Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT
J. Parallel Distrib. Comput. 64 (2004) 740–746
Journal of Parallel and Distributed Computing
http://www.elsevier.com/locate/jpdc

A computation offloading scheme on handheld devices

Cheng Wang* and Zhiyuan Li

Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA

Received 9 September 2003; accepted 22 October 2003

Abstract

In this paper, we present a computation offloading scheme on handheld devices. This scheme partitions an ordinary program into a client–server distributed program, such that the client code runs on the handheld device and the server code runs on the server. Our partition analysis and program transformation guarantee correct distributed execution under all possible execution contexts. We give a polynomial time algorithm to find the optimal program partition for given program input data. We use an option-clustering approach to handle different program partitions for different program execution options. Experimental results show significant improvement of performance and energy consumption on an HP IPAQ handheld device through computation offloading.
© 2003 Published by Elsevier Inc.

Keywords: Task partition; Wireless network; Distributed computing; Data consistency; Edge profiling; Program transformation

1. Introduction

Computation offloading is important for handheld devices. Certain applications are simply too time consuming to run on handheld devices. Therefore, the only feasible way to use those application programs is to offload all or part of the computation to a more powerful machine [10,6]. Recent years have seen widespread deployment of wireless LAN in campuses, in corporation buildings, and at other locations. In such environment, handheld devices with wireless LAN connection can access data in the user's regular desktop computer or a trusted server. Computation can be offloaded to these machines. For devices with wide area wireless connection like CDMA, the access to trusted machines has a much wider range.

This paper presents a computation offloading scheme on handheld devices. Our computation offloading scheme consists of a compiler-based tool which partitions an ordinary program into a client–server distributed program, such that the client code runs on the handheld device and the server code runs on the server. Our partition analysis and program transformation guarantee correct distributed execution under all possible execution contexts.

We build a constraint system for our computation offloading. Task mappings, data accesses and data validity states are represented simultaneously in the constraint formulation, which is a new approach to global cost modeling for distributed computing. Besides the computation cost and data communication cost, our cost analysis also considers the cost for the run-time bookkeeping which is necessary for correct distributed execution.

We give a polynomial time algorithm to find the optimal program partition for the given program input data. For cases in which the optimal program partitioning varies with different execution options, we compute optimal program partitions for a training set of execution options and determine which partition to use at run-time subject to the execution options.

We implement our computation offloading scheme in GCC. Experimental results show that our computation offloading scheme can significantly improve the performance on an HP IPAQ 3970 handheld device. As a side effect, the energy consumption on that handheld device is reduced proportionally.

2. Computation offloading scheme

Our computation offloading scheme divides the whole program computation into tasks. We do not restrict

*Corresponding author.
E-mail addresses: wangc@cs.purdue.edu (C. Wang), li@cs.purdue.edu (Z. Li).

0743-7315/$ - see front matter © 2003 Published by Elsevier Inc.
doi:10.1016/j.jpdc.2003.10.005

tasks at any specific level. A task can be a basic block, a loop, a function or a group of closely related functions. We divide the tasks into server tasks and client tasks such that server tasks run on the server and client tasks run on the handheld device.

It is important to note that although we execute client tasks and server tasks in a distributed way, we do not seek to execute a client task and a server task simultaneously. This is due to the significant computation speed gap between the two. Little can be gained in reality by parallel execution in this manner. In the absence of simultaneous tasks, program execution simply follows the original sequential control flow.

To transform a normal program into a distributed program for computation offloading, we must guarantee the correct control and data flow relations in the original program for all possible program execution contexts. Difficulties arise when programs contain run-time control and data flow information which is undecidable at compile time.

We take an approach that makes use of both the static information and the run-time information. Statically, we produce an abstraction of the given program in which all memory references are mapped to references to *abstract memory locations*. The program abstraction contains only statically available information. We then perform partition analysis on the program abstraction to statically determine the task allocations and data transfers of abstract memory locations subject to the control and data flow defined over the program abstraction. Our program transformation will insert efficient run-time bookkeeping codes for the correct mapping between abstract memory locations and run-time physical memory. The resulting distributed program guarantees the correct control and data flow at run-time.

