McFTP: A Framework to Explore and Prototype Multi-core Thermal Managements On Real Processors

Long Cheng†, Zhihao Zhao§, Kai Huang§, Gang Chen¶ and Alois Knoll†
†Technical University Munich, Germany §Sun Yat-Sen University
¶Northeastern University of China
Email: †{chengl, knoll}@in.tum.de §zackzhao0@gmail.com, huangk36@mail.sysu.edu.cn
¶chengang@cse.neu.edu.cn

ABSTRACT

Nowadays, multi-core processor architectures have been widely adopted in main domains e.g., embedded, general-purpose, real-time systems, etc. Diverse thermal managements have been proposed to manage the temperature under various constraints. This has made the selection of the right thermal management policy difficult. Designers need to validate any resource distribution decision in design phase on the target architecture, e.g., by using a re-configurable thermal framework running in the user-space. In this paper, we first analyze the requirements that such a framework should satisfy. Then, we propose McFTP: a thermal framework fulfilling all the requirements. For this purpose, an intermediate interface is defined to isolate thermal management policies from the low-level implementations. A set of commonly used temperature control mechanisms are implemented as a library which can be accessed via the interface. With these features, McFTP can not only implement a thermal management policy at high-level of abstraction, but also execute the user-defined task-set for real thermal evolution. We demonstrate the effectiveness and efficiency of McFTP by implementing it with two works in the literature on a Dell hardware platform.

1. INTRODUCTION

As technology for microprocessors swifts in the nanometer regime, power density is rapidly increased and has become one of the constraints to higher performance, especially for multi-core processors. Hot temperature, caused by high power density, severely hampers the reliability and performance of microprocessors. The traditional thermal managements which are designed for typical thermal conditions, i.e., physical cooling devices, are challenged by the significant spatial and temporal variation of chip temperature, for the sake of cost-effectiveness [24]. To meet such challenges, Dynamic Thermal Management (DTM) techniques have been proposed to control the temperature actively.

There have been plenty of DTM researches which are based on various temperature control mechanisms such as Dynamic Voltage and Frequency Scaling (DVFS), Dynamic Power Management (DPM), job scheduling and task migration. Designers need to select the proper thermal management policy to manage the temperature on the target platform under various constraints, e.g., peak temperature constraint or real-time constraint. These policies are often evaluated by simulation programs, which simulate the execution, power dissipation and temperature evolution of the processor according to user-defined models. The thermal management results obtained from simulation have little credibility since the adopted processor power and thermal models are usually simplified for efficiency. Moreover, when targeting commodity setups, that is, systems with off-the-shelf hardware and software environments, the timing behaviour of the system is influenced by the operating systems and the computer architecture. These concerns are often ignored in simulation programs. Nowadays, DTM researches show a trend towards multi-core architectures in which multiple cores work concurrently as a set of heat sources. Thermal management policies must properly arrange the execution of different tasks on different cores to optimize the temperature or performance while considering the heat influence between different cores. This makes comparing and selecting thermal management policy more complex.

We argue that validating the effectiveness of all selections in the early design phase on the target architecture is essential to select the right thermal management policy for commodity setups. These validating procedures can be accomplished by prototyping the policies on real hardware platforms with a high-level thermal framework. Such a framework should enable the designers to prototype the policies in a fast and efficient manner. To compare the performance of different policies, it also should offer results that can reflect the real influences of thermal policies to the temperature on the target platform. Specifically, such a framework must

- realize basic thermal-aware controlling mechanisms, i.e., a temperature control mechanisms library,
- allow the implementation of customized thermal management policies with minimal effort,
- evaluate thermal policies according to the temperature of real processors,
- have minimal requirements on the hardware and underlying software for better compatibility.

We study how to develop such a framework in this paper. The traditional frameworks of evaluating thermal management policies either are based on the power and thermal simulators of a processor [1, 2, 18, 20, 22, 23] or utilize a customized version of one standard operating system kernel to support the new thermal management technique [12, 16]. Therefore, these implementations either have little credibility in validating the effectiveness of the policies on real platforms or are difficult to maintain and place strict requirements on the hardware and software environment. Moreover, some researchers implement their work in user-space with a standard Linux kernel [11]. However, these implementations are limited to the specified policies and can be hardly re-used for validating other policies.

In this paper, we propose the Multi-core Fast Thermal Prototyping (MCFTP) framework, which is an open-source[1] thermal framework meeting all the aforementioned requirements. First, MCFTP utilizes the physical processors to execute real tasks or benchmarks. The temperatures of the cores are obtained by reading hardware thermal sensors built inside the processor instead of using thermal simulators. Second, MCFTP implements several basic thermal management mechanisms, including frequency-scaling, sleep state switching, task-migration and job scheduling. With such a thermal library, MCFTP enables the comparison and evaluation of a large set of thermal management policies. Third, MCFTP defines a Configuration Manipulation Interface (CMI), which separates the policies from the detailed low-level implementations. CMI defines a set of easy-to-use sub-interfaces to control the low-level execution of workload on the physical cores. Thus, customized thermal management policies can be quickly realized as the designer only needs to implement the high-level algorithms

[1]https://github.com/ThermalSimulationProgram/McFTP

of the policies. Finally, MCFTP has wide compatibility as it resides in the user-space and has little interaction to the kernel-space. In addition, MCFTP has few requirements on the hardware, i.e., only the Advanced Configuration and Power Interface (ACPI) and hardware thermal sensors, which are common features of modern processors. We also implement the proposed framework on the top of POSIX-compliant operating systems targeting a Dell Core-i7 desktop platform and study its performance. The effectiveness of MCFTP is demonstrated by two existing thermal and power management policies with 33 benchmarks. The efficiency of MCFTP, i.e., the running overheads of proposed framework, is also investigated by experiments on two platforms.

2. RELATED WORK

A large number of works have been proposed to evaluate multi-core thermal management policies in different levels of accuracy and for different applications. In this section, we briefly discuss the closest thermal evaluation frameworks.

