Performance Balancing: Software-based On-chip Memory Management for Effective CMP Executions

Naoto Fukumoto, Kenichi Imazato, Koji Inoue, Kazuaki Murakami
Department of Advanced Information Technology, Kyushu University
744 Motooka, Nishi-ku, Fukuoka City, Japan
{fukumoto,imazato}@c.csce.kyushu-u.ac.jp, {inoue,murakami}@ait.kyushu-u.ac.jp

ABSTRACT
This paper proposes the concept of *performance balancing*, and reports its performance impact on a Chip multiprocessor (CMP). Integrating multiple processor cores into a single chip, or CMPs, can achieve higher peak performance by means of exploiting thread level parallelism. However, the off-chip memory bandwidth which does not scale with the number of cores tends to limit the potential of CMPs. To solve this issue, the technique proposed in this paper attempts to make a good balance between computation and memorization. Unlike conventional parallel executions, this approach exploits some cores to improve the memory performance. These cores devote the on-chip memory hardware resources to the remaining cores executing the parallelized threads. In our evaluation, it is observed that our approach can achieve 31% of performance improvement compared to a conventional parallel execution model in the specified program.

Categories and Subject Descriptors
D1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming — *parallel programming*

General Terms
Performance, Management

Keywords
Chip MultiProcessors, Scratchpad Memory, Parallel Execution, Memory-wall Problem

1. INTRODUCTION
Integrating multiple processor cores into a single chip, or CMP, is a de-facto standard for high performance computing area. By exploiting thread level parallelism (or TLP), CMPs achieve high peak performance. Industry trends show that the number of cores in a chip will increase every process generation, so that improving efficiency of parallel executions becomes more important.

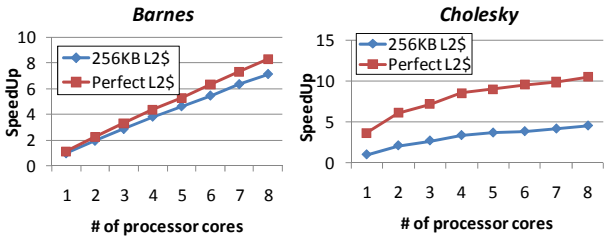

Figure 1: Impact of memory performance

Although the on-chip parallel processing is a promising way, CMPs cannot always achieve expected peak performance, especially for memory intensive applications. This is because the memory-wall problem becomes more serious in CMPs. In state-of-the art CMP designs, the off-chip memory access latency is almost two orders of magnitude longer than one operation time (or clock cycle time) of processor cores. Moreover, due to I/O pin limitations, off-chip memory bandwidth does not scale with the number of cores. Figure 1 shows the performance improvement achieved by increasing the number of cores on a CMP. Here, we assume CMP models which have 256KB realistic private L2 caches or ideal perfect L2 caches. The two benchmark programs from SPLASH-2 [14] were executed by using M5 cycle accurate CMP simulator [2]. For Barnes, the performance improvement is in almost proportional to the number of cores used. However, for Cholesky, it is clear that the CMP performance is limited by the inefficient memory system. As in the examples given above, in memory intensive programs, a conventional approach which attempts to exploit all cores to execute parallelized threads does not achieve high performance.

To solve the memory performance issue on CMPs, in this paper, we propose a parallel execution technique called *performance balancing*. In conventional approaches, all cores execute parallelized application threads to exploit TLP. On the other hand, our approach attempts to exploit some cores not only for executing parallelized threads but for improving memory performance. Our approach aims to improve the total performance by means of making a good balance between computing and memory performance according to application characteristics. Adjusting the ratio of the two kinds of cores can achieve good performance even in compute intensive programs and memory intensive programs. The contributions of this paper are as follows.

- The concept of performance balancing to improve the CMP performance is proposed. We focus on software-controllable

on-chip memory architecture such as Cell Broadband Engine (Cell/B.E.) [9], and show a performance model to discuss the efficiency of our approach.

