Journal of Information Processing Vol. 19 421–429 (Aug. 2011)

Regular Paper

A Library-based Performance Tool for Multicore Pervasive Servers

Sayaka Akioka,[†1] Yuki Ohno,[†1] Midori Sugaya[†1,∗1] and Tatsuo Nakajima[†1]

This paper proposes SPLiT (Scalable Performance Library Tool) as the methodology to improve performance of applications on multicore processors through CPU and cache optimizations on the fly. SPLiT is designed to relieve the difficulty of the performance optimization of parallel applications on multicore processors. Therefore, all programmers have to do to benefit from SPLiT is to add a few library calls to let SPLiT know which part of the application should be analyzed. This simple but compelling optimization library contributes to enrich pervasive servers on a multicore processor, which is a strong candidate for an architecture of information appliances in the near future. SPLiT analyzes and predicts application behaviors based on CPU cycle counts and cache misses. According to the analysis and predictions, SPLiT tries to allocate processes and threads sharing data onto the same physical cores in order to enhance cache efficiency. SPLiT also tries to separate cache effective codes from the codes with more cache misses for the purpose of the avoidance of cache pollutions, which result in performance degradation. Empirical experiments assuming web applications validated the efficiency of SPLiT and the performance of the web application is improved by 26%.

1. Introduction

Information appliances have become more sophisticated in order to provide more enhanced services, and are designed to support multi-tasking, multi-network, and intuitive user interface. It is a realistic projection of the future computational environment that people will be able to access information and services anytime anywhere via a high performance appliance at hand, which is connected to servers in the world on some kind of overlay network, such as a cloud computing environment. Such ubiquitous services strongly rely on web services and database systems, therefore, the service providers, called pervasive servers [1)], are expected to be highly optimized for web services and databases.

A multicore processor is a strong candidate to support pervasive servers from the hardware layer. A multicore processor is not only faster than a single core processor, but also superior to a flock of single core processors in terms of energy saving, which is one of the most important features for battery-powered information appliances. Many of the modern operating systems (OS) and applications are also ready to run on multicore processors, however, the techniques utilized in these softwares are basically migrations of the techniques developed for single core processors. This simple but immature approach will soon reveal several problems such as performance degradation caused by resource contention. Additionally, parallel applications are often optimized based on the characteristics of both the target application and the parallel environment, and there is no trivial clue universal for any environment.

Therefore, this paper proposes SPLiT (Scalable Performance Library Tool) to support any kinds of parallel applications on multicore processors from the OS layer, which knows very detailed process behaviors. Here, we use software such as MySQL or Apache applications. SPLiT provides both CPU and cache optimizations based not only on CPU utilization, but also on statistics and predictions of data accesses of the target application. The major contributions of SPLiT are 1) SPLiT improves the performance of applications up to 26%, 2) the overhead of SPLiT is small i.e., 2.5–5%, 3) all the features are available with small code modifications, and 4) the validations in this paper are highly applicable to power-efficient pervasive servers on multicore processors.

The rest of this paper is organized as follows. Section 2 defines the problems this paper addresses. Section 3 introduces conventional solutions. Section 4 illustrates detailed design of SPLiT, and Section 5 discusses the effect and performance of SPLiT through Empirical experiments. Finally, Section 6 concludes this paper.

†1 Waseda University
∗1 Presently with Yokohama National University

This article is the brushed up version based on the article in Proceedings of The 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2010).

© 2011 Information Processing Society of Japan

2. Problem Statement

SPLiT deals with three problems; programming, resource sharing, and thread scheduling. A detailed description of each problem is as follows.

Programming Recent speed-ups of CPUs basically come from the increase of parallelism, therefore, applications will never run faster unless the applications themselves run in parallel. Besides the parallelization, diversity of CPU architectures also intensifies the difficulty of software implementation. Considering these situations, programmers are required to optimize their applications according to the target architecture, however, this is not easy for every programmer, and great efforts should be taken to optimize one application to several architectures.