The majority of thermal frameworks are programs that obtain the temperature traces by simulating firstly the power dissipation and then the temperature evolution of the target processor. In general, such frameworks have three major components, namely the processor simulator, the power simulator and the temperature simulator. The processor simulator does the logical simulation of the processor and provides access and usage statistics to relevant architecture and microarchitecture blocks. A famous one is the Gem5 [3], which encompasses system-level architectures as well as processor microarchitectures. It supports various commercial ISAs (Instruction Set Architecture), including Alpha, ARM, SPARC, MIPS, POWER, RISC-V and x86 ISAs. It also supports processors of homogeneous and heterogeneous multi-core architectures. It performs cycle-accurate simulation and computes the number of accesses to all units during the execution of a benchmark. The second component, i.e., the power simulator, computes the power dissipation estimates of the processors and interconnect primitives. Wattch [4], a framework for analyzing and optimizing microprocessor power dissipation, enables architecture-level power dissipation exploration through a cycle-accurate model of a single-core processor. To accurately model the power of multi-core architectures, a novel power, area and timing modeling framework called McPAT [15] is proposed. Finally, the power estimation of the processor is fed to the temperature simulator to compute the temperature trace. A well-known thermal simulator is the HotSpot [14]. It calculates temperature evolution based on an equivalent circuit of thermal resistance and capacitance that correspond to microarchitecture floorplan blocks and essential aspects of the thermal package. Combining the aforementioned or other similar tools, many simulators and frameworks have been presented in literature. Typical examples are the SESC [19], the work proposed in [13], the framework presented in [23] and the work in [8]. Although the above frameworks can accurately simulate the logical behaviour w.r.t. thermal management policies, the correctness of the temperature evolution strongly depends on and could be limited by the power and thermal parameters, thermal model and floorplan description. Thus, evaluating thermal management policies in such methodology lacks evidence of the effectiveness of the policies.

Instead of adopting software simulators to get the temperature, some researchers validate their policies by implementing them on real platforms based on a customized version of standard operation system kernels. Zhu Changyun et al. implement the proposed ThermOS run-time thermal management algorithms within the Linux 2.6.8 kernel in [25]. Several parts of the kernel, including performance-counter based power modeling and power-thermal budgeting, have been modified in the implementation. Similarly, Hettiarachchi et al. in [12] test their theoretical results on an Intel i7-950 multi-core processor with modified Linux kernel (2.6.33.7.2-rt30 PREEMPT RT). Compared to the thermal-simulator-based methods, such implementations offer more evidence of the results. Since these policies are integrated within the modified kernel, high timing accuracy is also provided. The downside of such implementations is that it could be costly to extend them to new software platforms as they have specified requirements to the operating system kernel. Moreover, some implementations run, at least partly, in the kernel-space and could affect other functionalities of the system and increase the instability. There are also some thermal-aware policies that have been tested in the user-space of a standard operating system. The examples could be the feedback thermal controlling approach in [11] and the hierarchical power management in [17]. The main drawback of these test beds as well as the aforementioned kernel-customizing implementations is that they are merely designed for the proposed policies in their work. Thus, extending them to new thermal management policies could be costly or even impossible since it requires re-modification, re-verification and re-testing of the implementations. The framework proposed in this paper is designed to be a general platform and can implement a large set of thermal policies with little effort. To the best of our knowledge, this is the first user-space thermal framework that evaluates different thermal management policies by the temperature of processors on real hardware platforms.

3. BACKGROUND

3.1 Workload Model

The basic unit of the workload model is a *task* τ. An instantiation of a *task* is termed as a *job*. The jobs of a task can arrive with a period p and a jitter j. Moreover, the execution times of the jobs are bounded by the worst-case execution time C_{wc} and best-case execution time C_{bc}. To cope with the definition of real-time systems, a job might have a relative deadline D, which specifies the maximal allowed time between its release and complete instants.

3.2 Review of Thermal Management Policies

Thermal management policies aim to find the optimal resource management scheme which can effectively control the peak temperature, thermal gradient and occurrence of hot spots on the chip. Based on when such optimization procedure is performed, thermal management policies can be divided into two groups.

- Offline policy. Offline policies usually solve the resource management problem in design time or compile time according to the information of workloads and hardware platforms.
- Online policy. Online policies work online and adaptively manage the hardware and software resources according to the current state or the history of the system.

There have been plenty of temperature control techniques or mechanisms. Examples could be clock gating, power gating, dynamic voltage and frequency scaling, stop-go, job scheduling and task migration. Although implemented in different hierarchical levels of the system, such mechanisms share the same idea, i.e., controlling the power dissipation characteristics of a microprocessor for lower temperature or smoother heat distribution. Four temperature control mechanisms that have been widely adopted in various thermal management policies can be listed below.

- Dynamic voltage and frequency scaling (DVFS). This mechanism dynamically scales the supple voltage or clock frequency of a microprocessor to reduce the dynamic power.
- Dynamic power management (DPM). This mechanism dynamically switches a microprocessor to low power states in which both dynamic and leakage power can be decreased. Note that no workload can be handled in these states.

- Thermal-aware job scheduling. The execution of the jobs can be reordered via this mechanism to optimize the temporal variation of the temperature.
- Thermal-aware task migration. This mechanism dynamically adjusts the task mapping on the microprocessor to balance the temperature and thus reduces thermal gradient.

A thermal management policy is usually based on one or more of aforemention mechanisms. The proposed framework in this paper implements all the above mechanisms and supports offline and online thermal management policies that are based on any combination of these common mechanisms.

3.3 Power Model and Management

3.3.1 Review of Power Dissipation

Temperature strongly depends on the power dissipation of microprocessors. Many existing thermal management policies control the temperature by lowering the total power dissipation. The power consumption of a microprocessor consists of the dynamic switching power and the leakage power. The dynamic power can be calculated by below equation.

$$P^d = \alpha C {V_{dd}}^2 f \quad (1)$$

where C is the load capacitance, V_{dd} is the supply voltage, f is the clock frequency and α is the activity factor, i.e., the fraction of transistors that switch each cycle on average [5]. The leakage power is caused by leakage current and can be given as:

$$P^l = I_{leakage} V_{dd} \quad (2)$$

where $I_{leakage}$ is the leakage current and is influenced by the temperature. There exist various technologies to reduce the dynamic and leakage power consumption. The typical one for reducing dynamic power can be the *Clock Gating*, which removes the clock signal from a circuit and thus cuts off the dynamic power of the gated section. The supply voltage can be lowered or removed to decrease the leakage power consumption. Such technology is termed as *Power Gating*, which can reduce the temperature more effectively since the leakage as well as the dynamic power is lowered.

3.3.2 Advanced Configuration and Power Interface

To enable robust operating system-directed motherboard device configuration and power management of both devices and entire system, the Advanced Configuration and Power Interface (ACPI) [9] is developed as the common industry interface. In ACPI, several power states are defined for processors. These power states can be divided into two classes. A graphical representation of the power states is plotted in Fig. 1.

- Processor Performance States (P-states), which specify different levels of performance of operating processors.
- Processor Power States (C-states), which define different power saving levels of idle processors.

P-states are typically implemented with the Dynamic Voltage and Frequency Scaling technologies on microprocessors. When a microprocessor is in P0 state, it provides the maximal performance and may consume the maximal power. A performance state Pj is termed as a higher state than Pi if $i < j$. The microprocessor offers lower performance when it is in a higher performance state. Consequently, the power consumption is reduced. In Linux operating system, the P-states can be controlled manually via the interface provided by ACPI.

