WUCC: Joint WCET and Update Conscious Compilation for Cyber-physical Systems

Yazhi Huang, Mengying Zhao, Chun Jason Xue
Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong

***Abstract*—The cyber-physical system (CPS) is a desirable computing platform for many industrial and scientific applications. However, the application of CPSs has two challenges: First, CPSs often include a number of sensor nodes. Update of preloaded code on remote sensor nodes powered by batteries is extremely energy-consuming. The code update issue in the energy sensitive CPS must be carefully considered; Second, CPSs are often real-time embedded systems with real-time properties. Worst-Case Execution Time (WCET) is one of the most important metrics in real-time system design. While existing works only consider one of these two challenges at a time, in this paper, a compiler-level optimization, *Joint WCET and Update Conscious Compilation* (WUCC), is proposed to jointly consider WCET and code update for cyber-physical systems. The novelty of the proposed approach is that the WCET problem and code update problem are considered concurrently such that a balanced solution with minimal WCET and minimal code difference can be achieved. The experimental results show that the proposed technique can minimize WCET and code difference effectively.**

I. Introduction

The cyber-physical system (CPS) is a desirable computing platform for many industrial and scientific applications. A CPS typically consists of a number of spatially distributed autonomous sensor nodes. For example, many wireless sensor networks (WSNs) monitor different aspects of the environment and relay the processed information to a central node [3]. These sensor nodes in CPS, which are preloaded with application code for tracking the events of interest, are powered by batteries. The sensing results will eventually be constructed into data packets and sent back to the user side by wireless communication. A real-world example of such a system is the Distributed Robot Garden at MIT [4].

There are two challenges for the widespread application of CPSs. First, the remote sensor nodes in CPSs are usually energy constrained due to their battery-powered nature. This energy constraint can largely determine a CPS's lifetime. However, the preloaded code on remote sensors often need to be updated via wireless communication, which is an extremely energy-consuming process. For a wireless sensor node, sending a single bit of data can consume up to 1000 times the energy as executing a single instruction [13]. Therefore, the code update problem on the remote sensor nodes of CPSs needs to be carefully considered. Second, CPSs are often real-time embedded systems with real-time properties [5, 6]. Worse-case execution time (WCET) is an important real-time constraint for designing these systems which must be safely met to ensure the correctness of real-time properties. Most of existing work only consider one of these two challenges at a time. In this paper, we propose techniques that jointly consider these two problems for real-time cyber-physical systems.

Considering code updates, Li et al. [2] propose an update-conscious compilation (UCC) technique to improve the code similarity for energy consumption minimization in the wireless sensor network. Based on the compilation results of a program's previous version, a judicious compilation solution is determined for the current program to improve the code similarity (i.e. minimize the code difference) between the new and the old. The update energy consumption is in turn saved by reducing the number of instructions that need to be updated. Since the WCET problem for real-time systems in CPSs has not been considered, the increase in the number of inserted $Move$ operations by UCC to the updated program may have negative effect on the code quality, resulting in a worse WCET.

Considering WCET, Falk [1] proposes a WCET-aware compilation technique, namely graph coloring based WCET-aware register allocation, to avoid spill code generation along the critical path of a program for WCET reduction in real-time embedded systems. The variables on the worst-case execution path (WCEP) are allocated to registers as much as possible so that the expensive memory accesses are masked, which in turn results in a better WCET. Falk's approach takes care of the WCET problem but does not consider the code update problem in real-time CPSs. The replacement of the new code for CPSs may require larger amount of updates on the program instructions, resulting in larger energy consumption in CPSs due to the energy-consuming wireless update communication.

As a result, considering WCET reduction or code difference minimization alone is not enough for real-time cyber-physical systems. In this paper, we propose a compiler level optimization technique, Joint WCET and Update Conscious Compilation (WUCC), for joint WCET and code difference minimization in CPSs. The novelty of the proposed technique is that the WCET problem and code update problem are considered concurrently during the compilation process such that a balanced solution with minimal WCET and minimal code difference can be identified. The proposed technique tries to minimize a program's WCET and improve the code similarity as much as possible for CPSs at the compilation stage. The experimental results show that the proposed approach can minimize WCET and the code difference effectively. The main contributions of this paper are as follows:

- Propose an effective approach for real-time cyber-physical systems, where the WCET problem and code update problem are addressed jointly.

