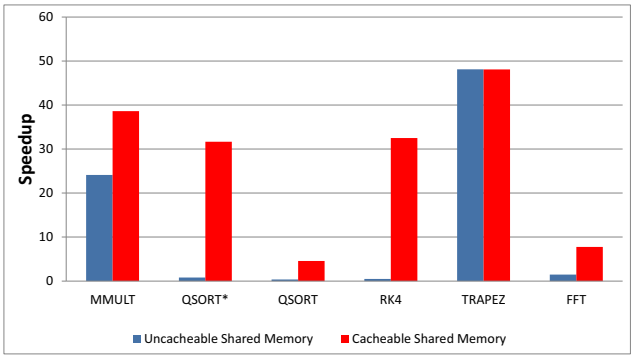


Figure 5. Comparing speedup results of *uncacheable* against *cacheable* shared memory on the Intel SCC

results when allowing the caching of the application data coming from the global address space.

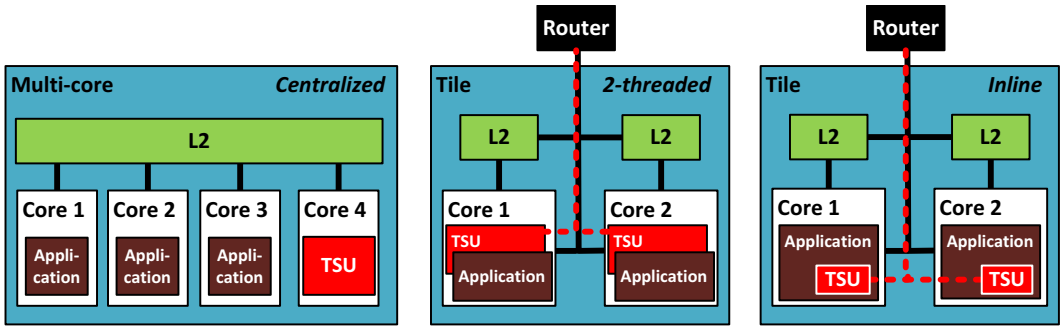
Conventional shared-memory parallel programming models require more sophisticated hardware components, that in systems of 100s -1000s of cores would be both power and performance inefficient due to the extra hardware and the overheads imposed by the coherency protocol [10]. Using TFluxSCC we manage to guarantee correct execution without major impact on the performance while at the same time maintain hardware simple.

A. TSU Implementations

The TSU implementation in the TFlux Platform is a semi-centralized implementation. This means that except from the TSU thread, part of the scheduling operations are executed by the application threads. This technique was used as an optimization that reduces the overheads of the TSU functionalities on a multi-core system [5]. Since there is only one instance of the TSU structures in TFlux this needs monitoring and locking of shared structures. Using shared structures on systems with a large number of cores will eventually create contention among the cores and locking can possibly result in starvation. For this reason no locking scheme is supported by the SCC.

In the original TFlux TSU (Figure 4(a)), the operation that handles the update messages involving all application threads is centralized. This means that the update messages that notify a thread that is ready to execute is sent explicitly by the TSU thread. In large-scale systems and in applications with a large number of update messages this can become a bottleneck to the performance due to contention in the network.

What we propose in TFluxSCC is a non-centralized runtime system that is able to scale and consume as few resources as possible, thus reducing the overheads to a minimum. In TFluxSCC we distribute the TSU functionalities to each core in order to achieve scalability regardless of the number of cores used. Also, we only allow for



(a) (b) (c)

Figure 4. TSU Evolution: (a) Original TFlux TSU, (b) *2-threaded* for TFluxSCC and (c) *Inline* for TFluxSCC

the TSU to take control of the execution unit at the time needed and only for as long as it is needed, as to reduce the runtime overhead to the minimum. In Figure 4 we show the evolution process of the TSU from the original TFlux implementation (Figure 4(a)) to what we propose for TFluxSCC (Figure 4(c)).