Processor power states are designed at C0, C1, C2, C3, ..., Cn. In ACPI, four standard C-states are defined, i.e., C0, C1, C2 and C3; The C0 power state is an active power state where processor can executes instructions. The performance level and power consumption at C0 are defined by the current P-state. The C1 through Cn power states are the processor sleeping states

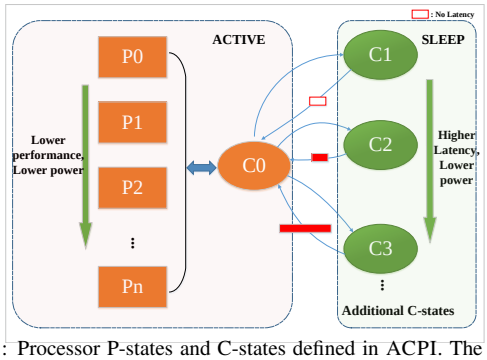


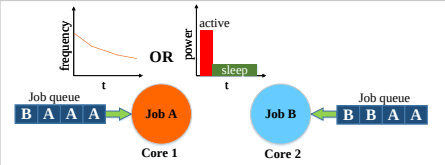
Figure 1: Processor P-states and C-states defined in ACPI. The power of C0 state depends on the currently used P-state. The red blocks at the curves connecting C0 and other C-states indicate the latency when the processor returns back to C0. Note the empty red frame indicates no latency.

where the processor consumes less power and dissipates less heat. Since the processor does not handle any workload when it's in a sleeping state, more aggressive power saving technologies such as power gating of whole cores can be applied. Temperature can be significantly lowered when a sleeping state is entered. However, exiting a C-state to normal working state introduces a certain latency which depends on the level of the C-state. Generally, the greater power saving when in the C-state , the longer the latency [9]. In [21], the actual wake-up latencies of C-states of several x86 processors are measured for various recover frequencies. When the operating system expects a certain time span before the next task, C-states will be used to save power. The specific C-state is determined based on the trade-off between power saving effect and the restore latency. Unlike P-states, C-states cannot be controlled directly in application level. However, they can be reached indirectly by eliminating workloads on the core, e.g., the Dynamic Concurrency Throttling [10] and the idle waiting policies.

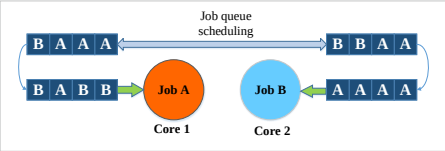
4. CHALLENGES AND DESIGN APPROACH

In this section, we discuss the challenges and the design approach of our MCFTP framework through a concrete example by adopting different multi-core thermal management policies. Suppose we have a dual-core processor executing two tasks, a hot task *A* and a cool task *B*. A task is termed as hot when executing it leads to a higher temperature on the core. Each core is associated with a buffer storing the waiting jobs. Now, consider the case that the temperature of core 1 is significantly raised by continuously executing jobs of task A while the temperature of core 2 is still in normal range, as shown in Fig. 2a. The large thermal gradient between two cores at this scenario hampers the stability and reliability of the processor. Moreover, modern processors usually require the temperature to be lower than certain threshold. Therefore, the temperature of core 1 should be decreased in this example.

To lower the temperature of a core in a multi-core processor, one may propose to use power management techniques, such as DVFS and DPM, as shown in Fig. 2a. The DVFS techniques could be utilized to lower the frequency on the core such that the dynamic power is reduced. In addition, the core can also be switched to C-states by using DPM techniques such as power gating. Power management techniques reduce the temperature at the expense of lowered performance. To maintain the same level of performance, the job queue scheduling technique serves as an alternative method to reduce the thermal gradient, as shown in Fig. 2b. One may switch the positions of the two task A jobs at core 1 and the two task B jobs at core 2. In this way, core 1 will execute two cool task jobs so that its temperature can be lowered. Moreover, one can further preempt the current running job on core 1 with the cool task B job in the waiting queue to reduce the temperature. It turns out



(a) Lowering performance



(b) Job scheduling

Figure 2: Examples of mechanisms to manage the temperature of multi-core processors.

that temperature can be controlled by diverse types of mechanisms. Note that in the example, we just consider online thermal policies that are based on only one of such mechanisms, not to mention offline policies and hyper policies combining two or more of these mechanisms. It's not clear how to implement these various thermal policies on top of a standard operating system nor how to abstract and extract their common characteristics such that we can reuse one in another. We aim to solve these problems with McFTP.

The objective of McFTP is to provide multi-core system designers a tool which enables the fast evaluation of various type of thermal management policies, e.g., offline or online, DVFS-based or task migration-based, or hyper ones combining two or more temperature control mechanisms, etc. The challenges in the design of McFTP, i.e., the implementation of various thermal management policies and the reuse of their common characteristics, are met by introducing an intermediate interface named configuration manipulation interface (CMI). Four basic thermal controlling mechanisms mentioned in Section 3.2, DVFS, DPM, job queue scheduling and task migration, are defined in CMI. Thermal management policies can access these basic mechanisms via a set of unified, pre-defined interfaces and do not need to handle the detailed implementation of the mechanisms on physical cores and the potential correlation between them. In this way, thermal management policies are isolated from the implementations of low-level mechanisms, thus can be evaluated in a fast and reliable manner.

This work does not concentrate on optimizing the temperature for different thermal management policies. Neither does this work claim that the temperature results of this proof-of-concept implementation on any (general purpose) operating systems are identical to those obtained from lower-level implementations. The primary goal of this work is to decouple the high-level description or principles of thermal management policies from the low-level implementations which depend on the system specification. This framework enables system designers to gain more complete understanding of the thermal management policies with temperature results from real hardware platforms instead of simulation programs.

5. CONFIGURATION MANIPULATION INTERFACE

In this section, the Configuration Manipulation Interface is introduced in detail by discussing the sub-interfaces defined in it.

As shown in Section 4, many temperature control mechanisms are available on multi-core processors to manage the temperature or heat distribution of the cores. Based on these mechanisms, various thermal management policies can be proposed to optimize the temperature or performance of multi-core processors. To

Table 1: The state table in CMI. Note that S_i could be an arbitrary state among the sleep state (0) and available frequencies. L_i should be a positive real number denoting the length of the state in pre-defined time unit.

state	length (microsecond)
S_1	L_1
S_2	L_2
...	...
S_i	L_i
...	...

