An Energy and Performance Efficient DVFS Scheme for Irregular Parallel Divide-and-Conquer Algorithms on the Intel SCC

Yu-Liang Chou, Shaoshan Liu, Eui-Young Chung, and Jean-Luc Gaudiot, Fellow, IEEE

Abstract—The divide-and-conquer paradigm can be used to express many computationally significant problems, but an important subset of these applications is inherently load-imbalanced. Load balancing is a challenge for irregular parallel divide-and-conquer algorithms and efficiently solving these applications will be a key requirement for future many-core systems. To address the load imbalance issue, instead of attempting to dynamically balancing the workloads, this paper proposes an energy and performance efficient Dynamic Voltage and Frequency Scaling (DVFS) scheduling scheme, which takes into account the load imbalance behavior exhibited by these applications. More specifically, we examine the core of the divide-and-conquer paradigm and determine that the *base-case-reached* point where recursion stops is a suitable place in a divide-and-conquer paradigm to apply the proposed DVFS scheme. To evaluate the proposed scheme, we implement four representative irregular parallel divide-and-conquer algorithms, tree traversal, quicksort, finding primes, and n-queens puzzle, on the Intel Single-chip Cloud Computer (SCC) many-core machine. We demonstrate that, on average, the proposed scheme can improve performance by 41% while reducing energy consumption by 36% compared to the baseline running the whole computation with the default frequency configuration (400MHz).

Index Terms—DVFS, Divide-and-conquer, Intel SCC, Load Imbalance

1 Introduction

Divide-and-conquer is a well-known and important algorithmic paradigm suitable for execution on multi-cores, clusters and grids due to the fact that distinct subproblems can be solved independently and simultaneously. However, parallel divide-and-conquer applications are notorious for exhibiting load imbalance [1] but this kind of irregular parallel divide-and-conquer algorithm is an important subset of the class of parallel divide-and-conquer algorithms [2]. Many dynamic load balancing schemes for irregular parallel divide-and-conquer algorithms have been proposed [1], [3]. These schemes seek to dynamically schedule the workloads and to make the workloads distributed among cores as evenly as possible. However, as applications become more complex and adaptive, balancing the load becomes increasingly difficult [4].

On the other hand, the Dynamic Voltage and Frequency Scaling (DVFS) technique can help balance the execution time of the cores with different workloads by adjusting the operating frequency of the cores [4]. This paper aims at developing a DVFS scheme that can be used to achieve performance and energy efficiency with inherently load-imbalanced parallel divide-and-conquer applications. Since current DVFS techniques experience a large overhead when adjusting the voltage, the proposed scheme in this paper focuses on solely changing the frequency level. We have implemented the proposed scheme and evaluated the corresponding performance and power on the Intel Single-Chip Cloud Computer (SCC) [5], a tiled, many-core processor with DVFS.

- *Y-L. Chou and J-L. Gaudiot are with the University of California, Irvine, USA. E-mail: d943010010@gmail.com and gaudiot@uci.edu.*
- *S. Liu is with Microsoft. E-mail: shaoliu@microsoft.com.*
- *E-Y. Chung is with the Yonsei University, Korea. E-mail: eychung@yonsei.ac.kr.*

Date of publication 27 Mar. 2012; date of current version 10 Oct. 2014.
Digital Object Identifier 10.1109/L-CA.2013.1

2 Background

2.1 The Power Management of SCC

SCC is a many-core research chip developed by Intel® Labs. It was targeted for many-core research projects [6], [7]. One of the key features of SCC is the user controllable dynamic frequency and voltage regulation capability. RCCE [8] (pronounced "*rocky*"), a small library for many-core communication, provides some power management APIs for programmers to vary frequency and voltage of SCC. RCCE provides a low latency power management API (about 20 cycles [9]), *RCCE_set_frequency_divider*, for programmers to only change the frequency level. While the frequency can in principle be changed on a per tile basis, *RCCE_set_frequency_divider* currently sets the frequency of cores collectively within a 2X2 array of tiles. In this paper, we have modified the *RCCE_set_frequency_divider* to support shifting the frequency on an individual tile basis.