- We propose a software technique to realize the performance balancing on the targeted CMP. To improve the memory performance, we dare to throttle TLP, i.e. the parts of cores execute parallelized application threads. The other remaining cores release their on-chip memory to the executing cores. Our technique appropriately allocates these two types of cores with the aims of maximum total performance.
- We implement five benchmark programs to evaluate the efficiency of the proposed approach. As the results, it is observed that using five cores for execution and two cores for memory assist achieves 31% of performance improvement compared to the conventional parallel execution in the specified program.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 describes our proposed technique called performance balancing. Section 4 shows the evaluation results, and Section 5 concludes the paper.

2. RELATED WORK

There are many techniques to improve memory performance in CMPs. These techniques fall into the two main categories: hiding memory access time and reducing the number of off-chip memory accesses. Data prefetching technique is categorized into the first type and has been proposed by several researchers [3][7]. Cantin et al. [3] proposed *stealth prefetching* which reduces the number of invalidation of prefetched data for cache coherency. To obtain the information on other cores, prefetcher does not issue prefetch requests for the data to be held in other cores. This leads to reducing the number of extra broadcast originated from data prefetching. Ganusov et al. [7] proposed helper threads for improving memory performance. Executing the helper thread which emulates hardware prefetching can improve memory performance for another core. Their approach appears similar to our approach in improving memory performance using processor cores. While their approach improves the single-thread application performance using an idle core to execute helper thread, our approach focuses on improving parallel program performance using core to assist other cores instead of executing parallelized thread.

In another type of category, several approaches which reduce the number of off-chip memory accesses have been proposed. Cooperative cache [4] reduces cache misses with copying data to remote cache if cache memory has the data exclusively. This approach is needed to modify coherency protocol. To be effectively allocated data into, software-managed local memory (call scratchpad memory or SPM in this paper) can achieve high performance and low power compared to cache memory. However, at the same time, this allocation requires extra effort to coding source code. To solve this issue, many researchers have proposed allocation techniques which allocate effective data automatically into SPMs at compile time [1] [11][13]. In these techniques, the number of SPM accesses is maximized under certain conditions. These techniques can be categorized into two types by the difference of conditions: dynamic allocation [13] and static
allocation [1][11]. In static allocation, the data in SPM are not replaced during program execution. Static allocation does not work well if the total size of the data set used in program execution is large. This is because static allocation only allocates the data set to SPM up to the size of SPM in program execution. Dynamic allocation replaces the data of SPM in program execution. Dynamic allocation can allocate the large capacity of data to SPM comparing static allocation. Thus, when the total size of data set used in program execution is large, dynamic allocation can effectively increase the rate of access to SPM.

A few prior studies have already started to manage processor cores in response to changing the situation on CMPs. Increasing the number of processor cores can degrades parallel processing performance due to synchronization overhead, limitation of off-chip memory bandwidth, and so on. To solve this phenomenon, Suleman et al. [12] and Curtis-Maury et al. [6] proposed concurrency throttling which changes the number of execution cores considering the characteristics of program. In these approaches, the best number of execution cores is estimated using profile information at runtime, and then program is executed with the number of cores. While their approaches improve performance only if increasing the number of execution cores degrades performance, our approach can improve performance in this case due to improving memory performance to use remaining cores.

3. PERFORMANCE BALANCING

3.1 Processor Model

Figure 2 shows the CMP model targeted in this paper. Each core has SPM and these SPMs are connected an on-chip interconnection such as ring or shared bus. Memory accesses operate by DMA transfer except for access to own SPM. The computation at the processor cores and the data transfer between the off-chip memory and SPM can be overlapped by using DMA controller.

3.2 Concept of Performance Balancing

For memory intensive programs, the conventional parallel execution method does not achieve high performance. In Figure 1, the six-core CMP with the perfect cache achieves higher performance than the eight-core CMP with 256KB private L2 caches. In other words, for the eight-core CMP, there is a possibility that best performance can be achieved by using the six cores for code execution and the two cores for memory performance improvement.

To improve the total performance of CMPs, we propose the concept of performance balancing. This approach attempts to exploit some cores for improving memory performance as shown in Figure 2. Here, we define the following terminology to help understanding our approach.

