Cache-Aware Cooperative Task Mapping in Multi-core Real-Time Systems

Endong Wang, Fan Ni, Jicheng Chen, Hongwei Wang, and Yihan Li

***Abstract*—Program execution can be accelerated with efficient use of cache in real-time systems. And each program has its own instruction access pattern, which causes uneven distribution of accesses to the sets of instruction cache. In multi-core real-time systems, mapping tasks with similar instruction access patterns to the same core will incur massive conflicts and degrade the utilization of the cache. This paper proposes a cache-aware cooperative task mapping method to improve system efficiency in multi-core real-time systems. Our method quantifies the access frequency of instruction cache sets for each task, and then select tasks with complementary distributions to run on the same core, which can reduce inter-task interference and shorten the cache refill delay during context switch. Evaluation results show that the utilization of the system is improved by about 8.92% with the method.**

***Index Terms*—Cache, multi-core, real-time system, cooperative task mapping.**

I. Introduction

Multi-core processors, which contain multiple processing cores on a single chip, have been adopted by most chip manufacturers due to the thermal- and power-related limitations of single-core designs. They will be increasingly used in embedded systems for high performance and low energy consumption. Fig. 1 depicts a popular multi-core architecture adopted in many commercial processors, such as Xbox360 Xenon processor [1] and FreeScale MPC8641D [2]. In such systems, extrinsic interference in the cache not only comes from different cores (termed as inter-core interference) as more than one cores share the last level cache, but also from tasks running on the same core (termed as *inter-task* interference). The *inter-task* interference will incur a cache refill delay. In real-time systems, corresponding cache lines have to be refilled when one task resumes its execution as the cache contents are more or less displaced by the previous task. This cache refill delay is defined as cache related preemption delay (CRPD) and taken into consideration when Worst Case Response Time (WCRT), a term used in multi-task context to replace WCET, is discussed.

Manuscript received October 9, 2015; revised December 29, 2015.

Endong Wang, Jicheng Chen, and Hongwei Wang are with the State Key Laboratory of High-end Server & Storage Technology Inspur (Beijing) Electronic Information Industry Co., Ltd, Beijing, 100085, China (e-mail: wangendong@inspur.com, chenjch@inspur.com, wanghongwei@inspur.com).

Fan Ni and Yihan Li were with Beihang University, Beijing, 100191, China. They are now with the State Key Laboratory of High-end Server & Storage Technology Inspur (Beijing) Electronic Information Industry Co., Ltd, Beijing, 100085, China (e-mail: nifan@inspur.com, liyihan@inspur.com).

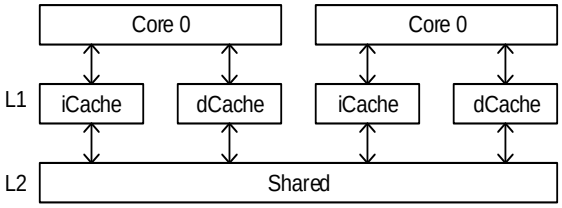

Fig. 1. A typical multi-core architecture with separated L1 and shared L2 cache.

The inter-core interference looses the WCET analysis when different cores access the shared L2 cache, which will potentially bring conflicts. And most attentions have been paid on studying the effect of L2 misses as it will cause serious access penalty. Some proposal [3] has managed to reduce the L2 cache contention through cooperative scheduling. While we believe that it is also of great importance to reduce the inter-task interference by carefully selecting tasks to run on the same core. There lies two reasons. Firstly, a cache refill delay is an important factor in the schedulability analysis of the real-time system, and it will significantly impact the WCRT analysis. The cache refill delay would be reduced if fewer conflicts be existed in the L1 cache. Secondly, extra hits in the L1 cache will decrease references to the shared L2 cache, which may in turn reduce inter-core interference and improve the WCET of the tasks. In this paper, we focus on reducing the L1 instruction cache interference through task mapping.

The cache is a hardware hash table with memory address as the key. It can be designed as direct-mapped, set-associative or fully associative, in which a memory block can be placed in one, several or all cache lines respectively. In reality, most instruction caches are designed as set-associative caches due to the tradeoff between performance and hardware cost. A cache conflict miss occurs when more than one blocks are mapped to the same set. When tasks with overlapped footprints co-run on the same core, accesses to same cache sets incur plenty of misses. On contrary, if tasks with complementary access patterns are scheduled to run together, part of the conflict misses can be avoided and the instruction access performance can be improved.

In this paper, we propose a cache-aware cooperative task mapping method for multi-core real-time systems. In the method, the distributions of instruction accesses in different cache sets are collected by quantizing the access patterns of instruction cache. Based on the distribution information, a model, defining the hot-or-cold of the cache sets, is presented to classify tasks. For each type of tasks, they follow similar hot-or-cold pattern of cache sets. A mapping decision is made by solving an *integer linear program* (ILP) problem to select tasks with different types to run on the same core. With the

doi: 10.18178/ijiee.2016.6.2.598

mapping decision, L1 cache conflicts and the system utilization in the worst-case execution are reduced.

