2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Scalability and Efficiency of Database Queries on Future Many-core Systems

Panayiotis Petrides,Andreas Diavastos, Constantinos Christofi, Pedro Trancoso

Department of Computer Science
University of Cyprus
Nicosia, Cyprus
Emails: {ppetrides, diavastos, christofi.c, pedro}@cs.ucy.ac.cy

***Abstract*—Decision Support System (DSS) workloads are known to be one of the most time-consuming database workloads that process large data sets. Traditionally, DSS queries have been accelerated using large-scale multiprocessors. In this work we exploit the benefits of using future many-core architectures, more specifically on-chip clustered many-core architectures. To achieve this goal we propose different representative data parallel versions of the original database scan and join algorithms. We also study the impact on the performance when on-chip memory, shared among all cores, is used as a prefetching buffer. For our experiments we study the behaviour of three queries from the standard DSS benchmark TPC-H executing on the Intel Single chip Cloud Computer experimental processor (Intel SCC). Our results show that parallelism can be well exploited by such architectures and how important it is to have a balance between computation and data intensity. Moreover, from our experimental results we show that performance improvement of *5x* and *10x* for the corresponding query implementation without data prefetching. Finally we show how we could efficiently use the system in order to achieve high power-performance efficiency when using the proposed prefetching buffer.**

I. Introduction

The multi-core architecture is the *de-facto* standard in processor design. This architecture offers the benefit of an increased degree of parallelism to provide better performance, without the drawbacks of previous monolithic designs, such as high power consumption and complex design. As technology improves, the integration level increases leading to an increase in the number of cores per chip. While this results in a further increase of the degree of parallelism, it may not necessarily lead to improved performance, even when considering highly parallel applications. The increasing number of processing units per chip results in a higher demand for "feeding" those units with both instructions and data. At the same time, neither the number of pins on the chip, nor the links to memory improve at the same rate as the number of cores. Moreover, the complexity of the interconnection network of large-scale multi-core architectures increases with the number of cores. The above mentioned multi-core issues result in limiting the scalability in terms of number of cores of these architectures. The proposed large scale many-core architecture by Intel, also known as the Intel SCC [1] addresses the above limitations.

Database applications are of the most demanding workloads. More specifically, Decision Support Systems (DSS) database applications combine the processing of large data sets along with the computation of statistical information extracted from data. Our goal in this paper is first to show the advantages that a future clustered many-core architecture, like the Intel SCC experimental processor [1], could have in a large scale data center that handles DSS applications. Secondly, we show the benefits of prefetching on the studied workloads when a shared on-chip memory is used as a prefetching buffer. Finally we present the power-performance efficiency analysis of the system for the different query implementations.

In order to achieve our goal we analyzed the performance of the basic database algorithms parallelized using the RCCE programming API provided for the Intel SCC. We have created different implementations for the data parallel sequencial scan, nested-loop and hash join query algorithms. For our implementations we also studied the impact on performance when we use a shared on-chip memory as a prefetching buffer. The algorithms are the basis for the execution of standard representatives DSS queries taken from the TPC-H benchmark suite [9]. We have selected real database workloads, which represent different database algorithms, and evaluate their performance on a real system, the Intel SCC experimental processor, using our proposed method of data prefetching. Our results show performance improvement by factors of *5x* to *10x* when data prefetching is used. Moreover, with a small performance loss we gain high benefits in power consumption resulting to high power-performance efficiency.

The rest of the paper is organized as follows: Section II describes the architecture of what we consider a scalable many-core architecture, Section III describes the database workloads selected and the different query algorithms implementations, Section IV shows the experimental setup and Section V analyzes the observed results. Section VI presents the relevant related work, and finally in Section VII we present the conclusions of this work.

1066-6192/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2013.14

II. Future Scalable Many-Core Architectures

The Intel SCC experimental processor is a 48-core 'concept vehicle' created by Intel Labs as a platform for many-core software research [1]. This processor consists of 24 dual-core tiles interconnected by a 2D-grid network. The tiles are organized in a 6x4 mesh with each tile containing:

- Two P54C cores with 16KB L1 and 256KB L2 cache dedicated to each core;
- A 16KB Message Passing Buffer (MPB)for storing messages to be sent to other cores (8KB per core);
- A Traffic Generator for testing the performance capabilities of the mesh network;
- A Mesh Interface Unit (MIU) to connect to the network;