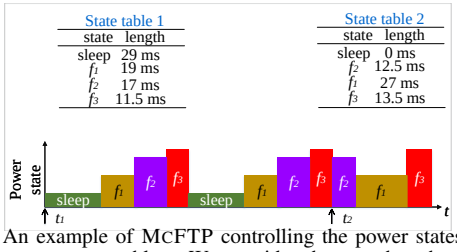


Figure 3: An example of McFTP controlling the power states of a core according to two state tables. We consider the core has three available frequencies, which are f_1, f_2 and f_3. State table 1 is applied at t_1 and repeated twice. Then, state table 2 is applied at t_2.

implement a customized thermal management policy in a fast manner with minimal effort, Configuration Manipulation Interface (CMI) is proposed as the intermediate interface between high-level algorithms of the policies and the low-level implementations of the basic temperature control mechanisms. CMI enables easy and robust control or combination of these basic mechanisms to manage temperature, hot spots and thermal gradient of multi-core processors. Next, we introduce the sub-interfaces defined in CMI for designers to access DVFS, DPM, job scheduling and task migration mechanisms.

5.1 Power Management

The evaluated thermal policy can control the power dissipation characteristics of a core in the processor via this interface. The dynamic and leakage power are managed through the DVFS and DPM mechanisms defined in the ACPI of the processor, respectively. The policy needs to provide a *state table* to specify how to control the power dissipation state of a core.

A state table has two columns, as shown in Tab. 1. The first column lists the order of demanded states of the core. A zero means to pause the execution of a job so that the core can switch to the sleep state. A positive number specifies the running frequency of that core. Since the available frequencies of a core in ACPI are defined by a set of discrete points, the given frequency will be rounded to the nearest available one if it does not equal any frequency in the set. The second column depicts the time length of the corresponding state. The start time of each state is the end time of the previous state. Specifically, the first state is adopted immediately once the state table is given. An example of the *state table* is demonstrated in Fig. 3. It is worth noting that the state table will be repeated continuously to control the power dissipation of the core until a new state table is provide to replace the old one. With the *state table*, the evaluated thermal policy can control not only the length each core stays in each power dissipation state but also the order of the states.

5.2 Job Scheduling and Task Migration

We consider that upon arrival, the jobs of all tasks are inserted into a set of queues associated with the cores and wait to be executed on the corresponding core. In default, the queue behaves as a First-In-First-Out (FIFO) buffer. New jobs are inserted at the back of the queue and the job at the front of the queue will be executed firstly. Depending on the temperature of the cores, a thermal management policy may need to change the order of

the job queue or move one job to another queue. In CMI, the following actions are defined for thermal policies to perform job queue scheduling.

- Advance. This action advances a job by a given number of job positions in the same queue.
- Recede. Similarly, this action recedes a job by a given number of job positions in the same queue.
- Move. This action moves a job in one queue to the specified position in another queue.
- Preempt. When the policy performs this action, the current running job, if exists, on the core connecting this queue will be preempted by the front job in the queue. Then, the preempted job is placed at the front of the job queue.

The above four actions can be accomplished by calling functions `advanceJobInQueue`, `recedeJobInQueue`, `moveJobToAnotherQueue` and `preemptCurrentJobOnCore`, respectively.

In addition to job queue scheduling, CMI also provides the interface for task migration. Thermal management policies can migrate the current running hot job from an overheated core to a cooler core to balance the temperature with such interface. Simply invoking the function `taskMigrate` with the source and target core indexes will make the framework perform the task migration.

5.3 Dynamic Information and Task Allocation

In addition to thermal-aware interfaces, CMI also provides the interface to collect dynamic information about the state of the cores as well as the job queues. For each core, such information structure contains the temperature, current power state, the on-going job, the length for which the on-going job has been executed, etc. For each job queue, its dynamic state is abstracted by a vector containing the pointers to the waiting jobs. Thermal management policies can use the dynamic information to make decisions during run-time.

A task allocation interface is also defined in CMI. When a new job arrives, this interface is called to decide where the job should be instantiated. This interface can be static, that is, defined by the designer in design phase, or dynamic, i.e., determined by the evaluated policy online according to the dynamic information of the cores. In default, CMI creates a static allocator which allocates all the jobs evenly on the cores. This default task allocator can be substituted by a customized one via the registration interface discussed in the next section.

5.4 Registration Interface

As discussed in Section 3.2, thermal management policies can be classified into two categories, namely offline and online policies. These two types of policies work in different manners and phases. An offline policy finds the optimal resource management scheme, e.g., the state table for controlling power and/or the task mapping on the processor, in design phase and applies the scheme at the beginning of the experiment. An online policy may dynamically change the power state of the cores or schedule the jobs according to the current state of the processor. To support the evaluation of both types of policies, CMI defines a registration interface to register an offline (function `setOfflineThermalPolicy`) or online policy (function `setOnlineThermalPolicy`) in design phase. The registered policy will be invoked automatically based on its type. As aforementioned in the previous section, CMI defines a task allocator to determine on which core a new job should be executed. Designers can also set the task allocator via this interface. To define a static allocator, designers can call `setTaskRunningCore` to statically link the jobs of a task to one specific core. In the same way, a dynamic task allocator can be explicitly set by calling `setTaskAllocator`.

In this section, we have introduced five major sub-interfaces defined in CMI. With them, our framework gains high flexibility in evaluating various thermal management policies working in different manners and replying on different temperature control

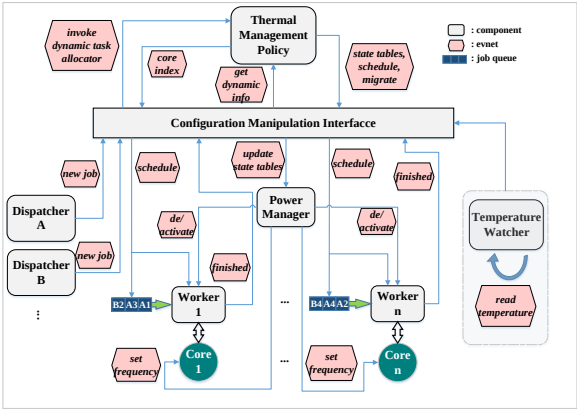


Figure 4: The proposed Multi-core Fast Thermal Prototyping Framework.

mechanisms. Note that it is not necessary to cover all the details of CMI in this paper. CMI also defines a set of interfaces and functions for the convenience of implementation of the policies. They are omitted due to their simplicity.

6. MULTI-CORE FAST THERMAL PROTOTYPING FRAMEWORK

After introducing the Configuration Manipulation Interface, we discuss the overall structure of Multi-core Fast Thermal Prototyping Framework. Fig. 4 graphically demonstrates the overall structure of MCFTP. As shown in the figure, MCFTP can be divided into two parts by CMI, one is composed of the functional components of the thermal management policy and the other part consists of the low-level implementations for executing the decisions of the policy on the actual processor. CMI isolates two parts and thus enables a predictive behavior of thermal management policies. MCFTP is composed of below components.

