This paper is author's private version of the paper published as follows:
Proc. 3rd Int'l Conf. on Networking and Computing (*ICNC'12*), pp.316-320
Copyright (C) 2012 IEEE
DOI: 10.1109/ICNC.2012.59

An Efficient Thread Recombining at Program Phase Changes

Kosuke SOBUE*, Tomoaki TSUMURA* and Hiroshi MATSUO*

*Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp

***Abstract*—Chip-multiprocessors now have become in wide use. For efficiently using the resources in chip-multiprocessors, programmers need to consider processor specifications and load balancing, but it is difficult for them. To address this problem, Thread Tailor has been proposed. Thread Tailor determines the number of threads based on processor specifications before execution, balances their loads based on the results of profiling and combines threads. However, Thread Tailor determines which threads should be combined based on only the number of executed cycles of each thread. Hence, the programs, whose threads change their computation loads according to program phases, may slow down with Thread Tailor. To solve this problem, we propose a method which dynamically recombines threads according to program phases for balancing the loads. The results of the experiment with SPLASH-2 benchmark suite and PARSEC benchmark suite show that the new method improves the execution time 6.0% in maximum.**

***Index Terms*—thread scheduling, multi-threading, multi-core processors**

I. Introduction

As electric power consumption and calorific power are increasing, and semiconductor devices keep downscaling, it becomes difficult to raise clock frequencies of microprocessors. In response to this distress, multi-core processors now attract a great deal of attention. A multi-core processor has several independent cores on a chip. On multi-core processors, multiple threads run in parallel for speed-up. For efficiently using the resources of multi-core processors, developers must write parallel programs and specify the number of threads. However, it is difficult for developers to determine the adequate number of threads with taking the resources into account.

To solve this problem, **Thread Tailor**[1] has been proposed. Thread Tailor adjusts the number of threads by combining threads to optimize system efficiency. The information such as the processor architecture and program characteristics are gathered and used for the thread combining. To combine threads, Thread Tailor first constructs an undirected graph. Each node of the graph represents the characteristics, such as the number of executed cycles, of an thread. Each edge represents communication patterns between the threads. To construct the graph, Thread Tailor uses offline analysis. Thread Tailor determines which threads should be combined by applying a graph partitioning algorithm to the graph. Therefore, by using Thread Tailor, programmers can utilize the resources in multi-core processors even if the execution time of some thread is much different from others. However, Thread Tailor combines threads only once at the beginning of a program.

In this paper, we propose an efficient thread combining, which recombines threads according to program phases at runtime. Thus, thread combining of the new model is more adaptive than that of the traditional model.

II. Research Background

In this section, we describe about related works. In addition, we describe about Thread Tailor and how it works.

A. Related Works

Many works for adjusting the number of threads considering the environment have been proposed in OS and hardware fields. However, for determining how many threads an application should use, the OS or the hardware may not be able to use enough information. To address this problem, Jung et al.[2] and Suleman et al.[3] have proposed monitoring the execution and predicting the appropriate number of threads.

Jung's work relies on OpenMP library to automatically adjust the thread decomposition in SMT. Suleman's work implements Synchronization-Aware Threading which predicts the optimal number of threads depending on the amount of data-synchronization, and Bandwidth-Aware Threading which predicts the minimum number of threads required to saturate the off-chip bus. However, these works are only effective on data-parallel loops. In addition, Jung's work can be used only with OpenMP.

B. Thread Tailor

1) Outline of Thread Tailor: When programmers write parallel programs, it is difficult to determine the adequate number of threads depending on the processor specifications, and to balance the loads. To address these problems, Thread Tailor determines the number of threads based on the processor specifications before execution, and balances the loads of the cores by combining threads based on the result of profiling. Fig. 1 shows the processing flow of Thread Tailor. The processing flow of Thread Tailor consists of two stages. They are the static stage before runtime and the dynamic stage at runtime. In the static stage, first, Thread Tailor acquires processor