3. Partition Analysis

3.1. Program abstraction

We build the directed control-flow graph $CFG = (V, E)$ for the program such that each vertex $v \in V$ is a basic block and each edge $e = (v_i, v_j) \in E$ represents the fact that v_j may be executed immediately after v_i.

We abstract all the memory accessed (including code and data) by a program at run time by a finite set of *typed abstract memory locations*. The abstraction of run-time memory is a common approach used by pointer analysis techniques [2,11] to get conservative but safe point-to relations. The type information is needed to maintain the correct data endians and data addresses during data transfer between the server and the handheld device. Each memory address is represented by a unique abstract memory location, although an abstract memory location may represent multiple memory addresses because a single reference in the program may cause multiple memory references at run time. We use D to denote the set of all abstract memory locations d.

With point-to analysis, we conservatively identify the references to abstract memory locations at each program point. For example, to get the value of $*x$, the program reads x as well as the abstract memory locations which x points to. To write into $*x$, the program reads x and writes all the abstract memory locations which x points to.

3.2. The constraint system

Dynamically, each basic block v and each flow edge e can have many execution instances. We define $f(v_i, v_j)$ as the execution count for the flow edge $e = (v_i, v_j)$ and $g(v)$ as the execution count for basic block v. The values of f and g may vary in different runs of the same program when using different input data. However, the following constraint always holds:

Constraint 1. For any basic block $v_i \in V$,

$$g(v_i) = \sum_{e=(v_i,v_j) \in E} f(v_i, v_j) = \sum_{e=(v_j,v_i) \in E} f(v_j, v_i).$$

Since each task will be mapped either to the server or to the client, but not both, the task mapping can be represented by a boolean function M such that $M(v)$ indicates whether basic block v is mapped to server. We define the M function for basic blocks because the computation of a basic block has little variance. All the instructions in a basic block always execute together.

By program semantics, the mapping of certain basic blocks may be fixed. For example, in interactive applications, basic blocks containing certain I/O functions are required to execute on the handheld device. Moreover, to partition the computation at level higher than a basic block, certain basic blocks are required to be mapped together. For example, for function-level partitioning, all the basic blocks in one function are required to be mapped together. The following constraint reflects these requirements.

Constraint 2. If basic block v is required to execute on the handheld, then $M(v) = 0$. If basic block v is required to execute on the server, then $M(v) = 1$. If basic blocks v_1 and v_2 are required to be mapped together, then

$$M(v_1) \Leftrightarrow M(v_2).$$

We use two boolean variables to represent data access information for abstract memory locations d such that $N_s(d)$ indicate whether d is accessed on the server and

$N_c(d)$ indicate whether d is *not* accessed on the handheld. By definition, we have:

Constraint 3. if d is accessed within basic block v then $M(v) \Rightarrow N_s(d)$ and $N_c(d) \Rightarrow M(v)$.

Distributed shared memory (DSM) systems [1] keep track of the run-time data validity states to determine the data transfers at run time. We analyze data validity states statically for the abstract memory locations to determine the data transfers statically. Due to constraint 1, it is obvious that inserting data transfers on control flow edges is always not worse, sometimes better, than inserting them in basic blocks. We consider the validity states of abstract memory location d before the entry and after the exit of each basic block v such that: $V_{si}(v,d)$ indicates whether the copy of d on server is valid before the entry of v. $V_{so}(v,d)$ indicates whether the copy of d on server is valid after the exit of v. $V_{ci}(v,d)$ indicates whether the copy of d on client is *not* valid before the entry of v. $V_{co}(v,d)$ indicates whether the copy of d on client is *not* valid after the exit of v.

For data consistency, the local copy of d must be valid before any read operation on d. After a write operation on d, the copy of d on the current host becomes valid and the copy of d on the opposite host becomes invalid. If there is no write operation on d within a basic block v, then the local copy of d is valid after the exit of basic block v only if it is valid before the entry of v. In cases where d is possibly or partially written in a basic block, we conservatively require d to be valid before the write. Otherwise, d may be inconsistent after the write. The following constraint is introduced for data consistency.