978-1-4673-3030-5/13/$31.00 ©2013 IEEE

- Propose a novel control flow graph node selection heuristic for better WCET and code difference minimization.
- Formulate the target problem and present an efficient greedy algorithm to solve the problem.

The rest of this paper is organized as follows. Section II presents a motivational example. Section III formulates the target problem. Section IV conducts the problem analysis. Section V presents the proposed algorithms, followed by the experimental results in Section VI. Section VII presents the related work and Section VIII concludes this paper.

II. Motivational Example

In this section, we present a motivational example to illustrate the effectiveness of the proposed technique, where the WCET problem and the code update problem are considered jointly. Consider the basic blocks of a program in Figure 1. Two basic blocks are presented, with one on the current worst-case execution path (WCEP) of the program and the other on non-WCEP. In Figure 1(a), the WCEP basic block $b1$ has two variables a and b with disjoint live ranges and they can be assigned to the same register `R1`. In Figure 1(b), the live range of a is assumed to be changed and it is extended into b's live range. If the number of free registers is sufficient, a standard register allocator will assign different registers to these overlapped live ranges (e.g. `R1` for b and `R2` for a as shown in Figure 1(b)). However, compared to the previous assignment, the new assignment of `R2` to a will result in a name change for all uses of a for the range $\{1, 12\}$. In contrast, an update-conscious compiler (UCC) processes this change in a different manner. As shown in Figure 1(c), the UCC will assign register `R2` to a for the range $\{8, 12\}$, where the live ranges of a and b overlap. Then for a in the range of $\{1, 7\}$, the assignment will be left unchanged so that less update is required. Only one $Move$ insertion is required to move a from `R1` to `R2` when the instruction execution reaches cycle 8. With $Move$ insertions, the UCC's solution can achieve small size of code updates and less update energy consumption.

However, when considering the real-time property of cyber-physical systems, the solution shown in Figure 1(c) needs to be re-considered. As can be seen, this basic block $b1$ on WCEP has an execution frequency of 1000. Therefore, additional $Move$ insertion in the basic block will have huge negative effect on the code quality, resulting in an even worse WCET. In contrast to basic block $b1$, $b2$ is a non-WCEP basic block with only 100 execution frequency. Figure 1(d) presents the original version of compilation code, with two disjoint live ranges c and d. Then similarly Figure 1(e) shows that a change extends the live ranges of c to d's. Figure 1(f) presents the solution of UCC, which keeps some of original assignment to c unchanged so that less number of update operations is achieved. Compared to basic block $b1$, $b2$ is not on the WCEP so that the UCC solution of $b2$ has less negative effect on a program's WCET. In addition, since the number of uses of variable c in block $b2$ is three, which is much larger than only one use of variable a in block $b1$ in the unchanged range, the update energy saving of $b2$ is three