- Main core: It executes parallelized application threads, and can exploit the SPMs provided by the helper cores as a new level of on-chip memory hierarchy.
- Helper core: It does not execute application threads, but release own SPM to main cores.

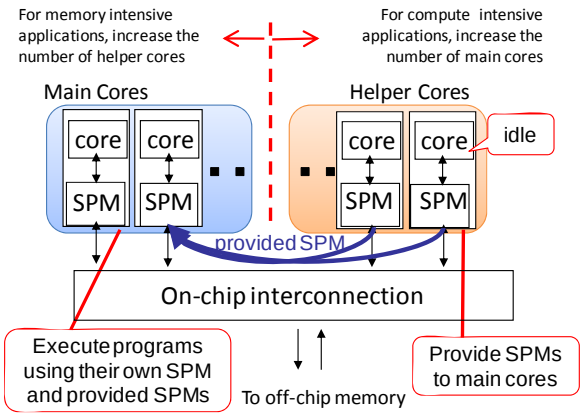

Figure 2: Concept of Performance Balancing

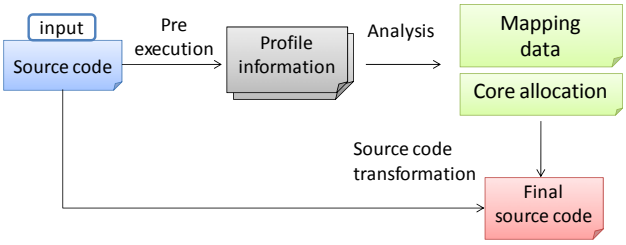

Figure 3: Workflow of Applying Performance Balancing

The total number of cores implemented in a chip is a constant. Thus, increasing the number of helper cores means throttling TLP, resulting in lower computation performance. At the same time, since the main cores can exploit the SPMs provided by the helper cores, its memory performance is improved. The number of main and helper cores largely impacts on the total performance. Our approach attempts to decide the appropriate number of cores to be used as main and helper cores in order to maximize the CMP total performance.

Figure 3 shows the workflow to which performance balancing is applied. First, our approach samples profile information to execute a target program for estimating the program behavior. Next, analyzing the profile information, we estimate an optimal data allocation of helper cores' SPM and the best number of main/helper cores. Finally, using the information, the original source code is transformed to objective source code.

3.3 Prediction of Optimal Core Allocation

3.3.1 Performance Modeling

In this section, we model the impact of our approach on CMP performance. Since this performance model is used for estimating the appropriate number of main and helper cores, we focus on the effects of changing the number of main and helper cores.

Let $T_{exe}(n)$ be the execution time except for the time of DMA transfers and $T_{mem}(m,n)$ be the time of DMA transfer when there are n main cores and m helper cores. Then, the execution time of a main core $T(m,n)$ can be approximated by the following equations.

$$T(m,n) = T_{exe}(n) + T_{mem}(m,n) \quad (1)$$

$$T_{exe}(n) = \left\{ \frac{F}{n} + (1-F) \right\} \times T_{exe}(1) \quad (2)$$

$$T_{mem}(m,n) = AC(n) \times AMAT(m,n) \quad (3)$$

$$AMAT(m,n) = HR_{SPMR}(m) \times AT_{SPM} + (1 - HR_{SPMR}(m)) \times AT_{main} \quad (4)$$

Let F be the rate of the whole execution time to execute the parallel part of the program with one main core. $T_{mem}(m,n)$ represents the product of the number of DMA transfers $AC(n)$ and average DMA transfer time $AMAT$. AT_{SPM} is the time consumed for accessing the SPM in a helper core from a main core. AT_{main} also represents access time to main memory. $HR_{SPMR}(m)$ denotes the rate that the data referenced by a main core is provided by a helper core, i.e. that data is provided by an SPM-to-SPM data transfer. $AC(n)$ can be expressed by the following equation.