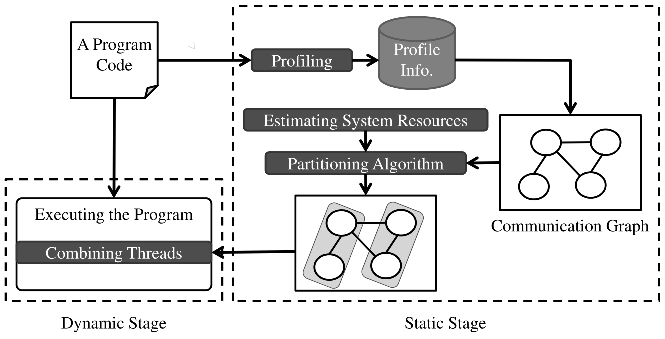
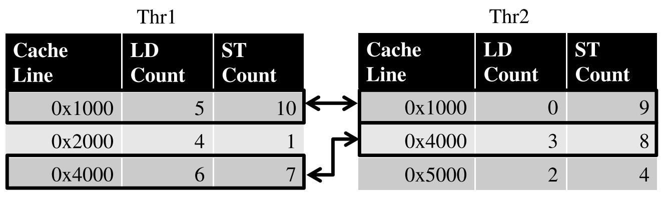


Fig. 1. The Processing Flow of Thread Tailor



Thr1

Cache Line	LD Count	ST Count
0x1000	5	10
0x2000	4	1
0x4000	6	7

Thr2

Cache Line	LD Count	ST Count
0x1000	0	9
0x4000	3	8
0x5000	2	4

Fig. 2. Cache Line Access Information

specifications such as the number of cores, the shared cache capacity, and so on. Next, Thread Tailor acquires information of each thread, such as the number of executed cycles, the memory bandwidth, and so on. Using the information gathered through the profiling, Thread Tailor constructs an undirected graph which is called a **Communication Graph**. Each node of the graph represents the number of executed cycles, the memory bandwidth, and the working set of a thread. Each edge represents communication patterns between the threads. Lastly, partitioning the Communication Graph, Thread Tailor determines which threads should be combined, or merged into one thread. When threads are created in a program, Thread Tailor applies a graph partitioning algorithm to the Communication Graph, and determines the thread combination.

2) Static Stage of Thread Tailor: Thread Tailor acquires processor specifications such as the number of cores, L2 shared cache capacity ($CacheCapacity$) and memory bandwidth ($MemBW$). Thread Tailor acquires the number of cores and $CacheCapacity$ from /proc/cpuinfo in Linux. $MemBW$ is acquired with STREAM benchmark[4].

Estimating processor specifications, Thread Tailor constructs a Communication Graph. Therefore, ahead of running program in the dynamic stage, Thread Tailor gathers the information of each thread. The information of the *i*-th thread is the cycles of computation without the barrier synchronization cycle ($Cycle_i$), the amount of the off-chip memory bandwidth (BW_i), and the working set size ($Workset_i$). The communication cost between *i*-th and *j*-th thread ($Comm_{i,j}$) is also gathered. $Cycle_i$ and BW_i are estimated by using Hardware Performance Counter (HPC) attached to each CPU core. However, $Workset_i$ and $Comm_{i,j}$ cannot be estimated by using HPC. Hence, Thread Tailor must approximate them by using another method. To estimate $Workset_i$, Thread Tailor examines cache line access information, which includes the accessed cache line and the number of accesses to the cache lines by each thread. All cache line accesses are sorted based on the number of accesses from every threads. Thread Tailor then determines how many unique cache lines are needed to comprise 90% of total accesses, and considers it as $Workset_i$.

Then, Thread Tailor estimates $Comm_{i,j}$ by using the cache line access information. In case of that all threads are executed in different cores, Thread Tailor finds approximate values of these costs. For example, if a program is executed with two threads Thr1 and Thr2, the cache line access information can be illustrated as Fig. 2. In this case, the cache lines which both threads access in common are `0x1000` and `0x4000`. When each thread accesses these lines, some communication overhead occurs. As shown in Fig. 2, Thr1 accesses the cache line `0x1000` by five load instructions and 10 store instructions. Thr2 accesses the same line by nine store instructions only. In this case, Thread Tailor estimates the number of communications between Thr1 and Thr2 as $min(5,9) + min(10,0) + min(10,9) = 14$. Likewise, Thread Tailor estimates the number of communications on the cache line `0x4000` as 16. Consequently, the total number of communications between Thr1 and Thr2 is regarded as 30. To estimate how many cycles does it take to communicate between threads, Thread Tailor assumes a non-uniform cache architecture with the 2D-mesh network and a directory-based coherence protocol. If threads are assumed to be randomly assigned to the cores, the average communication latency will be $3 \times \sqrt{Cores} \times L$, where L is the access latency to the lowest-level shared cache and $\sqrt{Cores}$ is the average distance between two cores in a grid. Using the information gathered through the profiling, Thread Tailor constructs a Communication Graph which is composed of nodes which represent $Cycle_i$, BW_i, $Workset_i$ and edges which represent $Comm_{i,j}$.