2.2 The Skeleton of A Generic Parallel Divide-and-conquer Paradigm

We use the parallel quicksort algorithm shown in Fig. 1, taken from Chen *et al*. [10], as an illustration of how the skeleton of a generic parallel divide-and-conquer paradigm is composed of three phases. In the first phase, the *Division Phase* (line 9 to 12), the original problem is recursively divided into subproblems and sent to correspond-

1556-6056 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

ing cores. The division phase ends when the total number of cores is committed. In the second phase, the *Computation Phase* (line 15), all cores begin to work in parallel on their own subproblems using a generic sequential procedure. In the third phase, the *Combination Phase* (line 13), the partial solutions are combined to obtain the solution of the original problem. Fig. 2(a) shows a representative execution pattern of the generic parallel divide-and-conquer paradigm using 4 cores.

2.3 The Benchmarks

N-Queens Puzzle: The goal is to find out how many ways there are to place N queens on an NxN chessboard so that no queen can take another. In our parallel implementation, the problem size is a 16 x 16 chessboard. A Core *i* is in charge of finding the subset 15 x 16 chessboard with all the possible solutions where the first queen have been placed in column *i* on the first row. Since the general algorithm to solve the N-Queens problem uses backtracking [11], the computation time of each core will be different even though we assign the same size of chessboard for each core. This is why load imbalance occurs.

Finding Primes: The goal is to find out how many prime numbers are in a range [a, b] ([1, 50000000] for us). This range is evenly divided into two parts which are sent to the corresponding cores and then divided recursively until all 16 cores are involved in the computation. The imbalance occurs because the execution time of the serial prime checking function varies not only with the input amount but also with the input value.

Quicksort: The goal is to sort an unsorted sequence. In our parallel implementation, the problem size is a 1MB unsorted sequence. This unsorted sequence is recursively divided and sent to the corresponding cores until all 16 cores are allocated. Since the division phase of quicksort randomly splits the unsorted sequence, the cores will receive different lengths of unsorted sub-sequences and thus load imbalance occurs.

Tree Traversal: The goal is to traverse a binary search tree and then calculate the sum of the values of all tree nodes. The problem size is a 10M-node binary search tree. We recursively remove the root of the tree and assign the subtrees to the corresponding cores until all 16 cores participate. Since we build the binary search tree by inserting a new node with a randomly generated value, the tree will be unbalanced and thus the size of the subtree assigned to each core is different, causing a certain load imbalance.

```
00  Input: Unsorted sequence A[1,N], 2^m processors are used
01  Output: Sorted sequence A[1,N]
02  Begin
03     para_quicksort(A,1, N, m, 0)
04  End
05
06  Procedure para_quicksort(A, i, j, m, id)
07  Begin
08     if m!=0 then
09        Cid calls patrition(A, i, j) to get pivot r
10        Cid sends A[r+1, j] to Cid+2^(m-1)
11        Cid calls para_quicksort(A, i, r-1, m-1, id)
12        Cid+2^(m-1) calls para_quicksort(A, r+1, j, m-1, id+2^(m-1))
13        Cid+2^(m-1) sends A[r+1, j] back to Cid
14     else
          Cid calls freq_select() to operate at proper Freq.
15        Cid calls quicksort(A, i, j)
          Cid reverts to default Freq.
16        return
17     end if
18  End
```

Figure 1. The parallel quicksort algorithm used in this paper and the proposed scheduling scheme (the shaded potions).

3 Proposed Approach

Fig. 2(b) shows the general idea of the proposed scheme. By increasing the frequency of the cores with a heavy workload, the critical execution path may actually be accelerated and thus the overall performance could be improved. At the same time, we can lower the frequency of the cores with light workloads to reduce the power consumption even if this implies a longer execution time, as long as the resulting increase in execution time for these cores in not so large to cause them to become an execution bottleneck. Moreover, the execution time of the cores with uneven workloads might not be perfectly balanced; therefore, some cores must be busy-waiting for messages from other cores, providing us with an opportunity for further energy saving without loss of performance by running the busy-waiting cores at the lowest frequency.

When to apply DVFS is one key point of our proposed scheme. Our scheme takes advantage of one fundamental trait of the parallel divide-and-conquer paradigm: workloads are recursively divided into smaller portions and sent to other cores until the base case, the point where recursion is ended, is reached, at which point the cores independently and simultaneously work on their own workloads. The beginning of the computation phase is a good point (*i.e.,* line 15 in Fig. 1) at which we can check the core workloads and then scale the core to the corresponding proper frequency.