$$AC(n) = AC(1) \times \left(\frac{F}{n} + 1 - F \right) \quad (5)$$

Based on these equations, we discuss the impact of our approach on the CMP performance. We firstly focus on how the number of main cores impacts on CMP performance. If the number of main cores n is increased with that of helper cores m fixed, $T_{mem}(m,n)$ and $T_{exe}(n)$ are decreased. This is caused by increasing n reduces the load of computation and the number of DMA transfer per a main core (see the Equation (2) and (5)). On the other hand, increasing the value of m with fixed that of n improves $T_{mem}(m,n)$ due to enlarging $HR_{SPMR}(m)$ in Equation (4). Increasing m or n causes the performance improvement, but the sum of m and n cannot exceed the all number of cores in the chip. Consequently, the value of m and n largely affects CMP performance and the optimal number of main and helper cores which achieves the maximum performance is depend on F , $AC(n)$, and $T_{exe}(1)$ which are varied with application program characteristic. Thus, it is important to specify the appropriate value of m and n with respect to each program. This way is explained at Section 3.3.2.

$HR_{SPMR}(m)$ is also changed by how effective data is loaded into the SPM of helper cores. Loading more data requested by main cores improves $HR_{SPMR}(m)$. We explain how to determine the data stored in SPM of helper cores in Section 3.4.

3.3.2 Implementation of Predicting Optimal Core Allocation

In our approach, program should be executed with the appropriate number of main/helper cores. The appropriate number of cores varies with characteristics of program, input set and machine configuration. Thus, we propose profile-based prediction technique for optimal core allocation.

First, our approach samples profile information to execute a program for estimating the program behavior. Next, we calculate execution time for each core combination based on the performance modeling. The optimal core allocation is estimated by choosing the best combination of cores to achieve best performance.

To calculate execution time with respect to each combination of cores, our approach uses the equations in Section 3.3.1. All terms except for access time to SPM and main memory are estimated using pre-execution information with one main core and no helper core. To measure F , *time function* is inserted at the begin and end of the parallel part in source code. F is calculated by the difference of this time and whole execution time. Identifying the parallel part of the source code and inserting the time function are done by hand. $AC(1)$ can be measured by performance counter. $HR_{SPMR}(m)$ can be estimated by analyzing the allocation list of data into the SPM of helper cores and DMA transfer trace. In other words, we can distinguish where each DMA transfer accesses by means of comparing the allocation data list to DMA transfer trace. The allocation data list for each number of helper cores is previously determined and the data targeted by DMA transfer are previously obtained. However, this estimation technique cannot obtain completely-accurate value of $HR_{SPMR}(m)$ because our technique assumes that varying the number of main cores does not make much difference on the DMA transfer information. Our technique ignores the effect produced by changes of the number of main cores, such as increasing the main cores enlarges the number of accesses to shared variables. $T_{exe}(1)$ can be obtained to assign $(m,n)=(0,1)$ to Equation (1) since $T(0,1)$ and $T_{mem}(0,1)$ can be calculated.

3.4 Data Mapping to SPM of Helper Cores

In our approach, main cores use SPM of helper cores as the second hierarchical memory. If helper cores have the data demanded by the main core, the main core can obtain the data by high-speed SPM-to-SPM data transfer without accessing main memory. Consequently, loading more effective data in helper cores corresponds to more memory performance improvement in main cores. This section describes how to determine the data loaded into SPM of helper cores.

Our approach decides the data which is loaded into SPM in helper cores for the purpose that minimizes the number of main memory accesses. Helper cores do not reload other data during program execution. This kind of technique has been proposed in area of SPMs as static allocation [1][11]. To allocate optimal data into SPM, static allocation minimizes the number of main memory accesses in a single core processor on the condition not to reload the data. We apply the static allocation technique to allocate the data of helper cores' SPM. Static allocation is applied as follows. First, static allocation obtains DMA transfer trace (of only data) by pre-executing with a main core. Next, the number of memory accesses is counted to each interval of address (224KB in this paper) based on the trace. Finally, the data are loaded into on-chip memory in decreasing order of counts of main memory accesses.

