

Temperature-Sensitive Loop Parallelization for Chip Multiprocessors

Sri Hari Krishna Narayanan, Guilin Chen, Mahmut Kandemir, Yuan Xie
Department of CSE, The Pennsylvania State University
{snarayan, guilchen, kandemir, yuanxie}@cse.psu.edu

Abstract

In this paper, we present and evaluate three temperature-sensitive loop parallelization strategies for array-intensive applications executed on chip multiprocessors in order to reduce the peak temperature. Our experimental results show that the peak (average) temperature can be reduced by $20.9^{\circ}C$ ($4.3^{\circ}C$) when averaged over all the applications tested, incurring small performance/power penalties.

©2005 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

This work is supported in part by NSF Career Award 0093082, NSF grants 0130143, 0202007 and a grant from the GSRC.

Temperature-Sensitive Loop Parallelization for Chip Multiprocessors *

Sri Hari Krishna Narayanan, Guilin Chen, Mahmut Kandemir, Yuan Xie
Department of CSE, The Pennsylvania State University
{snarayan, guilchen, kandemir, yuanxie}@cse.psu.edu

Abstract

In this paper, we present and evaluate three temperature-sensitive loop parallelization strategies for array-intensive applications executed on chip multiprocessors in order to reduce the peak temperature. Our experimental results show that the peak (average) temperature can be reduced by 20.9°C (4.3°C) when averaged over all the applications tested, incurring small performance/power penalties.

1 Introduction

Power dissipation is emerging as a critical constraint preventing hardware and software designers from extracting the maximum performance from computer systems. As technology scales, higher power consumption coupled with smaller chip area will result in higher power density, which in turn will lead to higher temperatures on chip [2, 3, 5, 10]. In fact, extrapolating the changes in microprocessor organization and the device miniaturization, one can project future power dissipation density to 200 W/cm² [14]. This requires extensive efforts on cooling techniques.

While hardware solutions to temperature management problem are very important, software can also play an important role because it determines the circuit components exercised during the execution and the period of time for which they are exercised. In particular, compilers determine the data and instruction access patterns of applications, which shape the power density profile.

We present and evaluate three temperature-sensitive loop parallelization strategies for array-intensive applications executed on chip multiprocessors. The proposed strategies start with a pre-parallelized code and re-distribute loop iterations across processors – at compile time – in such a way that the temperature of each

processor is reduced, without affecting performance and power consumption very adversely. To do this, we first divide the iteration space of the loop nest being optimized into multiple chunks of equal sizes, and for each chunk, our approaches determine the best processor to use, taking into account load balance, temperature, and data locality. If the scheduling algorithm is solely *temperature-driven* it may suffer from poor data locality, on the other hand, if it is solely *locality-driven* then it may suffer from poor load balancing. Thus we propose a *combined approach* that has the advantages of both temperature-driven and locality-driven scheduling.

Our experimental results reveal that the proposed parallelization strategy is very successful in practice. Specifically, it contains the peak temperature of the processors under a threshold temperature with little performance/energy penalties. In addition, the proposed approach reduces the peak (average) temperature by 20.9 °C (4.3 °C) when averaged over all the applications tested, incurring extra performance/power penalties within a small range.

2 Related work

Chip multiprocessors are architectures with multiple processors on a single chip [4]. Prior architecture-related studies in the context of chip multiprocessors include [1, 8, 13]. Reference [6] identifies the ideal number of processors to use when optimizing a particular execution to minimize runtime or the energy consumption.

Temperature hotspots have been identified as an important design concern in modern processors [10, 11, 12]. Research on solving this problem has mainly focused on runtime techniques. Other techniques involve hotspot reduction through dynamic power management which is in turn achieved by techniques such as voltage scaling. Activity migration [5] reduces temperature in a chip by dynamically ping-ponging jobs on multiple processors.

*This work is supported in part by NSF Career Award 0093082, NSF grants 0130143, 0202007 and a grant from the GSRC.

```

for (i=1; i≤600; i++)
  for (j=1; j≤1000; j++)
    B[i][j] = (A[i-1][j] + A[i+1][j] +
              A[i][j-1] + A[i][j+1]) / 4;
  
```

(a)

Slot	P ₀	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆	P ₇
1	0	6	12	18	24			
2	1	7	13	19	25			
3	2	8	14	20	26			
4	3	9	15	21	27			
5	4	10	16	22	28			
6	5	11	17	23	29			

(b)

```

for (i=k*120+1; i≤(k+1)*120; i++)
  for (j=1; j≤1000; j++)
    B[i][j] = (A[i-1][j] + A[i+1][j] +
              A[i][j-1] + A[i][j+1]) / 4;
  
```

(c)

Figure 1. Jacobi computation and a possible schedule.

Our work is different from these in that it is static in its approach and it targets parallel loops running on chip multiprocessors. All decisions on task assignments to processors are made at compile time. Furthermore, we optimize the cache data reuse in assigning jobs to processors by calculating the overlap of existing data in cache and the data that will be accessed by the task to be allocated to that processor. This is possible because the decision to migrate computation is not taken in the face of a thermal emergency, but rather in a calculated proactive manner dictated by the compiler.