Resource sharing When a parallel application runs on a multicore processor, functionalities to maintain consistency of data often cause bottle necks, such as locking mechanisms for threads and processes, and overhead caused by inappropriate cache coherency. More parallelism brings out more conflicts of data accesses. Data confliction limits the speed up by the parallelism.

Thread scheduling Wentzlaff, et al.[3] pointed out scheduling on multicore processors should be aware of both space and time sharing. That is, some applications are ideal for being evenly scattered over the available cores, rather than allocated onto one particular core. This is the problem of space sharing, and a similar discussion is effective for time sharing of each CPU core as well. Sometimes all the cores communicate with each other all at once, and the equality of core usage is the key to decrease the blocking for communication.

SPLiT provides library tools to optimize CPU allocation for applications on multicore processors, which predict data accesses of the target application, and provides optimum thread allocation avoiding data sharing and cache efficiency. SPLiT library tools also requires programmers only to add the start point and the end point of the target in the source code, and programmers benefit from all the optimizations SPLiT provides.

3. Related Work

There are several researches on optimization techniques looking at the application layers such as the work by Parello, et al.[4]. These optimization techniques, however, often require enormous trials, and are not portable to the other architectures are they are. There are many proposals on infrastructures[5)–11)] or optimization techniques on data parallel applications[12] as well, which are not applicable to applications with no data parallelism.

There are also several projects on optimization techniques approaching from OS kernel layers, which is the same approach to SPLiT. Meng, et al.[13] succeeded in an improvement of cache efficiency by handling thread private data on the cache, however, their approach possibly restricts memory usage and requires detailed control on thread private data and caches. Azimi, et al.[14], and Tam, et al.[15] successfully decreased cache misses, reducing the latency of web server applications.

4. SPLiT

4.1 Overview

SPLiT monitors applications, predicts process behavior based on the collected data, and allocates resources. **Figure 1** illustrates the overview of SPLiT. SPLiT

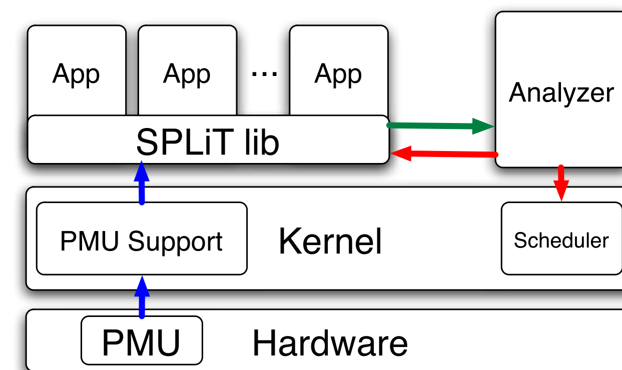

Fig. 1 Overview of SPLiT.

© 2011 Information Processing Society of Japan

Table 1 Performance data to be recorded.

Data	Features
Shared	
For Each Application	
app_id	application ID
related_app_id	ID of related application
hash_size	hash size of data storage
For each code	
code_id	code ID
count	counts executed the code
cycle_count	number of cycles for the code
cache_miss	number of cache misses
related_code_id	ID of related code
cpu_mask	core ID to be utilized
Non-Shared	
code_key_type	code key
code_key_table	hash data for code recognition

library provides APIs to user programs, and SPLiT library function, which is embedded at the entering (split_start()) and evacuating (split_end()) points of the target process by the programmer, and is required to start the sequence of monitoring and analysis. Programmers also have to call up the initialization function, split_init(), for the initialization of SPLiT in the beginning of their applications. The callee function counts CPU cycles and cache misses of the monitoring section, then, another library function saves the monitoring results onto the memory.

Monitoring is enabled via Performance Monitoring Unit (PMU), which many of the recent processors are equipped [14]. The overhead of monitoring is much smaller than the overhead by software monitoring, as PMU is a hardware supported functionality. PMU support by kernel is indispensable in order to complete this monitoring process. CPU does not recognize which thread is running, therefore, OS kernel has to save the register values and rearrange the events to observe on thread switching. PMU support also provides APIs to accept requests of PMU monitoring from user programs.