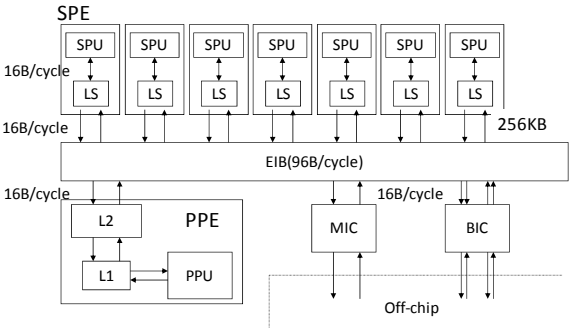

Figure 4: Cell/B.E. Processor Architecture

Table 1: Details of Executed workloads

Benchmark	Problem description	Input set
HIMENO	Poisson's equation	17x17x33(SSS), 33x33x65(SS)
SUSAN	Image processing	Large input
FFT	Fast fourier transform	229376 Point
LU	LU decomposition	512x512, 1024x1024
MATRIX_MUL	Matrix Multiplication	256x256 512x512

Table 2: Access Time to Each Memory Module

Memory component	Access time [CC]
Remote LS (AT_{SPM})	106+transfer size/8
Main memory (AT_{main})	300+transfer size/4

Note that our approach allocates statically data to SPMs of helper cores. Helper cores do not reload data dynamically even though this could be useful for long programs using the large amount of data. This is because frequently data reloading can cause performance degradation to increase synchronization overhead. Main cores must obtain the demanded data from helper cores or main memory to guarantee the expected behavior of program. If helper cores reload data into own SPM not to synchronize main cores, main core can obtain unexpected data. Using the unexpected data brings wrong behavior of program. Consequently, helper cores must synchronize main cores every time they replace the data. Also note that minimizing the number of main memory accesses does not correspond to maximizing the memory performance of main cores. Most processors to have SPMs can execute computation and DMA transfer at the same time. Processor core can continue computation during accessing to main memory. Consequently, there is no guarantee that stall time of cores is minimized in the case of minimizing the number of main memory accesses. The extensions to dynamic reloading and minimizing stall time are part of our future work.

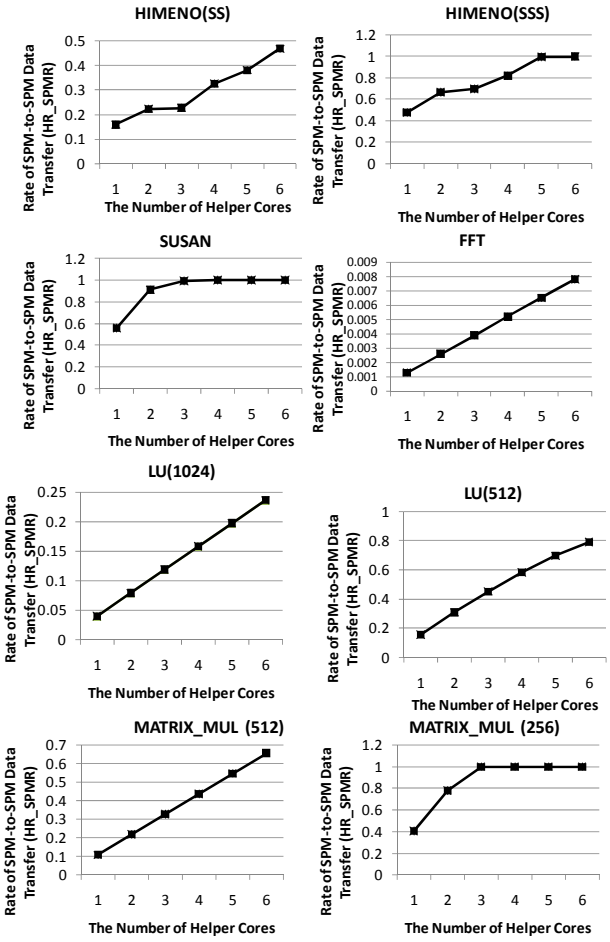

Figure 5: Rate of SPM-to-SPM data transfer (Read)