The contributions of the paper are twofold: (1) a quantitative model is given to depict the relative hot/cold distributions of accesses to the instruction cache; (2) a cache-aware cooperative mapping strategy is proposed to minimize the conflict misses in the L1 instruction cache and shorten the CRPD in the WCRT analysis. We also evaluate the mapping strategy with over 100 task set candidates, which are constructed with programs from typical real-time benchmarks. The results are discussed by comparing our mapping decision to those unaware of the instruction access distributions. The system utilization is improved by about 8.92%.

II. Related Works

For real-time systems, less attentions have been paid to improving performance through task scheduling than that in general-purpose computer systems. The main reason is that researchers in the real-time area pay much more attentions to safety than performance. They studied scheduling [4], [5], and [6] to determine whether the deadline of the task has been met rather than how to improve the performance.

While some architects focused on improving the performance of the shared L2 cache in multi-core systems due to the fact that L2 misses affect the system performance to a much greater extent than L1 misses or pipeline conflicts. Fedorova *et al.* [3] proposed a method to reduce the L2 contention by discouraging threads with heavy memory-to-L2 traffic from being co-scheduled. And Anderson *et al.* [7], [8], [9] applied the policy of encouraging or discouraging the co-scheduling of tasks, to improve the cache performance and also to meet the real-time constraints. All these works assumed that the WCETs of tasks are known in advance. However, the improved cache performance due to sophisticated task scheduling does not only affect the average execution cost but also impact the worst-case execution performance and WCET.

The work in this paper is orthogonal to many existing proposals that reduce WCET such as compilation optimization [10]-[12] and cache locking mechanism [13]. And it is different from existing task scheduling and mapping work in three aspects. Firstly, our strategy does not aim to determine whether a task set can be scheduled on certain underlying hardware. Instead, we focus on mapping tasks to cores to reduce cache misses and improve the system utilization in the worst-case execution. Secondly, the objective of our mapping strategy is to optimize the performance of the separated L1 cache rather than the shared L2 cache. Although L2 miss incurs much longer latency, it is not accessed as frequently as the L1 cache due to the fact that most instruction accesses hit in the L1 instruction cache. Avoiding misses in the L1 cache will reduce accesses to the shared L2 cache, which will reduce the opportunities of instruction conflicts from different cores. This will make the WCET analysis tighter and also benefit the schedulability analysis. Lastly, further partitioned scheduling is allowed once the tasks are mapped to different cores with our strategy.

III. Quantitative Task Classification Method

The WCET of a real-time program is calculated as follows,

$$\text{WCET} = \max \sum_{B \in \beta} N_B \times C_B \tag{1}$$

where B is a basic block in the program, N_B and C_B are the execution count and the WCET estimate of B respectively.

In static WCET analysis, a conflict is defined when two blocks with overlapped lifetime lie in identical cache set, and it turns into a miss when more than A blocks are mapped to a set in a A-way associative cache. In a multi-task context, tasks running on the same core may conflict as they access to the same cache set during their execution. This *inter-task* conflicts potentially bring in extra access misses and enlarge the WCET estimate.

Table I shows footprints to the instruction cache for some program. As basic blocks accessed frequently dominate the pattern of the footprint, only basic blocks with the top 20% highest access frequency are analyzed here and the results show that the footprints vary for different programs. For example, no instruction accesses are mapped to the middle set of the cache for program *cnt*, and accesses to the first half of the cache are much more frequent than the bottom half for *crc*. Programs with similar patterns running on the same core may cause some cache sets *hot* and some *cold*, which brings in extra conflicts or wastes some cache capacity.

In the paper, we use *standard deviation* (termed as *stdev* hereafter) of the distributions to evaluate quantitatively the deviation of accesses to different cache sets. A larger value means the accesses are less evenly distributed in the cache sets, vice versa.

A metric called *Relative Frequency* (RF) is defined here to quantify the hot-or-cold degree of a cache set. The RF of a cache set is defined as

$$RF(s) = \frac{\sum_{\text{set}(m_j)=s} Rm_j}{\sum_{m_k \in \gamma} Rm_k} \tag{2}$$

where m_j and m_k are instruction memory blocks and set(m_j)=s is true only when m_j is mapped to set s. R_{m_j} and R_{m_k} are reference counts of m_j and m_k respectively during the execution of the program. γ is a set of memory blocks under consideration and it can include part or all instruction memory blocks accessed by the program.