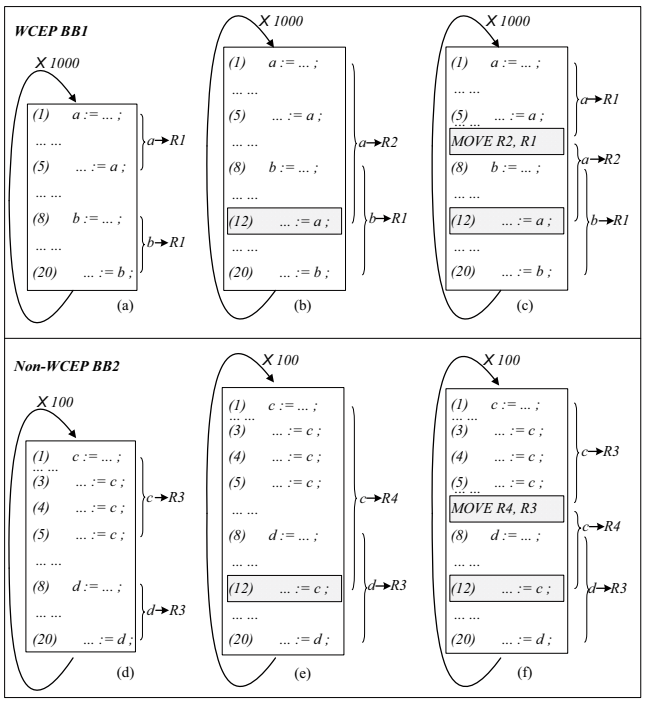

Fig. 1. Motivational example. (a) Original code of $b1$ on WCEP. (b) Changed code of $b1$ on WCEP. (c) UCC solution of $b1$. (d) Original code of $b2$ on non-WCEP. (e) Changed code of $b2$ on non-WCEP. (f) UCC solution of $b2$.

times greater than that of $b1$. Therefore, the proposed solution in this paper is based on the following observation: If $b2$ is conducted update-conscious compilation instead of $b1$, we can achieve less negative effect on the program's WCET and at the same time improve the code similarity and save most of the update energy, at 75% compared to the energy saving amounts of the pure UCC solution in this example. Table I presents the relative WCET effect, code difference, and update energy saving benefit for this example, where the maximum values are normalized to 100%. With simultaneous consideration of WCEP and code difference, a balanced solution with minimal WCET and minimal update energy consumption for CPSs can be achieved.

TABLE I
Relative WCET effect, code difference, and update energy saving amounts among different approaches.

Approach	Negative WCET effect	Code diff.	Energy saving
UCC [2]	100%	0%	100%
Proposed	0%–10%	25%	75%
WCET [1]	0%	100%	0%

III. Problem Formulation

In this section we first present the problem description for WCET and code difference minimization. Then the target problem is formulated.

A. Problem Description

As discussed previously, $Move$ insertion for code difference minimization by UCC technique can increase a program's WCET. We assume that the WCET is the main concern in

a CPS, where an acceptable increase in WCET after $Move$ insertion is α. The objective of the first problem is to maximize the code similarity (i.e. minimize the code difference) from a program's previous version under the given constraint α. Therefore, the target problem in this paper can be described as follows:

Problem: *Given an acceptable increase in a program's WCET, find an update-conscious compilation solution to maximize the overall code similarity (i.e. minimize the overall code difference) for the energy-sensitive CPS.*

The change in a program's WCET is set to be the constraint in the target problem. Under a given WCET constraint, we try to minimize the code difference of a program.

B. Problem Formulation

In this paper, we process programs in a block-based manner. Each time a single control flow graph (CFG) node is selected to be processed. We use b_i = 1/0 to represent that basic block i is or is not selected for WUCC processing. We also use $\Delta WCET_i$ to represent the WCET increment after processing basic block i. Then the goal described in the target problem can be mathematically described as follows:

$$max \sum_{i=1}^{n} b_i * CS_i \tag{1}$$

where b_i = 1 indicates block b_i is selected, CS_i is the code similarity benefit of processing b_i.

$$\sum_{i=1}^{n} \sigma_i * b_i * \Delta WCET_i \leq \alpha * WCET_{best} \tag{2}$$

where $\Delta WCET_i$ is the WCET increment of processing b_i, σ_i = 1/0 indicates block b_i is or is not on the current WCEP, percentage α is the acceptable WCET increment threshold, $WCET_{best}$ is the potential best WCET that WCET-aware compilation technique [1] can achieve before the code update activity.

Equation (1) suggests to improve the code similarity as much as possible such that the update energy saving can be maximized. Equation (2) suggests that overall WCET increment from the $WCET_{best}$ due to $Move$ insertion should be equal to or less than the given acceptable threshold α, where the $WCET_{best}$ is the potential best WCET by completely implementing the WCET-aware compilation technique in [1]. The original WCEP of a program before implementing the proposed WUCC is also evaluated based on the WCEP information obtained from the WCET-aware compilation technique.