Constraint 4. If basic block v has a (possibly or definitely) upward exposed read of d, then $M(v) \Rightarrow V_{si}(v,d)$ and $V_{ci}(v,d) \Rightarrow M(v)$. If d is (possibly or definitely) written in basic block v, then $V_{so}(v,d) \Leftrightarrow M(v)$ and $M(v) \Leftrightarrow V_{co}(v,d)$. If d is definitely not written in basic block v, then $V_{so}(v,d) \Rightarrow V_{si}(v,d)$ and $V_{ci}(v,d) \Rightarrow V_{co}(v,d)$. If d is possibly or partially written in basic block v, then $M(v) \Rightarrow V_{si}(v,d)$ and $V_{ci}(v,d) \Rightarrow M(v)$.

3.3. Cost analysis

There are four kinds of costs in our computation offloading scheme: computation cost for task execution, scheduling cost for task scheduling, bookkeeping cost for run-time bookkeeping, and communication cost for data transfer.

If we associate a computation cost $c_c(v)$ for each execution instance of basic block v running on the client, and a computation cost $c_s(v)$ for each execution instance of basic block v running on server, we get the total computation cost:

$$\sum_{v \in V} M(v)c_s(v)g(v) + \neg M(v)c_c(v)g(v). \tag{1}$$

If we associate a scheduling cost c_r for each instance of task scheduling from the client to the server, and a scheduling cost c_l for each instance of task scheduling from the server to the client, we get the total scheduling cost

$$\sum_{(v_i,v_j) \in E} \neg M(v_i)M(v_j)c_r f(v_i,v_j) + \neg M(v_j)M(v_i)c_l f(v_i,v_j). \tag{2}$$

Our program transformation discussed later performs run-time bookkeeping for allocations and releases of data *accessed by both hosts*. We assume each data release corresponds to a previous data allocation, so we include the release cost in the allocation cost. If we associate a bookkeeping cost c_a with each instance of data allocation, and let $A(v)$ denote the set of abstract memory locations d that are allocated in basic block v, we get the total bookkeeping cost

$$\sum_{d \in D, d \in A(v)} \neg N_c(d)N_s(d)c_a g(v). \tag{3}$$

We can derive the data transfer information from data validity states. On each edge (v_i,v_j), if $V_{so}(v_i,d) = 0$ and $V_{si}(v_j,d) = 1$, then the copy of d on the server is invalid after the exit of v_i but becomes valid before the entry of v_j. So there is a data transfer of d from the client to the server on edge (v_i,v_j). Similarly, if $V_{co}(v_i,d) = 1$ and $V_{ci}(v_j,d) = 0$, then there is a data transfer of d from the server to the client on edge (v_i,v_j). If we associate a communication cost $c_d(d)$ for each instance of data transfer of d from the server to the client, and a communication cost $c_u(d)$ for each instance of data transfer of d from the client to the server, then the total communication cost is

$$\sum_{(v_i,v_j) \in E} \neg V_{so}(v_i,d)V_{si}(v_j,d)c_u(d)f(v_i,v_j) + \neg V_{ci}(v_j,d)V_{co}(v_i,d)c_d(d)f(v_i,v_j). \tag{4}$$

3.4. Partition algorithm

For the given program input data, edge profiling techniques [3] can easily get f and g in formulas (1)–(4). The optimal program partitioning problem can then be expressed as

Problem 1. Find boolean values for M, N_s, N_c, V_{si}, V_{so}, V_{ci} and V_{co} subject to constraints 2–4 and minimize the sum of total cost (1)–(4).

Problem 1 can be reduced to a single-source single-sink *min-cut network flow problem* [4] which can be

solved in polynomial time. It is possible that the optimal program partitions vary with different program inputs. Problem 1 can be treated as a parametric problem with parameters f and g that satisfy constraint 1. However, the parametric problem 1 can be reduced from a 2-*path problem* [4] and is hence NP-hard. We omit the proofs of these claims due to the space limit.

We use an option-clustering heuristic for the parametric problem 1. Our heuristic groups a training set of options into a relatively small number of clusters and prepares one partition for each cluster such that, for any option in the training set, the cost difference between the prepared partition and its optimal partition is within a given error-tolerance ratio. For a program with r independent options, all the possible program options form an r-dimensional option space. We divide the whole option space into subspaces according to the clustering of the training set. At run time, we check to see which subspace the option belongs to, and the program will run based on the corresponding partition. We omit the details in the paper due to the space limit.