Suppose that the memory bock m_j belongs to basic block B_i, termed as $B_{i,j}$, and B_i is accessed R_{B_i} times during the execution. As a basic block is a set of instructions that cannot be branched into or out of, all memory blocks in it would have identical reference count, that is, $Rm_j = R_{B_{i,j}} = R_{B_i}$. The total number of memory blocks accessed in B_i is $R_{B_i} \times mbs(B_i)$, where $mbs(B_i)$ represents the number of memory blocks covered by B_i. So (2) can be deduced as

$$RF(s) = \frac{\sum_{\text{set}(B_{i,j})=s} R_{B_i}}{\sum_{B_i \in \beta} R_{B_i} \times mbs(B_i)} \tag{3}$$

where β denotes basic block set under consideration, and it may include all basic blocks in the program or just a subset, such as the top 20% hottest ones as listed in Table I.

TABLE I: DISTRIBUTIONS (PERCENTAGES) OF INSTRUCTION ACCESSES FOR THE TOP 20% HOTTEST BASIC BLOCKS TO CACHE SETS OF THE TARGET SYSTEM. THE INSTRUCTION CACHE IS 4-WAY ASSOCIATIVE 512-B YTE LARGE AND THE CACHE LINE IS 16-BYTE LONG

	0	1	2	3	4	5	6	7	Total (%)	STDEV
adpcm	13.85	17.9	21.46	10.38	6.77	11.73	7.6	10.3	100	5.04
cnt	22.22	22.22	0	0	0	11.11	11.11	33.33	100	12.51
crc	26.43	21.23	21.23	21.24	3.02	0.42	0.82	5.61	100	10.97
edn	3.07	3.2	16.02	15.75	15.61	16.15	15.57	14.64	100	5.8
fft1	11.74	11.53	13.08	15.08	13.82	13.96	7.51	13.28	100	2.33
fir	0	1.08	1.08	20.43	19.35	19.35	19.35	19.35	100	9.77
lms	15.42	8.47	8.47	11.47	7.47	13.4	15.88	19.42	100	4.27
matmult	14.89	13.07	13.07	13	13	13.65	16.78	2.54	100	4.24
qurt	15.62	15.49	22.66	7.68	7.81	7.68	7.68	15.36	100	5.63

To categorize applications with different distributions in their accesses to the instruction cache, *RF* is not enough as it only reflects how often a cache set is accessed among all accesses to the instruction cache. We use *Average Access Frequency (AAF)* here as a threshold to compare with *RF* and categorize *hot* and *cold* sets. For a cache with *n* sets, the *AAF* is *1/n*. For example, the *AAF* is 12.5% for a cache with 8 sets. By comparing the *RF* and *AAF*, cache sets can be classified into categories. For example, here all cache sets are classified into 4 categories as follows,

$$\begin{cases} HS: Hot\ Set & \text{if RF} \in [1.25\text{f},1] \\ NS: Normal\ Set & \text{if RF} \in [0.8\text{f},1.25\,f) \\ CS: Cold\ Set & \text{if RF} \in [0.5\text{f}, 0.8\text{f}) \\ ICS: Ice\text{-Cold}\ Set & \text{if RF} \in [0,0.5\text{f}) \end{cases} \quad (4)$$

where *f* is the *AAF* of the cache.

According to (4), cache sets in Table I are classified as Table II. Obviously, different programs have different frequency distributions in their accesses to the instruction cache. Take the distribution of "*Hot Set*" as an example, the *Hot Sets* lie in the low-number cache sets for *adpcm* and *crc*, in both ends for *cnt*, in the middle for *edn* and *qurt*, and in the high-number cache sets for *fir*, *lms* and *matmult*. Moreover, accesses to the cache seem even for *fft1* as there is no hot set according to (4).

TABLE II: CLASSIFICATION OF CACHE SETS WITH METHOD LIST IN (4). THE CACHE HAS 8 SETS, AND *AAF* IS 12.5%

Set	0	1	2	3	4	5	6	7
adpcm	NS	HS	HS	NS	CS	NS	CS	NS
cnt	HS	HS	ICS	ICS	ICS	NS	NS	HS
crc	HS	HS	HS	HS	ICS	ICS	ICS	ICS
edn	ICS	ICS	HS	HS	NS	HS	NS	NS
fft1	NS	NS	NS	NS	NS	NS	CS	NS
fir	ICS	ICS	ICS	HS	HS	HS	HS	HS
lms	NS	CS	CS	NS	CS	NS	HS	HS
matmult	NS	NS	NS	NS	NS	NS	HS	ICS
qurt	NS	NS	HS	CS	CS	CS	CS	NS

Except for the distribution of *HS*, the deviation of accesses to different sets also reflects the pattern of accesses to the instruction cache in a program. Standard deviation (*SD*) of *RF* reflects the degree of this deviation. The bigger *SD* is, the larger deviation is. Otherwise, the access is even among sets of the cache. As shown in Table I, The RFs of *cnt*, *crc* and *fir* have a larger *SD*, and accordingly they have more *HS* and *ICS* as shown in Table II, which means the deviation of accesses to different cache set is large.

