Scientific Programming 17 (2009) 285–294
DOI 10.3233/SPR-2009-0289
IOS Press

ePRO-MP: A tool for profiling and optimizing energy and performance of mobile multiprocessor applications

Wonil Choi, Hyunhee Kim, Wook Song, Jiseok Song and Jihong Kim *
School of Computer Science and Engineering Seoul National University, Seoul, Korea

Abstract. For mobile multiprocessor applications, achieving high performance with low energy consumption is a challenging task. In order to help programmers to meet these design requirements, system development tools play an important role. In this paper, we describe one such development tool, ePRO-MP, which profiles and optimizes both performance and energy consumption of multi-threaded applications running on top of Linux for ARM11 MPCore-based embedded systems. One of the key features of ePRO-MP is that it can accurately estimate the energy consumption of multi-threaded applications without requiring a power measurement equipment, using a regression-based energy model. We also describe another key benefit of ePRO-MP, an automatic optimization function, using two example problems. Using the automatic optimization function, ePRO-MP can achieve high performance and low power consumption without programmer intervention. Our experimental results show that ePRO-MP can improve the performance and energy consumption by 6.1% and 4.1%, respectively, over a baseline version for the co-running applications optimization example. For the producer-consumer application optimization example, ePRO-MP improves the performance and energy consumption by 60.5% and 43.3%, respectively over a baseline version.

Keywords: Multi-processor, multi-threaded software, embedded software profiling tool, performance, energy, energy model

1. Introduction

Achieving both high-performance and low-energy consumption is an important design goal for modern mobile embedded applications. Since most mobile embedded applications run on battery-operated mobile devices, the low-energy consumption has been one of the most important design constraints for many mobile devices. At the same time, as mobile embedded applications become more complex, a high level of computing power is necessary for adequate application operations. In order to meet these design requirements, programmers should understand the performance and the energy consumption of their applications. Furthermore, when the applications do not satisfy the design requirements, it should be easy to predict and identify performance and energy bottlenecks of the applications at the application level.

*Corresponding author: Jihong Kim, Room 315-2, Building 302, Seoul National University, Shillim-dong, Kwanak-gu, Seoul 151-742, Korea. Tel.: +82 2 880 1861; Fax: +82 2 871 4912; E-mail: jihong@davinci.snu.ac.kr.

To help embedded developers to analyze their applications, many profiling tools have been developed for embedded systems. For example, many existing profiling tools estimate the performance characteristics of applications using hardware performance counters available in most microprocessors [3,10,13]. For energy profiling, most existing tools depend on extra power measurement equipments [3,9,12] to measure the power consumption of applications. Although using power measurement equipments can provide accurate power consumption data, average embedded software developers are reluctant to use power measurement equipments. For many embedded programmers, these equipments are not comfortable to use and they are often too expensive to be widely used by an individual programmer. For energy profiling tools to be widely employed by embedded programmers, the tools should present a familiar user interface without using any special equipment, thus relieving the programmers from learning many different interfaces and equipments.

With multicore/multiprocessor architectures emerging as a practical alternative to a traditional single-CPU architecture, tools support for performance/energy pro-

1058-9244/09/$17.00 © 2009 – IOS Press and the authors. All rights reserved

filing and optimization is getting more important in building efficient mobile embedded applications. In order to efficiently utilize multiple cores on mobile embedded systems, programmers need to understand the power and performance characteristics of their programs in multiple levels (such as the per thread, per core, and per function levels). Since multiprocessor-based systems typically operate on top of an operating system, performance/power analysis of embedded applications is almost impossible without an adequate tools support.

Furthermore, as pointed out in [2], in multiprocessor-based embedded systems, automatic optimization support is becoming more important because an efficient implementation often requires to explore a large design space. For example, when several multi-threaded applications are executed simultaneously, determining the optimal number of threads for each co-running application is a challenging task. Therefore, an automatic optimization function should be an integral part of a performance/power profiling tool for mobile multiprocessor applications.

In this paper, we present ePRO-MP, which profiles and optimizes both performance and energy consumption of multi-threaded applications for ARM11 MPCore-based embedded systems, satisfying the tools requirements discussed above. The main contributions of our work can be summarized as follows. First, ePRO-MP provides both performance profiling and energy profiling for multi-threaded applications running on embedded multiprocessors. Using ePRO-MP, programmers can achieve not only high-performance but also low-energy consumption when developing parallel programs. In addition, developers can find bottlenecks easily because ePRO-MP presents the analysis results in the program, thread, and function levels. Second, ePRO-MP is based on a model-based energy profiling approach which does not require an extra power measurement equipment, thus making ePRO-MP more accessible for average embedded programmers to analyze power/energy consumption of their programs. Finally, ePRO-MP supports a limited automatic optimization function. In this paper, we demonstrate that ePRO-MP can be effective in determining the number of co-running threads for two multi-threaded applications. In this case study, the energy consumption and execution time are improved by 4.1% and 6.1%, over a baseline version respectively. We also use ePRO-MP's automatic optimization function to optimize a producer-consumer application using a matrix multiplication program (as a producer) and a matrix transpose program (as a consumer). In this example, ePRO-MP explores two kinds of problem space: one for determining the number of threads for the producer and consumer and the other one for determining the tile size for the matrix multiplication job (when a simple tiling technique is applied). The experimental results of this case study show that the energy consumption and the execution time are improved by 43.3% and 60.5%, over a baseline version respectively.