Thread Tailor determines which threads should be combined by analyzing the Communication Graph. For this purpose, Thread Tailor uses Kernighan-Lin (KL) algorithm[5], which is one of the graph partitioning algorithm. KL divides nodes in Communication Graph into two groups, exchanges nodes between the groups, and minimizes the number of edges across the groups. Each of the group represents a set of combined threads. KL estimates parameters of the *k*-th group as $Cycle_{Group_k} = \sum_{i \in Group_k} Cycle_i - \sum_{i,j \in Group_k} Comm_{i,j}$, $BW_{Group_k} = \sum_{i \in Group_k} BW_i$, and $Workset_{Group_k} = \sum_{i \in Group_k} Workset_i$. In Thread Tailor, KL finds the best combination of nodes, with which the largest number of execution cycles of the thread groups will be reduced, by exchanging nodes between groups with greedy search, while both $CacheCapacity/N > Workset_{Group_k}$ and $MemBW > BW_{Group_k}$ are fulfilled, where N is the number of groups. KL is applied repeatedly until the largest number of executed cycles in each group does not be improved any more.

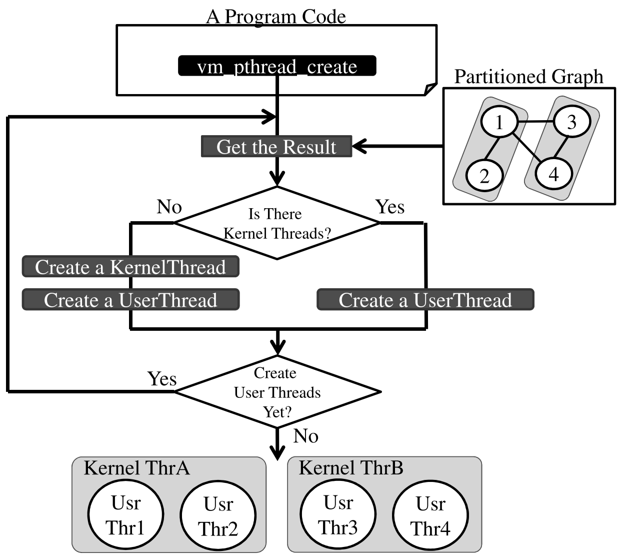


Fig. 3. The Combining Flow of Dynamic Stage

3) Dynamic Stage of Thread Tailor: Using the information acquired in the static stage, Thread Tailor combines threads in the dynamic stage. When `vm_pthread_create` function in Thread Tailor is called in a program, Thread Tailor creates both a kernel thread and a user thread, or only a user thread using the information. Fig. 3 shows the combining flow of dynamic stage. First, when the main thread creates a user thread UsrThr1, the main thread identifies the group, which UsrThr1 belongs to, by using KL. Now, the kernel thread corresponding to the group, which UsrThr1 belongs to, is not created yet. Hence, the main thread calls `pthread_create` function to create a kernel thread for UsrThr1. Next, when the main thread creates UsrThr2, the main thread identifies the group which UsrThr2 belongs to. If UsrThr2 belongs to the same group with UsrThr1, only the UsrThr2 is created. Therefore, by creating user threads within the kernel thread, Thread Tailor achieves thread combining. However, to execute a program with user threads and kernel threads, Thread Tailor needs to switch user thread contexts and have user thread barrier synchronization function. Thread Tailor manages the context information of user threads and switches a user thread to another user thread in the same group. When Thread Tailor detects that all user threads arrive at a barrier function, the user threads terminate the barrier synchronization.

III. Recombining Threads at Phase Changes

In this section, we propose a new model which will improve the total performance of Thread Tailor by recombining threads.