According to the classification of cache sets based on the relationship between *RF* and *AAF*, programs in Table I can be divided into five types,

1) **LSI** (*Low-Set Intensive*): hot sets are not rare and lies in low-number cache sets, such as *adpcm* and *crc* in Table II.
2) **HSI** (*High-Set Intensive*): hot sets are not rare and lies in high-number cache sets, such as *fir* and *lms* in Table II.
3) **HLI** (*High-and-Low set Intensive*): hot sets are not rare and lies in both ends of the cache, such as *cnt* in Table II
4) **MSI** (*Middle-Set Intensive*): hot sets are not rare and lies in the middle of the cache, such as *edn* in Table II.
5) **ED** (*Evenly distributed*): the accesses to the cache are even and the ice-cold and hot sets are rare. *fft1*, *matmult* and *qurt* in Table II fall into this category.

It is noted that, a task may fall into different types when it runs on processors with different cache configurations. Also, the classification depends on the basic block set chosen for calculating *RF*. For example, a task may fall into different type when 50% rather than 20% top hottest basic blocks are selected as β as that used in Table I.

IV. COOPERATIVE TASK MAPPING

With the program classification method described above, we propose a cache-aware cooperative task mapping strategy to make optimal mapping decisions for multi-core real-time systems. By mapping tasks with complementary instruction access patterns to run on the same core, the utilization of the instruction cache is improved and the misses caused by inter-task mapping conflicts is reduced.

To carry out the mapping strategy, following steps have to be performed in advance,

1) Perform control flow analysis, generate control flow graph (*CFG*), and collect basic block information for each task in the task set. For each basic block, the start address and block size are recorded;
2) Collect the worst-case reference count of all basic blocks in each task with static WCET analysis tools, such as *Chronos*;
3) Gather indexes of the cache sets accessed by each basic block according to the target cache configuration;
4) Calculate the RF of each set gotten in Step 3 for each task as shown in (3);
5) According to the RF values, for each task, determine the

distribution of instruction accesses in different sets and classify cache sets into HS/NS/CS/ICS;

6) Based on the HS/NS/CS/ICS classifications, divide tasks into LSI/HSI/MSI/HLI/ED types;

After the above steps are finished, the mapping strategy is carried out step by step as follows:

1) Tasks with high timing priority are mapped to different cores as much as possible. This principle ensures that high timing priority tasks are authorized to get resources as soon as possible to run;
2) Tasks falling into types other than ED are mapped first. This is because tasks of ED type are unlikely to gain much for cooperative mapping strategy;
3) Except tasks of ED type, tasks of the same type are mapped to different cores preferentially. This is due to the fact that mapping tasks of the same type to the same core brings in a lot of conflicts, which may degrade the performance of the locking content selection algorithm and enlarge the WCET estimates;
4) Tasks of HLI type are preferentially mapped to the same core with those of MSI type;
5) Tasks of LSI and HSI type are mapped to the same core preferentially;
6) Tasks of LSI, HSI and MSI type are mapped to the same core as far as possible.

Given principles given above to make mapping decisions, extra approach is needed to evaluate which mapping is better when more than one mappings follow the principles. For example, if more than one tasks fall into HLI type, some scheme must be used to decide which one is preferentially chosen to run together with a MSI type task. Also, it is essential to assess how evenly the tasks have been mapped and when the mapping process can be terminated under certain system constraints. The process to make an optimal mapping decision can be deduced as solving a 0/1 knapsack problem, which is a NP-hard and impossible to find a polynomial-time solution. So extra constraints are attached to accelerate convergence process. And the mapping process is finished when one of the break conditions, such as that the system utilization meets the requirement, is met.

For these reasons, a *RF* vector is defined for a task set to assess a mapping decision. That is,

$$RF(\mathrm{S}) = (\sum_{t\in S} p_t \times RF_t^0, \sum_{t\in S} p_t \times RF_t^1, ..., \sum_{t\in S} p_t \times RF_t^{n-1}) \quad (5)$$

where RF_t^i represents the *RF* value of cache set *i* when task *t* runs alone on a given core (see (3)), and *n* is set number of the cache. Also, a constant p_t is attached to *RF* to reflect the priority of task *t*, and it is large for a task with high priority.

To make a mapping decision, the *RF* vector of the task set *S* mapped to the same core for each mapping decision candidate is calculated as (5). The standard deviation of all the elements of the vector is then calculated, and if it is lower than the threshold defined in advance, the task mapping decision is accepted. Otherwise, replace the task which has the lowest priority (termed as T_k) in *S* with a new task of the same type as T_k, termed as T_l, and (S-{*Tk*}) U T_l forms a new mapping decision candidate. If all tasks with lowest priority has been tested while the mapping decision is not accepted, try to replace task with the second lowest priority, and so forth. The mapping process would not complete until the requirement meets or a failure is reported.