4. Program transformation

The task execution control and data transfer are implemented by message passing. At any moment, only one host (active host) performs the computation. The other host (passive host) waits in a message processing function which acts in accordance to the incoming messages. The active host sends a message to start a task on the opposite host. Upon receiving the message, the receiver becomes active by starting the execution of corresponding task. Meanwhile, the sender becomes passive by blocking its current task execution and entering the message processing function. The active host becomes passive by sending a message to the opposite host indicating the termination of tasks. Upon receiving the message, the passive host becomes active by exiting the message processing function and resuming the blocked task execution.

Two mechanisms (shown in Fig. 1(b)) dynamically perform bookkeeping for the correct mapping between abstract memory locations and their physical memory. The *registration mechanism* keeps track of the local mapping between abstract memory locations and their corresponding physical memory with registration table. Entries in the registration table are indexed by the abstract memory location ID for lookup. Each entry in the table contains a list of memory addresses for that abstract memory location. The *translation mechanism* keeps track of the mapping of the same data between different hosts. Only the server has the translation mechanism which translates the data representation back and forth for the server and handheld device. For translation of data addresses, we maintain a mapping table on the server. Entries in the mapping table contain the mapping of memory addresses for the same data on the server and the handheld device. To reduce the run-time overhead, the registration and translation mechanism only apply to memory locations that are accessed on both hosts. Both the registration table and the mapping table are updated at run time only when new (stack or heap) memory is allocated or released.

With the registration and translation mechanisms, we can now transfer data safely between the server and handheld device. We use two methods for data transfer, namely the *push* method and the *pull* method. The push method sends abstract memory locations to the opposite host. The pull method lets the intended receiver make a request for modified abstract memory locations from the opposite host.

5. Experiments

The handheld device in our experiments is an HP IPAQ 3970 Pocket PC which has a 400 MHz Intel XScale processor. The Server is a P4 2 GHz Dell Precision 340 machine. We run Linux on both machines. The wireless connection is through a Lucent Orinoco (WaveLan) Golden 11Mbps PCMCIA card inserted into a PCMCIA expansion pack for the IPAQ. Besides the program execution time, we also measure the program energy consumption. We connect an HP 3459A high precision digital multimeter to measure the current drawn by the handheld device during program execution. In order to get a reliable and accurate

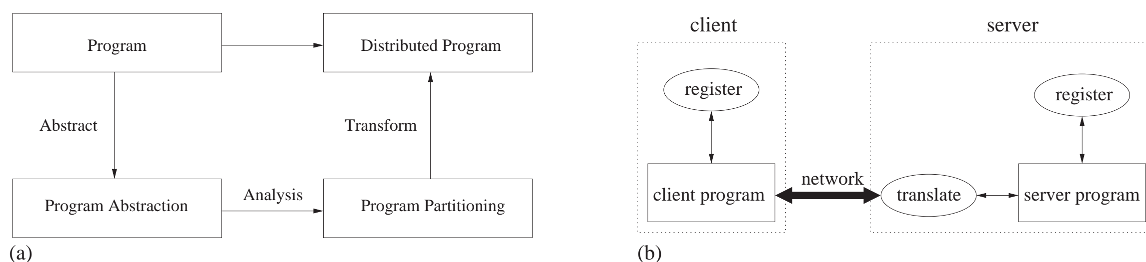

Fig. 1. Computation offloading scheme.

reading, we disconnect the batteries from both the IPAQ and the extension pack and we use an external 5 V DC power supply instead.

We implement our computation offloading scheme in GCC. A pointer analysis similar to [2] is used to get the point-to information. Such information is then used to identify references to the abstract memory locations. We partition the program in function level and restrict all the I/O functions to execute on the handheld during the computation offloading.

5.1. Cost model

In our experiments, we model the cost as the program execution time during task partitioning. We estimate the execution time of each instruction by averaging over repeated execution of that instruction. We then get the cost c_s and c_c for each basic block by adding the execution time of the instructions in the same basic block. For small data size, c_d and c_u are simply the measured network latency time. For large data transfer, we also consider the network transfer time which can be calculated by the data size and network bandwidth. We obtain other cost c_a, c_d and c_u by physical measurement. Each cost item is averaged over a large number of synthesized workloads. In our experiments, with our cost model, the difference of the whole program execution time between the measured result and estimated cost is in range of 10–30%. We will consider more accurate cost models [7,5] in our future work.