The description of the notations used in this paper is presented in Table II, where column "Equ" lists the index of the equations.

IV. Problem Analysis

In this section, detail analysis for efficiently conducting WUCC is presented. We first present an overview of the proposed technique. Then a novel CFG node selection strategy is proposed. Finally a priority based solution is formulated.

TABLE II
DESCRIPTIONS OF THE NOTATIONS.

Equ	Notation	Description
(1)	CS_i	Code similarity benefit of processing block i
(1)	b_i	b_i = 1 indicates the selection of block i
(2)	$WCET_{best}$	Potential best WCET that can be achieved
(2)	α	A given constraint (%) in WCET increment
(2)	σ_i	σ_i = 1 indicates block i is on current WCEP
(2)	$\Delta WCET_i$	WCET increment of processing block i
(3)	P_i	Priority of block i
(3)	M_i	The number of $Move$s required in block i
(3)	$Freq_i$	The execution frequency of block i

A. Overview of the Proposed Technique

This paper proposes a compiler level approach to minimize WCET and code difference for CPSs. Note that with each $Move$ operation inserted by update-conscious compilation technique [2], a program's WCEP may change. Therefore, the optimization should be aware of the change of WCEP such that the follow-up $Move$ operation insertion for minimizing code difference can have less negative effect on a program's WCET. A judicious decision can be made by considering a program's up-to-date WCEP and code difference simultaneously. In this way, a balanced solution with minimal negative effect on WCET and minimal code difference can be achieved.

The proposed process is as follows: With the CFG input in its intermediate representation (IR) form, the WCET analysis and code similarity analysis will be conducted simultaneously during the compilation process. In each iteration, an appropriate CFG node is selected to be processed. Then an update-conscious solution for one CFG node is determined with the consideration of the previous version of the program. After that, the new WCEP information is calculated and the new version code is used for next iteration of WCET and code similarity analysis. This iteration continues until a balanced solution for minimal WCET and minimal code difference is obtained. Overview of the technique is presented in Figure 2.

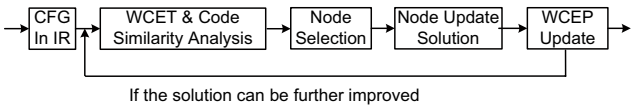

Fig. 2. Overview of the proposed technique.

B. Strategy for CFG Node Selection

Since the input program will be processed in a block-based manner, we need to determine which block to select such that the code difference and WCET can be minimized. In general, it is desirable if the update-conscious solution can improve more code similarity and at the same time have zero or minimal negative effect on a program's WCEP. Processing this type of node first will leave more space for processing the rest of nodes such that more nodes have potential to be selected and processed by considering the WCET increment status and code difference. Based on this principle, we propose to mark a less frequently executed node on non-WCEP with more number of executions and less variables to be updated for processing

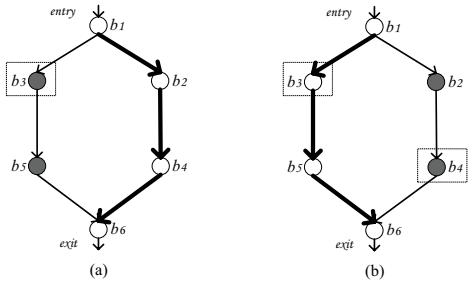

Fig. 3. CFG node selection and candidate set re-construction. Execution path in bold is assumed to be the current WCEP. (a) Non-WCEP block b_3 is assumed to be the appropriate block for selection. (b) After processing b_3, check the new WCEP and update the candidates to be processed. Block b_4 is assumed to be the next selected block.