In the rest of this paper, we call the continuous part of the program to be monitored a "code". SPLiT library creates code_id, records performance data, and schedules threads for each code. Here, code_id is provided as the return value of split_start() call. SPLiT lib also creates app_id and records performance

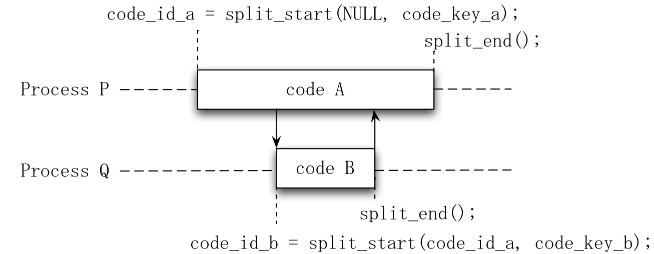

Fig. 2 Illustration of how code_id and split_start() are utilized.

Table 2 Library API.

API functions	features
app_id split_init(app_id, related_app_id, code_key_type)	initialize the application
code_id split_start(related_code_id, code_key_type)	the beginning of a code
void split_end()	the end of a code
void split_set(setting_type, value)	modify the settings
void split_get(setting_type, value)	get the settings

data for each application in the similar fashion, and app_id is the return value of split_init() call.

Table 1 shows the data to be recorded by SPLiT lib. Here, As already mentioned, code_id is the return value of split_start(), and app_id is the return value of split_init() that is called only one at the beginning of the application. These return values are utilized to declare the relationship among codes. That is, code_id is expected to be passed as related_code_id to split_start(), and app_id is expected to be passed as related_app_id to split_init(). **Figure 2** illustrates how code_id and split_start() are utilized. The relationship between code_id and split_start() is applicable to the relationship between app_id and split_init().

Table 2 summarizes functional calls of SPLiT, return values of these functional calls, and parameters to the functional calls.

4.2 Statistics

SPLiT counts CPU cycles and the number of cache misses for each application. The current version of SPLiT calculates an average for each metric, however, it switches the way to calculate the average depending on the number of samples (Eq. (1)). SPLiT utilizes an arithmetic average with smaller samples in order to

© 2011 Information Processing Society of Japan

moderate the effect by jutted measurements. On the other hand, an exponential moving average is utilized with a sufficient number of measurements. An exponential moving average weights more on recent values, taking over older values. Here, a_n is the current statistics, n is the counts of codes, a is the new measurement, and a_{n+1} is the updated statistics. θ is the threshold to switch the methodology of statistics, and w is the weight of the exponential moving average.

$$a_{n+1} = \begin{cases} (a_n \times n + a)/(n+1) & (n < \theta) \\ a_n \times w + a \times (1-w) & (n \geq \theta) \end{cases} \quad (1)$$

4.3 Resource Allocation Based on Predictions

SPLiT predicts the behavior of the application based on the statistics, and then decides resource allocations. First of all, SPLiT categorizes codes into two groups according to cache misses. Secondly, SPLiT decides the number of cores to be allocated to each code, and then finally, starts allocation considering both cache efficiency and the related codes. In the rest of this section, we describe each step more specifically.

4.3.1 Code Categorization Based on Cache Efficiency

In order to increase cache hit rates, codes sharing the same data should be allocated on the same core. Therefore, SPLiT categolizes codes based on cache efficiency, and puts more priority on code categorises with better cache efficiency.

Dirty-code/clean-code

We call a code with more cache misses than the threshold a dirty-code, and all the other codes clean-codes. A clean-code is eligible for better cache efficiency, while a dirty-code is not.

Dirty-core/clean-core

We also categorize cores into dirty-cores, and clean-cores, depending on the codes to process. We do this categorization in order not to degrade the performance of clean-code by cache pollutions by dirty-codes.

4.3.2 Load Balancing Based on CPU Cycles

For the purpose of load balancing, SPLiT decides the number of cores to allocate for each code depending on the ratio of cycle clocks for clean-codes, and dirty-codes.