The maximum main memory the system can support is 64GB. The 32-bit memory addresses of the core are translated into system addresses by the MIU through a lookup table (LUT) [7]. The systems' main memory is located outside the chip and four DDR3 Memory Controllers (MC) are used to move data on and off chip. In Figure 1 we present an overview of the SCC Memory Architecture and how the programmer can view the system and the core's memory through the RCCE message passing API [7]. The SCC supports both distributed and shared memory programming models and the systems' memory is configured and separated in four regions:

- Private off-chip memory: Each core's LUT is configured such that a specific region of the off-chip DRAM (equally divided to all) is only accessible by that core.
- Shared off-chip memory: This region of the off-chip DRAM is mapped by all LUTs and all cores have direct access to it through any MC. These data are not cached as the system does not provide cache-coherence;
- Shared on-chip memory: Also called Message Passing Buffer (MPB), this on-chip SRAM is cached in the L1 caches of the cores;
- L2 cache: used only by the private off-chip memory

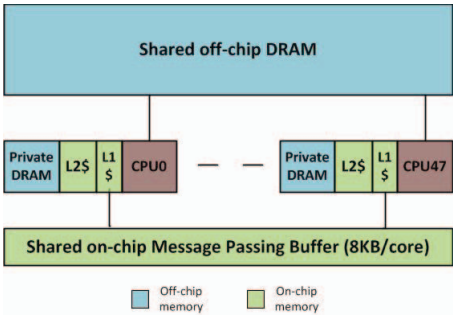

Figure 1. Intel SCC Memory Architecture as used by the programmer through the RCCE message passing API

In this work we study how we could use the shared on-chip memory as a prefetching buffer. More specifically we study how we could store data for all cores in this buffer in order to be used when needed. Therefore we minimize the off-chip memory accesses and eviction of useful data from L2 caches. We use the MPB in two ways: (i) as a whole, where each core writes only to its' own MPB and reads from all and (ii) as 48 different buffers, where each core writes and reads only to and from its' own MPB.

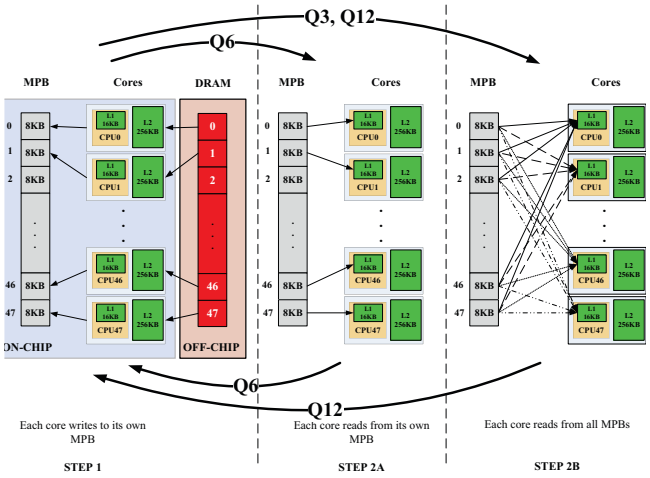

Figure 2. Description of data prefetching using the MPB.

III. Database Workloads And Implementations

For our work we focus on the execution of the basic database algorithms and their parallel implementation. As such, the queries analyzed in this work were implemented as programs that execute the operations determined by the queries and their results were validated. We have ported 3 different queries from TPC-H benchmark suite [9] of different complexity and demands. More specifically we ported Queries 3, 6 and 12 [10], from now on referenced as Q3, Q6 and Q12. We format the processing data in two ways in order to evaluate different implementations of the query algorithms. In the first format data are stored row-wise, *i.e.* all attributes of a particular record are stored in the same *row* of a two-dimensional array. Let's consider a table which is composed of records containing three attributes: attr1, attr2, and attr3. For each record a new row is created that stores all its attributes. The second format is hashing the data according to the key attribute on which tables are joined, creating discrete linked lists of records based on the tables' attribute on which join operation is performed.

A. Data-Parallel Sequential Scan (DPSS) and Parallel Join

Given the data layout as presented above, for this work, we use the simple sequential scan algorithm as to exploit both load balancing and locality while traversing the data. In our algorithm, all records are traversed and the records' attributes are checked against a certain condition. The condition may be a simple attribute comparison or a complex boolean function. We have mapped this operation to the Intel

SCC by implementing the condition to be tested and by sending to each core the input parameters which are the data streams that are used to evaluate the scan condition.

In this work we parallelize two join algorithms: nested loop and hash join. In order to perform the nested loop join, we compare each record of the outer loop with all the records of the inner loop in an iterative way. Parallelization is achieved by splitting the data of the outer loop to all cores. Regarding the hash join, the operation is performed on each lists' key attribute and if the condition is satisfied then we proceed for examining all its records. Parallelization of the hash join implementation of the queries algorithms is achieved by splitting the data in the first level of join by splitting the first Table among cores.