The rest of this paper is organized as follows. In Section 2, we survey previous profiling tools and energy profiling techniques. Section 3 describes the overall architecture of the ePRO-MP tool. The performance profiling module and energy profiling module are described in detail in Sections 4 and 5, respectively. Section 6 reports our experiences of using the automatic optimization function with two examples. Finally, Section 7 concludes with a summary and directions for future works.

2. Related work

There have been several investigations on performance or energy profiling for multi-threaded applications. SCALEA [13] presented a performance analysis tool for distributed and parallel applications. SCALEA instruments user programs running on SMP clusters and finds the performance bottlenecks. PAPI [7] provides the tool designer and application programmer with a consistent interface and methodology for using the hardware performance counter supported in most major microprocessors. PerfSuite [10] provides tools, utilities, and libraries for software performance analysis without extensive source code changes. In this paper, ePRO-MP uses PAPI and PerfSuite by modifying it for ARM11 MPCore. On the other hand, Hsu [9] analyzed the power consumption of parallel programs on the Beowulf cluster. This measurement-based profiling can reduce the CPU power consumption. These existing tools for distributed and parallel applications, however, focus on either performance profiling or energy profiling. In mobile embedded application, however, the performance and the energy consumption should be considered together.

Existing energy profiling techniques can be divided into three groups: simulation-based techniques, measurement-based techniques, and regression-based modeling techniques. SimplePower [15] and Wattch [4] are examples of the simulation-based approach. Energy profiling using simulators is time consuming as

well as inaccurate. The measurement-based approach (e.g., ePRO [3] and SES [12]) profiles the energy consumption of a target application using a power measurement equipment. Although the accuracy of energy profiling can be significantly improved under the measurement-based approach, this approach is usually not widely adopted among software developers because it requires an expensive power measurement equipment.

In order to produce accurate profiling results without using a power measurement equipment, regression-based modeling techniques were proposed for energy profiling. Contreras [6] proposed a power prediction model for Intel PXA255 processors using hardware performance counters. Through a linear regression analysis, the power model is estimated using five performance events captured from hardware performance counters. While this approach produces an accurate result, it was limited to a single processor system.

ePRO-MP is distinct from existing tools in several aspects (as will be discussed later). ePRO-MP provides both performance profiling and energy profiling, which help developers build multi-threaded high-performance applications with low-energy consumption. Especially, to avoid the limitations of measurement-based techniques, our tool extends a regression-based modeling technique for energy profiling to multiprocessor-based embedded systems.

3. Overview of ePRO-MP

An overall architecture of ePRO-MP is shown in Fig. 1. ePRO-MP components are divided into 2 parts: softwares which are executed in a target system and an analysis part which can be deployed in any other machine (host system). In the current implementation, our target system is ARM11 MPCore [1] where four ARM11 cores are integrated on a single chip. For performance monitoring, ARM11 MPCore supports various types of hardware performance counters for each core. By using the hardware performance counters of each core, ePRO-MP estimates various performance metrics such as the cache miss rate and IPC. In the target system, three logical modules are running: a target application, an operating system, and a performance profiling module. The target application is the program to be profiled and in the current version, multi-threaded parallel programs using the POSIX thread library are assumed to be main target programs. For an OS, we use Linux 2.6 for ARM11 MPCore which on the target application is running. The Linux kernel was slightly modified for performance profiling. However, it works well when the kernel version is updated because only a small part of the scheduler in the linux kernel should be patched. The performance profiling module, which collects performance data during runtime, is executed with the target application. (We will describe this module in detail in the next section.)

Once the target application completes its execution, the collected performance data are transferred to a host system for an analysis. The host side components of

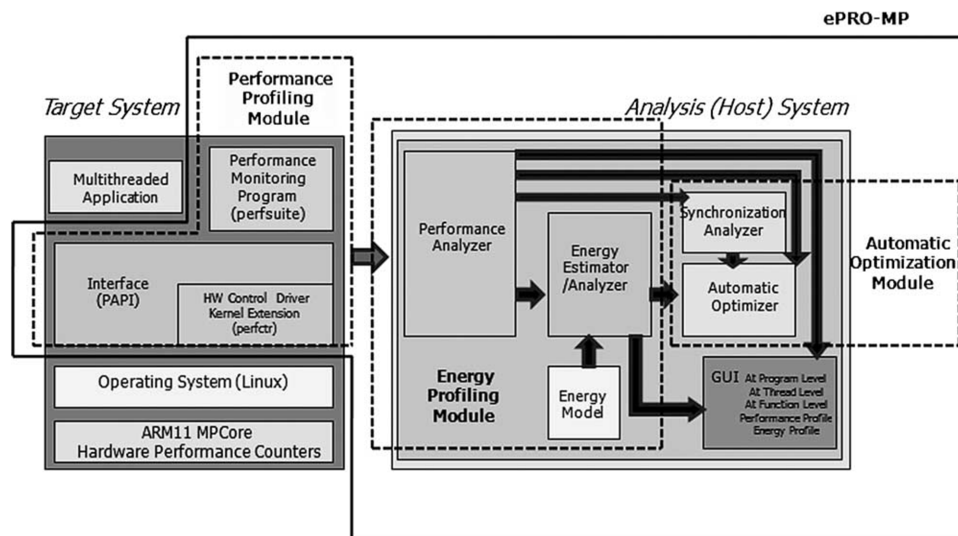


Fig. 1. An architectural overview of ePRO-MP.