4. EVALUATION

4.1 Experimental Setup

In order to clarify the efficiency of the proposed approach, we have evaluated the performance of a Cell B.E. processor. Figure 4 shows the microarchitecture of Cell/B.E. processor. Cell/B.E. has one PPE (PowerPC Processor Element) and eight SPEs (Synergistic Processor Elements) [9]. PPE which is general purpose processor assigns threads to SPEs and controls standard I/O. Each SPE has 256KB controllable on-chip memory call Local Store (LS). We apply our approach to SPEs without PPE. Since only seven SPEs are available on the Cell/B.E. chip, the maximum number of main cores is seven. Cell/B.E. accesses main memory or remote LSs with DMA transfer. The access time of DMA write does not vary where to access. Consequently, we allocate the data to minimize the read requests to main memory. Table 2 shows read access time to each module for performance modeling. We decided these values based on the reports [5], [10].

For the evaluation, we used five multi-threaded applications from scientific domain and embedded domain. Since Cell/B.E. processors are used for CPU accelerator in super computers and embedded computer systems such as game console, we chose benchmark programs from multiple domains. These programs are modified for executing in Cell/B.E and are tuned for memory performance improvement. In *HIMENO* benchmark program [15], there is a three dimension array, and the main function includes a fourfold loop. We parallelized at the second level loop and did not adapt special code tuning except for double buffering. *Susan* in Mibench [8] has four steps of program, which are (1) *initialize*, (2) *edge reinforcement*, (3) *edge correction*, and (4) *stacking raw image*. We parallelized the part of (2) and (4). We do not arrange algorithm in *LU* in SPLASH-2 [14] since SPLASH-2 is parallelized benchmark program suit. We obtained FFT and MATRIX_MUL source codes from sample codes of IBM Cell SDK 3.1. These codes are optimized for Cell/B.E. processors. Source code of FFT is modified to be able to change the number of SPEs other than power of two SPEs. The input set of benchmarks is shown in Table 1. We used -O3 as a compile option, and took the average execution time of 10 times.

We measure the real execution time of the following evaluation models.

- ***CONV***: This is a conventional model which uses all of the cores to execute parallelized threads, i.e. the number of main cores is equal to seven.
- ***PB-IDEAL***: This model is applied performance balancing with the optimal number of main/helper cores. The optimal core allocation is estimated by pre-execution of all combination of main/helper cores with the same input as the evaluated input.
- ***PB-PREDICT***: This model is applied performance balancing with the predicted number of main/helper cores. The predicted number of main/helper cores is calculated by the method explained in Section 3.3. We use the same input size as the evaluated input size.

4.2 Reduction of Main Memory Accesses

Figure 5 illustrates the proportion of SPM-to-SPM data transfer counts to whole DMA transfer counts for each benchmark programs. The x-axis represents the number of helper cores and the other remaining cores are used as that of main cores. In this graph, the proportion of SPM-to-SPM data transfer is higher in increasing the number of helper cores in all programs. This is because increasing the number of helper cores enlarges the capacity of on-chip memory available to main cores. Increasing the data capacity to be used as second hierarchical memory enlarges the rate of SPM-to-SPM transfer. Another important aspect of Figure 5 is that the rate of SPM-to-SPM transfer even in the same program and the same number of helper cores is quite different among the input size. This is because executing program with large input set increases the data capacity to be used during program execution. Our approach does not replace the data of helper cores after the data allocation for helper cores is determined. Consequently, the amount of data to be used in program execution largely affects the proportion of SPM-to-SPM data transfer rate.