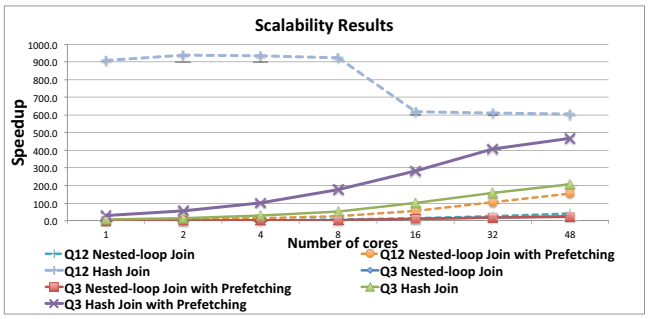

Figure 3. Normalized scalability of Q3 and Q12 query.

B. Data Prefetching

In order to use data prefetching and minimize the number of off-chip memory accesses we use the MPB of the system. To achieve this we implemented the previously mentioned algorithms using the MPB as a prefetching and storing buffer for most commonly used data, in order to optimize the performance. Data fetched or read from the MPB are controlled at user level specifying which data are to be used.

For the Q6 query which is a simple scan operation, we used the MPB of each core to fetch and store portions of data, 8KB long, from main memory. From Figure 2, in Step 1 each core copies data from the off-chip main memory to its' local MPB and in Step 2A it copies those data to its' L1 cache for calculation. These two steps are iterative steps until all data are processed. For the Q12 query we use the data prefetching scheme only for the nested-loop join implementation. The reason that we could not use the prefetching on the hash-join implementation is related to the organization of the data and that if we moved data from one table to the MPB then the hashing implementation would no longer exist. On the other hand if we move the hash table into the MPB we would still need to go off-chip in order to access the data directed by the hash table (currently located in the main memory). This would result in a larger overhead, minimal cache misses reduction and eviction of necessary data from L2 cache, since main memory is cached in L2.

For the nested-loop join algorithm of Q12 we use the MPB to store data from the inner-most table. From Figure 2, each core will copy data to its' local MPB (Step 1) and then each core will read and perform calculations on data from all the MPB chunks (Step 2B). Step 1 and 2B are iterative because data cannot fit in the total of 384KB of the systems' MPB. For the Q3 query we used the data prefetching on both nested-loop join and a hybrid implementation of hash-join and nested-loop join. For the nested-loop algorithm we store the inner-most table in the MPB. This decision was made since it is the table that is most frequently used which would mean a significant reduce of the last level cache misses and it also has the smallest size and in our case could fit for both input sizes in the MPB of the system. For the hybrid implementation, where we use both hash-join and nested-loop join, we perform the hash-join operation for the first join operation and for the records satisfying the query condition we perform a nested-loop join. In Figure 2 Steps 1 and 2B depict this implementation showing that Step 1 in Q3 will only be executed once and each core will copy a portion of data to its' local MPB. In the case that data do not fit in the MPB the algorithm will become iterative and continue fetching data from main memory just as it happens with Q12.

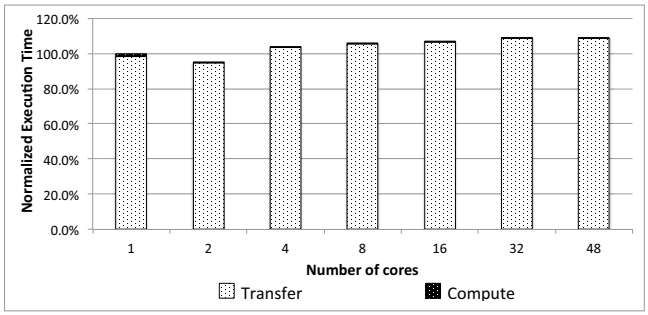

Figure 4. Data-parallel sequential scan (Q6) normalized execution time and breakdown.

IV. Experimental Setup

For the evaluation of our work we used the Intel SCC experimental processor, RockyLake version. The operating system used for the Intel SCC cores is the default Linux kernel provided by the RCCE SCC Kit 1.4.0 which was used for porting the applications as well. The workloads used for our work were selected from the TPC-H benchmark suite [9] using different input sizes, in order to study their performance scalability, generated using the *dbgen* tool. The input sizes 0.01 and 01 as well as the number of tables used for each query execution are: *(i)* Q3 use of 3 Tables of total size of 4.24MB and 93.56MB, *(ii)* Q6 use 1 Table of size 4.24MB and 93.56MB and *(iii)* Q12 use of 2 Tables of total size 3.74MB and 91.14MB. For calculating the power consumption of the chip we developed an application that

measures the power consumption using the same technique used by the SCC GUI performance meter. For our experiments we scaled the frequency of the cores to three different frequencies: (i) 100MHz, (ii) 266MHz and (iii) 533MHz. In order to calculate the power-performance efficiency of the system we have normalized the power and execution time product of the different implementations to the product of the single core execution of the base line scenario of each query executing on the 533MHz.