A. Dynamic Recombining according to Program Phases

When `vm_pthread_create` function is called, user threads are combined by Thread Tailor. However, Thread Tailor cannot change the combination of user threads through the program execution. Therefore, if the loads of the user threads fluctuate as the program phases change, the performance will decline because Thread Tailor cannot keep track

TABLE I
Execution Cycles in Program Phase in Barnes

	Thr0	...	Thr3	...	Thr15
...	...	...	...	...	...
Phase A	2,336,700	...	1,699,472	...	<u>2,355,357</u>
Phase B	<u>296,850</u>	...	176,227	...	288,654
...	...	...	...	...	...
Sum	342,073,821	...	<u>529,612,583</u>	...	449,629,402

of it. In this paper, we propose a new model of Thread Tailor which recombines user threads at each barrier, considering that the program phases will change at the barriers. The new model recombines user threads at each barrier to minimize the number of executed cycles of the last kernel thread, which arrives at the barrier, in each thread group.

To estimate the effectiveness of the new recombining model, we have observed the number of executed cycles between barrier synchronizations in a program. TABLE I shows the execution cycles of barnes from SPLASH-2 benchmark suite[6] with 16 threads on a 4-core processor. *Phase* in TABLE I represents the number of executed cycles of each thread between a barrier and the next barrier, and *Sum* represents the total cycle of all phases. Each of the numbers with underline represents the cycles of the last thread in the phase. TABLE I shows that the threads with the most largest number of executed cycles in *PhaseA*, *PhaseB* and *Sum* are different. This means that the load of each thread increases or decreases every phase. However, the traditional model combines threads only once at the beginning considering only *Sum*. On the other hand, the recombining model adequately combines threads every barrier synchronization considering the information of each *Phase*. Hence, user thread combining of our model is more fine-grained than that of the traditional model, and some speed-up is expected.

B. Additional Parameter for Partitioning Algorithm

As mentioned before, the new recombining model can change the combination of user threads every phase. Hence, the user thread migration, which the traditional model need not assume, will occur. Therefore, it should be prevented that the overhead for the user thread migration has a bad influence on the user thread combining. Hence, we install the number of L1 cache misses as the new parameter which reflects the cost of user thread migration. Now, assume that when a processor executes a parallel program, a user thread UsrThr1 accesses the same cache line before and after a barrier synchronization. If UsrThr1 is migrated to another core at the barrier synchronization, UsrThr1 will cause L1 cache misses afterward. Therefore, the new recombining model estimates this L1 cache miss penalty in the static stage and uses this penalty as additional parameter for KL.

In addition, there is a problem that BW_{Group_k} used by KL is estimated incorrectly larger in the traditional model. Therefore, we also improve the evaluation formula of KL.

IV. Implementation

A. Improvement of Profiling and Partitioning Algorithm

In the new recombining model, the method for profiling in the static stage needs to be changed to get information of each phase. In addition, KL should be improved for considering the migration costs of the user threads and improving the constraint on the memory bandwidth.

In the implementation of the recombining model, $Cycle_i$, BW_i, $Workset_i$ of each thread, $Comm_{i,j}$, and the number of L1 cache misses ($MigrationCacheMiss_i$) caused by user thread migrations should be estimated every phase. Through a program phase, $Cycle_i$, BW_i, $Workset_i$ of each thread and $Comm_{i,j}$ can be acquired as same as the traditional model. To estimate $MigrationCacheMiss_i$, the recombining cost, the cache line access information of each thread can be used. In this paper, the number of the cache lines, which are accessed in common before and after a barrier synchronization, is regarded as $MigrationCacheMiss_i$. Ahead of dynamic stage, these parameters are acquired every phase.

To consider $MigrationCacheMiss_i$ and to improve the constraint on memory bandwidth, we improve KL. In the improved KL, when the number of executed cycles of each group is estimated, L1 cache misses which are involved by user thread migration are considered. If KL determines that a thread is migrated, KL adds $L2AccessLatency \times MigrationCacheMiss_i$ to $Cycle_i$.

In the traditional model, the constraint on memory bandwidth, $MemBW > \sum_{i \in Group_k} BW_i$, is used by KL. However, to execute a program with switching user threads in each group, the user threads in the same group are not simultaneously executed. Therefore, the constraint on memory bandwidth in the traditional model is too hard. Hence, we change the constraint on the memory bandwidth as $MemBW > \max_{i \in Group_k} BW_i$.