5.2. Computation offloading result

Fig. 2 shows the performance for several programs. We compare the results between two versions for each program. One version is the original program running completely on the handheld device. The other version is partitioned between the handheld device and the server which is obtained by applying our partition algorithm for that particular execution. To generate the machine code (for both the server and the handheld), all the programs, including the transformed ones, are compiled using the GCC compiler with the `-O2` optimization level. Fig. 3 shows the measured results of energy consumption which we can see, are proportional to the performance.

We should note that not all the options for these programs can get benefits from computation offloading. Here, we only show the results for a subset of options that can benefit from computation offloading. In each figure, we append the program names by the execution option and by the number of repeated execution. The program `SUSAN` performs photo processing. We run this program using the option `-s` (smoothing) and `-e`

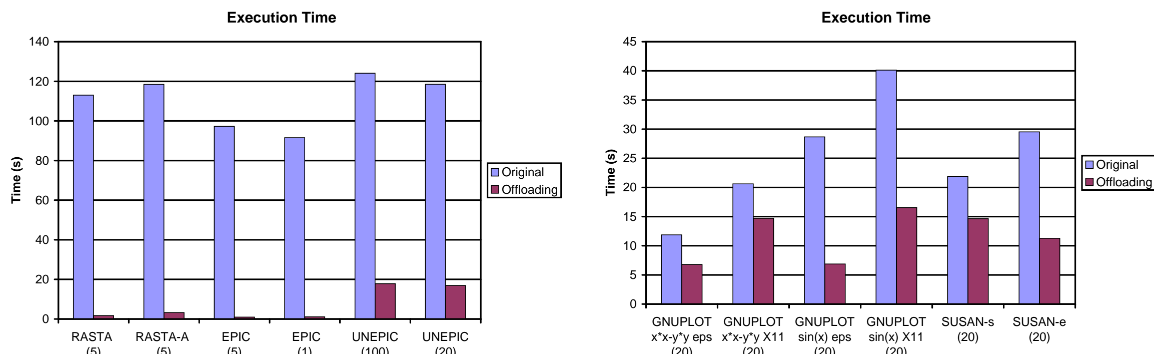

Fig. 2. Performance.

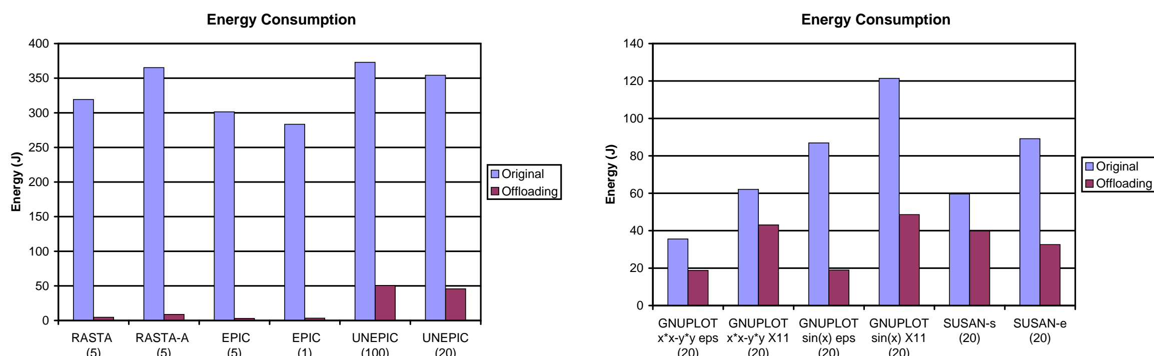

Fig. 3. Energy consumption.

(recognizing edges). The program `RASTA` performs speech recognition. It can generate results in two formats: binary data (default) and ASCII data (option `-A`). The program `EPIC` compresses graphics. Program `gnuplot` runs interactively with a command interface, and the options are specified for various commands. We generate three-dimensional figures with the *splot* command, and $x^*x\text{-}y^*y$ is the plotted function. The option *eps* means generating an `eps` file, and *X11* means generating X11 events for display. For figures with the same options, we use different input files.