3 Preliminaries

The architecture being simulated is an 8 core chip multiprocessor. The details of each core is given in Table 1.

The loop parallelization problem can be regarded as a process of *partitioning* the iteration space of a given loop nest into *chunks* (sets) of successive iterations and *scheduling* the iteration chunks into appropriate processors. The resultant parallelized code can be represented using a *schedule table*. Figure 1 illustrates an example. Figure 1(a) gives the loop nest for the Jacobi Solver computation. Each iteration of this loop nest can be represented using a two-entry vector $(i, j)^T$, and there are a total of 600×1000 iterations in this loop nest. Let us assume that we divide the iteration space into 30 equal size chunks with each chunk having 20×1000 iterations. Each chunk can be represented using \mathcal{I}_k ($0 \leq k < 30$), where \mathcal{I}_k is defined as $\mathcal{I}_k = \{(i, j)^T \mid 20 * k < i \leq 20 * k + 20 \ \& \ 1 \leq j \leq 1000\}$. The mapping of the iteration space to the data space of array A which is accessed by the references shown can be explained as $\mathcal{D}_A = \mathcal{D}_A^{(-1,0)} \cup \mathcal{D}_A^{(1,0)} \cup \mathcal{D}_A^{(0,-1)} \cup \mathcal{D}_A^{(0,1)}$, where

$$\mathcal{D}_A^{m,n} = \{(i, j) \rightarrow [a, b] \mid a = i + m \ \& \ b = j + n\} \quad (1)$$

In this work, we use the Omega Library [7] and Presburger arithmetic [9] to represent the iteration chunks

Benchmark	N1	N2	N3	N4	N5	N6	N7
3step-log	1	1	1				
adi	4	1					
btatrix	1	1	1	1	1	1	3
efflux	1	2					
tsf	1	2	1	1			

Figure 2. Optimum number of processors.

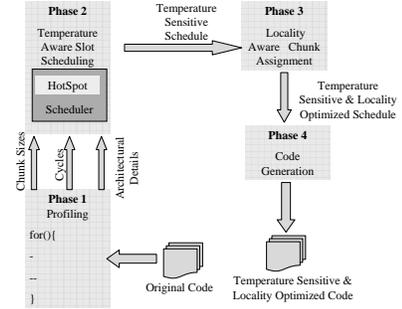


Figure 3. Overview of our methodology.

and the data set they access. Figure 1(b) gives a possible schedule for the 30 iteration chunks on 5 out of 8 processors. If iteration chunk \mathcal{I}_k is scheduled to be executed on processor P_n at time slot m , the corresponding table entry is k . An entry is left empty if the processor is idle at that time slot. For example, \mathcal{I}_9 is scheduled on P_1 at time slot 4, and processors P_5 , P_6 , and P_7 are idle during the entire execution. Figure 1(b) represents in a sense a parallelized version of the original loop nest in the form a schedule table, and the corresponding code to be executed on processor P_k ($0 \leq k \leq 4$) is given in Figure 1(c).

Prior study [6] shows that, in many cases, using only a subset of the available processors generates the best result in terms of performance and energy. That is, increasing the number of processors used can degrade performance and/or can increase energy consumption in some circumstances. Such a scenario may happen due to parallelization overheads, data dependencies, synchronization, etc. Figure 2 presents the optimum number of processors for each nest of each benchmark code used in our experiments (the information on our benchmarks is given in Section 5). Note that increasing the number of processors (for any nest) further increases overall energy consumption without any benefits. In our discussion of the different scheduling algorithms, we will stick to such predetermined *active processor count limit*, referred to as A in the rest of the paper. In all the schedules obtained from our scheduling algorithms we use no more than A processors at each time slot .

Due to the reducing scale of technology, the power density of chips has increased. As a processor expends power, the increased power density causes a rise in the operating temperature of the processor.

If a processor operates above a safe threshold tem-

perature, it could lead to the burning and permanent damage of the chip. In order to prevent this, the chip should never be allowed to exceed the threshold temperature. This is called the *threshold temperature constraint*. Our work is a compiler driven approach that determines a schedule at compile time that will prevent the chip from operating at a super threshold temperature. Such a schedule lets a processor be idle periodically so that it cools down. However, this should be performed in a performance-sensitive manner.

To calculate the rise and fall of the temperature of a processor, we use the following framework which is based on the Hot-Spot tool [12]. Assuming that a processor’s temperature at cycle c is T_c , a processor’s temperature after running for δ cycles can be estimated using:

$$T_{c+\delta} = F(T_c, \delta, \text{Computation}_\delta, \text{floorplan}, \text{power}) \quad (2)$$

That is, a processor’s temperature at a specific time is determined by its previous temperature (T_c), the power it expends in δ , the number of cycles during which it expends this power as well as the physical layout and properties of the processors.

The granularity of our scheduling and hence thermal model is an iteration chunk, i.e., we estimate a processor’s temperature at the end of one time slot (the time for running an iteration chunk) based on its temperature at the end of the previous time slot and its power consumption and computation during the current time slot. The power consumption is a function of the computation performed by the processor in the current time slot. Since all the iteration chunks are of equal size, we assume that the computation to execute each iteration chunk is the same. The time and energy required for executing a chunk is determined through profiling.