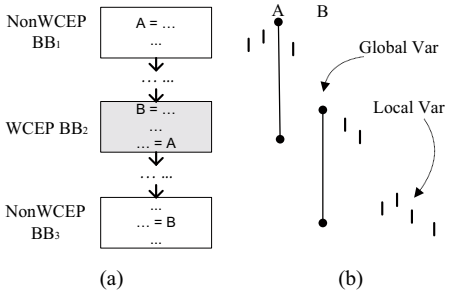

Fig. 4. Processing WCEP-crossing global variables in a selected CFG node. Basic block on WCEP is marked in dark. (a) WCEP basic block BB_2 and non-WCEP basic block BB_1 & BB_3. (b) Live ranges of local and global variables, where WCEP-crossing variables A and B is preferred to be spilled if the number of registers is not enough.

first. With the highest priority given to such kind of nodes, we can improve code similarity while the negative effect of $Move$ insertions on WCET can be minimized.

Figure 1(f) is a good example to illustrate this principle. As shown in Figure 1(f), block $b2$ is a non-WCEP basic block with less execution frequency comparing to block $b1$. At the same time, $b2$ has more number of usage than that of $b1$, with only one variable c to be processed. In other words, only one $Move$ operation is required to be inserted so that we can cover all register changes of the uses of variable c, which will thereby have less negative effect on the program's WCET, while the code similarity can be maintained as much as possible.

Note that the candidate node set during the node selection phase might change due to the potential change of WCEP after a block is processed. Therefore each time after a block is processed, we will calculate the new WCEP. If WCEP has changed, the block candidate set will be re-constructed. The basic idea of candidate updating is presented in Figure 3. Figure 3(a) presents a CFG with the current WCEP marked in bold lines. As we can see, block b_3 and b_5 are non-WCEP blocks so they are candidates to be selected for WUCC processing. Block b_3 is assumed to be selected to insert $Move$ operations for code similarity improvement. After that, a new WCEP is calculated. We assume that the WCEP has changed, which is shown in Figure 3(b). Then the candidate set will be re-constructed, with b_5 removed and b_2 b_4 included. We also assume that block b_4 is the appropriate block among all new candidates. Then b_4 is selected to be processed. This process continues until a balanced solution has been found.

Another problem we need to address is the register allocation problem for CFG nodes on WCEP. In update-conscious compilation, a large number of $Move$ operations inserted could become the bottleneck of a program's WCET. Therefore the node selection strategy for the proposed WUCC technique is to give nodes on non-WCEP preference for selection. On the other hand, WCET-aware register allocation technique aims at avoiding spill code generation along the critical path (i.e. WCEP) of a program for WCET reduction . They mark the nodes on WCEP for allocation first to ensure variables on WCEP are assigned registers as much as possible. To process this difference in CFG node selection, after a non-WCEP node is selected under our proposed approach, we prefer to spill those global variables that are used along the current WCEP. This is because spilling these WCEP-crossing variables with long live ranges can leave other variables in WCEP nodes additional available registers, so that more variables can reside in registers when these WCEP nodes are processed later using traditional compilation technique without $Move$ insertion. Figure 4 presents this basic idea. We assume that non-WCEP block BB_1 and BB_3 are two selected blocks. Then global variable A and B will be spilled if the number of registers is not enough since they are WCEP-crossing variables in WCEP block BB_2. In this way, more variables on WCEP can be allocated into registers instead of expensive memory access so that a program's WCET can be reduced, which is consistent with the strategy of WCET-aware compilation technique.

C. A Model for CFG Node Selection

As discussed in the last subsection, the goal is to select the appropriate CFG nodes for processing such that they have less negative effect on WCET and have better code similarity benefit. Note that there can be many variables within a single node to be processed. Therefore, more than one $Move$ operation is required to be inserted for these variables when applying WUCC. In addition, the number of uses of each variable is different. To summarize, we observe that a CFG node has the following properties:

- For each node, the more variables to be processed it contains, the more $Move$ operations are required.
- For each variable in a node, the more number of usage of that variable has, the more code similarity and update energy can be saved by inserting a single $Move$ operation.
- For each node with the same number of $Move$ insertion, the higher the execution frequency is, the more potential negative effect it has on WCET.