B. Execution Model

In the new recombining model, when all user threads arrive at a barrier, the threads are recombined. Now, the recombining model prepares a user thread called *temporary user thread* per kernel thread, and uses it at each barrier synchronization. The reason why the temporary user threads are prepared is that the user threads may be migrated at each barrier synchronization. In the traditional model, each kernel thread always has only one active user thread, because the combination of user threads never change. However, in the recombining model, a kernel thread may have no active user thread right after a barrier synchronization, because the active user thread of the kernel thread can be recombined to another thread group. Similarly, a kernel thread may have multiple active user threads. Therefore, in the recombining model, the thread *activeness* should be delegated to other thread after recombining. However, if the activenesses are delegated directly between user threads, an active user thread may delegate its activeness to another active user thread. In this situation, execution will be stopped by segfault. Therefore, to prevent this situation, the recombining model uses temporary user threads.

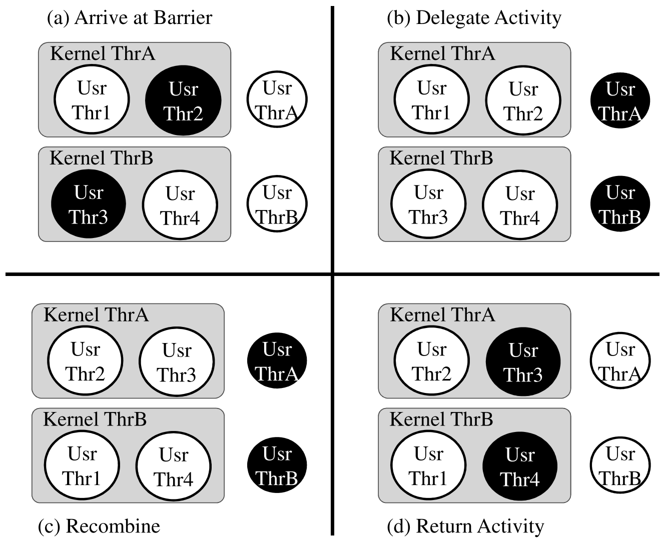


Fig. 4. An Example at a Barrier Synchronization

TABLE II
EVALUATION ENVIRONMENT

OS	Fedora 15
CPU	Intel Core2 Quad
Frequency	2.83 GHz
Shared L2 Cache	6 MB×2
Memory	3 GB
Compiler	llvm-gcc 4.2.1
Compile options	-O3

An example at a barrier synchronization is shown in Fig. 4. In Fig. 4, UsrThrA and UsrThrB are the temporary user threads. In the recombining model, (a) when the last user threads UsrThr2 and UsrThr3 arrive at a barrier, (b) their activenesses are delegated from UsrThr2 to UsrThrA and from UsrThr3 to UsrThrB. Next, (c) Thread Tailor recombines the user threads. In this example, Thread Tailor exchanges UsrThr1 and UsrThr3. After recombining, (d) UsrThrA delegates back its activeness to UsrThr3 and UsrThrB delegates back to UsrThr4.

However, in this model, there is a problem that some processor resources must be assigned to temporary user threads. Nevertheless, the number of these user threads is equivalent to the number of kernel threads at the most. Therefore, there will be little impact on the execution time.

V. Performance Evaluation

A. Evaluation Environment

We will discuss the performance of the new recombining model proposed in this paper. The evaluation environment is shown in TABLE II. We assume that no other program runs on the processor, and the CPU and cache resources can be occupied by the benchmark program. The parameters, $L2AccessLatency$ and the L2 cache line size, cannot be estimated at the system level. Hence, we assume the latency

TABLE III
BENCHMARK PROGRAMS AND THEIR INPUT PARAMETERS

SPLASH-2	
barnes	32768 BODIES
water-nsquared	4096 MOL, 6 STEP
PARSEC	
fluidanimate	simlarge
blackscholes	simlarge
swaptions	simlarge

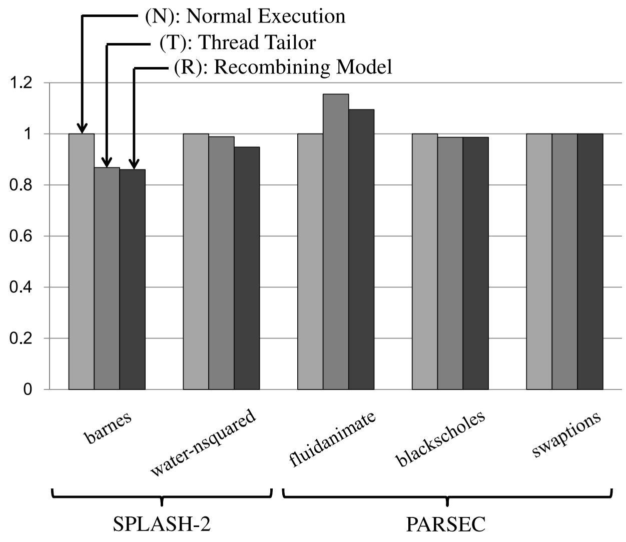


Fig. 5. Ratio of Execution Time

as 50 cycles and the line size as 64bytes. The number of user threads in the static and the dynamic stage is defined as 16.