Dirty-core allocation

As a dirty-code often rewrites a cache, the cache is hard to be utilized effectively. Therefore, SPLiT does not consider cache efficiency for dirty-codes, however, it balances loads. OS scheduler usually takes care of CPU load balancing, and SPLiT mostly relies on OS scheduler for detailed process scheduling with one exception to prohibit dirty-codes from allocation over clean-cores.

Clean-core allocation

A clean-code has a chance for better cache utilization depending on core allocation, therefore, SPLiT decides to allocate clean-codes to clean-cores considering dependency among codes. Hints to analyze code dependency are app_id, related_app_id, code_id, and related_code_id. Clean-core allocation policy is as follows.

(1) When code_ids are equal, the codes are allocated on to the same core, as codes with the same ID often share data.
(2) When app_ids are equal, the codes are allocated on to the cores sharing the cache, as applications with the same ID often share data. Related applications are also allocated close to the application with related_app_id.
(3) Based on the ratio of CPU clocks of each application to the total CPU clocks of the environment, SPLiT decides the number of cores to be allocated for each application.
(4) SPLiT decides core allocation for codes, in descending order of CPU clocks of the corresponding code. On this allocation, SPLiT considers core allocation starting with cores allocated to the application, cores allocated to the related codes, and then finally considers all the cores.

The overall sequence of core allocation is as follows.

(1) SPLiT calculates the average clock count for each core in advance. The average clock counts is calculated as total clock counts for all the codes divided by the number of cores.
(2) Among the cores allocated to the application, SPLiT chooses the core with minimum clock counts in total.
(3) If the total clock counts spent will not exceed the average clock counts even after the allocation of codes, SPLiT allocates code onto the core.

© 2011 Information Processing Society of Japan

(4) If the total clock counts spent exceeds the average clock counts after the allocation of codes, SPLiT tries to allocate the codes to the cores which the related codes are allocated to.

(5) If there is no related code, SPLiT chooses the core with the minimum clock counts spent among all the cores.

When all the core allocation is decided, Analyzer writes to cpu_mask to specify the allocated cores. When split_start is called in the beginning of the code, threads are moved to the core related to code_key_type.

5. Empirical Performance Evaluation

5.1 Setup

We implemented SPLiT onto the environment with features listed as **Table 3**. Intel Core i7 975 equips four cores inside one CPU. Each core has two logical processors, and two threads run in parallel on one core with HyperThreading Technology [16]. Intel Core i7 975 has three levels of cache structure. Each core has L1 and L2 cache shared by the two threads, and L3 cache and the main memory shared by all the threads [17]. We also utilized Perfmon3 [20] and mBrace [21] for the full functionalities of SPLiT.

The effect of the resource management by SPLiT is measured through the experiments with Apache 2.2.11 [18], and MySQL 5.1.32 [19]. We chose Apache and MySQL as workloads on the assumption that SPLiT is utilized for Web applications. As already described in Section 1, ubiquitous services come to rely on web servers more and more, and pervasive servers utilized for these services comes to be sufficient. Therefore, we can easily imagine the situation that pervasive servers provide web services via Apache, and provide information requested by clients referring to MySQL servers. Therefore, we picked up Apache and MySQL for this experiment as strong candidates of near-future killer applications for pervasive servers. **Table 4** shows the hardware information of a client. For the Web server, we utilized the same configuration shown in Table 3. Ideally, the hardware is not a PC, but a more likely embedded hardware, however, such a rich hardware is not available for embedded equipment yet. Therefore, we believe PC with Intel CPU is a reasonable selection as the near-future pervasive servers. Another reason for this hardware is PMU support. As SPLiT relies heavily on PMU, therefore, PMU support is indispensable. Fortunately, Intel CPUs are very common in practical use, and we believe the validation and efficiency for other CPUs with PMU support. Further validation on this point, however, is reserved a future work. Parameters such as the threshold for dirty-code recognition, θ, w, and any others are determined through some experiments in advance. That is, we ran through the experiments with several settings, and chose the best combination. The proper methodology for parameter decision is reserved as future work.