To select a CFG node for processing, we introduce a model to reflect these properties of each node. We use CS_i to denote the benefit of code similarity we can save after processing a node. $Freq_i$ is used to denote the execution frequency of a node i. M_i represents the number of $Move$ operations that a node i requires. Then a CFG node i's priority P_i can be

defined as follows:

$$P_i = \frac{CS_i}{M_i \times Freq_i} \quad (3)$$

In Equation (3), the denominator $M_i \times Freq_i$ indicates the increase in path length that may have negative impact on a program's WCET. The numerator CS_i indicates the code similarity benefit, where 1 unit code similarity is assumed to be obtained if one instruction $instr$ is kept unchanged under WUCC from the update of $instr$ under traditional non-update conscious compilation approach. For example, in the new program shown in Figure 1(e), 4 updates of variable c are needed for the changes in registers from its previous version (i.e. from `R3` to `R4`). In Figure 1(f), only one update of variable c is needed, while the other 3 occurance of c are kept unchanged. As a result, 3 unit code similarity benefit is obtained. Therefore Equation (3) suggests that the more code similarity profit per unit potential increase in WCEP a node can bring, the higher priority it should be given. This strategy results in less negative effect on WCET and more energy saving benefit.

V. Algorithm

This section presents the algorithm for the target problem. The input is a program P in IR form, a given threshold α, and P's previous compiled version P'. The output is a balanced compilation solution for P with minimal code difference from P' and minimal increase from P's original WCET.

Algorithm 1 WUCC: Joint WCET and Update Conscious Compilation

Require: a program P in IR form that consists of n variables $P = \{v_1, v_2, v_3, \ldots, v_n\}$. a given threshold α. P's previous compiled version P'.

Ensure: determine a compilation solution for P with minimal code difference (i.e. maximal code similarity) under a given threshold α for WCET increment.

```
1: ΔWCET ← 0;
2: while ΔWCET < αWCET_best; do
3:     Calculate_WCEP();
4:     Construct_Candidate_Set();
5:     for each basic block b in CFG do
6:         Calculate_Priority();
7:     end for
8:     Sort_By_Priority();
9:     Select_A_Block_Based_On_Priority();
10:    Update_Conscious_Compilation();
11:    Update_ΔWCET();
12: end while
13: return;
```

The proposed Algorithm WUCC is presented in Algorithm 1. Algorithm WUCC first sets the WCET increment counter to be 0. Then the current WCEP information is calculated (line 3). All basic blocks on non-WCEP are included in the candidate set (line 4). Then for each basic block in CFG, the priority P is calculated based on Equation (3) (lines 5-7). All blocks are sorted based on their priority P (line 8). After that, the basic block with the highest priority P is selected (line 9). Then update-conscious compilation technique is implemented on that block selected. Necessary $Move$ operations are inserted such that code difference for that block can be minimized (line 10). Subsequently the effect of that processed block on WCET is recorded and the WCET increment counter is updated (line 11). This iteration continues while the WCET increment counter is smaller than the given threshold (line 12).

Experimental results show that the process of block sorting and $Move$ insertion can be completed in a short time. Algorithm WUCC can improve the code similarity for CPSs effectively with only a small increase in a program's WCET.

VI. Experimental results

In this section we present the experimental results by implementing the proposed algorithms on Trimaran compiler [17]. Benchmarks tested in the experiments are mainly from DSPStone. Each basic block's WCET is statically predicted by using the Trimaran compiler/simulator infrastructure, following the setup of [8-11]. Modifications are mainly conducted in procedure "process coloring" according to the proposed approach. The number of available registers is set to be 6. A $Load$ from memory takes 4 unit times while a $Store$ takes 2 unit times. All other operations such as $Move$, addition, multiplication are set to be 1 unit time. One unit code similarity is improved if a single use of a variable is saved from *update* state to *unchanged* state by applying WUCC. The potential best $WCET_{best}$ and the original WCEP before implementing WUCC are calculated by implementing the WCET-aware compilation technique in [1]. The acceptable WCET increment threshold is set to be 5%, 10%, and 15% respectively. The results of UCC without considering a program's WCET are used as the comparison base.