We have evaluated the execution time of some workloads from SPLASH-2 benchmark suite and PARSEC benchmark suite[7]. These benchmarks are also used for the evaluation of Thread Tailor in [1]. The workloads are executed with the input parameters shown in TABLE III. All workload programs are compiled by llvm-gcc[8] 4.2.1 with -O3 option. We have evaluated following three models,

(N) Normal Execution (baseline)
(T) Thread Tailor
(R) Recombining Model proposed in this paper

and we tried 20 times on each workloads. All models are evaluated with four kernel threads.

B. Results with SPLASH-2 & PARSEC

The evaluation results are shown in Fig. 5. It shows the execution time of three models and each bar is normalized to the execution time of (N).

The results show that the recombining model achieved more speed-up than traditional model in barnes and water-nsquared, and mitigated the performance deterioration in fluidanimate. The reason why the execution time of these three programs is improved, compared to the traditional model, is that these programs include barrier synchronizations. There are 18 barrier synchronizations in barnes, 38 barrier synchronizations in water-nsquared, and 41 barrier synchronizations in fluidanimate. Hence, the grain of thread combining become finer than the grain with traditional Thread Tailor. Moreover, fluidanimate is a program which uses much memory bandwidth. Hence, due to the hard constraint on memory bandwidth in KL, the traditional model cannot balance loads between the combined threads. By relaxing this constraint, fluidanimate with the recombining model is 6.0% faster than that with the traditional model. On the other hand, the execution time of blackscholes and swaptions with the recombining model is as fast as that with the traditional model. The reason is that these programs do not include any barrier synchronization. In conclusion, the performance of the recombining model (R) is better than the traditional model (T) as a whole. The model (R) improves the execution time 2.1% in average, and 6.0% in maximum. In addition, the model (R) improves the execution time of the programs,which include barriers, 3.6% in average.

VI. CONCLUSION

In this paper, we have proposed a new recombining model of Thread Tailor, which can balance the loads according to the program phases. In addition, we have improved the constraint on memory bandwidth in the graph partitioning algorithm. Through an evaluation with SPLASH-2 and PARSEC benchmark suites, it is found that the new model improves the execution time 2.1% in average, and 6.0% in maximum, compared to the traditional model.

One of our future works is to improve the execution time of a program without barrier synchronizations. In this paper, we have considered that the program phases will change at the barriers. However, not only barrier synchronizations will cause the change of phases. In addition, we need to prevent programs, such as fluidanimate, from performance deterioration with the recombining model, compared to the performance of normal execution.

REFERENCES

[1] J. Lee *et al.*, "Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications," in *Proc. 37th Annual Int'l Symp. on Computer Architecture*, 2010, pp. 270–279.
[2] C. Jung *et al.*, "Adaptive Execution Techniques for SMT Multiprocessor Architectures," in *Proc. 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2005, pp. 236–246.
[3] M. A. Suleman *et al.*, "Feedback Driven Threading: Power-Efficient and High-Performance Execution of Multithreaded Workloads on CMPs," in *Proc 13th Int'l Conf. on Architectural Support for Programming languages and Operating Systems (ASPLOS)*, 2008, pp. 277–286.
[4] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.
[5] B. W. Kernighan *et al.*, "An Efficient Heuristic Procedure for Partitioning Graph," *Bell System Technical Journal*, pp. 291–307, 1970.
[6] S. C. Woo *et al.*, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc. 22nd Annual Int'l Symp. on Computer Architecture (ISCA'95)*, 1995, pp. 24–36.
[7] C. Bienia *et al.*, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. 17th Int'l Conf. on Parallel Architectures and Compilation Techniques*, Oct. 2008, pp. 72–81.
[8] C. Lattner *et al.*, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proc. 2004 Int'l Symp. on Code Generation and Optimization (CGO'04)*, 2004, pp. 75–86.