ePRO-MP includes six main modules: performance and energy analyzer modules, graphical user interface module, synchronization analyzer module, and automatic optimization module. The collected performance data is analyzed by two analyzers, performance analyzer and energy analyzer. The performance analyzer classifies and arranges the performance profiling results. On the other hand, the energy analyzer applies the energy model, which is developed offline (by following the procedure described in Section 5), to the performance profiling data to estimate the energy consumption. Both performance and energy consumption analysis results are presented in multiple levels (e.g., the thread level and function level) so that developers can identify bottlenecks in target applications. The current version of ePRO-MP adopts the GUI of the Eclipse platform. Users can profile the performance and energy consumption of a target application easily by selecting a profiling menu of GUI. As a hint for optimization of multi-threaded applications, the synchronization analyzer module profiles the waiting times among coordinating threads. Finally, the automatic optimization module uses the feedback from the profiling results for various optimizations.

4. Performance profiling module

Performance profiling depends on hardware performance counters available in microprocessors. The counter values are collected by a performance monitoring program which runs with a target application. In order to make a performance monitoring process portable across different target systems, we used a layered organization for the performance profiling module. Specifically, two layers were added: one is the hardware control driver layer and the other one is the interface layer between the monitoring program and the hardware control driver.

The hardware control driver is the lowest layer of the performance profiling module. To allow user level programs to access the hardware counters, the operating system should provide a device driver to initialize, start, stop, and read the hardware counters. For the hardware control driver, we adopted Performance Monitoring Counters Driver [11], *perfctr*. Since *perfctr* was not supported in ARM11 MPCore, we ported the existing *perfctr* to ARM11 MPCore.

The interface layer provides an interface between hardware control driver and the performance monitoring program. For this layer, we ported PAPI to our multiprocessor target system. The performance monitoring program, which runs with the target application, is based on Perfsuite. When using this performance monitoring program, users do not have to instrument their code. After running the target application with the performance monitoring program, the profiling result files are created in the XML format. There is an overhead due to the performance monitoring program because it should run with a target application. However, it is less than 1%, which is negligible.

5. Energy profiling module

ePRO-MP employs the regression-based modeling approach for energy profiling. Although regression-based energy models for single processors have been proposed before, there have not been investigations on regression-based energy models for multiprocessor embedded systems. We extend the regression-based energy model for single processor systems by adding the characteristics which multiprocessor systems have, such as the number of cache coherence transactions and the number of shared L2 cache accesses. In this section, we describe an energy model development procedure based on the linear regression analysis, which uses the performance hardware counter. Following the proposed methodology, we derive an energy model for ARM11 MPCore. As shown in Fig. 2, our methodology consists of four main steps. After the energy model is developed offline, it can be used in the energy analysis with performance profiling data.

Random program generation: In order to build an energy model that can accurately predict the en-

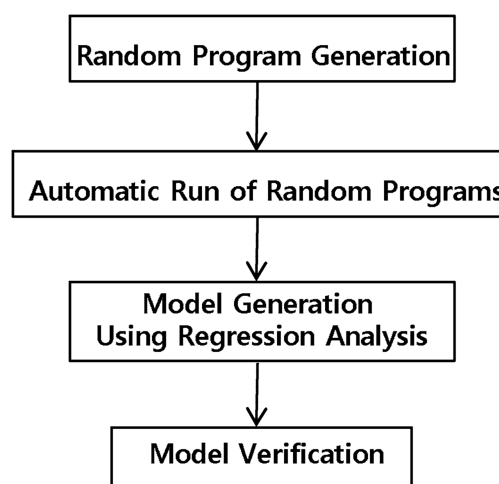


Fig. 2. A procedure for deriving an energy model.

ergy consumption of an arbitrary program based on hardware performance counters, we generate various random test programs with different execution characteristics (e.g., memory-intensive programs, CPU-intensive programs). A random test program is generated by combining several basis program segments. The basis program segments represent different program execution behaviors. In the current version, there are 30 basis program segments. In the current version of ePRO-MP, about 200 random test programs were generated to build an energy model.

Automatic run of random programs: In this step, training data for regression analysis are produced by executing the random programs generated by the random program generator. Because we do not know an exact regression model yet, all the hardware performance counters are collected at this step. At the same time, the energy consumption value is gathered from a power measurement equipment. (Note that a power measurement equipment is necessary only for this step. Once the energy model is constructed, when ePRO-MP is used by developers, there is no need for a power measurement equipment.) Figure 3 shows the setup for this step using NI DAQ-6016. We have used a script to automate the measurement and collection tasks.