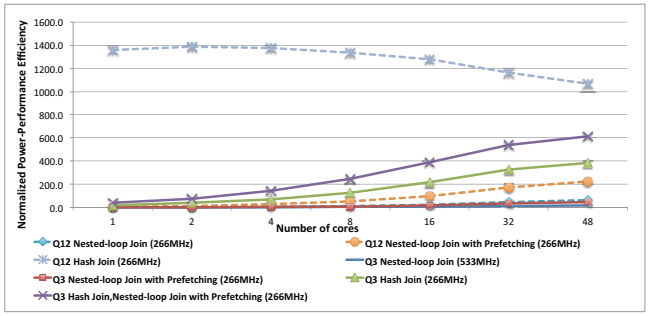

Figure 5. Normalized power-performance efficiency of Q3 and Q12 query.

V. Experimental Results

In our first analysis we compare the performance behaviour of the different queries algorithms executed on the Intel SCC experimental processor as described in Section III and study their scalability for the target architecture. Moreover, we study how the data prefetching scheme proposed can improve their performance. We present the results for 533MHz cores frequency due to the limited space available, even though 266MHz and 100MHz show the same behavior.

In Figure 3 we present the performance behavior and scalability of the different implementations of Q12 and Q3 query. The execution times are normalized to the corresponding execution of the nested-loop join implementation for a single core. Our results show that for Q12 query the hash join algorithm outperforms the nested-loop join implementation by a factor larger than *10x*. This performance difference can be explained from the way data are mapped and the efficiency of the hash join implementation. The second important observation is when we use the prefetching scheme for the nested-loop join algorithm we observe an improvement up to 5x. In particular we prefetch data to MPB and these data are stored without being influenced from evictions of L2 cache, therefore data reusage can be achieved. Q12 scalability shows that nested-loop join (with or without data prefetching) scales well whereas hash join implementation is stable. Hash join implementation algorithm complexity is limited and data transfers dominate the computation time, in a ratio of 1:20 computations over data transfers, resulting in no performance improvement as the number of cores increases. As for Q3 scalability the execution times are normalized to the corresponding nested-loop join implementation on a single core. Our first observation is that the nested-loop join implementation using data prefetching does not improve the performance compared to the implementation without data prefetching. This is explained from the fact that the table stored in the MPB can fit in the L2 cache of the core compared to the limited size of the MPB. This results in higher overheads on fetching data from the main memory to the MPB until all data are processed. In the case of scenarios with 4,8 and 48 cores, records are equally divided among cores, but the number of each cores records that satisfy the condition varies among cores in an average of 20%. In addition data chunks of 8KB are always read from MPB regardless if they satisfy or not the where clause condition. For the nested loop-join implementation data from main memory will only be read until the where condition is satisfied. The most important observation is the fact that the hybrid implementation of Q3 using first hash join and the data that satisfy the condition are then passed to a nested-loop join. This implementation shows important performance improvement as a result of gaining the most from both the hash join implementation and the prefetching scheme. In Figure 4 we present the normalized execution time of the data-parallel sequential scan implementation for Q6. The execution time is normalized to the execution time of the corresponding sequential scan on a single core. Due to the simplicity of the query algorithm and the fact that no data reuse is observed there is no performance improvement using the proposed prefetching scheme.

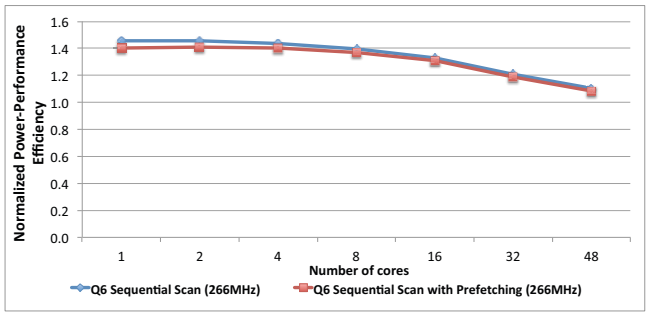

Figure 6. Normalized power-performance efficiency of Q6 query.