How to select the proper frequencies for the cores is yet another key point. We select the top five frequency levels (800 MHz, 533MHz, 400MHz, 320MHz, and 266MHz) provided by SCC as the available frequencies the core can dynamically switch and we assume the default frequency to be 400MHz. The frequency selecting strategy of this research is shown in Fig. 3. Since the total time taken by a sequential procedure is approximately proportional to the total size of the workload, if we know the ratio between the execution time and the workload size, we can estimate the computation time of each core in computation phase at different frequency levels by multiplying the *actual workload* with the ratio. This ratio can be obtained

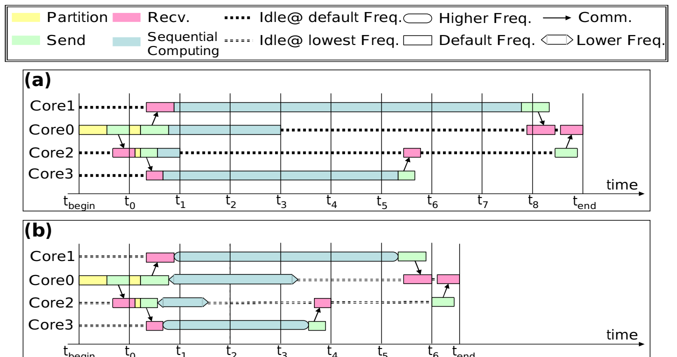


Figure 2. (a) A representative execution pattern of the generic parallel divide-and-conquer paradigm running on four cores. (b) The corresponding frequency scheduling idea proposed in this paper.

off-line or it can be derived by running a small piece of the sequential code when the program starts. The notation "exe_time@*num*" in Fig. 3. refers to the predicted execution time of a core handling the actual workload at *num MHz*.

Also, we need a reference point so that we can decide whether the core frequency should be scaled up or down. In general, we expect each core to receive an equally-sized subworkload, referred to as the *expected balanced workload*. We choose the computation time of the *expected balanced workload* running at the top frequency (800MHz), referred to as balanced_exe_time@800 in Fig. 3, as our reference point. Then, we strive to select the proper frequency for each core to make them complete their computations around this time. For example, if exe_time@400 is smaller than balanced_exe_time@800, which means there is some room for us to slow down the core frequency to save power, we could keep the frequency at 400MHz or lower it to 320MHz or 266MHz. If exe_time@320 is larger than balanced_exe_time@800, which means the execution time of the core scaled down to 320MHz might become the bottleneck, we keep the core running at the default frequency. Other selecting strategies in Fig. 3. can be similarly designed.

Defining the actual workload and the expected balanced workload of applications is another key point. Defining the actual workload and the expected balanced workload varies from benchmark to benchmark. We identify them for the four benchmarks as follows:

N-Queens Puzzle: since the execution of a backtracking algorithm can be illustrated using a recursion tree and the recursion tree is unique to each input problem size, we can consider the recursion tree as known information. The expected balanced workload can be defined as *total nodes of the recursion tree/16*. Also, the actual workload is the size of the sub-recursion tree assigned to the core.

Finding Primes: the imbalance of finding primes comes from the input value as well as input amount. Nevertheless, our parallel implementation distributes workloads evenly among the cores; therefore, the computation time is in proportion to the input values the cores get. Since the original range [1, 50000000] is evenly divided into 16 subranges for 16 cores, Core 0 will handle the [1, 3125000], Core 1 will handle the [3125001, 6250000], and so forth. Based on this workload assignment, the execution time will linearly increase from Core 0 to Core 15 and the execution time of Core 7 will be the average point; therefore,

```
00 Procedure freq_select ()
01 Begin
02   if exe_time@400 > balanced_exe_time@800 then
03     if exe_time@533 > balanced_exe_time@800 then
04       Let Cid operate at Freq. 800MHz
05     else
06       Let Cid operate at Freq. 533MHz
07   else
08     if exe_time@320 < balanced_exe_time@800 then
09       if exe_time@266 < balanced_exe_time@800 then
10         Let Cid operate at Freq. 266MHz
11       else
12         Let Cid operate at Freq. 320MHz
13     else
14       Let Cid operate at Freq. 400MHz
15 End
```

Figure 3. The frequency selecting strategy proposed in this paper

we can define the expected balanced workload as *[21875001, 25000000]*, the subrange that Core 7 received, and the actual workload is the subrange assigned to the core.

Quicksort: since the problem size is known at the beginning of program execution, we can easily define the expected balanced workload as *1MB/16*. Moreover, even if the sequence is split randomly in the division phase, we can easily know the actual workload of each core at the beginning of the computation phase.

Tree Traversal: just as in Quicksort, since the tree size is known at the beginning of program execution, we can define the expected balanced workload as *10M/16 nodes*. However, since each core only receives the pointer to the root of its own subtree, the subtree size (the actual workload) of each core cannot be known during the computation phase. We need an additional parameter in each tree node to record its corresponding subtree size during the generation of the tree.

How to identify the busy-waiting cores is yet another key point. It is easy to identify whether a core is busy-waiting or not in an RCCE program, which is an MPI-like SPMD programming style. In an RCCE program, a busy-waiting core will enter a busy-waiting state by calling *RCCE_wait_until*. *RCCE_wait_until* implements a busy-waiting loop (spinlock) continuously checking whether a flag in memory is set or not. We have modified the *RCCE_wait_until* to make the core calling this function run at the lowest frequency, 100MHz, to reduce power consumption. The core frequency will return to the default frequency after the core exits *RCCE_wait_until*.

4 Experimental Evaluation

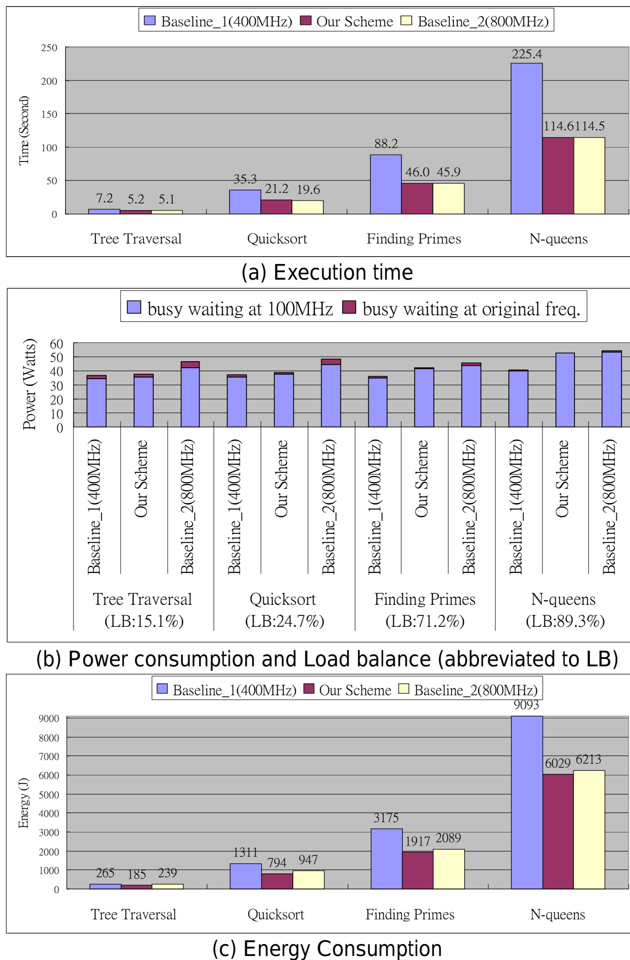
4.1 Experimental Environment

We conducted our experiments by running the four benchmarks on 16 cores of SCC. Cores run at 533 MHz, the mesh and the memory run at 800MHz, and the voltage for all settings is nominally 1.1V. We run the benchmarks at the default frequency (400MHz) for the whole execution as the baseline, and then at the top frequency (800MHz) for the whole execution as another baseline, and finally with the proposed scheme.

4.2 Experimental Results

Fig. 4(a) shows the execution time of the proposed scheme and the two baselines. Overall, the proposed scheme running with 16 cores can obtain 28%, 40%, 48%, and 49% (about 41% on average) of improvement in execution time on tree traversal, quicksort, finding primes, and n-queens puzzle, respectively, with respect to the baseline_1 (400MHz). In addition, it can achieve almost the same execution time (about 2% higher on average) as that of baseline_2 (800MHz).