Model generation using regression analysis: Regression analysis is applied to the training data gathered in the previous step. While multiprocessors can support various hardware performance counters, not all the performance events are related with the energy consumption of the processors. In order to construct a linear regression model, we start with a very general linear model that includes all the hardware performance counters. Using the model adequacy test of regression analysis, we eliminate those hardware performance counters that are not significantly related to energy consumption. For the current implementation, five performance events, the number of instructions (*Instr*), the number of L1 data cache accesses (*DL1Access*), the number of L2 cache accesses (*L2Access*), the number of stall cycles due to data dependency (*DataDep*), and the number of coherence transactions (*cohTrans*), are selected for our power model. That is, the power model for ARM11 MPCore is given as follows:

$$\begin{aligned} \mathit{Power} = {} & A \times (\mathit{Instr}/\mathit{time}) \\ & + B \times (\mathit{DL1Access}/\mathit{time}) \\ & + C \times (\mathit{L2Access}/\mathit{time}) \\ & + D \times (\mathit{DataDep}/\mathit{time}) \\ & + E \times (\mathit{CohTrans}/\mathit{time}) + F_{const}. \end{aligned} \quad (1)$$

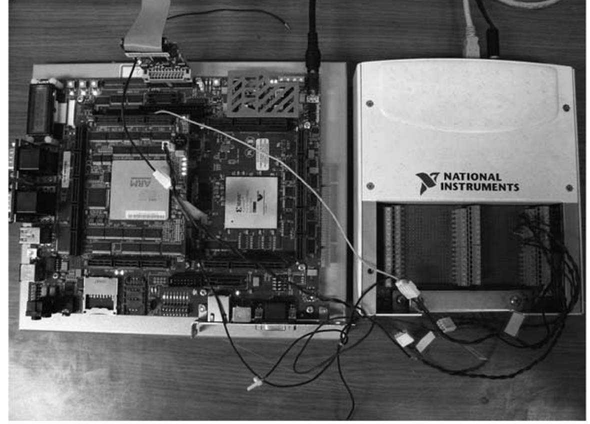


Fig. 3. A setup for power measurement and performance data collection.

Table 1

Coefficients values of our power model

Performance event	Coefficient	Performance event	Coefficient
Instr	2.56E−07	*DataDep*	−3.97E−07
DL1Access	9.32E−07	*CohTrans*	−1.99E−07
L2Access	2.76E−05	*Const*	7.89E+02

We have used SAS9 Enterprise Miner to derive the energy model. Table 1 lists five coefficients values of the power model 1. In order to compute the energy consumption of a thread running on a single core, we use $\frac{1}{4} \times F_{const}$ as a constant term in Eq. (1), assuming that all four ARM11 cores contribute equally to F_{const}. Other performance counter values are all collected per core basis. Note that we cannot measure the power consumption of each core because our current target system, ARM11 MPCore, provides only a power probe to measure the entire chip power consumption.

Model verification: The derived model should predict the energy consumption of arbitrary programs with different characteristics. To evaluate our energy model, we verified our power model using several benchmark programs as well as the random test programs used in the model generation step. Figure 4 shows the comparison results of the average measured power (by NI power measurement equipment) and predicted power (by our power model) for six SPLASH-2 benchmark programs [14] and two sorting algorithms. As shown in Fig. 4, our model is very accurate. The average prediction error was less than 2% while the maximum prediction error was 4.9% for the Quick sorting algorithm.

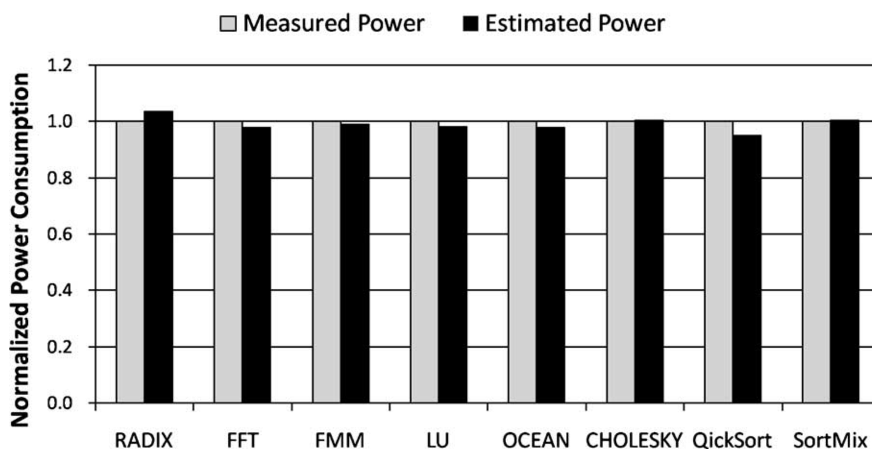


Fig. 4. Comparison results of measured power and predicted power.

6. Profile-based automatic optimizer

As the number of cores and the number of threads increase, supporting automatic optimization becomes more important because an optimization problem space becomes often too complex for average programmers to deal with efficiently. ePRO-MP supports a couple of automatic optimizations to relieve the programmers' burden of time-consuming optimization problems. Targeting multi-threaded applications, we focus on two particular situations for automatic optimizations. First, we consider when several multi-threaded applications run simultaneously. (For example, reading/editing multimedia messages while browsing/processing digital pictures could be such a case.) Since co-running applications and their threads share system resources such as shared L2 cache and memory, often competing with each other, determining the number of threads optimally for each co-running multi-threaded program has a large implication on the overall system performance. One important requirement is to minimize the interference among competing applications for the shared resources. Second, we consider when a single multi-threaded application is expected to run alone. In this case, interactions/load balancing among co-running threads affect the overall application performance significantly, especially at synchronization points among the co-running threads. In this paper, we focus on the producer-consumer relationship among the threads, which is one of the most commonly used parallel implementation models used in implementing multi-threaded applications.

In this section, we describe ePRO-MP's restricted automatic optimization function using two examples: *co-running applications optimization* and *producer-consumer application optimization*.