6.1 Dispatcher

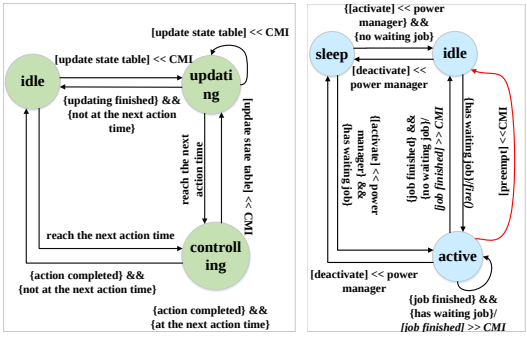
A dispatcher is defined for each task. The dispatcher supports periodic, periodic with jitter and sporadic task timing models. A dispatcher creates jobs of the task based on the task timing model. When a new job is created, the dispatcher sends the job instance to CMI by calling `addJob` instead of directly appending the job to one of the job queues. Then, if the pre-defined task allocator is a static one, CMI directly gets the index of the core on which the job should be executed from the allocator. Otherwise, CMI invokes the dynamic task allocator defined in the thermal management policy to determine the core index. Finally, the new job is appended to the corresponding job queue.

6.2 Thermal Management Policy

This component is defined by the designer. It should contain full functional descriptions of the policy such as calculating frequencies, determining the state tables for the cores, job scheduling policies, etc. The comprehensive information about the processor can be obtained via the dynamic information interface in CMI. Then, the thermal policy can manage the resources on the processor via the power management, job scheduling and task migration sub-interfaces. Moreover, the designer may also define a dynamic task allocator in the policy to dynamically assign new jobs to proper cores, as aforementioned in Section 5.4.

6.3 Temperature Watcher

This component periodically reads and saves the temperatures of all the cores of a real processor. When the dynamic information interface in CMI is invoked, the temperature watcher provides the latest temperatures of the cores.



(a) Power Manager (b) Worker

Figure 5: The operation semantics for Power Manager and Worker entities. $[e]<<s$ indicates receiving an event e from sender s. $[e]>>r$ refers to sending an event e to recipient r.

6.4 Power Manager

The power manager controls the frequencies and power states of all cores according to the *state tables* obtained from CMI. It controls the power states of the cores based on the clock frequency and C-states switching mechanisms defined in ACPI. The operation semantics of a power manager is outlined in Fig. 5a. After receiving one or more new stable table from CMI, the power manager is in *updating* state. It saves the stable table and then calculates the time instance when the next frequency or power state transition happens. If the next action time instant has not been reached, the power manager stays at *idle* state until the next action time. Then, the power manager is in *controlling* state and takes the corresponding action. After the action, the next action time instant is also updated. For a frequency transition, a power manager simply changes the core frequency via the interface provided by ACPI. In the case of switching one core to sleep, it sends a *deactivate* event to suspend the worker associated with that core. Then, the idle-waiting policy is adopted to stop the worker occupying CPU times so that the core can switch to sleep state, i.e., C-states. At the next action time, the power manager first sends an *activate* event to wake up the worker from sleep state and then switch the core to target state.

6.5 Worker

The operation semantics of a worker is depicted in Fig. 5b. For a n-core processor, n workers are created to virtually represent the cores. Each worker is associated with a job queue storing the waiting jobs. After receiving a *deactivate* event from the power manager, a worker switches to *sleep*, whichever state it is current in. When the power manager sends an *activate* event to it, a worker switches to *idle* state if there is no on-going job, otherwise it switches to *active* state to execute that job. If current job is finished and the job queue is empty, a worker sends an event to CMI to inform the job completion and then goes to the *idle* state, waiting for new jobs. Moreover, to perform job preemption or task migration, a worker can also be interrupted from *active* state by CMI with a *preempt* event. After being preempted, a worker is in *idle* state to load new jobs from the queue, if they exist. If the job queue is not empty, a worker switches back from *idle* state to *active* state and executes the first job in the queue by calling function `fire`.

7. PORTABLE IMPLEMENTATION

In previous sections, we defined Configuration Manipulation Interface and presented the overall structure of MCFTP framework. The abstract operation semantics of the power manager and worker was also described. In this section, we discuss a specific implementation of MCFTP which utilizes the API provided by POSIX standard. The POSIX standard has been widely supported by operating system including many variants of UNIX and Real-Time Operating System (RTOS).

The main goal of MCFTP is to evaluate and compare the performance of different thermal management policies on actual hardware platforms in an efficient manner. The evaluation process should be fast, safe and reasonable accurate. We implement MCFTP in user-space level as we argue that a user-space tool that can be accessed easily is the first choice if the designer wants to compare different thermal management policies in early design phase. The main concern is the relative thermal optimization performance of the policies among each other, not the absolute performance. Although implementations in kernel-space are more accurate in timing and power consumption controlling, it may be infeasible to prototype thermal management policies in early design phase when the actual hardware and software environment has not been specified. Moreover, the flexibility of MCFTP for different operating systems is also significantly limited if it modifies the kernel. Prototyping thermal management policies in user-space is more efficient and provides greater interoperability.

7.1 Implementation Requirements

To implement MCFTP, two basic features should be supported by the hardware environment. First, the processor must support the Advanced Configuration and Power Interface (ACPI) such that MCFTP can control the power dissipation by putting the processor in different P-states and C-states. Second, the temperatures of different cores can be sampled by sensors as the comparison criteria among thermal management policies. In Linux environment, the sensors built inside processors can be read via the tool *lmsensors*.

Besides the hardware environment, the operating system must support several functionalities to realize the framework in user-space. First, MCFTP should have the access to the aforementioned ACPI and thermal sensors. Second, the concurrent execution of multiple entities must be provided for running the components in MCFTP. Third, preemptive priority scheduling of the concurrent execution should be supported. Finally, timers are necessary to support the time-triggered power manager.

7.2 Multi-thread Implementation

Given a POSIX-compliant operating system, we implement MCFTP by a set of interacting threads that are assigned different priorities. The priority-based scheduler in the kernel selects the thread with higher priority to execute on the cores. The thermal policy thread has the highest priority p_0. In this thesis, p_i refers to a higher priority than p_j if $i < j$. The dispatcher is assigned priority p_1 and the power manager has priority p_2. Then, the temperature watcher gets the priority p_3. For a n-core processor, at most n worker threads can be created and each worker is assigned to one core. The workers run on different cores but have the same priority p_4. Note that apart from worker threads, the aforementioned threads can be attached to arbitrary cores in the processor, which can be customized by designers. Moreover, these threads work in a time-triggered manner. They execute their tasks at the pre-determined time instant. When finishing their tasks, they update the next time instant and then block themselves such that lower-priority threads on the same core can get chance to execute.