Finding a set of tasks to schedule on a core (termed as C_T) to meet certain conditions can be deduced as an ILP (*Integer Linear Programming*) problem. The objective function is

$$\text{Minimize } (stdev(S)) \quad (6)$$

where *stdev*(*S*) is the standard deviation of all elements of the RF vector of task set *S*, and it is defined as

$$stdev(S) = \sqrt{\frac{n\sum f_i^2 - (\sum_{i\in[0,n)} f_i)^2}{n(n-1)}} \quad (7)$$

In (7), f_i is an element of the *RF* vector, and defined as $f_i = \sum_{t\in S} x_t \times p_t \times \mathrm{RF}_t^i$, where x_t is an element of *X*(*S*), which is the ILP variable to solve. Suppose there are *n* tasks in *S*, *X(S)* is denoted as (x_0, x_1,…, x_{n-1}). For task T_i, x_i is defined as,

$$x_i = \begin{cases} 1 & \text{if } T_i \text{ is mapped to } C_T \\ 0 & \text{if } T_i \text{ is NOT mapped to } C_T \end{cases} \quad (8)$$

To guarantee the utilization of the cache, the constraint is defined as,

$$\sum_{i\in[0,1)} f_i > F \quad (9)$$

where *F* is a pre-defined constant. A large *F* promises the cache is kept busy. Other constraints can also be defined to accelerate the solving, such as some tasks are restricted to run on a subset of all cores of the target system.

V. Evaluation

In the section, we show how the evaluation is carried out, what assumptions are made and what the results are. We also discuss the results in typical mapping scenarios.

A. System Configurations and Assumptions

TABLE III: Underlying Hardware Configurations

Item	Value
Cores	2
iL1 size	256-byte
L2 cache	1Mbyte
dL1 cache	Perfect
Cache block size	16-byte
Way of cache	4
iL1 hit latency	1-cycle
L2 hit latency	4-cycle
Memory access latency	10-cycle
Pipeline	In-order
Branch predictor	Perfect

The hardware structure is shown as Fig. 1 and configurations are listed in Table III. It can be seen that the iL1 cache is small. This is because the benchmarks used in the evaluation are only the kernel of the real-time programs, using a small cache can mimic the conflicts in real systems. To eliminate the impacts of data accesses to the evaluation, a "*perfect*" data L1 cache is used during the evaluation, which means all data accesses hit in it and no data accesses are

issued to the L2 cache. The same explanation also fits the use of in-order pipeline and perfect branch predictor.

We also assume that the two cores are symmetric, that is, mapping the task set to either core will make no difference to the final evaluation results.

B. Benchmarks

We select 9 programs from MRTC benchmark suite [14] as candidates to map on the target system, as shown in Table IV. The selected benchmarks have different size and their patterns of accesses to the instruction cache (see Table V) fall into different types. Table VI gives the classification of cache sets when the programs run on the target system alone, according to the relationship of RF and AAF as shown in (4).

TABLE IV: CHARACTERISTICS OF THE BENCHMARKS

Program	Description	Bytes	LOC	NOI
adpcm	Adaptive pulse code modulation algorithm.	26852	879	928
cnt	Counts non-negative numbers in a matrix	2880	267	106
crc	Cyclic redundancy check computation on 40 bytes of data.	5168	128	150
edn	Finite Impulse Response (FIR) filter calculations.	10563	285	550
fft1	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm.	6244	219	312
fir	Finite impulse response filter (signal processing algorithms) over a 700 items long sample.	11965	276	74
lms	LMS adaptive signal enhancement. The input signal is a sine wave with added white noise.	7720	261	342
matmult	Matrix multiplication of two 20x20 matrices.	3737	163	122
qurt	Root computation of quadratic equations.	4898	166	170

LOC: Lines Of source Code;
NOI: Number of assembly Instructions.

TABLE V: PERCENTAGES OF ACCESSES TO SET (0-3) OF THE 4-WAY ASSOCIATIVE 256-BYTE INSTRUCTION CACHE WITH 16-BYTE BLOCKS

Set No.	0	1	2	3
adpcm	17.7	33.6	31.87	16.83
cnt	17.67	15.51	20.28	46.54
crc	32.16	28.09	1.14	38.61
edn	29.53	5.16	65.04	0.27
fft1	11.71	11.19	26.96	50.15
fir	7.15	14.29	7.14	71.43
lms	20.19	27.27	15.37	37.18
matmult	70.61	4.44	24.72	0.23
qurt	13.99	24.95	46.5	14.56

TABLE VI: SET CLASSIFICATION OF THE 4-WAY ASSOCIATIVE 256-BYTE INSTRUCTION CACHE WITH 16-BYTE BLOCKS WHEN THE PROGRAMS RUN ON THE TARGET SYSTEM ALONE