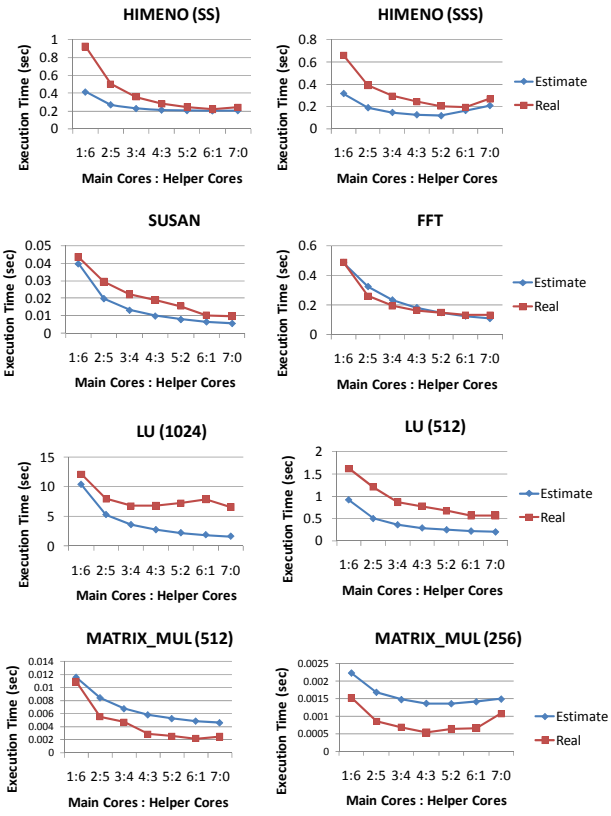

Figure 6: Execution Time Varying the Number of Cores

4.3 Accuracy of Core Allocation

In this section, we evaluate the accuracy of our prediction technique which estimates the optimal number of main and helper cores. Figure 6 shows the execution time applying *performance balancing* as the number of main/helper cores varies. X-axis represents the number of main and helper cores. For example, *1:6* means one main core and six helper cores. *Estimate* of graph legends represents the execution time estimated by the performance modeling (explained in Section 3.3). *Real* represents the actual measurement execution time with Cell/B.E. Our predict technique considers the number of main/helper cores with the minimum execution time in *Estimate* as the optimal core allocation. To evaluate our prediction technique, we compare the number of main/helper cores with minimum execution time in *Estimate* to the number of main/helper cores with minimum execution time in *Real*.

With all programs in Figure 6, the number of main/helper cores which achieves the minimum execution time in *Estimate* approximately corresponds with the optimal number of main/helper cores in *Real*. In *HIMENO (SS)*, *Susan*, *FFT*, and *LU (1024)*, our approach can predict the best number of cores which achieves the highest performance. In the other programs, our approach estimates the second best number of cores.

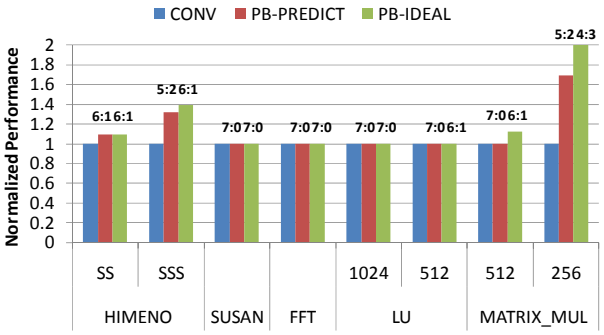

Figure 7: Performance Normalized with Conventional Approach

4.4 Performance

Figure 7 illustrates the performance of the three evaluation models normalized by the *CONV* model. X-axis is the name of benchmark programs and input size. The numerical character above the bars represents the ratio of the number of main cores/helper cores. For example, 6:1 means six main cores and one helper core. In *HIMENO (SSS)*, *HIMENO (SS)*, and *MATRIX_MUL (256)*, performance balancing in optimal core allocation improves performance by 39%, 9%, and 100%, respectively. In these programs, adding helper cores achieves significant memory performance improvement (see Figure 5), and reducing main cores has small negative effect on computing performance due to low thread level parallelism. Consequently, memory performance improvement exceeds computation performance degradation originated form decrease in the number of main cores. In *PB-PREDICT*, our approach improves 31%, 9% and 69% in *HIMENO (SSS)*, *HIMENO (SS)* and *MATRIX_MUL (256)*. This result shows that our prediction technique works well. In *MATRIX_MUL (512)*, allocating optimal number of main and helper cores achieves 13% performance improvement.