In a first attempt to de-centralize the runtime system we created one TSU thread for every core (Figure 4(b)): the *2-threaded* implementation. This implementation leads to one extra thread per core. As the SCC cores do not support hardware multi-threading, depending on the scheduling policy, the OS will switch the execution from the application to the TSU thread allowing the TSU to hold the execution unit from the application thread. The TSU implements a busy wait loop in the background until an update message is ready to be send. Thus, in many cases the TSU will take control of the execution unit without having any useful work to do that will help the application progress. To find the possible cost of this context switching, we control the time that the TSU thread uses the execution unit. At runtime, we deactivate the TSU thread for a certain amount of time by adding a *sleep* call. This way we allow for the application thread to take control of the execution unit for a longer period of time and we also reduce the number of times that the OS switches the execution from the application thread to the TSU thread. By limiting the time that the TSU thread uses the execution unit (*sleep* implementation) we manage to reduce the total execution time from our baseline implementation (*2-threaded*) as shown in Figure 6. Although the *sleep* is effective, it can not be considered for an implementation as the best sleep time varies with the applications and can not be determined in advance. Any runtime mechanism used to determine the best value will impose significant overhead to the execution. While the absence of multi-threading on the Intel SCC is a limitation, it is also a factor of scalability of the hardware. The simpler the cores are, the more that can be integrated on the same processor.

To avoid the above restrictions and limitations we propose the *Inline* implementation. In this implementation we integrate the TSU functionality with the application thread as shown in Figure 4(c). We remove the busy wait loop from the TSU and call its' operations at the end of the execution of an application thread, which is the only time that the TSU will have real operations to execute (send update messages to consumers). This solution allows us to utilize the execution unit of the core to the maximum. In Figure 6, we show the execution time of the three approaches, normalized to the *2-threaded* implementation, for two applications. The results observed verify what we discussed earlier.

In Figure 7 we show the execution time breakdown into the time for the application. *i.e.* the time spent to execute pure application code, and the time for the *Inline* implementation of the TSU, *i.e.* the time spent to execute the different scheduling procedures, such as the update of the local TSU structures after a thread execution, the exchange of update messages with consumer threads on different cores. We also test two different network frequencies to find out whether the message exchange process of the TSU can be improved by the current hardware. In our baseline scenario, the SCC network was operating at 800MHz. We then increased the network frequency to the maximum, which is 1600MHz and observe that the TSU time remains constant. This means that the time spent by the TSU for message exchanging in the *Inline* implementation is not significant. In this Figure, we also observe that the overall TSU overhead is small and it reduces as we apply unrolling to the applications. This happens because in DDM every loop iteration is considered a separate thread, thus the more iterations we have, the more threads we have to schedule. Using unrolling, we reduce the total number of threads we have to schedule, thus reducing the total time consumed by the TSU procedures.

B. Compilation Toolchain

As described in Section III, Intel SCC processor comes with it's own programming API, called RCCE. Therefore,

Table II
EXPERIMENTAL WORKLOAD DESCRIPTION AND PROBLEM SIZES

Benchmark	*Source*	*Description*	*Characteristics*	*Unroll Factor*	*Problem Size*		
					Small	*Medium*	*Large*
MMULT	kernel	Matrix multiply	Compute+Memory-Bound	16	64x64	128x128	256x256
QSORT	MiBench	Total Array sorting	Memory-Bound	N/A	100K	200K	400K
QSORT*	MiBench	Partial Array sorting	Memory-Bound	N/A	100K	200K	400K
RK4	kernel	Differential equations	Compute-Bound	1	1024	2048	4096
TRAPEZ	kernel	Trapezoidal rule for integration	Compute-Bound	256	30	31	32
FFT	NAS	FFT on a matrix of complex numbers	Compute-Bound	1	32	64	N/A

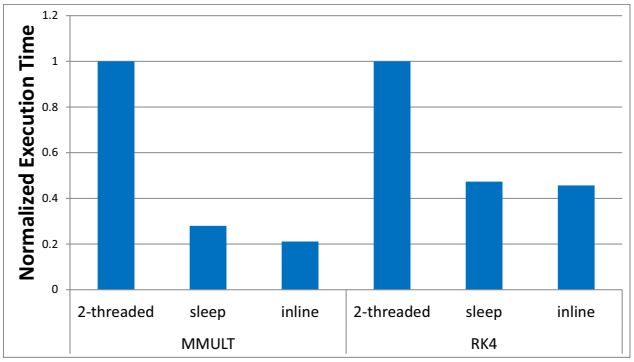


Figure 6. Execution time of the implementations of the TSU for 48 cores normalized to the *2-threaded*