Set No.	0	1	2	3
adpcm	CS	HS	HS	CS
cnt	CS	CS	NS	HS
crc	HS	NS	ICS	HS
edn	NS	ICS	HS	ICS
fft1	ICS	ICS	NS	HS
fir	ICS	CS	ICS	HS
lms	NS	NS	CS	HS
matmult	HS	ICS	NS	ICS
qurt	CS	NS	HS	CS

C. Evaluation Methods

We use *Chronos* [15] as the framework to carry out the evaluation. *Chronos* is a popular open-source static WCET tools used by architects in real-time system area to evaluate their work. It can model many critical micro-structures such as pipeline, cache hierarchies and their interactions for tight WCET estimates. In the paper, we extend *Chronos* to support multi-core platform and collect WCET estimates of the benchmarks. We also use it to collect the basic block information needed by the mapping strategy, such as the start address, size and reference counts.

Every 4 programs from the 9 benchmarks are selected to form the task set to map on the two cores of the system, with 2 on each core. Once two programs are selected to map on the same core, the remaining two map to the other core. In our evaluation, we consider all possible combinations of the benchmarks to form the mapping task set, that is, total 126 (C_9^4) task set candidates.

Moreover, to exclude inter-core interaction of instruction accesses to the L2 cache, we split the L2 cache equally, with each core using 512-byte capacity.

To make the description clear, a task set candidate with four tasks $T1$, $T2$, $T3$ and $T4$ are termed as TK= ($T1$, $T2$, $T3$, $T4$). For each TK, there are three mapping options M1, M2 and M3 (see Table VII). So for all 126 task set candidates, there are total 378 mapping options.

TABLE VII: THREE TASK MAPPING OPTIONS FOR TASK SET TK

M1: ($T1$,$T2$)|($T3$,$T4$)	$T1$ and $T2$ are mapped to one core, $T3$ and $T4$ are mapped to the other
$M2$: ($T1$,$T3$)|($T2$,$T4$)	$T1$ and $T3$ are mapped to one core, $T2$ and $T4$ are mapped to the other
$M3$: ($T1$,$T4$)|($T2$,$T3$)	$T1$ and $T4$ are mapped to one core, $T2$ and $T3$ are mapped to the other

We use utilization as the metric to evaluate the mapping strategy. The utilization of the system is defined as,

$$U = \frac{\sum_{i=0}^{n-1} U_i}{n} \tag{10}$$

where n is the number of cores in the system, and it is 2 in our evaluation. U_i is the utilization of the i^{th} core and it is defined as,

$$U_i = \sum_{T_j \in TK_i} \frac{C_j + \gamma_j}{P_j} \tag{11}$$

In (11), TK_i is the task set mapped to the i^{th} core and T_j is a task in it. Cj and P_j are the worst-case execution time and the period of task j, respectively. And γ_j is an upper bound on the refill penalty after a context-switch, it is assessed as the time to refill the intersection of lines between the preempting and preempted tasks.

Moreover, in the evaluation, it is assumed that the period of each task is identical and it is set to make the utilization of the system not lower than 90%. And the tasks on each core are scheduled to run with fixed-priority scheduling policy. A

program with shorter deadline is assigned a higher priority and p_t in (5) is calculated as

$$p_t = 1 - \frac{C_t}{\sum_{k \in S} C_k} \quad (12)$$

where C_t and C_k are WCET estimates of tasks t and k respectively, and S is the task set assigned to a given core. It is clear that, a larger p_t is assigned to a task with smaller WCET estimate, vice versa.

D. Results

After comparing the utilizations of all 378 scenarios of mapping, 7 scenarios with most significant improvement (see Table VIII) and 7 scenarios with least significant improvement (see Table IX) are picked out as examples to discuss. The column "*Ratio of utilization*" in the table lists the ratio of the system utilization in the best mapping to that of the worst one. The lower the value is, the better the mapping strategy works. After careful studies of all the 378 scenarios, we find that, in almost all scenarios, our mapping strategy makes the best mapping decisions, which achieve the lowest system utilization.

TABLE VIII: THE 7 MAPPING SCENARIOS WITH THE MOST SIGNIFICANT IMPROVEMENT IN UTILIZATION U. STDEV. 1(2) IS THE STANDARD DEVIATION OF THE RF VECTOR IN THE BEST (WORST) MAPPING