In the other programs, our approach does not improve performance even with the best number of main and helper cores. This result demonstrates that conventional approach which executes programs with all cores is the best case. In these programs, performance balancing does not improve total performance. This is mainly because adding helper cores cannot achieve higher memory performance than the degradation of computation performance. In *FFT* and *LU (1024)* which use large amount of data, the effect of reducing the number of main memory accesses is small observed in Figure 5. To achieve high performance in these programs, it is necessary to effectively load important data into helper cores' LS such as the dynamic allocation technique. In *SUSAN*, on the other hand, even though the largely number of main memory accesses replaces SPM-to-SPM transfer, our approach does not improve the total performance. This is why memory performance is not performance bottleneck and performance scales in proportional to the number of cores. Even in these programs, our proposed technique does not degrade the performance to predict the best allocation of cores.

5. CONCLUSIONS

In many-core era, it is important to bring out potential performance in processor chip to operate processor cores concertedly. We have researched techniques of orchestrating processor cores in multicore systems for performance, energy, reliability, and so on. As part of this project, this paper proposes the parallel execution technique called performance balancing. This approach exploits processor cores not only for executing parallelized threads, but also for improving memory performance considering the balance of memory and computation performance. Experimental results show up to 31% performance improvement to predict the appropriate number of main/helper cores.

For future work, we develop more effective method for data mapping to SPMs of helper cores such as replacing mapping data during program execution. We also evaluate energy on our approach adding benchmark programs.

6. ACKNOWLDEGMENT

We would like to express our thanks to Semiconductor Technology Academic Research Center (STARC) for their support. This research was supported in part by New Energy and Industrial Technology Development Organization and the Grant-in-Aid for Young Scientists (A), 21680005. We thank all members of the System LSI laboratory of Kyushu University for discussing at technical meetings.

7. REFERENCE

[1] O. Avissar, R. Barua, and D. Stewart, An Optimal Memory Allocation Scheme for Scratchpad-based Embedded Systems. ACM Transactions on Embedded Computing Systems, pp.6-26, 2002.

[2] N. L. Binkert, E. G. Hallnor and S. K. Reinhardt. Network Oriented Full-System Simulation using M5. 6th Workshop on Computer Architecture Evaluation using Commercial Workloads, 2003.

[3] J.F. Cantin, M.H. Lipasti, and J.E. Smith. Stealth prefetching. Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 274-282, 2006.

[4] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. The 33rd Intl. Symposium on Computer Architecture, pp 264-276, 2006.

[5] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation performance view. IBM Journal of Research and Development, 51(5): pp.559-572, 2007.

[6] M. Curtis-Maury, K. Singh, S. A. McKee, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, M. Schulz. Identifying Energy-Efficient Concurrency Levels using Machine Learning. The Intl. Workshop on Green Computing, pp.488-495, 2007.

[7] I. Ganusov and M. Burtscher. Efficient Emulation of Hardware Prefetchers via Event-Driven Helper Threading. The 15th Intl. Conference on Parallel Architectures and Compilation Techniques, pp. 144-153, 2006.

[8] 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. The IEEE 4th Annual Workshop on Workload Characterization, pp.3-14, 2001.

[9] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer and D. Shippy. Introduction to the Cell multiprocessor. IBM journal of Research and Development, 49(4-5), 2005.

[10] M. Kistler, M. Perrone, F. Petrini. Cell Multiprocessor Communication Network: Built for Speed. Micro, IEEE, 26(3), pp. 10-23, 2006.

[11] S. Steinke, L. Wehmeyer, B. S. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. Proceedings of the conference on Design, automation and test in Europe, pp.409-415, 2002

[12] M. A. Suleman, M. K. Qureshi and Y. N. Patt. Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-threaded Workloads on CMPs. 13th Intl. Conference on Architectural Support for Programming Languages and Operating Systems, pp.277-286, 2008

[13] S. Udayakumaran, A. Dominguez, and R. Barua, Dynamic allocation for scratchpad memory using compile-time decisions. ACM Transactions on Embedded Computing Systems, pp.472-511, 2006.

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. The Intl. Symposium on Computer Architecture, pp.24-36, 1995.

[15] Himeno Benchmark: http://accc.riken.jp/HPC/HimenoBMT/index_e.html