6.1. Co-running applications optimization

In this section, we discuss the optimal thread allocation problem for two multi-threaded applications, matrix multiplication (MM) and insertion sort (IS) as one example of co-running applications. We assume that the baseline thread allocation policy to assign is four threads per each program, because programmers usually parallelize their programs into the maximum number of physical cores (in our target system, which is four) to fully exploit the available cores. When 4-threaded MM and 4-threaded IS are executed simultaneously, the profiled performance and the energy consumption results are shown in Fig. 5 using the ePRO-MP's thread level GUI. The block A indicates the four threads of MM while the block B indicates the four threads of IS. The total energy consumption is 1,155,528 mJ which is computed by summing up the values in the Energy column.

Our automatic optimizer tries to find a near-optimal number of threads using a simple heuristic. Starting from using one thread for each program, the optimizer increments the number of threads by one at a time and executes the applications. After each execution, using the profiled data, the automatic optimizer compares the performance or the energy consumption of the current application with the previous execution. When there is no more improvement in both the performance and the energy consumption, the optimizer stops searching and finally compares the current optimal profiling information with that of the baseline (4, 4). Figure 6 illustrates the problem space and its exploration orders of the optimizer. The pairs and the arrows indicate the number of threads of each co-running program and exploration orders, respectively. In this example, the search stops

Program View | Function View | Thread View

Refresh

Brief

Thread ID	Cycle	IPC	DL1MissRatio(%)	SPI	Energy(mJ)
thread1	181,402,407,285	0.215	3.870	3.678	163,708.420
thread2	181,258,645,863	0.216	3.880	3.498	166,260.150
thread3	180,923,039,271	0.216	3.870	3.653	163,642.690
thread4	190,283,707,395	0.205	3.890	3.505	174,946.740
thread5	112,002,049,989	0.470	0.990	1.159	121,288.450
thread6	111,668,147,760	0.471	0.980	1.134	121,507.060
thread7	112,392,556,940	0.469	0.990	1.131	122,356.300
thread8	112,420,600,848	0.468	0.990	1.162	121,818.490

A (thread1–thread4), B (thread5–thread8)

Fig. 5. Profiling result of (4, 4) thread allocation.

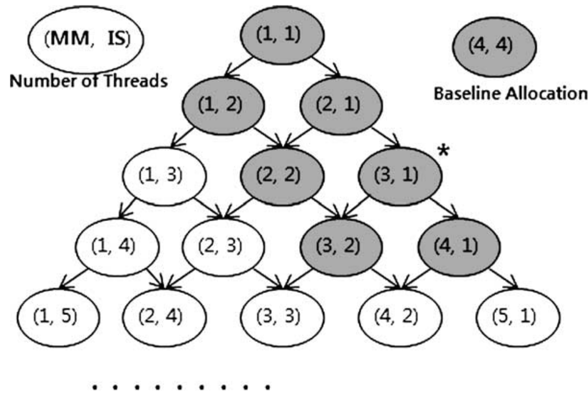


Fig. 6. Problem space of our case study.

after eight different configurations were executed. The asterisked pair, three threads for MM and one thread for IS, is the near-optimal number of threads determined by the optimizer.

Figure 7 shows the result of the automatic optimization when three threads are used for MM and one thread for IS. Over the baseline (4, 4) configuration, the (3, 1) thread configuration for MM and IS improves the total execution time by 6.1% and reduces the total energy consumption by 4.1%. Although the energy improvement is less than the maximum prediction error 4.9%, the prediction error for Matrix Multiplication and Insert Sorting is much less around 1%, which makes the energy improvement meaningful.

Furthermore, in order to better understand why (3, 1) configuration worked better than (4, 4) configuration, we profiled the performance and the energy consumption of each application alone by changing the number of threads from one to four. From the analysis results, we found two interesting trends. Multi-threaded MMs show better performance and energy consumption than a single-threaded MM. Because a matrix array is shared among multiple threads in MM, the higher the number of threads in MM is, the higher the L2 cache hit ratio is in general. On the other hand, IS is not sensitive to the number of threads because each thread sorts its local array with little sharing of the data among different threads. Therefore, increasing the number of threads for MM and decreasing the number of threads for IS generally improves the energy consumption. The other finding is that the speedup of programs is bounded by the number of cores not threads. For four cores of ARM11 MPCore, the four threads, three for MM and one for IS, are sufficient.

6.2. *Producer-consumer application optimization*

In this section, we describe the automatic optimization support for producer-consumer applications. As pointed out in [8], for high-performance producer-consumer applications, an efficient implementation of synchronization between the producer and consumer is critical. (For example, if a consumer is much faster than the corresponding producer, the consumer spends a large number of cycles waiting for the results of the producer, thus wasting a significant amount of energy as well as the CPU cycles.) In multiprocessor embedded systems, the waiting times and the total execution time can be reduced by allocating different number of threads for the producer and consumer and balancing the speed of the producer and consumer. As the number of cores as well as the number of producers and consumers increase, determining the optimal number of threads for the producer and consumer becomes a challenging task.