TK	Best mapping	Worst mapping	Ratio of U	Stdev. 1	Stdv. 2
(cnt,crc,fft1,qurt)	((cnt,qurt)|(crc,fft1))	((cnt,crc)|(fft1,qurt))	89.29%	0.433	0.491
(cnt,edn,fft1,qurt)	((cnt,qurt)|(edn,fft1))	((cnt,edn)|(fft1,qurt))	89.99%	0.481	0.493
(adpcm,cnt,crc,fft1)	((adpcm,cnt)|(crc,fft1))	((adpcm,crc)|(cnt,fft1))	90.15%	0.382	0.447
(adpcm,cnt,edn,fft1)	((adpcm,cnt)|(edn,fft1))	((adpcm,edn)|(cnt,fft1))	90.96%	0.430	0.662
(adpcm,cnt,fft1,qurt)	((adpcm,fft1)|(cnt,qurt))	((adpcm,cnt)|(fft1,qurt))	91.63%	0.332	0.342
(cnt,fft1,matmult,qurt)	((cnt,qurt)|(fft1,matmult))	((cnt,fft1)|(matmult,qurt))	92.26%	0.439	0.657
(crc,fft1,fir,qurt)	((crc,fft1)|(fir,qurt))	((crc,fir)|(fft1,qurt))	93.10%	0.541	0.656

TABLE IX: THE 7 MAPPING SCENARIOS WITH THE LEAST SIGNIFICANT IMPROVEMENT IN UTILIZATION U. STDEV. 1(2) IS THE STANDARD DEVIATION OF THE RF VECTOR IN THE BEST (WORST) MAPPING

TK	Best mapping	Worst mapping	Ratio of U	Stdev. 1	Stdv. 2
(adpcm,cnt,crc,edn)	((adpcm,cnt)|(crc,edn))	((adpcm,crc)|(cnt,edn))	99.78%	0.279	0.389
(crc,edn,fir,qurt)	((crc,edn)|(fir,qurt))	((crc,fir)|(edn,qurt))	99.79%	0.438	0.856
(cnt,crc,fir,matmult)	((cnt,crc)|(fir,matmult))	((cnt,fir)|(crc,matmult))	99.86%	0.556	0.809
(cnt,fft1,fir,lms)	((cnt,fir)|(fft1,lms))	((cnt,fft1)|(fir,lms))	99.88%	0.706	0.723
(crc,edn,fir,matmult)	((crc,matmult)|(edn,fir))	((crc,edn)|(fir,matmult))	99.88%	0.619	0.455
(adpcm,crc,lms,qurt)	((adpcm,crc)|(lms,qurt))	((adpcm,lms)|(crc,qurt))	99.91%	0.238	0.134
(adpcm,crc,edn,fir)	((adpcm,crc)|(edn,fir))	((adpcm,edn)|(crc,fir))	99.95%	0.386	0.766

As shown in Table VIII, accesses are evener in the best mapping, which can be deduced from the *Stdev* values. This means our cooperative mapping strategy has made the best mapping decisions. When tasks mapped to the same core have even access to different cache sets, the conflicts are rare, and fewer lines of the preempting task have to be loaded into the cache and thus the refill penalty is low. Moreover, if cache locking is performed on the core and all instruction blocks of the tasks are candidates for content selection, our cooperative mapping strategy will improve the performance of the cache locking algorithm as the mapping conflicts are reduced. According to Table V, it is also clear that the tasks in the task sets which achieve most significant improvement have complementary distributions in their accesses to the instruction cache. For the 7 task mapping scenarios, the utilization of the system is reduced by about 8.92% on average when using cooperative task mapping strategy.

Compared to the results listed in Table VIII, for tasks in the scenarios in Table IX, the improvement of system utilization is less significant. There are two reason for this. Firstly, tasks in these task sets have similar or overlapped distributions in their accesses to the instruction cache. For example, *adpcm* and *crc* both have frequent accesses to the 1st set. Secondly, if the selected tasks only access part of the cache, the improvement will be insignificant. For example, when *cnt* and *edn* co-run on the same core, the first half of the cache cannot be used effectively as most of the accesses in them lie in the bottom half of the instruction cache, that is the 2nd and 3rd set of the cache respectively.

Similar to the results in Table VIII, for most scenarios in Table IX, the standard deviations in the best mapping (Stdev. 1) are smaller than those in the worst mapping (Stdev. 2) as expected, but there are also some exceptions as those marked in bold italics. This is due to the fact that tasks in the sets almost have the same instruction access patterns, which leaves little space to optimize for our cooperative mapping strategy. In this case, tasks' access distributions to the cache dominate the standard deviations, which cannot be reduced by cooperative mapping. This kind of abnormality provides some extra insights into the task mapping strategy to further improve the performance of the cache. Before performing the task mapping based on the standard deviation of RF vector, we can estimate the space left for optimization by a rough profile of the access patterns of instructions to different cache sets.

For all the 378 scenarios, the system utilization is reduced by about 3.22% on average when our cooperative task mapping strategy is used.