For Apache, SPLiT utilizes URL of the requested web page as code_key_type. For this experiment, code_key_type is utilized to recognize the process, and is passed as the parameter for split_start() or split_init(). SPLiT starts measurement when the process, or thread, which is generated by Multi-Processing Module (MPM) by Apache in advance, accepts an HTTP request from the client. When the requested data is completely sent to the client, SPLiT finishes the measurement. We modified Apache to send code_id with an SQL query, as a query is sent through SQL library when an application on Apache sends the query to MySQL server. By this modification to Apache, an application on Apache always sends code_id without any modification to modules or applications.

For MySQL, SPLiT utilizes a query type and a target table as code_key_type. SPLiT starts the measurement when the SQL server process accepts a query. On this point, the code_id passed with the SQL query from Apache is passed over to the split_start function as related_code_id. SPLiT ends the measurement, similar

Table 3 Hardware and operating system for this implementation.

Processor	Intel Core i7 965 (3.20 GHz)
Memory	DDR SDRAM 3 GB
Network Device	Marvell Yukon 88E8056 PCI-E Gigabit Ethernet Controller
OS	Linux 2.6.30

Table 4 Hardware for a client.

Processor	Intel Core 2 Duo (2.0 GHz)
Memory	DDR3 SDRAM 2 GB (1067 MHz)
Network Device	10/100/1000BASE-T (Gigabit) Ethernet
OS	Mac OS X 10.5.8

© 2011 Information Processing Society of Japan

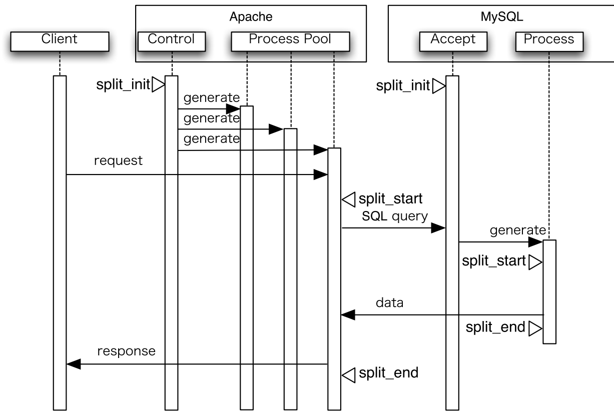

Fig. 3 Overview of measurement of Apache, and MySQL application.

to the case of Apache, when the requested data is completely sent to the client. **Figure 3** summarizes the flow of the target application, and how SPLiT is utilized.

We utilized Rice University Bidding System (RUBiS) [22], and Apache JMeter (JMeter) [23] as the workload for the benchmarks. RUBiS is a prototype of auction sites, which is modeling eBay [24], and sales products are managed by MySQL with seven tables. JMeter is an application for load testing and performance measurements. The experiments here utilized JMeter in the client, and let the client access RUBiS as a web server for performance measurements.

5.2 Prediction Accuracy

In order to measure the prediction accuracy of SPLiT, each code is repeated 1,000 times for statistics, and the same code is repeated 1,000 times again with the same settings for measurement. The target codes are one code for Apache, and another code for MySQL, however, be aware that these codes are utilized repeatedly whenever a new request comes from a client. We also tested the case with and without core allocations by the proposed methodology in order to compare the efficiency of core allocation. **Table 5** is the summary of the empirical results. Here, error is defined as Eq. (2) with s_n for statistics, and m_n for the actual measurement. The maximum error of cycles, or the maximum error of cache misses is the maximum value of $((s_n - m_n)/m_n)$ for each case.

Table 5 Prediction accuracy.

	W/O Core Allocation		W/ Core Allocation	
	Apache	MySQL	Apache	MySQL
Error for Cycles [%]	11.2	5.7	9.0	1.3
Error for Cache Misses [%]	9.0	6.7	4.1	1.8
Maximum Error for Cycles [%]	129.3	230.1	134.2	148.1
Maximum Error for Cache Misses [%]	100.4	227.2	94.7	170.0

$$e = \frac{1}{N}\sum_{n=1}^{N}\frac{s_n - m_n}{m_n} \tag{2}$$

Based on Table 5, we can conjecture that core allocation reduces both errors and the maximum error with the exception of the maximum error for cycles of Apache. The reason for this exception is that variations of cycles, and cache misses are suppressed by restricting available cores by core allocation.

5.3 Response Time

Utilizing JMeter in the client, we measured the turn around times of web applications. Here, the turn around time of a web application is defined as the duration of the process at a client. During the processing time, the client sends HTTP requests to the Web applications according to the scenario in JMeter, and completely accepts the responses for all the requests sent. We prepared two kinds of scenario of JMeter. One scenario is that each process accesses pages with a small amount of data to exchange frequently (*many accesses*). The other scenario is that the total number of accesses to a page is small, however, the total amount of data to exchange for each access is huge (*huge data*). **Table 6** summarizes the number of files to access, the total amount of data to receive, and the number of SQL queries for each scenario. For each scenario, we measured the performance of 1) the original versions of Apache, and MySQL (Original), 2) the modified version only with performance prediction by SPLiT (Prediction), and 3) the modified version with performance prediction and core allocation by SPLiT (Prediction + Core Allocation).

© 2011 Information Processing Society of Japan

Table 6 Details of scenarios.

	Many Accesses	Huge Data
Number of Clients	1,000	100
Number of Files to Access	14,000	1,300
Number of SQL Queries	10,000	4,542,500
Lines of Database [millions]	184.4	23.0
Total Data Files Received [MB]	218.2	409.7

Table 7 Turn around time for many accesses scenario.

	Turn Around Time [ms]	Overhead [%]
Original	21,788	—
Prediction Only	22,888	+ 5.0
Prediction + Core Allocation	20,759	− 4.7

Table 8 Turn around time for huge data scenario.

	Turn Around Time [ms]	Overhead [%]
Original	152,426	—
Prediction Only	156,275	+ 2.5
Prediction + Core Allocation	112,738	− 26.0

Table 7 summarizes the results with the scenario of many accesses. In this scenario, the overhead of SPLiT is 5.0%, and the overall performance is improved with 4.7% utilizing both prediction and core allocation by SPLiT.

Table 8 summarizes the results with the scenario of huge data. In this scenario, the overhead of SPLiT is 2.5%, and the overall performance is improved with 26.6%, which is better than the results the with scenario of many accesses. SPLiT optimizes applications with both many accesses, and huge data, however, there are more effects on applications with huge data transfer. Regardless of senario, the network cost for the same process in the same senario is identical, therefore, the key for the speed-up is the process on CPU, which is the very part SPLiT optimizes. Compared with the many accesses senario, the number of SQL queries is huge in the huge data senario. We can conjecture that the optimization on SQL query thread by SPLiT speeds up the corresponding process at a certain level, and then the effect cumulatively made a big difference on the speed-ups on the two senarios.

5.4 Core Allocation

We measured the turn around times of two kinds of core allocation policies with the huge data scenario in order to validate the effect of the core allocation policy by SPLiT. One policy is that each application is allocated onto each physical core, and the two logical cores on the same physical core execute the same application. We call this allocation *intensive allocation*, which is implemented as the core allocation policy of SPLiT. The other policy is that one logical core executes Apache, and the other logical core on the same physical core executes MySQL. We call this type of core allocation *extensive allocation*. **Table 9** compares the turn around times of the original system, intensive allocation, and extensive allocation. The intensive allocation is 16.4% better than the original system, while the extensive allocation is 60.0% worse. We can conjecture that core allocation has a strong impact on the performance of applications, and the intensive allocation is the right approach.

Table 9 Turn around time with core allocation.

	Turn Around Time [ms]	Overhead [%]
Original System	152,426	—
Intensive	127,462	− 16.4
Extensive	243,898	+ 60.0

Table 10 Overhead of each part of SPLiT.