Furthermore, achieving the high speed of each thread itself can lead to the high-performance application. For example, if there is a matrix multiplication job as either the producer or consumer, finding the op-

Program View | Function View | Thread View

Refresh

Brief

Thread ID	Cycle	IPC	DL1MissRatio(%)	SPI	Energy(mJ)
thread1	228,772,422,056	0.227	3.850	3.390	211,664.800
thread2	229,130,279,563	0.228	3.850	3.399	211,646.840
thread3	228,897,090,718	0.229	3.850	3.365	211,949.190
thread4	423,198,349,513	0.497	0.940	1.013	472,384.910

A B

Fig. 7. Profiling result of (3, 1) thread allocation.

timal size of the matrix tile can improve the cache efficiency, thus achieving high performance and low energy consumption. (Tiling [5] is a cache-aware optimization technique which divides the entire matrix into small matrices that fit better with the cache size, thus reducing the number of cache misses significantly.) Finding the optimal tile size (that maximizes the number of cache hits) also requires to explore another large problem space. The ePRO-MP's automatic optimizer can help programmers find the near-optimal number of threads for the producer and consumer and the tile size for the matrix multiplication job.

As a case study, we executed a producer-consumer application using a matrix multiplication task (MM) as a producer and a matrix transpose task (MT) as a consumer. To apply two optimization strategies discussed above, the application was implemented to support multiple threads (for the thread allocation problem) and different tile sizes (for the tile size selection problem). We assume that the baseline configuration is $(2, 2)$ thread allocation without the tiling technique used. When programmers do not know the speed balance in advance between the producer and consumer, they might start the optimization from allocating the same number of threads to each side.

Our automatic optimizer tries to find the near-optimal number of threads for MM and MT and the near-optimal tile size for MM using a simple heuristic shown in Fig. 8. At first, the automatic optimizer starts to find the near-optimal number of threads for the producer and the consumer. After the application with the $(1, 1)$ thread allocation is executed, the waiting times of both the producer and consumer are calculated using the synchronization analyzer. If the waiting time of the consumer is longer than that of the producer, the speed of the producer should be increased (by allocating more threads) to reduce the waiting time of the consumer, and vice versa. We call a task τ_{slow} when the task (τ_{slow}) makes other tasks wait. Starting from one thread, the optimizer increments the number of threads for τ_{slow} by one at a time and executes the applications. After each execution, the performance or the energy consumption is profiled and compared with the previous execution. When there is no more improvement in the performance and the energy consumption, for the second problem space exploration, the optimizer tries to find the near-optimal tile size for the matrix multiplication job. Starting from the original matrix size as the tile size, the optimizer decrements the tile size by $\frac{1}{n}$ of the original matrix size at a time and executes it (with the thread allocation determined in the first problem space exploration). The variable n, initially set to two, is incremented by one at a step. After each execution, the performance or the energy consumption is profiled and compared with the previous execution. When there is no more improvement, the optimizer explores repeatedly the thread allocation problem space for the producer and consumer because the exploration result from the tile size selection problem can make other thread configurations better. If there is no change in the thread allocation, the optimizer stops searching and finally outputs the number of threads and the tile size for the target application. In case of the MM-MT application, because MM was much slower than MT, MM was decided to be τ_{slow}. The more threads were allocated to MM, the better performance and energy consumption was measured until the $(4, 1)$ thread configuration. From 500×500 to 100×100 for the tile size (the baseline tile size is same as the original matrix size, 500×500), the performance and the energy consumption were getting better. When the tile size was less 100×100, the performance and the energy consumption started to be degraded and the exploration was finished.

The performance and the energy consumption results of our case study are shown in Fig. 9. The results are normalized to the baseline $(2, 2)$ thread allocation without the tiling technique because programmers might start the optimization from allocating same

```
INPUT: Target Application (TA) (e.g., multi-threaded MM-MT), Original Matrix Size
OUTPUT: Best Alloc. & Best Tile Size
1:   Best Alloc. = (1,1)
2:   Measure Perf./Energy of TA with (1,1) thread allocation (baseline execution)
3:
4:   If (WaitTimeProducer < WaitTimeConsumer)    T_slow = Producer
5:   else   T_slow = Consumer
6:   While (true)
7:       Current Alloc. = Allocate one more thread to T_slow
8:   Measure Perf ./Energy with Current Alloc.
9:   if (Current Alloc. is better Perf ./Energy than Best Alloc.)   Best Alloc. = Current Alloc.
10:  else if (PrevDecidedAlloc == Best Alloc.)    Goto 25.
11:  else break
12:
13:  PrevDecidedAlloc = Best Alloc.
14:  Best Tile Size = Original Matrix Size
15:  SizeAdjuster = 1
16:  Measure Perf ./Energy with Best Alloc. & Best Tile Size
17:  While (true)
18:      While (true)
19:          SizeAdjuster ++
20:          Current Tile Size = Original Matrix Size / SizeAdjuster
21:          if (Original Matrix Size % SizeAdjuster == 0) break
22:      Measure Perf ./Energy with Best Alloc. & Current Tile Size
23:      if (Current Tile Size is better than Best Tile Size)   Best Tile Size = Current Tile Size
24:      else Goto 3.
25:  Return OUTPUT
```

Fig. 8. ePRO-MP's Heuristic for optimizing MM-MT application.