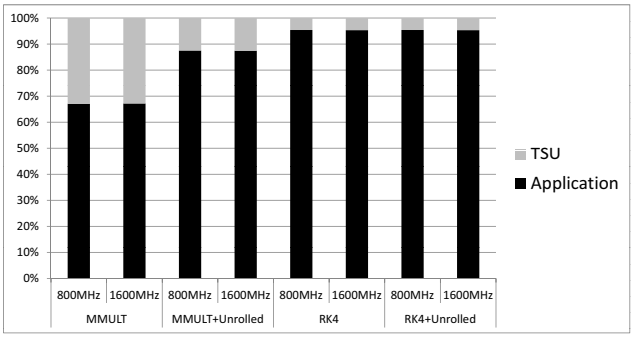


Figure 7. Application and TSU execution time breakdown for different frequencies

we integrated the RCCE API into the TFluxSCC preprocessor. We updated the DDM C source-to-source Preprocessor [7] in order to be able to generate code for the Intel SCC Processor. The basic operations added on top of the previous implementations of TFlux are the initialization of the platforms' API, the support for memory allocation on the global address space and most importantly, the flushing of a cores' cache after writing data in the global address space. This later operation is necessary for the model to synchronize explicitly the cache contents to memory as to ensure correctness, given that no hardware cache-coherency is available.

Although the intended platform for our implementation is different, we made sure that TFluxSCC can be programmed in the exact same way as the original TFlux. This means that the interface of TFlux (*i.e.* the directives in Table I) is kept the same in the TFluxSCC implementation as well. This way, the updated preprocessor provides backward compatibility with the previous implementations of the TFlux Platform. Using a predefined command line flag (*scc*), we can inform the preprocessor that we want to produce code for the Intel SCC platform. Thus, no change is needed to the previously implemented applications.

V. EXPERIMENTAL SETUP

We have evaluated our implementation of TFluxSCC using six different benchmarks. Three of them are kernels that represent common scientific operations [11], two belong to the *MiBench* [12] suite and one to the NAS [13] suite. QSORT* is a subset of QSORT and represents the partial sorting of an array. In QSORT* we remove the reduction operation and measure only the fully parallel part of QSORT, where each core sorts its' own portion of the input array. The benchmarks are briefly described along with their different problem sizes in Table II. For our experiments we used three different input problem sizes: *Small*, *Medium* and *Large*. We measure execution time and in order to minimize the statistical error, we executed each experiment ten times. The results shown are the arithmetic average of the measurements after excluding the outliers. The baseline execution for every scenario is the best sequential execution of the benchmarks on a single SCC core.

Our hardware setup was an Intel SCC experimental processor, RockyLake version. The system has a total of 32GB of main memory and we used a balanced frequency setting for our experiments of 800MHz for the tile, the mesh interconnection network and the DDR3 memory and Memory Controllers. The operating system used for the Intel SCC cores was the Linux_dcm kernel provided by Intel SCC Communities repository that supports caching the data coming from the off-chip shared-memory to L2 cache. To cross compile our benchmarks for the SCC we used the GCC v.3.4.5 compiler with the optimization flag O3. For porting and executing the applications on the SCC we used the RCCE v1.4.0 tool-chain [8]. Finally, since our study