	Overhead [ms]	Ratio [%]
Shared Memory Access for Statistics	1,731	34.0
PMU Control by Perfmon3	839	16.5
Thread Migration	101	2.0
Analyzer	410	8.1
SPLiT lib and Related Codes	570	11.2
Modification of Apache, and MySQL	1,439	28.3
Total	5,090	100.0

5.5 Overhead of SPLiT

We measured the duration of the huge data scenario in order to clarify the overhead of each part of SPLiT, disabling the corresponding part of SPLiT. **Table 10** summarizes the measured overheads. Here, the sum of all of the ratio is not equal to 100% because of the effect of rounding errors.

5.6 Total Amount of Code Modification

One huge advantage of SPLiT is that a small modification to the application enables all the features of SPLiT including performance monitoring and resource

© 2011 Information Processing Society of Japan

Table 11 Total amount of code modifications.

Application	Number of Lines	
	modified or deleted	added
Apache (moduled)	0	73
Apache (embedded)	0	10
MySQL library	0	162
MySQL server	6	48

optimizations. All SPLiT requires of programmers is adding several lines to the applications for SPLiT library calls. **Table 11** shows the total amount of code modifications for Apache with moduled SPLiT, Apache with embedded SPLiT, MySQL library, and MySQL server respectively. Here, the modification to MySQL library is to let web applications on Apache send code_id with SQL query to SQL server. As summarized in Table 11, only small changes to the application source codes are enough to embed SPLiT library calls.

6. Conclusion

In this paper, we proposed SPLiT, which predicts the behaviors of processes and threads, and optimizes resource allocations based on the predictions on the fly. For predictions, SPLiT utilizes the statistics collected via PMU. All the programmers are required to do is simply to specify the target codes of the application by surrounding the function calls to tell SPLiT which part of the codes should be optimized. Once this small modification to the application is made, all the monitoring, analysis, and optimizations through resource allocation provided by SPLiT are turned on.

We implemented SPLiT library on the linux environment and validated the efficiency of SPLiT through experiments assuming optimizations on web services. The experiments are set up realistically utilizing RUBiS and JMeter, and SPLiT improved the performance of web applications by 26%, and the total code modification was only 10–162 lines.

SPLiT will play a more important role for a good solution of optimum resource usage, which is an especially challenging goal in a multicore environment. For better resource management and further optimizations by SPLiT, we are planning a greater improvement of predictions, as well as support for other hardwares and multi-threading.

Acknowledgments This material is based in part on work supported by "New IT Infrastructure for the Information-explosion Era", MEXT Grant-in-Aid for Scientific Research on Priority Areas.

References

1) Nakajima, T.: Pervasive Servers: A framework for creating a society of appliances, *Pers Ubiquit Comput*, Vol.7, pp.182–188 (2003).
2) Veal, B. and Foong, A.: Performance scalability of multi-core web server, *Proc. 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems* (*ANCS'07*), pp.57–66 (2007).
3) Wentzlaff, D. and Agarwal, A.: Factored operating systems (FOS): The case for a scalable operating system for multicores, *SIGOPS Oper. Syst. Rev.*, Vol.43, No.2, pp.76–85 (2009).
4) Parello, D., Temam, O., Cohen, A. and Verdun, J.-M.: Toward a systematic, pragmatic and architecture-aware program optimization process for complex processors, *Proc. 2004 ACM/IEEE Conference on Supercomputing* (*SC'04*), p.15 (2004).
5) Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks, *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (*EuroSys'07*), pp.59–72 (2007).
6) Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al.: *OpenMPI: Goals, concept, and design of a next generation MPI implementation*, *Proc. 11th European PVM/MPI Users' Group Meeting* (2004).
7) Dagum, L. and Menon, R.: OpenMP: An industry-standard api for shared-memory programming, *IEEE Computer Science and Engineering*, Vol.5, No.1, pp.46–55 (1998).
8) Munshi, A.: OpenCL: Parallel Computing on the GPU and CPU, *SIGGRAPH*, Tutorial (2008).
9) Dean, J. and Ghemawat, S.: MapReduce: Simplified data processing on large clusters, *Proc. 6th OSDI*, pp.137–150 (2004).
10) Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G. and Kozyrakis, C.: Evaluating mapreduce for multi-core and multiprocessor systems, *Proc. High Performance Computer Architecture 2007* (*HPCA2007*) (2007).
11) Reinders, J.: *Intel threading building blocks*, O'Reilly & Associates, Inc., Sebastopol, CA, USA (2007).
12) Chen, S., Gibbons, P.B., Kozuch, M., Liaskovitis, V., Ailamaki, A., Belelloch, G.E., Falsafi, B., Fix, L., Hardavellas, N., Mowry, T.C. and Wikerson, C.: Scheduling threads for constructive cache sharing on cmps, *Proc. 19th Annual ACM Symposium*