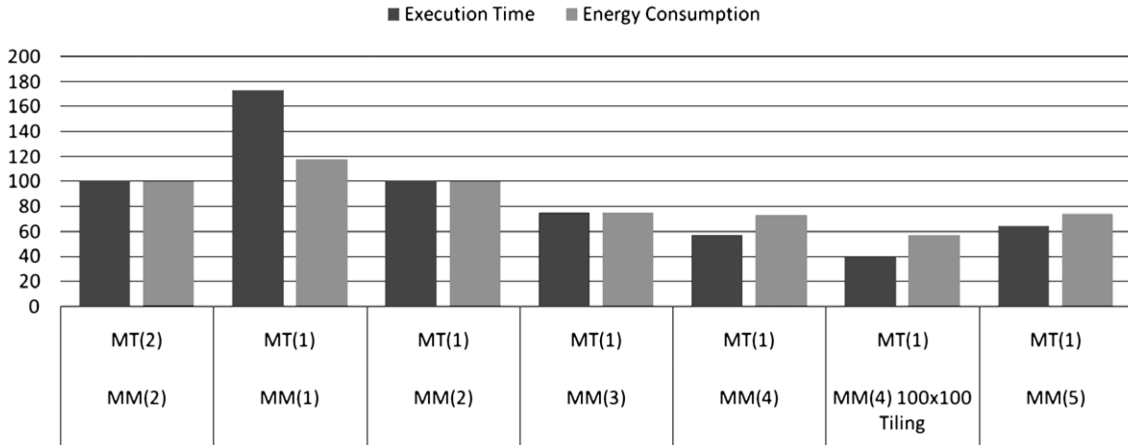


Fig. 9. Optimization result of MM-MT application.

number of threads to each when the programmers do not know the producer and the consumer's loads. Exploring only thread allocation problem space, the (4, 1) thread configuration for MM and MT improves the total execution time by 43.5% and reduces the energy consumption by 26.8%. After the near-optimal tile size is determined, the performance and the energy consumption are further improved by 17% and 16.5%, respectively. By allocating three more threads to MM from the (1, 1) thread allocation, MM's producing speed and MT's consuming speed were balanced. However, because one more thread allocation for MM (the (5, 1) thread allocation) makes MM's speed faster than MT's speed and the entire six threads are not efficiently scheduled to get the four physical cores, the execution time and the energy consumption start to increase. For the tile size, when the tile size close to the L1 cache size was used, its performance was the best.

7. Conclusions

We described ePRO-MP, an energy and performance profiler and optimizer for embedded multiprocessors. ePRO-MP provides both energy profiling information and performance profiling information that can be important in developing high-performance and low-energy embedded multi-threaded applications. One of

main strengths of ePRO-MP is that ePRO-MP can accurately estimate the energy consumption of multi-threaded applications without using extra power measurement equipments. Furthermore, we demonstrated the usefulness of ePRO-MP's automatic optimization capability using the thread allocation problem of two co-running multi-threaded applications. Experimental results show that we can improve the performance and the energy consumption over the baseline thread allocation by 6.1% and 4.1%, respectively. We also use the optimizer for optimizing a producer-consumer application where there is a matrix multiplication job as either the producer or consumer. After the optimizer determined the number of threads for both the producer and consumer and the tile size for the matrix multiplication job, the execution time and the energy consumption were reduced by 60.5% and 43.3%, respectively.

Although the current version of ePRO-MP can be useful in building efficient multi-threaded embedded applications, it can be extended in several directions. First, we plan to implement different kinds of multiprocessor task schedulers in the Linux running on the target system. The default task scheduler used by the current version of ePRO-MP limits the improvement of the performance and the energy consumption because the critical performance factors such as the cache efficiency and waiting times among threads are affected by the scheduling policy as well as the program efficiency itself. Based on the profiled characteristics of the target application, if ePRO-MP can select the best task scheduler for it, the performance or the energy consumption can be more improved. We also plan to develop a more fine-grained automatic optimizer that can work on the compiler option level. For example, we are interested in finding compiler options for given multi-threaded applications that can generate binary executable with the lowest power/energy consumption, possibly employing different compiler options for each thread.

Acknowledgments

This work was supported by the Korea Science and Engineering Foundation (KOSEF) Grant funded by the Korea government (MEST) (No. R0A-2007-000-20116-0). This work was supported by World Class University (WCU) program through the Korea Science and Engineering Foundation funded by the Ministry of Education, Science and Technology (R33-2008-000-10095-0). This work was also supported in part by the Brain Korea 21 Project in 2008 and Samsung Electronics Inc. The ICT at Seoul National University provides research facilities for this study.

References

[1] ARM11 MPCore [Online], available at: http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html.

[2] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams and K.A. Yelick, The landscape of parallel computing research: A view from Berkeley, Technical Report No. UCB/EECS-2006-183, University of California, Berkeley, CA, 2006.

[3] W. Baek, Y. Kim and J. Kim, ePRO: A tool for energy and performance profiler for embedded applications, in: *Proceedings of International SoC Design Conference*, Seoul, Korea, 2004, pp. 372–375.

[4] D. Brooks, V. Tiwari and M. Martonosi, Wattch: A framework for architectural-level power analysis and optimizations, in: *Proceedings of International Symposium on Computer Architecture*, Vancouver, BC, Canada, 2000, pp. 83–94.

[5] S. Coleman and K.S. Mckinley, Tile size selection using cache organization and data layout, in: *Proceedings of Conference on Programming Language Design and Implementation*, La Jolla, CA, 1995, pp. 270–290.

[6] G. Contreras and M. Martonosi, Power prediction for Intel XScale processors using performance monitoring unit events, in: *Proceedings of International Symposium on Low Power Electronics and Design*, San Diego, CA, 2005, pp. 221–226.

[7] Performance application programming interface [Online], http://icl.cs.utk.edu/papi/.

[8] A. Grama, A. Gupta, G. Karypis and V. Kumar, *Introduction to Parallel Computing*, Addison-Wesley, Boston, MA, 2002.