Figure 5 presents the code similarities as well as their WCET increment for 15 benchmarks under WUCC approach. The code similarities under UCC technique are used as the comparison base. As can be seen, WUCC can effectively achieve most of code similarity from UCC with only a small increase in WCET. In other words, Most of update energy consumption can be saved without significantly sacrificing a program's WCET comparing to UCC. For example with a threshold of 10% increase in WCET, we observe that the code similarity of different benchmarks range from 70% to 85% compared to the traditional UCC without considering WCET effect. The largest gain in terms of all these benchmarks is measured for *4-lattice*, where the code similarity can achieve up to 85% of UCC under the proposed approach, with only 10% increase in WCET. We obtain a code similarity of 76% from UCC on average. In addition, the code similarity are also effectively improved under a small WCET increment of 5%, at 64% from UCC on average. With a threshold of 15% increase in WCET, we can achieve 84% of code similarity comparing to UCC on average.

Figure 6 presents the code difference among three different approaches, the WCET-aware compilation technique proposed in [1], the UCC compilation technique in [2] and the proposed WUCC technique in this paper. We adopt the code difference results of WUCC with WCET increment α = 10% for comparison. The results of WCET-aware compilation

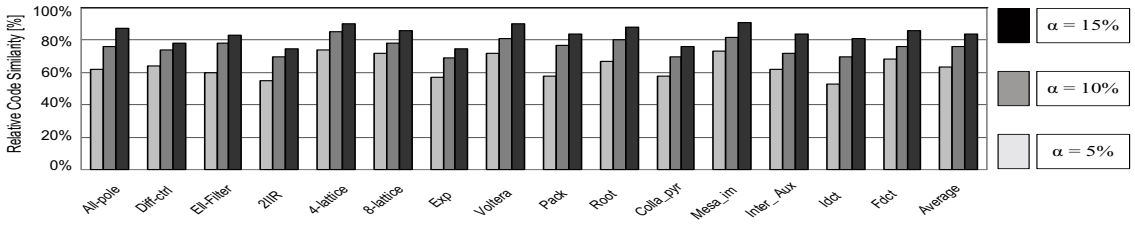

Fig. 5. The code similarity with different WCET increments under the proposed approach. The results of UCC technique are used as the comparison base.

technique is used as the comparison base. As can be seen, UCC achieves minimal code difference (i.e. maximal code similarity) among these techniques. The code difference under UCC only amounts for an average of 51% to that of WCET-aware technique without considering code update problem. The proposed WUCC technique achieves the moderate results for code difference, with 66% on average, leading to a saving of 34%. With a small increase in a program's WCET, the proposed approach can achieve most of code similarity from its previous version comparing to UCC.

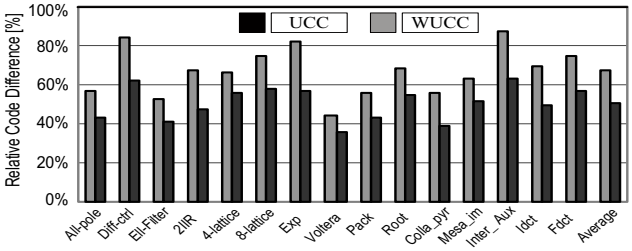

Fig. 6. The code difference of different benchmarks under WUCC, UCC [2], and WCET-aware technique [1], with [1] as the comparison base.

VII. Related Work

The cyber-physical system has recently emerged as a desirable computing platform. [14] pointed out that reprogramming sensors are more economical and practical for sensor networks. Li et al. [2] proposed the update-conscious compilation technique for achieving energy efficiency in these reprogramming-based wireless sensor networks. Li et al. [15] proposed an update-conscious code update scheme to reduce path size for DSP applications. Parolini et al. [16] presented a new control strategy for energy management in cyber-physical systems. J. Hu et al. [19] aimed at minimizing transferred data for code update on wireless sensor network.