7.3 Power Management Implementation

MCFTP controls the frequency of each core via the ACPI interface provided in the operating system. Firstly, our framework tries to set the Linux CPUFreq governor as the *userspace* with the module modified from the tool *cpupower*. Then, the `scaling_governor` files in the kernel are dynamically modified by the power manager thread according to the *state tables* given by the thermal policy.

Switching a core to sleep state is accomplished by the idle-waiting policy with the POSIX semaphore library. A worker thread

pauses its execution and blocks itself only corresponding to user-defined suspend checkpoints when it receives a *deactivate* event from the power manager. When reaching a suspend checkpoint, the worker thread first calls the function `sem_trywait(&suspend)`. The value of semaphore `suspend` is initialized as zero and can only be incremented by the power manager via sending a *deactivate* event, i.e., calling `sem_post(&suspend)`. The worker thread exits from the suspend checkpoint and performs normal functionalities if the return value of function `sem_trywait(&suspend)` indicates no *deactivate* event has been detected. Otherwise, the worker thread blocks itself by calling function `sem_wait(&resume)`. Similarly, the value of `resume` is also initialized as zero and can only be incremented by the power manager via invoking `sem_post(&resume)`, that is, sending an *activate* event. The code of the suspend checkpoints is provided as a library to the designer.

7.4 Task Preemption Implementation

The job scheduling and task migration interfaces defined in CMI both require that our framework can preempt the task currently executed by the worker thread. To enable task preemption, we implement a specific class named *TaskLoad* holding all the information related to task preemption. Similar to the sleep mechanism mentioned in previous section, *TaskLoad* defines the preempt checkpoints to stop the execution when CMI wants to preempt the job. Another semaphore named `stop` is defined to check whether CMI has notified a job preempt request. If so, the thread firstly saves all the data related to the job execution and records the unique identity of the checkpoint. Then, it stops the execution of the job. The object of *TaskLoad* holding all the dynamic information is returned to CMI for job scheduling. When this job is executed by a worker again, the thread first jumps to the preempt checkpoint where the preemption happens by checking the identities of the checkpoints. Then, it continues the execution of the job with the saved data in the object.

8. EXPERIMENTAL EVALUATION

In this section, the performance of proposed MCFTP framework is evaluated. Firstly, we investigate the effectiveness of MCFTP in managing the temperature via the configuration manipulation interface. Then, we implement two existing thermal managements in MCFTP, one is proposed in [7], namely O-PBOO, and the other policy is presented in [6], which is termed as BWS. Finally, we report the running overhead of MCFTP on two platforms that have different computing power abilities.

8.1 Temperature Experiments

In this section, we investigate the effectiveness of our framework by reporting the temperature evolution of the cores when the frequency scaling, sleep switching and task scheduling are utilized to manage the temperature. We used a Dell Optiplex 9020 desktop personal computer as the experiment hardware platform. It contains an Intel i7-4770k processor with four physical cores. Each core has 15 available running frequencies between 800MHz and 3.4GHz, when the 'acpi-cpufreq' driver is adopted. The C-states defined for every core are C0, POLL, C1, C3, C6 and C7 if the CPUidle drive is 'intel_idle'. The cores can switch to C-state C7 when they have no workload to handle. The Hyper Thread feature of the processor is disabled in the BIOS (Basic Input/Output System) of the computer. Three air cooling fans are built inside the computer as the cooling system. The experiment ambient temperature is 20 °C. All experiments are done in the 3.16.0-53-generic Linux kernel environment. During the evaluations, the system runlevel of the operating system is set to the lowest level 1 such that only essential system services are running. Four worker threads are created and each one is attached to one core. The other threads such as dispatcher, power manager are evenly attached to all the cores. MCFTP framework is implemented in C++ language and compiled by G++ 4.8.4 with optimization level O3 turned on.

Table 2: The state table applied in our experiment. Note the zero frequency means sleep state.

state (MHz)	length (second)
3400	500
800	500
2100	500
3200	500
0	500
3400	500

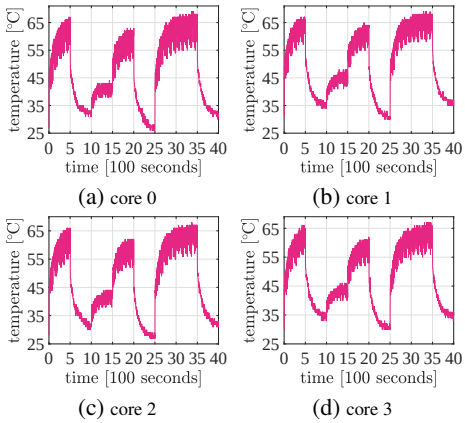


Figure 6: The temperature evolutions of the processor cores when state table Tab. 2 is applied to them.

In the first experiment, we test the temperature evolution of the cores when they execute jobs in different frequencies. The benchmark `SQRT-RAND` are executed on all the four cores. For clear demonstration, the four cores apply the same state-table, which is shown in Tab. 2. The temperature evolutions of four cores are depicted in Fig. 6. From the figure, we observe that (1) the temperature of the cores increases quickly from the initial idle state-steady temperature (around 25 °C) to a temperature about 65 °C. Then the temperature changes depending on the running frequency. Note that when the cores switch to sleep state (2000 to 2500 second), the temperature deceases to the initial idle state-steady value and is lower than that of 800MHz frequency state. (2) The increase in temperature is not linear to the increase in frequency. For example, the temperature is raised about 10 °C from 800 MHz to 2100 MHz while a nearly 20 °C temperature increase is resulted by a frequency increase from 2100 MHz to 3200 MHz. The reason is that the dynamic power dissipation is linear to the square of the frequency, as we discussed in Section 3. (3) We can also observe that although all the cores adopt the same state table, the temperature of them are not identical. For instance, when running in 800MHz and 2100MHz (500 to 1500 second), the temperatures of core 1 and core 3 are higher than that of core 0 and core 2. This is expected because the cores are on different locations in the processor floorplan. Thus, the heat removing capacity may be different for different cores.

In the second experiment, we show the effect of task scheduling policy. Two tasks, one is a hot task τ^A and the other is a cool task τ^B, are adopted. We implement a simple task scheduling policy, which assigns the hot task τ^A to the cores dynamically according to the experiment time. Similarly, the cool task is assigned to the next cores in a circular manner. The temperature results are plotted in Fig. 7.

From this experiment, we can make some interesting observations. First, the temperature of a core strongly depends on the workload it executes. This validates that task migration and job queue scheduling can be effective in controlling the temperature.