© 2011 Information Processing Society of Japan

on Parallel Algorithms and Architectures (*SPAA'07*), pp.105– 15 (2007).
13) Meng, J. and Shadron, K.: *Avoiding Cache Thrashing due to Private Data Placement in Last-level Cache For Manycore Scaling*, *Proc. 2009 IEEE International Conference on Computer Design* (*ICCD'09*) (2009).
14) Azimi, R., Tam, D.K., Soares, L. and Stumm, M.: Enhancing operating system support for multicore processors by using hardware performance monitoring, *SIGOPS Oper. Syst. Rev.*, Vol.43, No.2, pp.56–65 (2009).
15) Tam, D., Azimi, R. and Stumm, M.: Thread clustering: Sharing-aware scheduling on smp-cmp-amr multiprocessors, *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (*EuroSys'07*), pp.47–58 (2007).
16) Kaufaty, D. and Marr, D.T.: Hyperthreading thechnology in the netburst microarchitecture, *IEEE Micro*, Vol.23, No.2, pp.56–65 (2003).
17) Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2 (2009).
18) Apache. http://www.apache.org/
19) MySQL. http://www.mysql.com/
20) Perfmon project. http://perfmon2.sourceforge.net/
21) van der Zee, A., Courbot, A. and Nakajima, T.: mBrace: action-based performance monitoring of multi-tier web applications, *Proc. 3rd Workshop on Dependable Distributed Data Management* (*WDDM'09*), pp.29–32 (2009).
22) RuBiS. http://rubis.ow2.org/
23) JMeter. http://jakarta.apache.org/jmeter/
24) eBay. http://www.ebay.com/

(Received December 5, 2010)
(Accepted May 14, 2011)
(Released August 10, 2011)

Sayaka Akioka is a researcher in Information Technology Research Organization of Waseda University. She received her Ph.D. in computer science from Waseda University in 2004. Prior to joining Waseda University in April 2010, she was an assistant professor in The University of Electro-Communications. Her research interests lie in the area of a parallel computing and data mining. In particular, her research focuses on the resource management in parallel computing environment. Recent projects have included the dynamic scheduling of data intensive applications in a cloud environment, and the design of efficient algorithms for machine learning algorithms on GPU processors.

Yuki Ohno received his B.S. degree in computer science from Waseda University, Japan, in 2008, and Master Degree in 2010 in the Department of Computer Science at Waseda University, Japan. His research interests are in monitoring, operating system and resource management in distributed system.

Midori Sugaya is a lecturer of Yokohama National University. She has eight years of work experience in the software industry. She received her Master Degree in computer science from Waseda University, Japan, in 2004, and belonged to Dependable Embedded OS R&D Center, Japan Science and Technology Agency (JST) in 2008 and received her Ph.D. in computer science from Waseda University in 2010. Her research interests include operating and dependable systems and proactive fault management system.

Tatsuo Nakajima is a professor of the Department of Computer Science and Engineering at Waseda University. His research interests are distributed systems, embedded systems, ubiquitous computing and interaction design. Currently, his group is working on three topics. The first topic is to develop a virtualization layer for multicore processor based embedded systems. The second topic is to develop ambient media that are new media to help human decision making. The third topic is to develop a crowdsourcing services to exploit human computation.

© 2011 Information Processing Society of Japan