It is difficult to measure the time spent on bookkeeping. We count the operations instead. Fig. 4 shows the estimated bookkeeping overhead ratio for the test programs. This figure only shows the results for execution time. The results for energy consumption are similar. The average run-time bookkeeping overhead is about 13%.

We should note that optimal program partitions vary with different program inputs. The optimal partition for one option may slow down the program execution for other inputs. Fig. 5(a) shows the program speedup for different inputs of SUSAN using the program partition got by the option *-s*. With our option-clustering heuristic, we can group the program options into two clusters and prepare one partition for each cluster. The resulting program speedup is shown in Fig. 5(b).

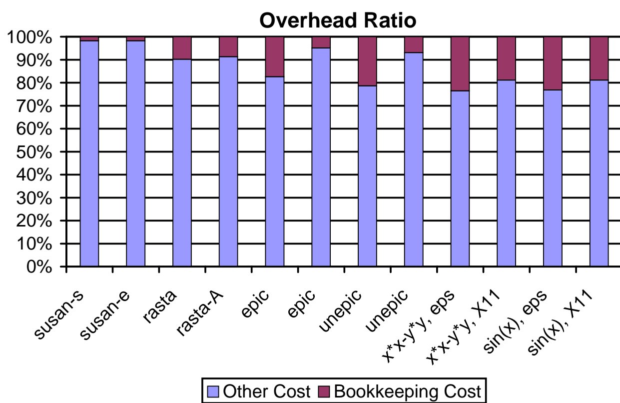

Fig. 4. Bookkeeping overhead.

6. Related work

Li et al. [9,8] propose a partition scheme for computation offloading which results in significant energy saving for half of the multimeter benchmark programs. The scheme is based on profiling information about computation time of procedures and inter-procedural data communications. It minimizes the cost and guarantees correct communication within a specific execution context only. Their work does not study the important issue of how to guarantee the correct distributed execution under all possible execution contexts. Neither does it consider the different partitions for different program inputs.

Kremer et al. [6] introduce a compilation framework for power management on handheld computing devices through remote task execution. Their paper does not address details about the data consistency issue. They only consider the procedure calls in the main routine as the candidates for remote execution and they evaluate the profitability of remote execution for each individual task separately. For tasks that share common data, evaluating the profitability of remote execution of each individual task separately may result in the overcount of data communication cost. Using their approach, due to the overcount of data communication cost, only 7 out of the 12 testing programs in the previous section can get benefits from computation offloading.

7. Conclusion and future work

In this paper, we have presented a computation offloading scheme on handheld devices. Our computation offloading scheme guarantees the correct distributed execution under all possible execution contexts.

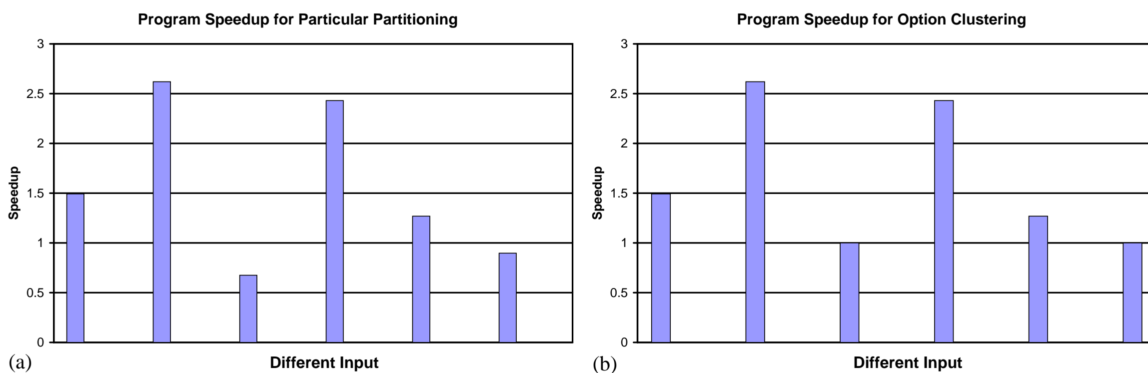

Fig. 5. Experimental result for option clustering.