Works regarding WCET reduction for embedded systems can be found in [7, 18]. Compiler optimizations for WCET minimization are an emerging area of research with only few related works. Falk [1] presented a WCET-aware register allocator for real-time embedded systems, which was one of the early works to handle the WCET problem at compiler stage. Deverge et al. [12] proposed a hybrid approach, that combined instruction-level parallelism with an iterative heuristic, for WCET-oriented dynamic data allocation on SPM.

VIII. Conclusions

This paper proposes a compiler level optimization technique, joint WCET and update conscious compilation, for WCET and code difference minimization in cyber-physical systems. To the best of our knowledge, this work is the first to consider the WCET problem and the code update problem jointly at the compilation stage for CPSs. Experimental results show that the proposed approach can reduce the CPS's code difference effectively with only a small increase in a program's original WCET, which is a promising solution for the energy-sensitive real-time cyber-physical systems.

Acknowledgements

This work is supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 123811 and 123210].

References

[1] H. Falk. "WCET-aware register allocation based on graph coloring," in *DAC '09,* 2009, pp. 726–731.
[2] W. Li, Y. Zhang, J. Yang, and J. Zheng, "UCC: update-conscious compilation for energy efficiency in wireless sensor networks," in *PLDI '07,* 2007, pp. 383–393.
[3] E. H. Callaway, "Wireless sensor networks: architectures and protocals," in *CRC Press,* 2003.
[4] The distributed robotics garden, http://people.csail.mit.edu/.
[5] CarTel [MIT Cartel], http://cartel.csail.mit.edu/.
[6] C. O. Yang, "Cyber-physical System: real-time internet-based structure health monitoring system," http://mtc.engr.siu.edu/data/publication/
[7] T. Liu, M. Li, and C. J. Xue, "Minimizing WCET for real-time embedded systems via static instruction cache locking," in *RTAS '09,* 2009, pp. 35–44.
[8] J. Yan, and W. Zhang, "WCET analysis of instruction caches with prefetching," in *LCTES '07,* 2007, pp. 175–184.
[9] S. S. Lim, Y. H. Bae, G. T. Jang, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim, "An accurate worst case timing analysis technique for RISC processors," in *RTSS,* 1994.
[10] R. Arnod, F. Muller, D. Whalley, and M. Harmon, "Bounding worst-case instruction cache performance," in *RTSS,* 1994.
[11] C. A. Healy, D. B. Whalley, and M. G. Harmon, "Intergrating the timing analysis of pipelining and instruction caching," in *RTSS* 1995.
[12] J.-F. Deverge and I. Puaut, "WCET-directed dynamic scratchpad memory allocation of data," in *ECRTS,* 2007, pp. 179–190.
[13] W. Shnayder, M. Hempstead, R. Chen, G. W. Allen, and M. Welsh, " Simulating the power consumption of large-scale sensor network applications," in *SenSys,* 2004, pp. 188–200.
[14] P. Levis, and D. Culler, "Mate: a tiny virtual machine for sensor networks," in *ASPLOS,* 2002, pp. 85–95.
[15] W. Li, and Y. Zhang, "An efficient code update scheme for DSP applications in mobile embedded systems," in *LCTES '10,* 2010, pp. 105–114.
[16] L. Parolini, N. Tolia, B. Sinopoli, and B. H. Hrogh, "A cyber-physical systems approach to energy management in data centers," in *ICCPS '10,* 2010.
[17] Trimaran. http://www.trimaran.org.
[18] T. Liu, Y. Zhao, M. Li, and C. J. Xue, "Task assignment with cache partitioning and locking for WCET minimization on MPSoC," in *ICPP '10,* 2010, pp. 573–582.
[19] J. Hu, C. J. Xue, M. Qiu, W. Tseng, C. Q. Xu, L. Zhang, and E. H. Sha, "Minimizing transferred data for code update on wireless sensor network," in *WASA '08,* 2008, pp. 349–360.