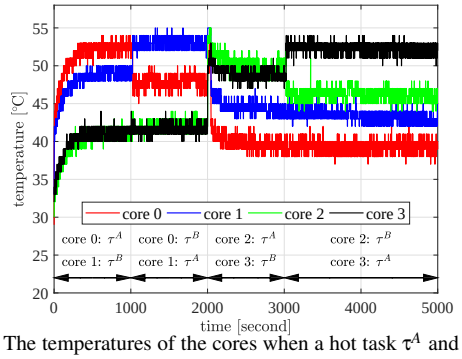


Figure 7: The temperatures of the cores when a hot task τ^A and a cool task τ^B are executed on different cores.

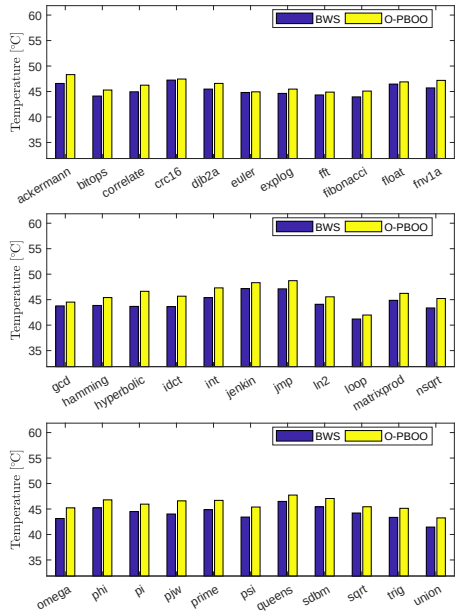


Figure 8: The temperatures of PBOO and BWS for the benchmark set.

Thermal management policies can select proper tasks to the cores according to their temperatures for thermal balance. Second, the temperature of a core is also influenced by other cores. In the first 2000 seconds, core 2 and core 3 have no workload but are both heated up by core 0 and core 1. Third, similar to previous experiment, the temperature of different cores can be different even for same workload. Fig. 7 shows that core 2 is less sensitive to the hot task τ^A, compared to other cores.

In the third experiment, we implement two thermal-aware management policies in MCFTP with 33 benchmarks from the CPU stress tool `Stress-ng`. The first approach, termed as O-PBOO, is a static one that periodically switches the cores to sleep state [7]. The second approach, namely BWS, is an online one and switches the core to sleep state dynamically. For each policy, we run the benchmark for 100 seconds and wait for 100 seconds before next benchmark to cool the processor. The average temperatures of the four cores in different cases are shown in Fig. 8. We can observe that BWS outperforms O-PBOO in all cases. This is reasonable since O-PBOO is a static policy which is determined in design phase while BWS is an online one which works in run-time. Compared to O-PBOO, the average and maximal temperature reductions of BWS are 1.5 K and 3 K. Another observation is the temperature of the processor changes when different benchmarks are executed. This further strengthens the conclusion made in the second experiment that the temperature strongly depends on the workload it executes.

8.2 Efficiency Experiments

In this section, we study the efficiency of MCFTP by reporting the overhead introduced by the framework in different scenarios. To study the efficiency on platforms with different computing capacities, in addition to the aforementioned Dell platform, we also use another embedded environment, a Raspberry Pi (RPI) Model B v1.2 with a 1.2GHz 64-bit quad-core ARMv8 CPU running the Linux 4.1.19-v7 kernel. In this platform, MCFTP framework is compiled by G++ 5.4.0 with optimization level O3 turned on.

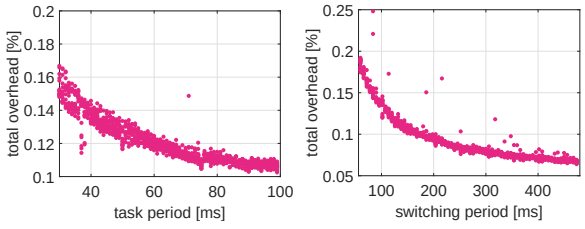
MCFTP has the following roles. The first one is to execute the user-defined tasks. The second one is to run the thermal management policy, if the policy works in run-time. Anything else can be considered as overhead. Specifically, we consider the sum of the CPU times spent by the power manager, temperature watcher, dispatchers and the checkpoints in workers as the overhead of MCFTP. The overhead is incurred by creating and registering new jobs, reading thermal sensor interface, managing job queues, parsing state tables, sending de/activate signals, writing the frequency controlling interface and checking the state of the checkpoints. Since the overhead of checkpoints depends on how the designer programs the task code, i.e., the number of checkpoints, we first study the overhead of MCFTP without checkpoints and then report the overhead of checkpoints separately.

In the first experiment, we do not consider the overhead of checkpoints. Therefore, the total overhead is the sum of the CPU times spent by the power manager, temperature watcher and dispatchers. We investigate how the overhead varies when (1) the job arriving period changes and (2) the power state switching frequency changes. For the first scenario, we vary the arriving period of jobs from 30ms to 100ms. In the second scenario, a two-state state table is used and the switching period increases from 60ms to 480ms with step 6ms. In each scenario, 750 task-sets are generated. Each task set contains five tasks with the same period. The total utilization of the task set is set as 0.5. The execution times of the tasks are randomly chosen between 1ms and the period. The experiment run-time is set as 10 seconds. The overhead is measured using the POSIX-CPU-timers for the power manager, temperature watcher and dispatcher threads, normalized over the total run-time. Fig. 9 shows the measured overheads plotted against the job arrival period and state switching period, for both platforms. From the figure, we can make following observations.

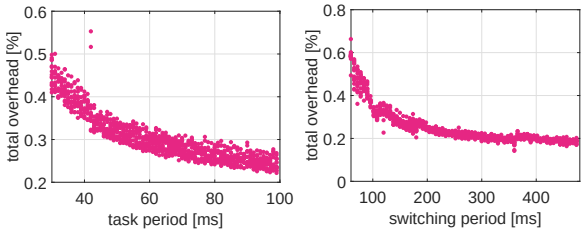
- *Task period-dependence*. The total overhead decreases when task period increases. This is expected since less jobs are created and managed.
- *Power state switching period-dependence*. Similar to the above observation, increasing the switching period mainly decreases the overhead incurred by the power manager thread.
- *Platform-dependence*. The overhead is higher (around 3×) on the Raspberry Pi platform than the overhead on Dell desktop platform. However, the overhead is still below 1% of the total run-time.

The second experiment investigates the overhead incurred by the checkpoints in our framework. We study how the overhead changes when the total number of checkpoints increases. The two types of checkpoints, i.e., the suspend and preempt checkpoints, have same number in the experiment. We vary the total number from 20000 to two million with step 20000 and repeat the experiment 50 times for each scenario. Again, we adopt the POSIX-CPU-timers to measure the time spent by checkpoints. Fig. 10 plots the overheads of different numbers of checkpoints. Following observations can be made from the figure.

- *number-dependence*. The overhead increases (approximately) linearly with respect to the total number of checkpoints, which is straightforward as the framework has to check more states.

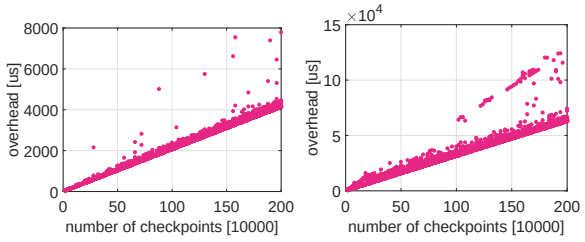


(a) Desktop, task period (b) Desktop, switching period



(c) RPI, task period (d) RPI, switching period

Figure 9: McFTP overhead in different scenarios on two platforms having different computing capabilities.



(a) Desktop (b) RPI

Figure 10: Checkpoints overhead for different platforms.

- *Platform-dependence*. On the Raspberry Pi platform, the overhead is higher (around 10×) than that on Dell desktop platform.

9. CONCLUSION

In this paper, we present the Multi-core Fast Thermal Prototyping (McFTP) framework, a new thermal framework for evaluating general thermal management policies. The variety of thermal management policies is supported by pre-implementing a set of widely used temperature control mechanisms and combining them freely. An intermediate interface named configuration manipulation interface is defined to separate the thermal management policies from low-level implementations. McFTP is designed in user-space and has little interaction to the kernel-space, thus supporting a large variety of target platforms. We implement McFTP with four basic temperature control mechanisms on top of POSIX-compliant operating systems. Its effectiveness is demonstrated by implementing two existing two works on a Dell desktop platform with a four-core Inter-I7 processor. We also investigate the efficiency of McFTP by reporting its overheads on two platforms, i.e., the Dell platform and a Raspberry Pi.

10. ACKNOWLEDGEMENTS

This work has been partly funded by China Scholarship Council, German BMBF projects ECU (grant number: 13N11936), and the Fundamental Research Funds for the Central Universities (Grant No: N161604002)

11. REFERENCES

[1] David Atienza and et al. A fast hw/sw fpga-based thermal emulation framework for multi-processor system-on-chip. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 618–623. IEEE, 2006.

[2] Andrea Bartolini, Matteo Cacciari, Andrea Tilli, Luca Benini, and Matthias Gries. A virtual platform environment for exploring power, thermal and reliability management control strategies in high-performance multicores. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, pages 311–316. ACM, 2010.

[3] Nathan Binkert and et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[4] David Brooks, Vivek Tiwari, and Margaret Martonosi. *Wattch: A framework for architectural-level power analysis and optimizations*, volume 28. ACM, 2000.

[5] J Adam Butts and Gurindar S Sohi. A static power model for architects. In *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pages 191–201. IEEE, 2000.

[6] Gang Chen, Kai Huang, and Alois Knoll. Adaptive dynamic power management for hard real-time pipelined multiprocessor systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2014.

[7] Long Cheng, Kai Huang, Gang Chen, Biao Hu, and Alois Knoll. Minimizing peak temperature for pipelined hard real-time systems. In *Design, Automation Test in Europe Conference Exhibition*. European Design and Automation Association, 2016.

[8] Simone Corbetta, Davide Zoni, and William Fornaciari. A temperature and reliability oriented simulation framework for multi-core architectures. In *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*, pages 51–56. IEEE, 2012.

[9] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd, and Toshiba Corporation. Advanced configuration and power interface specification. revision 5.0 errata a. http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf, 2013.

[10] Matthew Curtis-Maury, Karan Singh, Sally A McKee, Filip Blagojevic, Dimitrios S Nikolopoulos, Bronis R De Supinski, and Martin Schulz. Identifying energy-efficient concurrency levels using machine learning. In *Cluster Computing, 2007 IEEE International Conference on*, pages 488–495. IEEE, 2007.

[11] Yong Fu, Nicholas Kottenstette, Chenyang Lu, and Xenofon D Koutsoukos. Feedback thermal control of real-time systems on multicore processors. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 113–122. ACM, 2012.

[12] Pradeep M Hettiarachchi, Nathan Fisher, and Le Yi Wang. Achieving thermal-resiliency for multicore hard-real-time systems. In *ECRTS*, pages 37–46. IEEE, 2013.

[13] Ming-yu Hsieh, Arun Rodrigues, Rolf Riesen, Kevin Thompson, and William Song. A framework for architecture-level power, area, and thermal simulation and its application to network-on-chip design exploration. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):63–68, 2011.

[14] Wei Huang, Shougata Ghosh, Siva Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R Stan. Hotspot: A compact thermal modeling methodology for early-stage vlsi design. *IEEETransVLSI*, 14(5):501–513, 2006.

[15] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[16] Fabrizio Mulas and et al. Thermal balancing policy for multiprocessor stream computing platforms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(12):1870–1882, 2009.

[17] Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Hierarchical power management for asymmetric multi-core in dark silicon era. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–9. IEEE, 2013.

[18] Gergely Nagy and András Poppe. Simulation framework for multilevel power estimation and timing analysis of digital systems allowing the consideration of thermal effects. In *Test Workshop (LATW), 2012 13th Latin American*, pages 1–5. IEEE, 2012.

[19] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. Sesc simulator, 2005.

[20] Carta Salvatore and et al. Multi-processor operating system emulation framework with thermal feedback for systems-on-chip. In *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, pages 311–316. ACM, 2007.

[21] Robert Schöne, Daniel Molka, and Michael Werner. Wake-up latencies for processor idle states on current x86 processors. *Computer Science-Research and Development*, 30(2):219–227, 2015.

[22] Yuan Xie and Wei-Lun Hung. Temperature-aware task allocation and scheduling for embedded multiprocessor systems-on-chip (mpsoc) design. *The Journal of VLSI Signal Processing*, 45(3):177–189, 2006.

[23] Jianxun Yang and Shan Cao. An accurate power and temperature simulation framework for network-on-chip. In *Integrated Circuits and Microsystems (ICICM), International Conference on*, pages 166–171. IEEE, 2016.

[24] Jun Yang, Xiuyi Zhou, Marek Chrobak, Youtao Zhang, and Lingling Jin. Dynamic thermal management through task scheduling. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 191–201. IEEE, 2008.

[25] Changyun Zhu, Zhenyu Gu, Li Shang, Robert P Dick, and Russ Joseph. Three-dimensional chip-multiprocessor run-time thermal management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(8):1479–1492, 2008.