Our partition algorithm finds optimal program partitioning for given program input data. We use a heuristic method to deal with different program partitioning for different execution options. Experimental results show that our computation offloading scheme can significantly improve the performance and energy consumption on handheld devices. However, we believe our computation offloading scheme can still be improved. First, our cost model for execution time can be improved. More accurate time estimation models [7,5] may get better result. Second, the point-to analysis [2] used in our experiment is control and flow insensitive, we plan to implement more accurate analysis [11]. Last, our computation offloading may be improved by dynamically adapting the program partition to the inputs and system information at run time. We plan to integrate adaptive analysis in our scheme as the next step.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel, Treadmarks: shared memory computing on networks of workstations, IEEE Comput. 29 (2) (1996) 18–28.

[2] L. Andersen, Program analysis and specialization for the C programming language, Ph.D. Thesis, DIKU, University of Copenhagen, 1994.

[3] T. Ball, J.R. Larus, Optimally profiling and tracing programs, ACM Trans. Programming Languages Systems 1994, Vol. 16, No. 4, pp. 1319–1360.

[4] J. Bang-Jensen, G. Gutin, Digraphs: Theory, Algorithms, and Applications, Springer, London, 2001.

[5] J. Engblom, A. Ermedahl, Modeling complex flows for worst-case execution time analysis, Proceedings of RTSS'00, 21st IEEE Real-Time Systems Symposium, 2000, Orlando, FL, USA.

[6] U. Kremer, J. Hicks, J.M. Rehg, A compilation framework for power and energy management on mobile computers, 14th International Workshop on Parallel Computing (LCPC'01), August 2001, Comberland Falls, KY, USA.

[7] Y.-T.S. Li, S. Malik, A. Wolfe, Efficient microarchitecture modeling and path analysis for real-time software, IEEE Real-Time Systems Symposium, 1996, Washington, DC, USA.

[8] Z. Li, C. Wang, R. Xu, Computation offloading to save energy on handheld devices: a partition scheme, Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, November 2001, Atlanta, GA, USA.

[9] Z. Li, C. Wang, R. Xu, Task allocation for distributed multimedia processing on wirelessly networked handheld devices, Proceedings of 16th International Parallel and Distributed Processing Symposium, April 2002, Ft. Lauderdale, FL, USA.

[10] A. Rudenko, P. Reiher, G.J. Popek, G H. Kuenning, Saving portable computer battery power through remote process execution, Mobile Comput. Comm. Rev. 2 (1) (January 1998) 19–26.

[11] R.P. Wilson, M.S. Lam, Efficient context-sensitive pointer analysis for C programs, Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI), June 1995, LaJolla, CA, USA.

Cheng Wang is a Ph.D. student and research assistant at the Department of Computer Science in Purdue University. He received a B.S. in Mathematics Science and an M.S. in Computer Science from Fudan University of P.R. China in 1995 and 1998, respectively. His research interests include compilation techniques for parallel computing, memory performance optimization, and compiler support for power and energy managements on handheld and embedded systems in distributed environments.

Zhiyuan Li is Associate Professor at the Department of Computer Science in Purdue University. He received a B.S. in Mathematics from Xiamen University of P.R. China in 1982, and an M.S. and a Ph.D., both in Computer Science, from University of Illinois, Urbana-Champaign, in 1985 and 1989, respectively. Before he joined Purdue University in 1997, he worked as a senior software engineer at Center for Supercomputing Research and Development in University of Illinois, Urbana-Champaign, and as Assistant Professor at the Department of Computer Science, University of Minnesota, Minneapolis-St. Paul.

Li received a Research Initiation Award an Early-Faculty Career Award from National Science Foundation (NSF) in 1992 and 1995, respectively. He has served as a member of several NSF review panels and program committees of several computer conferences, including IEEE/ACM International Parallel and Distributed Processing symposium (IPDPS), ACM International Conference on Supercomputing (ICS), International Conference on Parallel Processing (ICPP) and SIGPLAN Symposium on Languages, Compilers, and Tools for Embedded Systems (LCTES). He has also co-edited special issues for journals such as IEEE Transactions on Parallel and Distributed Systems and International Journal on Parallel Programming. Li's current research interests include compilers and run-time support for parallel and distributed systems, for memory performance optimization, and for energy-saving on handheld and embedded systems in distributed environments.