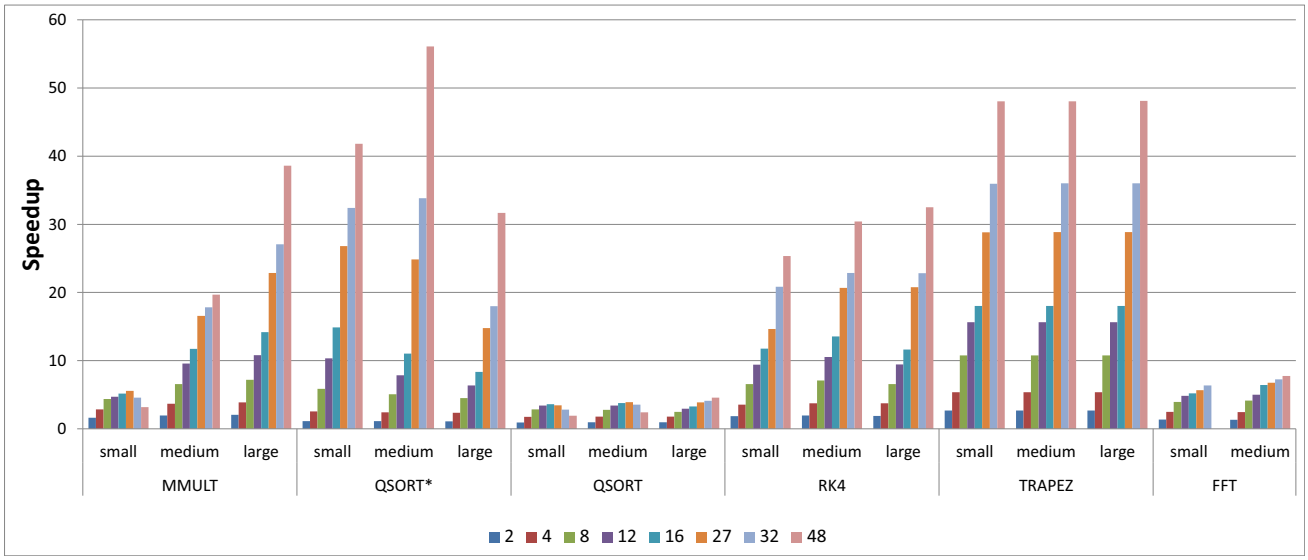


Figure 8. Performance scalability of TFluxSCC for different number of cores and input sizes

emphasizes on the scalability of the DDM model, all results are reported as *Speedup* compared to the baseline execution.

VI. Experimental Results

With this work we perform a scalability study of the performance for applications with different characteristics. In our workload we have included applications that are embarrassingly parallel like QSORT*, applications that are compute-bound like TRAPEZ and others that have a combination of memory- and compute-bound nature, as well as more complex dependencies among the different parallel threads. We executed the applications using up to 48 cores and with 3 different input sizes and present in Figure 8 the speedup results compared to the best sequential execution.

The results in Figure 8 show a large speedup for most applications. The application with the largest overall speedup is TRAPEZ, which is an application that is compute-bound and suffers no memory overheads. RK4, which has a considerable number of threads and dependencies achieves also a good speedup and thus it shows that the execution of the TSU code does not incur in a large overhead for the execution of the application. QSORT* shows very impressive speedup, especially for the *medium* input size. The reason that the speedup in this case exceeds 48 is that the input size of QSORT* fits in the cache thus, creating a super-linear speedup phenomenon. MMULT, that is both a compute- and memory-bound application, shows smaller but still large speedup. Finally, FFT and QSORT show the smallest speedup of all applications. QSORT is split into two phases. The first one is like QSORT* and thus has linear speedup. The following phase combines the results of all sorted parts as to build the complete sorted vector. This is done as a reduction using the merge sort algorithm. For this implementation we observed that a binary reduction was too costly so we implemented a more optimized n-way reduction, where at each step one thread combines the results of 4 sorted portions and managed to get better performance.

To see if the data set size affects the performance results we also tested the same applications with 3 different data set sizes. For MMULT and RK4 we observe that as we increase the size of the input data the speedup increases as expected for compute-bound applications. TRAPEZ achieves linear speedup for the all input sizes. For FFT, we could not scale to the large input size as it required more memory space than what is available from the global address space of the SCC. For QSORT*, which is a memory intensive application, we observe a different behavior as for the larger set the speedup is smaller than for the medium set. This is due to the fact that while for the medium set the portions of data to sort for the 48-core case are still fitting in the cache, for the larger set this is not valid any longer. In QSORT we observe a small difference in the performance while increasing the input size but it is not able to scale more as the reduction phase is a bottleneck for the performance.

Overall, the results show that we manage to scale well with different applications on a real many-core system. The fact that we manage to get 48x speedup for 48 cores, for compute-bound applications shows that our runtime does not add any overhead in applications that don't have a performance bottleneck in the algorithm.

VII. Related Work

There are several projects currently, targeting the exploitation of parallelism for many-core architectures. On the one hand, the industry is moving closer to the development of many-core processors with Intel proposing recently the

Many Integrated Cores (MIC) Architecture [14] and the 48-core SCC processor [6]. The MIC architecture has only 60 cores but exploits parallelism using wide vector units, while the latter adopts the clustered architecture with simple hardware design as to allow efficient scaling.

Totoni *et al.* in [15] use CHARM++ and MPI message passing paradigms to implement parallel applications with different characteristics in order to evaluate the Intel SCC platform in matters of performance. They get speedup results of up to 32.7x for 48 cores and they propose more sophisticated cores for the future many-cores in order to increase the performance, mainly for the sequential execution. RCKMPI [16] by Intel and SCC-MPICH [17] by RWTH Aachen University implement customized MPI libraries aiming to improve the message passing model with respect to the SCC many-core architecture. These two implementations use an efficient mix of MPB and DDR3 shared memory for low-level communication in order to achieve higher bandwidth and lower latency. In [18] the authors examine various performance aspects of the SCC using a stream benchmark and the NAS Parallel Benchmarks [13] *bt* and *lu*. Their findings show that for these benchmarks the data exchange based on message passing is faster than shared memory data exchange and in order to improve the memory access behaviour you must increase both the clock frequency of the mesh network and the memory controllers.

Intel SCC avoids the hardware-based cache coherency and introduces a software-oriented message passing based architecture instead. A software cache-coherency implementation for the SCC system can act as another potential solution for creating simpler many-core architectures, free of complex hardware. As X. Zhou *et al.* propose in [19], Software Managed Cache Coherence (SMCC) shows a comparable performance to hardware coherency while offering the possibility of having dynamically reconfigurable coherence domains on the chip. The unnecessary complex hardware support for applications with little sharing and the inability to support heterogeneous platforms make the SMCC achieve better use of silicon with significant reduction of hardware budget.

In addition, we also have other architectures like the GPUs' [20] that have been around for some time and support parallel execution with hundreds of cores. GPU's offer large computing power with low cost but the programming of such engines is still trivial and the programmer must have all the information of the underlying hardware in order to achieve good performance. GPUs are special purpose hardware units and the range of applications that offer significant performance increase on a GPU system is limited to data parallel applications. TFluxSCC on the other hand targets a more conventional hardware approach and can be used for a wider range of applications.

Programmability in many-core architectures is another challenge the community has to address. OpenCL represents a parallel programming standard especially for heterogeneous computing systems. SnuCL [21] is an OpenCL framework for heterogeneous CPU/GPU clusters that provides ease of programming for such systems. This framework achieved high performance on a cluster architecture with a designated, single host node and many compute nodes equipped with multi-core CPUs and multiple GPUs. The scalability though refers only to medium-scale clusters, since large-scale clusters may lead to performance degradation due to the centralized task scheduling model followed. Lee *et al.* in [22] presents a new OpenCL framework, this time for homogeneous many-cores with no hardware cache coherency, such as the Intel SCC. The framework includes a compiler and an OpenCL runtime which together with the dynamic memory mapping mechanism preserve coherency and consistency between CPU cores on the SCC architecture with a small overhead.

TERAFLUX [23] was a large-scale project funded by the European Union aiming to solve the challenges of programmability, manageable architecture design and reliability of a 1000+ core chips by using the Data-Flow principles. The idea was to develop new programming models, compiler analysis and optimization technologies in order to build a scalable architecture based mostly on off-the-shelf components while simplifying the design of such Tera-device systems.

Our implementation differs from other projects as we propose a complete programming and execution platform, for a many-core system, based on Data-Flow principles. Using a software approach helps our implementation to be easily ported to new architectures without the need of specialized hardware.

VIII. Conclusions and Future Work

In this work we exploit the Data-Flow model on a many-core system. Specifically, we propose the TFluxSCC platform that supports the programming and execution of DDM applications on the Intel SCC processor. Using a set of applications with different characteristics we were able to show that the performance scales well and good speedup is observed for most applications. It is relevant to notice that these are executions of real applications on a real many-core processor and TFluxSCC is the first software implementation of the DDM model for a many-core processor.

We believe that TFluxSCC is suitable for future generations of many-core processors as it is a software implementation that can be easily configured to any many-core architecture. It has limited demands on hardware support that allows for a simpler design with a larger number of execution units and thus increases the parallelism offered by the hardware. Our implementation only requires a global address space for storing application data and a selective data-cache flush policy for data that were written and came from the global address space. Cache-coherence is not a

requirement and this leads to using simpler, more scalable hardware.

In a future work we plan on scaling our implementation to a larger number of cores. To do so, we are going to use a simulation-based approach that will allow us to experiment with different hardware components. We also plan on expanding our evaluation benchmark suite to cover more application domains and compare our results with other parallel platforms and models.

REFERENCES

[1] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[2] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[3] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-driven multithreading using conventional microprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1176–1188, 2006.

[4] K. Stavrou, P. Evripidou, and P. Trancoso, "DDM-CMP: data-driven multithreading on a chip multiprocessor," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Springer, 2005, pp. 364–373.

[5] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso, "TFlux: A portable platform for data-driven multithreading on commodity multicore systems," in *Proceedings of the 37th ICPP '08*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 25–34.

[6] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *IEEE International ISSCC 2010*. IEEE, 2010, pp. 108–109.

[7] P. Trancoso, K. Stavrou, and P. Evripidou, "DDMCPP: The data-driven multithreading c pre-processor," in *The 11th Workshop on Interaction between Compilers and Computer Architectures*, 2007, p. 32.

[8] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl *et al.*, "The 48-core SCC processor: the programmer's view," in *Proceedings of the ACM/IEEE SC '10*, 2010, pp. 1–11.

[9] M. Loghi, M. Poncino, and L. Benini, "Cache coherence tradeoffs in shared-memory MPSoCs," *TECS'06*, vol. 5, no. 2, pp. 383–407, 2006.

[10] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems," in *Proceedings of the MICRO 42*. New York, NY, USA: ACM, 2009, pp. 413–422.

[11] M. C. Seiler and F. A. Seiler, "Numerical recipes in C: the art of scientific computing," *Risk Analysis*, vol. 9, no. 3, pp. 415–416, 1989.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proceedings of WWC '01 IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14.

[13] D. Baliley, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber *et al.*, "The NAS parallel benchmarks-summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, vol. 91. New York, NY, USA: ACM, 1991, pp. 158–165.

[14] (2013) Intel many integrated core architecture. [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html

[15] E. Totoni, B. Behzad, S. Ghike, and J. Torrellas, "Comparing the power and performance of Intel's SCC to state-of-the-art CPUs and GPUs," in *Proceedings of the ISPASS '12*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 78–87.

[16] I. A. C. Ureña, M. Riepen, and M. Konow, "RCKMPI–lightweight MPI implementation for Intels Single-chip Cloud Computer (SCC)," in *Recent Advances in the Message Passing Interface*. Springer, 2011, pp. 208–217.

[17] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor," in *HPCS International Conference*. IEEE, 2011, pp. 525–532.

[18] P. Gschwandtner, T. Fahringer, and R. Prodan, "Performance analysis and benchmarking of the Intel SCC," in *Proceedings of the IEEE CLUSTER '11*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 139–149.

[19] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha, "A Case for Software Managed Coherence in Manycore Processors," in *Poster on 2nd USENIX Workshop on Hot Topics in Parallelism HotPar10*, 2010.

[20] Q. Huang, Z. Huang, P. Werstein, and M. Purvis, "GPU as a general purpose computing resource," in *Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008*. IEEE, 2008, pp. 151–158.

[21] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of the 26th ICS '12*. New York, NY, USA: ACM, 2012, pp. 341–352.

[22] J. Lee, J. Kim, J. Kim, S. Seo, and J. Lee, "An OpenCL framework for homogeneous manycores with no hardware cache coherence," in *PACT'11*. IEEE, 2011, pp. 56–67.

[23] R. Giorgi, R. M. Badia, F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R. Gao, A. Garbade, R. Gayatri *et al.*, "TERAFLUX: Harnessing dataflow in next generation teradevices," *Microprocessors and Microsystems*, 2014.