Another aspect of our work is to study the power-performance efficiency of the executed query implementations on the Intel SCC research processor for different core frequencies. As defined earlier the higher the results the better for the different scenarios investigated. We depict the most relevant results. In Figure 5 we present the power-performance efficiency results of Q3 implementations using as a baseline the nested-loop join implementation without data prefetching. Our results show that we can achieve high power-performance efficiency if we use our hybrid implementation which consists of hash and nested-loop join using prefetching even if we scale the frequency of cores to 266MHz. In Figure 5 we also present the power-performance

efficiency results of Q12 implementations using as a baseline the nested-loop join implementation without data prefetching. Our results depict that high power-performance efficiency, increased by a factor of up to *1400*, can be achieved using the hash join implementation of Q12 even if we scale cores' frequency to 266MHz. From our results we can conclude that up to 4 cores the power-performance efficiency of the selected implementation remains stable and afterwards reduces. Therefore we can achieve higher system execution efficiency if we consider executing multiple instances of the same implementation in fewer cores per instance. This applies for Q6 as well as depicted in Figure 6.

VI. Related Work

Different works study the performance evaluation and optimization of database workloads. Porting and evaluating the performance of such workloads in different systems is being of a large interest due to the fact that large data centrs are executing such workloads and performance and power efficiency is of their most interest. Data prefetching is a technique widely used for reducing the memory latencies by fetching data to the processors cache before it is requested in order to avoid misses that would otherwise occur. Data prefetching can be done automatically in hardware [6], where the prefetcher predicts the next data to be requested, or in software where data requests are explicitly placed by the programmer of the compiler in the code [3]. An example of data prefetching for accelerating the execution of database workloads is in [5]. Pre-execution [4] is an alternative prefetching mechanism using an extra thread called helper thread that executes portions of the code ahead of the execution threads. Trancoso et al [8] investigated the acceleration of decision support queries when executed on Cell/Be and GPUs using Rapidmind as a common platform.

In our work we are studying the performance and parallelism scalability of database queries algorithms using different techniques in order to optimize their performance like data prefetching. Moreover, power-performance efficiency analysis of database algorithms using different implementations is presented. Finally our experiments were executed on a representative many-core architecture platform, the Intel SCC, and we show how the selected workloads can be benefit in such architectures.

VII. Conclusions

Database applications are of the most demanding workloads. We have ported three different queries from the TPC-H benchmark suite on the Intel SCC experimental processor and we have study their performance behaviour when data prefetching is applied using the on-chip shared memory of the system. Our results were very encouraging for the use of data prefetching for future large-scale many-core processors even for demanding database applications. We observed a performance improvement by factors of *5x* to *10x* when data prefetching is used. Finally, from our analysis we show that with a small performance loss we achieve high benefits in power consumption resulting in high power-performance efficiency. Our results show that in the case of simple query algorithms scaling down the systems' cores frequency and reducing the number of cores (executing the respective implementation) can result in both high power-performance efficiency and throughput.

Acknowledgment

The authors would like to thank Intel Labs for lending the Intel SCC research processor.

References

[1] J. Howard et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Proceedings of the International Solid-State Circuits Conference*, Feb, 2010.

[2] N. Ioannou et al. Phase-based Application-driven Hierarchical Power Management on the Single-chip Cloud Computer. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 131–142, 2011.

[3] D. Koufaty and J. Torrellas. Compiler support for data forwarding in scalable shared-memory multiprocessors. In *Proceedings of the 1999 International Conference on Parallel Processing*, ICPP '99, pages 181–, Washington, DC, USA, 1999. IEEE Computer Society.

[4] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, pages 40–51, New York, NY, USA, 2001. ACM.

[5] K. Papadopoulos, K. Stavrou, and P. Trancoso. Helpercoredb: Exploiting multicore technology to improve database performance. *Parallel and Distributed Processing Symposium, International*, 0:1–11, 2008.

[6] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 42–53, New York, NY, USA, 2000. ACM.

[7] M. Timothy G. et al. The 48-core scc processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking and Storage Analysis*, April, 2007.

[8] P. Trancoso, D. Othonos, and A. Artemiou. Data parallel acceleration of decision support queries using cell/be and gpus. In *Proceedings of the 6th ACM conference on Computing frontiers*, CF '09, pages 117–126, New York, NY, USA, 2009. ACM.

[9] Transaction Processing Council. *TPC Benchmark H (Decision Support) Standard Specification Revision 2.6.1*. June 2006.

[10] P. Petrides, A. Diavastos and P. Trancoso. Exploring Decision Support Queries on Futured Many-Core Architectures. In *Proceedings of the Third Many-core Applications Research Community (MARC) Symposium*, pages 81-84, 2011.