VI. Conclusion and Future Work

In this paper, we propose a cache-aware cooperative task mapping method for multi-core real-time systems. A relative frequency based model is built to quantize the deviation of accesses to cache sets. With the model, tasks are classified into several types. The mapping between types and cores is treated as an ILP problem. By solving the problem, tasks with complementary distributions are picked out to run on the same core. The method can minimize inter-task conflicts and reduce the cache-related preemption delay in multi-core context. Evaluation results show that our cooperative task mapping strategy provides the best mapping decisions in almost all cases.

Currently, the evaluation is carried out on dual-core platform, but for the method, it has no limitations to the number of cores and can be scaled to system with more cores, such as 4 or 8.

Acknowledgment

The authors would like to thank National University of Singapore (NUS) for their release and continuous support of the open-source WCET static analysis tool-*Chronos*.

References

[1] J. Brown. (2005). Application-customized CPU design: The Microsoft Xbox 360 CPU story. [Online]. Available: http://www-128.ibm.com/developerworks/power/library/pa-fpfxbox/?ca=dgr-lnxw07XBoxDesign

[2] FreeScale Semiconductor. (2008). MPC8641D integrated host processor family reference manual. [Online]. Available: http://www.freescale.com

[3] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, "Throughput-oriented scheduling on chip multithreading systems," Technical Report TR-17, 2004.

[4] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46-61, 1973.

[5] T. P. Baker, "Multiprocessor EDF and deadline monotonic schedulability analysis," in *Proc. IEEE 34th Real-Time Systems Symposium*, 2003, pp. 120-120.

[6] J. H. Anderson and A. Srinivasan, "Early-release fair scheduling," in *Proc. 12th Euromicro Conference on Real-Time Systems*, 2000, pp. 35-43.

[7] J. H. Anderson and J. M. Calandrino, "Parallel real-time task scheduling on multicore platforms," *RTSS*, pp. 89-100, 2006.

[8] J. H. Anderson, J. M. Calandrino, and U. C. Devi, "Real-time scheduling on multicore platforms," in *Proc. Real-Time and Embedded Technology and Applications Symposium*, 2006, pp. 179-190.

[9] J. M. Calandrino and J. H. Anderson, "Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study," in *Proc. Euromicro Conference on Real-Time Systems*, 2008, pp. 299-308.

[10] W. Zhao, W. Kreahling, D. Whalley, C. Healy, and F. Mueller, "Improving WCET by optimizing worst-case paths," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2005, pp. 138-147.

[11] P. Lokuciejewski, H. Falk, and P. Marwedel, "WCET-driven cache-based procedure positioning optimizations," in *Proc. Real-Time Systems Symposium*, 2008, pp. 321-330.

[12] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches," in *Proc. Real-Time Systems Symposium*, 2009, pp. 68-77.

[13] T. Liu, M. Li, and C. J. Xue, "Minimizing WCET for real-time embedded systems via static instruction cache locking," in *Proc. Real-Time and Embedded Technology and Applications Symposium*, 2009, pp. 35-44.

[14] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks — past, present and future," *WCET*, 2010.

[15] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, no. 1, pp. 56-67, 2007.

Wang Endong was born in 1966, Jinan, Shandong Province. He got a master degree in computer architecture from Tsinghua University, Beijing, China, 1991. Now he is a fellow member of China Computer Federation.

His research interests include computer architecture, hybrid storage system and interconnect protocols for scalable system.

Ni Fan was born in 1984, Tuanfeng, Hubei Province. He got a PhD degree in computer architecture from the School of Computer Science and Technology, Beihang University, Beijing, China, 2014, and now is working as a research fellow in the State Key Laboratory of High-end Server & Storage Technology Inspur (Beijing) Electronic Information Industry Co., Ltd, Beijing, China.

His research interests include computer architecture, real time system and operating system design.

Chen Jicheng was born in 1976, Huainan, Anhui Province, China. He got a PhD degree in information and communication engineering from the Department of Information Science & Electronic Engineering, Zhejiang University, 2005, and now working as a research fellow in the State Key Laboratory of High-end Server & Storage Technology Inspur (Beijing) Electronic Information Industry Co., Ltd, Beijing, China. He is also a member of China Computer Federation.

His research interests include computer architecture, cache coherence and integrated circuit design.

Wang Hongwei was born in 1982, Harbin, Heilongjiang Province, China. He received his PhD degree in computer architecture from the School of Computer Science and Engineering, Harbin Institute of Technology, China in 2013. He works as a research fellow in the State Key Laboratory of High-end Server & Storage Technology Inspur (Beijing) Electronic Information Industry Co., Ltd, Beijing, China.

His main research interests include computer architecture and artificial intelligence.

Li Yihan was born in 1985, Putian, Fujian Province, China. He received his PhD degree in software engineering from the School of Computer Science and Engineering, BeiHang University, 2015 and now is working as a research fellow in the State Key Laboratory of High-end Server & Storage Technology Inspur (Beijing) Electronic Information Industry Co., Ltd, Beijing, China.

His main research interests include software debugging and computer architecture.