[9] C. Hsu and W. Feng, A feasibility analysis of power awareness in commodity-based high-performance clusters, in: *Proceedings of International Conference on Cluster Computing*, Boston, MA, 2005, pp. 1–10.

[10] Perfsuite [Online], http://perfsuite.ncsa.uiuc.edu.

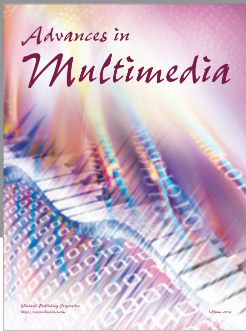
[11] Linux x86 performance monitoring counters driver [Online], http://www.csd.uu.se/mikpe/linux/perfctr/.

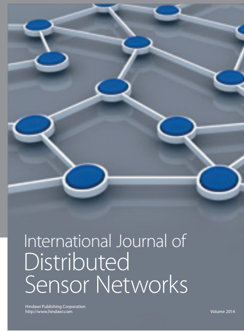
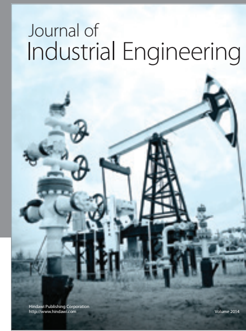
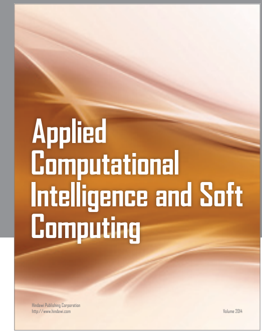
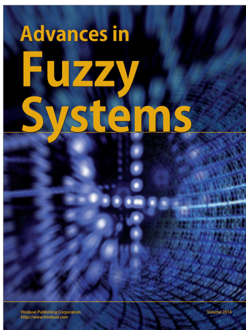
[12] D. Shin, H. Shim, Y. Joo, H. Yun, J. Kim and N. Chang, Energy-monitoring tool for low-power embedded programs, *Design and Test of Computer* **19**(4) (2002), 7–17.

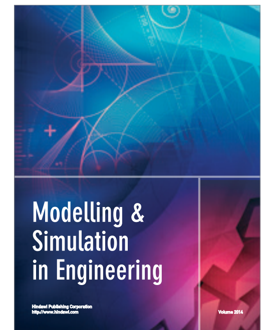
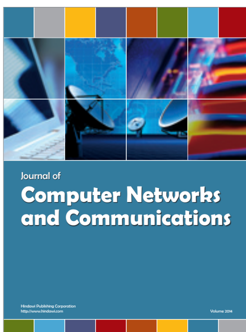
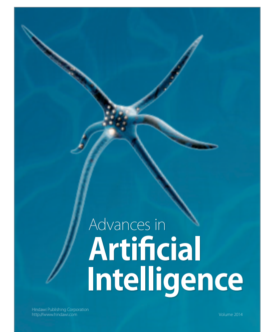
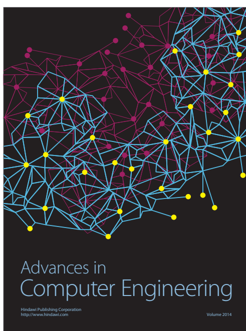
[13] H.L. Truong and T. Fahringer, SCALEA: A performance analysis tool for distributed and parallel programs, in: *Proceedings of Euro-Par Conference on Parallel Processing*, Parderborn, Germany, 2002, pp. 75–85.

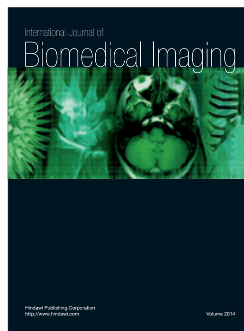
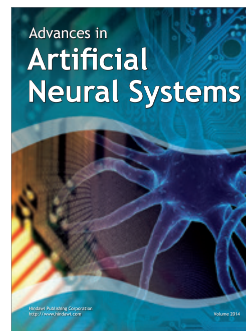
[14] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh and A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, in: *Proceedings of International Symposium on Computer Architecture*, St. Margherita Ligure, Italy, 1995, pp. 24–36.

[15] W. Ye, N. Vijaykrishnan, M. Kandemir and M.J. Irwin, The design and use of simple power: A cycle accurate energy estimation tool, in: *Proceedings of Design Automation Conference*, Los Angeles, CA, 2000, pp. 340–345.

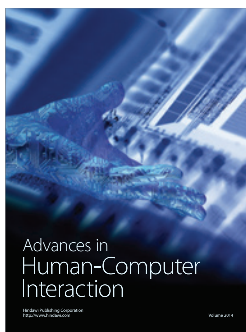
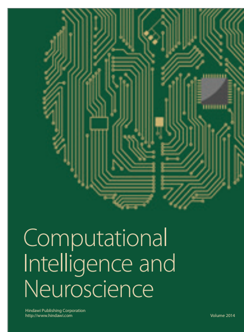

Advances in
Multimedia

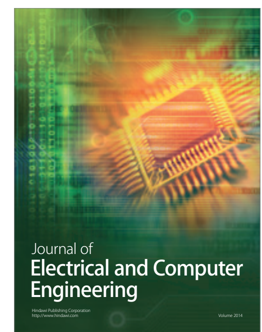
The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems

Hindawi
Submit your manuscripts at
http://www.hindawi.com

Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering