Dynamic Thermal Management via Architectural Adaptation

Ramkumar Jayaseelan, Tulika Mitra
School of Computing
National University of Singapore
{ramkumar,tulika}@comp.nus.edu.sg

ABSTRACT

Exponentially rising cooling/packaging costs due to high power density call for architectural and software-level thermal management. Dynamic thermal management (DTM) techniques continuously monitor the on-chip processor temperature. Appropriate mechanisms (e.g., dynamic voltage or frequency scaling (DVFS), clock gating, fetch gating, etc.) are engaged to lower the temperature if it exceeds a threshold. However, all these mechanisms incur significant performance penalty. We argue that runtime adaptation of micro-architectural parameters, such as instruction window size and issue width, is a more effective mechanism for DTM. If the architectural parameters can be tailored to track the available instruction-level parallelism of the program, the temperature is reduced with minimal performance degradation. Moreover, synergistically combining architectural adaptation with DVFS and fetch gating can achieve the best performance under thermal constraints. The key difficulty in using multiple mechanisms is to select the optimal configuration at runtime for time varying workloads. We present a novel software-level thermal management framework that searches through the configuration space at regular intervals to find the best performing design point that is thermally safe. The central components of our framework are (1) a neural-network based classifier that filters the thermally unsafe configurations, (2) a fast performance prediction model for any configuration, and (3) an efficient configuration space search algorithm. Experimental results indicate that our adaptive scheme achieves 59% reduction in performance overhead compared to DVFS and 39% reduction in overhead compared to DVFS combined with fetch gating.

Categories and Subject Descriptors

C.1.0 [**Processor Architectures**]: General

General Terms

Design, Performance, Reliability

Keywords

Dynamic Thermal Management, Architecture Adaptation

1. INTRODUCTION

Exponentially increasing power density due to technology scaling has made thermal management an important aspect of computer systems design. Traditionally, the problem of high on-chip temperature has been solved by employing more advanced packaging and cooling solutions. But modern high-performance processors are already pushing the limits of what the cooling solutions can offer. This has led to widespread interest in thermal-aware design at all levels of the computer systems. Recently, researchers have explored architectural and software-based techniques for thermal management with the aim to maximize performance while maintaining the on-chip temperature below a specified threshold. The on-chip temperature is continuously monitored and when it exceeds a predefined threshold, appropriate mechanisms are engaged to lower the temperature.

The most popular choice of mechanisms for thermal management include dynamic voltage/frequency scaling (DVFS), clock gating, and fetch gating. Unfortunately, each of these mechanisms, once engaged, is associated with significant performance degradation as well as invocation overhead. For instance, a system employing DVFS for thermal management incurs performance loss due to lower operating frequency plus non-negligible overhead in scaling voltage/frequency [17]. Hence, thermal management techniques must be judicious in choosing the severity of the response mechanism in proportion to the severity of the thermal stress.

We show that runtime micro-architectural adaptivity, such as scaling instruction window size and issue width, is more effective at managing thermal stress with minimal performance impact. These mechanisms are easy to configure at runtime and have significant impact on temperature. As most applications exhibit only limited instruction-level parallelism (ILP), they cannot exploit the wide issue width and large instruction window available in high performance processors. If these micro-architectural parameters can be scaled appropriately to track the available ILP of the program, we get significant reduction in on-chip temperature (due to reduced power dissipation) with hardly any impact on performance. On the other hand, for an application with higher ILP, we have no choice but to reduce the operating frequency of the processor to maintain on-chip temperature below the threshold. Therefore, we argue that a combination of multiple mechanisms (architectural adaptation with DVFS) can provide significantly better performance that current techniques while still satisfying the thermal constraints.

The motivation behind exploiting an adaptive micro-architecture (DVFS, fetch gating, issue width scaling and issue window scaling) for DTM is illustrated in Figure 1. The figure shows the peak temperature and throughput (in billion instructions per second or BIPS) at different processor configurations for benchmark `crafty`. Clearly, *there exist multiple configuration points with varying per-*

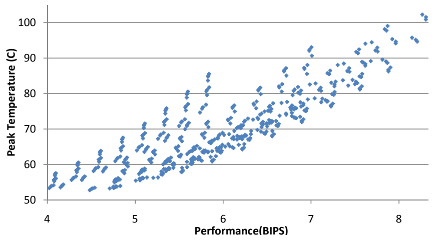

Figure 1: ***Performance versus temperature for different configuration points with benchmark crafty.***

formance that result in the same peak temperature. In other words, given a thermal constraint, appropriate choice of configuration parameters can lead to significantly higher performance.

The main challenge in using multiple mechanisms is to chose at runtime an appropriate configuration for a time varying workload. Conventional single response DTM techniques employ feedback control to determine the best performing operating point (frequency level) for a given workload/thermal stress [16]. Architecture adaptation such as window scaling and issue width scaling have not been exploited for thermal management but have been used for power/performance tradeoffs. Techniques such as [4, 12] opportunistically exploit the workload characteristics to scale micro architecture structures for power savings. However, like single response DTM, these techniques are single point optimizations and do not employ multiple adaptations synergistically.

To control multiple response mechanisms, we rephrase the thermal management problem as a configuration space exploration problem. We design a software-based framework that identifies the optimal configuration under thermal constraints. The configuration management routine wakes up once every adaptation interval (in the order of milliseconds) to collect workload statistics. These statistics are used to choose the optimal configuration that is thermally safe for the next interval.

As our DTM framework is prediction based (relies on neural network classifier to determine if a configuration is thermally safe), we assume the presence of a simple hardware DTM mechanism such as clock gating as a backup technique. In case we choose a configuration that exceeds thermal threshold (which is very rare), clock gating will be engaged to lower the temperature.

2. RELATED WORK

With temperature constraints becoming one of the key limiters of performance in computer systems, there has been widespread interest in the design of efficient thermal management techniques [16]. The goal is to maximize performance of the system while maintaining the temperature below a critical point. The on-chip temperature is monitored continuously and when it exceeds a threshold, appropriate mechanisms are engaged to lower the temperature. Commonly employed mechanisms to control temperature include fetch gating [15], activity migration [18], and DVFS [7]. There is a performance loss associated with all of these mechanisms and the design of thermal management schemes involve adjusting the extent of response (performance loss) to the severity of thermal stress. Feedback controllers have been used for this purpose and it has been shown that they achieve near optimal results [6].

We show that employing multiple mechanisms synergetically for thermal management can provide better performance. One possible approach to engage more than one response for thermal management is to determine a crossover point between the thermal management techniques as in [15] where fetch gating and dynamic voltage scaling are combined. Similarly Jung et al. [8] employ exhaustive simulation and stochastic modeling to determine the best power management policy with two configuration parameters, namely, cache size and frequency. It is unclear how to determine the cross-over point when multiple mechanisms are employed. Similarly, the large configuration space for multiple mechanisms makes exhaustive simulation like [8] infeasible. In our approach, we view the thermal management problem as a configuration space exploration problem and design an efficient online technique to determine the configuration that results in maximum performance for the workload under a given temperature constraint.

Orthogonal to existing hardware based schemes, software based thermal management have also been explored. These schemes exploit the variation in the thermal behavior of different tasks in a multitasking scenario and perform scheduling to maintain thermal constraints. Common approaches involve adjusting time slices between hot/cold tasks [10, 5] and migration [13]. Instead of changing the workload executing on the processor in response to a thermal stress, our DTM technique alters the hardware configuration for a given workload.

Adaptive hardware components that can change complexity at runtime in terms of width and size have been used previously to provide power/performance tradeoffs. Existing hardware adaptation techniques are single point optimizations that rely on local information about the workload [4] to reduce power. We use multiple adaptations synergistically along with frequency scaling for thermal management. [17] is the only other work that uses architectural adaptivity for thermal management. However, it uses off-line profiling to guide adaptation and hence is only applicable for multimedia applications. In contrast, ours is an online technique that is applicable to any workload (including multimedia applications).

3. ADAPTIVE MICRO-ARCHITECTURE

The micro-architectural parameters that we control at runtime for effective thermal management are (1) instruction window size, (2) issue width, and (3) fetch gating. These structures have been chosen as (a) it is easy to reconfigure them at runtime, and (b) they contribute (either directly or indirectly) to the thermal hotspots of the processor. In addition, we also scale the operating frequency/voltage.

We model an adaptive instruction window similar to the design in [4] that contains four partitions each of which can be enabled or disabled at runtime. The issue width can be altered between two and six instructions per cycle. When the issue width is reduced, the additional functional units are disabled and the corresponding register file ports are not precharged. The fetch unit is controlled by setting the appropriate gating level. Fetch gating level T implies instruction fetch is halted once after every T cycles. We assume special instructions to resize the adaptive structures in software [17].

Moving along each axis in our configuration space has different impact on performance and temperature. Fetch gating lowers the active power dissipation by reducing the number of instructions delivered to the back-end. Scaling the window size and issue width changes the power dissipation per operation (adaptive structures consume less power when scaled down) in addition to altering the activity factor (less number of instructions are issued per cycle at smaller issue width and window size). Finally, power dissipation reduces with reduced operating voltage/frequency.

4. DYNAMIC THERMAL MANAGEMENT

We now present our software-based DTM framework. Figure 2 presents the components of our software-based dynamic thermal management framework that exploits the adaptive micro-architecture presented in Section 3. The configuration management routine (on the right in Figure 2) runs in software. It collects the performance

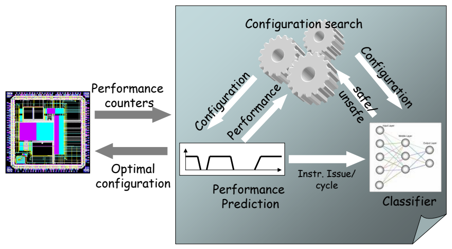

Figure 2: ***Components of the Adaptive DTM Framework.***

counters from the processor once every adaptation interval (10^7 cycles or 2.8 ms at 3.6 GHz). As temperature change occurs slowly [16], the adaptation interval is set in the order of milliseconds, which is the period for timer interrupts in many systems. These workload statistics are used to guide the choice of configuration parameters for the next interval. The goal of the configuration search routine is to find the configuration with the maximal performance that satisfies the thermal constraints. At a particular configuration $\mathcal{C}'$ in the search space, we need to answer two questions.

1. What is the expected performance of this configuration? For this purpose, we develop a model (Section 4.2) that predicts the performance of configuration $\mathcal{C}'$ given the counter values for the currently running configuration $\mathcal{C}$.
2. Is the configuration $\mathcal{C}'$ thermally safe? We design a neural network classifier (Section 4.1) that takes in configuration $\mathcal{C}'$ plus the number of instructions of each class (integer, floating point, branch, and load/store) issued per cycle as input and predicts if $\mathcal{C}'$ is thermally safe.

Note that the classifier requires the number of instructions issued per cycle as input as the temperature depends on the issued instructions. The performance, on the other hand, is determined only by the committed instructions. To bridge this gap, our performance model also estimates the number of instructions of each class issued per cycle corresponding to configuration $\mathcal{C}'$.

The configuration space of our adaptive micro-architecture consists of 1,280 points (8 fetch gating levels $\times$ 4 window sizes $\times$ 5 issue widths $\times$ 8 frequency levels). Clearly, it is not feasible to evaluate all the configurations and find the optimal one. Instead, we design an efficient search strategy (Section 4.3) that (a) reduces the four-dimensional configuration space (fetch gating levels, windows sizes, issue widths, frequency levels) to two dimensions (IPC and frequency levels) based on insights gained from the performance model, and (b) further prunes the two-dimensional configuration space based on some properties of the space. Due to these optimizations, our search strategy evaluates only a small subset of the configuration space (32 points in the worst case).

4.1 Neural Network Classifier

While searching for the optimal configuration, we need to determine if a particular configuration is thermally safe. The thermal profile of a processor typically shows large variations among the different components of the processor (up to 15^oC difference) [16]. In our adaptive micro-architecture, the temperature of a processor component depends both on the configuration parameters as well as the usage pattern of the component (workload). However, an analytical framework to determine the temperature of the different components is too computationally expensive to be employed in an online DTM framework like ours. Instead, we model the problem of determining if a particular configuration is thermally safe for the current workload as a classification problem. A classification problem consists of a set of input features, output classes and a trained classifier. When an input is given (i.e., the input features are assigned values), the classifier predicts the class to which the input belongs. In our framework, we design a neural network classifier that partitions the {configuration, workload} pairs into thermally safe and thermally unsafe classes.

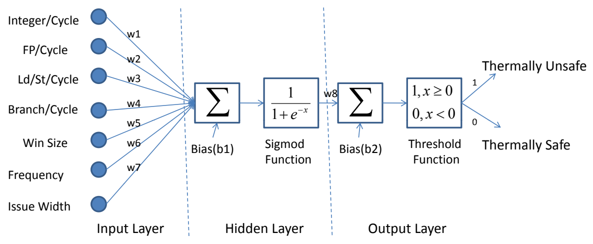

Figure 3: ***Neural network classifier architecture.***

Classifier Architecture. Figure 3 shows the structure of our neural network classifier. The input features consist of three configuration parameters: (1) instruction window size, (2) issue width, (3) operating frequency plus four workload parameters: number of integer, branch, load/store and floating point instructions issued per cycle. We choose the workload features that correlate well with the usage pattern of the branch predictor and the execution core (instruction window, register file, and execution units) — the hottest components of the processor [16]. We verify this by employing principal component analysis [1], and observe that these features account for 98% of the variance in the observed temperature.

The values of the workload features vary according to the configuration as well as the workload. For example, the number of integer instructions issued per cycle depends on the issue width, instruction window size as well as fetch gating level. Given a configuration, the workload parameters are obtained from the performance prediction model (see Section 4.2). Fetch gating level is excluded from input features of the classifier as it only alters the usage patterns of the processor components, which is reflected sufficiently in the four workload features. The rest of the configuration parameters, on the other hand, also impact power consumption per usage.

We use a neural classifier with a single hidden layer and one neuron in the hidden layer as shown in Figure 3. The hidden layer neuron uses a sigmod transfer function and the output layer neuron uses a threshold transfer function. Our classifier architecture results in high prediction accuracy with minimal classification time.

Training the Classifier. We use the Levenberg-Marquardt training algorithm [1] for training our classifier. The training algorithm is an iterative off-line procedure that adjusts the weights and bias values in the neural classifier to minimize the classification error. We train the classifier during system installation and/or when the system conditions (heat sinks, ambient conditions, etc.) change.

The training set is generated by running a set of micro-benchmarks under different configurations and checking if the resulting execution hits the thermal threshold. Each micro-benchmark consists of a loop body with 100 instructions. The number of loop iterations is large enough to ensure that the loop execution time is longer than the thermal time constant of the different processor components. The loop body of each micro-benchmark contains a mix of integer, floating point, load/store, and branch instructions. The shares of the instruction classes in this mix are generated randomly for each micro-benchmark.

Accuracy of the Classifier. We first train the classifier with 30 micro-benchmarks each running on 10 randomly chosen configurations from the configuration space. After training, we test the accuracy of our classifier for real programs by comparing the classification results against actual benchmark runs. We simulate each benchmark at 160 configuration points and determine if the execution hits threshold temperature and compare it with the corresponding classifier prediction. The results show that our neural network is highly accurate at predicting if a {workload,configuration} pair is thermally safe. The average classification error is less than 3%. Wrong classifications are of two types: (a) the classifier predicts a thermally safe configuration as unsafe (false positive), (b) the classifier predicts a thermally unsafe configuration as thermally safe (false negatives). In the case of false negative errors, the fail safe hardware DTM (clock gating) is engaged if the corresponding configuration is chosen by our search algorithm. False negative errors are observed only for 1.14% of the configurations and hence the backup hardware mechanism is rarely engaged in our scheme.

4.2 Performance Prediction Model

The performance prediction model is used as part of the configuration search process to predict the performance at a given configuration. It uses the performance counter values collected in the previous interval as input to characterize the workload and predict the performance for this workload under any given configuration.

First, we present the input and output parameters of the performance model. Let $\mathcal{C} = \langle T, IW, W, F \rangle$ denote the configuration in the current adaptation interval, where T is the fetch gating level, IW is the issue width, W is the instruction window size, and F is the frequency level. We collect the following statistics from performance counters: Number of committed instructions $N_{useful}(\mathcal{C})$, Number of cycles in the interval $Cycles$, Number of committed instructions of type X: $N^X_{useful}(\mathcal{C})$ where X can be of type integer, floating point, branch, or load/store, Instruction cache misses IC_{miss}, Data cache misses DC_{miss} and Branch mispredictions Br_{miss}.

The model is used to produce two outputs. First, it estimates performance for each configuration $\mathcal{C}' = \langle T', IW', W', F' \rangle$ that is visited; the performance is expressed as number of useful instructions committed per second (to include the effect of frequency scaling). Second, given a configuration $\mathcal{C}'$, the neural network classifier needs the number of instructions issued per cycle for each instruction class to predict if $\mathcal{C}'$ is thermally safe. Therefore, the performance prediction model has to estimate number of issued integer, floating point, branch, and load/store instructions per cycle. Note that number of issued instructions is typically more than the number of committed instructions due to branch misprediction.

Our performance prediction model is an extension of the interval analysis [9] by Karkhanis and Smith. Interval analysis is based on the notion that any superscalar processor has a sustained background level of performance that is interrupted by miss events such as cache misses and branch misprediction. Based on this assumption, the CPI (cycles per instruction) of a processor is

$$CPI = CPI_{steady} + CPI_{miss}$$

where CPI_{steady} is the background sustainable performance when there are no miss events and CPI_{miss} is the loss in performance due to branch mispredictions, instruction cache misses and data cache misses. For the current configuration $\mathcal{C}$, we get

$$CPI_{steady}(\mathcal{C}) = CPI(\mathcal{C}) - CPI_{miss}(\mathcal{C}) \text{ where } CPI(\mathcal{C}) = \frac{Cycles}{N_{useful}(\mathcal{C})}$$

CPI_{miss} can be computed from the number of miss events and their corresponding latencies and is largely independent of the configuration [9] because changing window size, fetch gating level or issue width has minimal impact on miss ratios and their penalties.

$$CPI_{miss} = CPI_{miss}(\mathcal{C}') = CPI_{miss}(\mathcal{C})$$

Now the CPI of configuration $\mathcal{C}'$ for which we are estimating the performance can be expressed as

$$CPI(\mathcal{C}') = CPI_{steady}(\mathcal{C}') + CPI_{miss}(\mathcal{C}') = CPI_{steady}(\mathcal{C}') + CPI_{miss}$$

Thus for $\mathcal{C}'$, we only need to compute the steady background performance $CPI_{steady}(\mathcal{C}')$ or $IPC_{steady}(\mathcal{C}') = \frac{1}{CPI_{steady}(\mathcal{C}')}$.

Karkhanis and Smith [9] observe that the IPC in the absence of miss-events and unbounded issue width is approximately $\sqrt{(W)}$. Under limited issue width, the IPC follows the unbounded characteristics and saturates at the issue width. Thus

$$IPC_{steady}(IW, W) = min(IW, \sqrt{W})$$

The fetch gating level affects steady IPC by changing the number of instructions delivered to the window. When the throttling level is T, the fetch unit is inactive one cycle after every T cycles of activity and $\frac{T}{T+1} \times FW$ instructions are delivered per cycle, where FW is the fetch width. At steady state, the number of instructions fetched per cycle should be equal to the number of instructions issued per cycle. Therefore, the steady IPC at configuration $\mathcal{C}' = \langle T', IW', W', F' \rangle$ can be expressed as

$$IPC^{ideal}_{steady}(\mathcal{C}') = min(\frac{T'}{T'+1} \times FW', IW', \sqrt{W'})$$

We call this ideal IPC as the characterization does not account for non-unit latency instructions, limited number of functional units of different types, and commit of multi-cycle operations [9]. To factor in these effects on the steady IPC, we compute a ratio η between *ideal* steady IPC and *observed* steady IPC for configuration $\mathcal{C}$.

$$\eta = \frac{IPC_{steady}(\mathcal{C})}{IPC^{ideal}_{steady}(\mathcal{C})} = \frac{1}{CPI_{steady}(\mathcal{C}) \times IPC^{ideal}_{steady}(\mathcal{C})}$$

As we do not adapt the latency of the functional units etc., the factor η remains constant across different configurations. So

$$IPC_{steady}(\mathcal{C}') = \eta \times IPC^{ideal}_{steady}(\mathcal{C}') = \eta \times min(\frac{T'}{T'+1} \times FW', IW', \sqrt{W'}) \tag{1}$$

$$CPI(\mathcal{C}') = \frac{1}{IPC_{steady}(\mathcal{C}')} + CPI_{miss};$$

Finally, the performance for $\mathcal{C}'$ is estimated in terms of the number of instructions committed per second

$$Performance(\mathcal{C}') = IPC(\mathcal{C}') \times F' = \frac{1}{CPI(\mathcal{C}')} \times F'$$

Estimating Issued Instructions. We also estimate the number of instructions issued per cycle at configuration $\mathcal{C}'$ ($IPC_{issue}(\mathcal{C}')$) by extending our model to count both correct path and wrong path instructions [9]. The instruction mix at issue needed by the neural classifier can be computed as

$$IPC^X_{issue}(\mathcal{C}') = IPC_{issue}(\mathcal{C}') \times \frac{N^X_{useful}(\mathcal{C})}{N_{useful}(\mathcal{C})}$$

where $IPC^X_{issue}(\mathcal{C}')$ is the number of instructions of type X issued per cycle and X can be of type integer, floating point, branch, or load/store. $N^X_{useful}(\mathcal{C})$ are input to the performance model.

Accuracy of the Performance Prediction Model. We evaluated the accuracy of our performance model for 64 randomly selected configuration points. The error is less than 5% for any benchmark and the average error for all the benchmarks is only 3.8%.

4.3 Configuration Search Strategy

We perform an intelligent search of the configuration space to determine the best performing configuration that is within the thermal limit. The search process explores the configuration space employing the neural classifier and the performance model.

Reducing Search Space. An exhaustive evaluation of all the 1,280 points in the configuration space is infeasible. We exploit insights derived from the performance prediction model in Section 4.2 to reduce the search space. From Equation 1 it is clear that that the steady IPC is constrained either by the fetch gating level (T), the window size (W) or the issue width (IW). Therefore, the performance of a processor cannot be improved by over-designing along one of the configuration parameters while restricting the other parameters — a balanced architecture provides the best performance. In other words, given a target steady IPC, we can compute appropriate values of T, W, and IW from Equation 1 as follows.

$$W = \left(\frac{IPC_{steady}}{\eta}\right)^2;\ IW = \frac{IPC_{steady}}{\eta};$$

$$T = \frac{K}{1-K} \text{ where } K = \frac{IPC_{steady}}{\eta \times FW}$$

So we can reduce the four dimensional configuration space (T, W, IW and frequency F) into a two dimensional search space consisting of only frequency and steady IPC (see Figure 4).

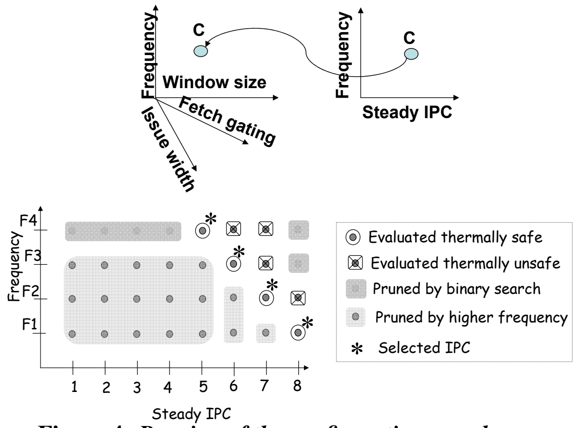

Figure 4: ***Pruning of the configuration search space.***

Pruning Search Space. In the reduced search space {steady IPC, Frequency}, any increase in either steady IPC or Frequency would result in both higher temperature and higher performance. We exploits this to further prune the search space.

1. If a configuration $\langle F, IPC\rangle$ is thermally unsafe, then all the configurations $\langle F, X\rangle$ where $X > IPC$ are also thermally unsafe and can be pruned.
2. If $\langle F, IPC\rangle$ is thermally safe, then all the configurations $\langle F, X\rangle$ where $X < IPC$ have lower performance than the known thermally safe point $\langle F, IPC\rangle$ and can be pruned.
3. If $\langle F, IPC\rangle$ is the highest performing, thermally safe configuration at frequency level F, then we can prune all the configurations $\langle F', IPC'\rangle$ where $F' < F$ and $IPC' \leq IPC$ as they are guaranteed to have lower performance than $\langle F, IPC\rangle$.

We perform a linear search along the frequency dimension and binary search along the steady IPC dimension. Our search strategy for a hypothetical search space with four frequency levels and eight IPC values is shown in Figure 4. The search starts at the highest frequency $F4$. It evaluates $\langle F4, 5\rangle$, which is the midpoint of the IPC space and determines that it is thermally safe. Therefore, points with lower values along the IPC axis are pruned. The search proceeds to the remaining points along the IPC axis and evaluates the midpoint $\langle F4, 7\rangle$ as thermally unsafe. Now the higher IPC part ($\langle F4, 8\rangle$) of the search space is pruned because they will be thermally unsafe. The search returns $\langle F4, 5\rangle$ as the feasible point with highest IPC at frequency $F4$. At the next frequency level, the configuration space $\langle F3, 1\rangle \ldots \langle F3, 5\rangle$ is pruned as the points within this space are guaranteed to have lower performance than the point $\langle F4, 5\rangle$. In this fashion, the search selects the best performing thermally safe point at each frequency level and finally chooses the highest performing point among them.

Complexity of Search Algorithm. Our algorithm has a worst case complexity of $O(L_f \times \ln(L_{IPC}))$ where L_f and L_{IPC} are the number of frequency levels and steady IPC levels. In our implementation, $L_f = 8$ and $L_{IPC} = 9$ (between 2 and 6 in increments of 0.5) resulting in 32 configuration points in the worst case. Our optimized search routine (with key optimizations such as using constants and pre-computations wherever possible, fast exponentiation [14], etc.) takes around 8,000 cycles on our simulated architecture in the worst case (32 search points). This represents an overhead of about 0.3% for a configuration interval of 1 ms.

5. EXPERIMENTAL RESULTS

We now present our experimental methodology and an evaluation of our software-based DTM management scheme against state-of-the-art DTM management techniques.

5.1 Processor Model and Workloads

We use Simple Scalar-3.0 simulator with Wattch power models [2, 3] for our experimental evaluation. We model an out-of order superscalar processor with an issue width of 6 instructions per cycle, 128 entry active list (reorder buffer), 64 entry issue window and 64 KB instruction and data caches. The model also includes a 128 entry fully associative TLB, 2MB unified L2-cache, and 4KB entry bimod branch predictor. As mentioned earlier, our adaptive architecture has four possible window sizes (16,32,48,64), five possible issue widths (2–6), and eight fetch gating levels.

We use linear scaling in Wattch to obtain the power consumption with a supply voltage of 1.4 Volt and a frequency of 3.6 GHz at 100 nm, which corresponds to the supply voltage and frequency of the Pentium 4 processor. The power consumption at different window sizes are obtained based on [17]. When the issue width is altered, we assume that the additional functional units are switched off (no leakage power) and the corresponding lines of the wakeup logic and register ports are not driven [17]. For dynamic voltage/frequency scaling, we consider eight different levels between 3.6 GHz and 2.5 GHz. We obtain the corresponding supply voltages following the methodology proposed in [16]. We assume a penalty of $10\mu s$ per frequency/voltage transition [16].

We use HotSpot-3.0 [16] for thermal simulation with a floor-plan similar to Alpha floor-plan scaled to 100 nm and a convection resistance of 1.0 K/W. The power values are collected once every $2.8\mu s$ (10^4 cycles at 3.6 GHz). We feed the power trace to the thermal model and calculate the temperatures. The leakage power is obtained based on a simple model that computes the ratio between the active power and leakage power as a function of temperature [16]. The maximum allowed temperature is 85^oC and adjusting for sensor placement and reading errors, we get a threshold of 82^oC.

We use 14 benchmarks from the SPEC 2000 benchmark suite. For each of these benchmarks, we fast forward to the simulation point specified by [11] and simulate a total of 500 million instructions. Our simulation consists of an architectural warmup phase and a thermal warmup phase after which the statistics are collected.

5.2 Dynamic Thermal Managements Schemes

We compare our software-based thermal management scheme exploiting architectural adaptation (called *adaptive DTM*) against two state-of-the-art hardware based schemes namely *DVS* and *hybrid DTM* (DVS + fetch gating). We use a PI-control based DVS scheme and use Matlab to design a PI controller with a set point of 81.8^{o}C which includes a low-pass filter to prevent frequent voltage transitions [16]. The hybrid DTM scheme [15] combines fetch gating and DVS for thermal management.

As discussed earlier, the configuration search algorithm (implemented in software) of *adaptive DTM* has worst-case overhead of 8,000 cycles. In our experiments, we assume this worst-case overhead for each invocation of the search routine. Another important parameter for *adaptive DTM* is the configuration interval or the interval at which the search routine is invoked. We set the configuration interval to 2.8ms (10^7 cycles at 3.6GHz).

5.3 Performance Comparison

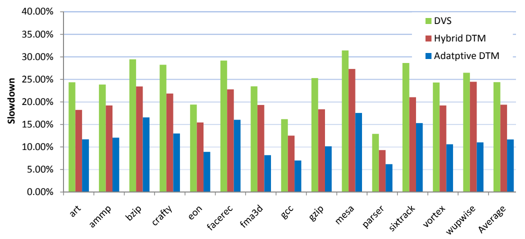

Figure 5: ***Performance comparison of different DTM schemes.***

Figure 5 plots the slowdown of the three DTM schemes compared to the baseline architecture operating at the maximum frequency (i.e., without any thermal constraints). Any DTM schemes incurs some slowdown when the temperature of the processor exceeds the threshold. It is clear that *adaptive DTM* has significant performance benefit (lower slowdown) compared to the existing DTM schemes. On an average, *adaptive DTM* has 11.68% slowdown while *DVS* and *hybrid DTM* have 24.4% and 19.37% slowdown, respectively. In other words, *adaptive DTM* has 52% reduction in slowdown compared to *DVS* and 39% reduction in slowdown compared to *hybrid DTM*. Next, we try to explain where this performance benefit of *adaptive DTM* is coming from.

5.4 Temperature Profiles and Throughput

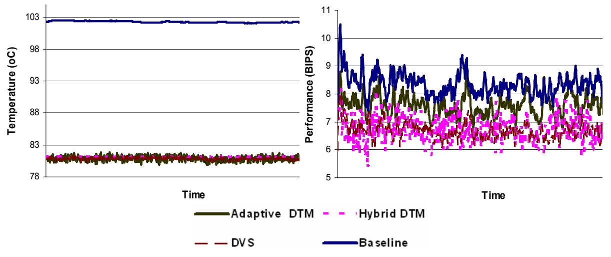

Figure 6: ***Temperature profile and throughput for crafty.***

Figure 6 plots the time varying temperature profiles and throughput for benchmark `crafty`. The temperature of the baseline configuration remains above the thermal threshold for the entire duration of execution. The DTM mechanisms keep the temperature below the threshold. The corresponding performance plots show that keeping the temperature below the threshold results in loss of performance (billion instructions per second or BIPS). The performance of all three DTM schemes are lower than the baseline. However, the performance of *adaptive DTM* is higher than *DVS* and *hybrid DTM* for most points in the plot.

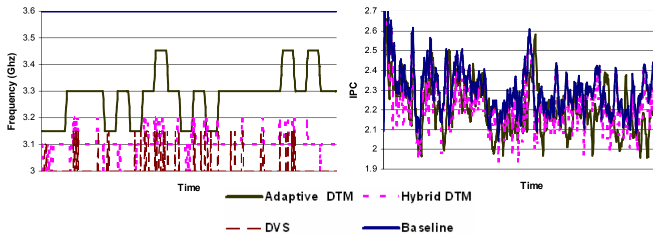

Figure 7: ***Operating Frequency and IPC plots for crafty.***

We further analyzed the loss in performance in terms of frequency and IPC components. Figure 7 shows the IPC and frequency plots for crafty. Clearly, *adaptive DTM* resulted in a higher operating frequency than *DVS* and *hybrid DTM* for the same thermal constraint and this results in an improved performance. This in because unlike the other DTM schemes, *adaptive DTM* scales the micro-architecture structures in conjunction with frequency scaling. However, this impacts IPC. Our configuration search strategy optimizes along both the frequency and IPC axes and hence achieves better performance than existing techniques.

6. CONCLUSIONS

In this paper, we explore micro-architectural adaptivity, such as scaling instruction window size and issue width, for dynamic thermal management (DTM). We formulate the thermal management issue as a configuration space exploration problem and present a software-based framework that determines the best performing configuration under thermal constraints. Our method results in 59% reduction in performance overhead in comparison to DVS based DTM scheme and 39% reduction in comparison to a hybrid scheme that combines fetch gating and DVS.

7. REFERENCES

[1] Matlab Neural Network Toolbox. `www.mathworks.com/access/helpdesk/help/pdf_doc/nnet/nnet.pdf`.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2), 2002.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-level Power Analysis and Optimizations. In *ISCA 2000*.

[4] A. Buyuktosunoglu et al. A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors. In *GLSVLSI*, 2001.

[5] J. Choi et al. Thermal-aware task scheduling at the system software level. In *ISLPED 2007*.

[6] A. Cohen et al. On Estimating Optimal Performance of CPU Dynamic Thermal Management. *IEEE Computer Architecture Letters*, 2, 2003.

[7] H. Hanson et al. Thermal response to DVFS: Analysis with an Intel Pentium M. In *ISLPED 2007*.

[8] H. Jung, P. Rong, and M. Pedram. Stochastic modeling of a thermally-managed multi-core system. In *DAC 2008*.

[9] T. S. Karkhanis and J. E. Smith. A First-Order Superscalar Processor Model. In *ISCA 2004*.

[10] A. Kumar et al. HybDTM: A Coordinated Hardware-Software Approach for Dynamic Thermal Management. In *DAC 2006*.

[11] E. Perelman, G. Hamerly, and B. Calder. Picking Statistically Valid and Early Simulation Points. In *PACT 2003*.

[12] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing Power Requirements of Instruction Scheduling through Dynamic Allocation of Multiple Datapath Resources. In *MICRO 2001*.

[13] R. Rao, S. Vrudhula, and K. Berezowski. Analytical results for design space exploration of multi-core processors employing thread migration. In *ISLPED 2008*.

[14] N. N. Schraudolph. A Fast, Compact Approximation of the Exponential Function. In *Technical Report INDISA-07-98*.

[15] K. Skadron. Hybrid Architectural Dynamic Thermal Management. In *DATE 2004*.

[16] K. Skadron et al. Temperature-aware Microarchitecture: Modeling and Implementation. *ACM TACO*, 1(1), 2004.

[17] J. Srinivasan and S. V. Adve. Predictive Dynamic Thermal Management for Multimedia Applications. In *ICS 2003*.

[18] X. Zhou, C. Yu, and P. Petrov. Compiler-driven register re-assignment for register file power-density and temperature reduction. In *DAC 2008*.