Fig. 4(b) shows the power consumption for the proposed scheme and the two baselines running the busy-waiting periods at 100MHz and original frequencies. The original load balance values, defined by Etinski *et al.* [12], of four benchmarks without the proposed DVFS scheduling are also shown along with the benchmark names. We can observe that the more load balance the benchmark exhibits the less power reduction that busy-waiting at 100



(a) Execution time

(b) Power consumption and Load balance (abbreviated to LB)

(c) Energy Consumption

Figure 4. Results of the proposed scheme and the two baselines.

MHz can achieve. The proposed scheme running the busy-waiting periods at the lowest frequency can further yield 6%, 4%, 2%, and 1% (about 3% on average) of reduction in power on tree traversal, quicksort, finding primes, and n-queens puzzle, respectively, with respect to the proposed scheme running the busy-waiting periods at default frequency. On average, the proposed scheme running the busy-waiting periods at the lowest frequency can yield 9% and 14% of reduction in power consumption compared to the 800MHz baseline running the busy-waiting periods at the lowest and original frequencies, respectively.

Fig. 4(c) shows the overall energy consumption of the proposed scheme and the two baseline schemes. These values are calculated by multiplying the overall power consumption with the total execution time for each scheme. On average, the proposed scheme running with 16 cores obtains 36% of reduction in energy consumption with respect to the 400MHz baseline and 13% with respect to the 800MHz frequency baseline.

5 Conclusions

In this paper, we have presented an energy and performance efficient DVFS scheduling scheme to address the load imbalance issue often encountered in parallel divide-and-conquer algorithms. We have validated the primary idea on the Intel SCC many-core platform with four representative irregular divide-and-conquer algorithms. On average, the simulation results have shown that the proposed scheme is able to improve performance by 41% while reducing energy consumption by 36% compared to the baseline configuration running the whole computation with the default frequency (400MHz).

Acknowledgments

This work is partly supported by the US National Science Foundation under Grant No. CCF-1065448, by the National Research Foundation of Korea (NRF) under Grant No. 2012S1A2A1A01031420, by the Ministry of Education, Science and Technology under Grant No. 2012-047670, and by the National Science Council under Grant No. NSC 101-2917-I-564-079. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of these sponsors.

References

[1] M. Eriksson, C. Kessler, and M. Chalabine," Load Balancing of Irregular Parallel Divide-and-Conquer Algorithms in Group-SPMD Programming Environments" , *Proc. 8th Workshop on Parallel Systems and Algorithms*, pp. 313–322, Mar. 2006.

[2] J. C. Hardwick," Practical Parallel Divide-and-Conquer Algorithms," PhD thesis, Carnegie-Mellon University, 1997.

[3] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, T. Kielmann, and H. E. Bal," Adaptive Load Balancing for Divide-and-Conquer Grid Applications," *Journal of Supercomputing*, 2004.

[4] V.W. Freeh, N. Kappiah, D.K. Lowenthal, and T.K. Bletsch, "Just-in-Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs," *Journal of Parallel and Distributed Computing*, vol. 68, pp. 1175-1185, 2008.

[5] Intel Corp., "The SCC Platform Overview," http://communities.intel.com/docs/DOC-5042

[6] J. Tang, S. Liu, Z. Gu, C. Liu, and J.-L. Gaudiot," Acceleration of XML Parsing Through Prefetching," *IEEE Transactions on Computers*, in press.

[7] J. Tang, P. Thanarungroj, C. Liu, S. Liu, Z. Gu, and J.-L. Gaudiot," Pinned OS/Services: A Case Study of XML Parsing on Intel SCC," *Journal of Computer Science and Technology*, in press.

[8] Intel Corp.," RCCE: a small library for many-core communication," http://communities.intel.com/docs/DOC-5628

[9] T G. Mattson, R. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, "The 48-core SCC Processor: The Programmer's View," *Proc. ACM/IEEE Conference on Supercomputing*, Nov. 2010.

[10] G.-L.Chen, *Implementation of parallel algorithms*, Beijing: Higher Education Press, 2004.

[11] V. Rao and V. Kumar, "On the efficiency of parallel backtracking," *IEEE Transactions on Parallel and Distributed Systems*, vol.4, pp. 427-437, 1993.

[12] M. Etinski, J. Corbalan, J. Labarta, and M. Valero, "Understanding the future of energy-performance trade-off via DVFS in HPC environments," *Journal of Parallel and Distributed Computing*, vol. 72, pp. 579-590, 2012.