As a result, at compile time, we are aware of the computation chunks, the time they take to execute, and the power each chunk expends. Furthermore, the floor plan of the system is constant. Hence, given a starting temperature, our compiler can estimate the temperature changes due to the execution of the application statically. In order to do so, a scheduler based around the Hot-Spot tool was created. Hot-Spot is used to return the temperature of a processor at the end of each iteration chunk. This information is used to decide in the scheduler whether an iteration can be scheduled to that particular processor or not. To prevent the temperature of a processor from rising above the threshold temperature, our scheme lets a processor be idle periodically to it cool down. Computation is scheduled on the processor once its temperature has fallen sufficiently.

Algorithm 1 *Temperature driven scheduling.*

```

1:  $TimeSlot \leftarrow 0$ ;
2: while exist nonscheduled iteration chunk do
3:   for all processors do
4:      $\mathcal{P} \leftarrow$  the  $A$  coolest processors;
5:     delete from  $\mathcal{P}$  the processors with threshold temperature;
6:   end for
7:   schedule an iteration chunk for each processor in  $\mathcal{P}$  at  $TimeSlot$ ;
8:   remove these iteration chunks;
9:    $TimeSlot \leftarrow TimeSlot + 1$ ;
10: end while

```

Figure 3 illustrates our scheme. In the first phase, the code is profiled and information such as the number of cycles it takes to execute, the size of iteration chunks, and the energy they consume are extracted. In the second phase, this information along with the details of the architecture being simulated are fed as input to our scheduler, which returns a temperature-sensitive schedule. Next, in the third phase, this schedule is optimized for data cache locality. Finally in Phase 4, the Omega Library is used to generate code based on the schedule.

4 Our approach

4.1 Temperature-driven loop scheduling

Let us now discuss an approach to temperature-driven loop scheduling. Algorithm 1 gives a sketch of the algorithm for the temperature-driven loop scheduling approach. In this algorithm, we schedule the iteration chunks one time slot at a time. At each step, we select the coolest (in terms of temperature) processors, and the number of such processors does not exceed A . We estimate a processor’s temperature from its temperature and activity in the previous time slot using the temperature estimation function (Equation (2)). By selecting the coolest processors at each time slot, we reduce the overall temperature. It is possible that there are less than A schedulable iteration chunks at some time slots because of the threshold temperature. We repeat this process until all the iteration chunks have been scheduled.

Figure 4(a) gives the schedule generated by this algorithm for the example program given in Figure 1, assuming that the value of A is 5. For all the algorithms presented in the paper, we assume the following simple temperature estimation function for each processor p :

$$T_{c+1}(p) = \begin{cases} T_c(p) + 1 & \text{if } p \text{ runs at time slot } c; \\ T_c(p) - 1 & \text{if } p \text{ is idle at time slot } c. \end{cases} \quad (3)$$

We assume further that the initial temperature is 0, and the threshold temperature is 2. This temperature-driven schedule has better thermal behavior than the original one (in terms of both the average and peak

temperatures), and it uses only one more time slot than the original schedule. However, an important problem with this schedule is that it does *not* consider data locality. By looking at the program given in Figure 1, one can see that data reuse happens between the neighboring iteration chunks. In the original schedule given in Figure 1(b), data locality is optimized since most of the iteration chunk pairs that share data are scheduled on the same processor successively. But the schedule given in Figure 4(a) does not have this good locality behavior. Actually, none of the iteration chunks scheduled on processor P_4 have data reuse among them. The degraded data locality in this temperature-driven schedule is mainly due to the fact that we did not consider any data reuse when we schedule the iteration chunks. In the next subsection, we discuss a data locality-driven scheduling approach.

4.2 Data locality-driven loop scheduling

Algorithm 2 gives a sketch of our locality-driven loop scheduling algorithm. This algorithm is locality-driven since, at each step, it tries to determine the chunk-processor pair, (\mathcal{I}, p) , that is the best in terms of data reuse (lines 8–10 in Algorithm 2). In our work, the potential data reuse achieved by scheduling iteration chunk \mathcal{I} on processor p is obtained by calculating the intersection set of the data elements accessed by \mathcal{I} (calculated using a mapping similar to Equation 1), and those accessed by the last-scheduled iteration chunk on p . That is, if the set of data elements accessed by \mathcal{I} is \mathcal{D}_1 , and the set of data elements accessed by the last-scheduled iteration chunk on p is \mathcal{D}_2 , the data reuse is determined by the size of set $\mathcal{D}_1 \cap \mathcal{D}_2$. As has been discussed in Section 3, in this work, the data elements accessed by an iteration chunk are obtained using the Omega Library tool.

Note that we still need to make sure that the resultant schedule satisfies the active processor count limit and threshold temperature constraint. The array $NextSlot$ in Algorithm 2 captures such requirement. Specifically, for each processor p , $NextSlot[p]$ gives the next available time slot to which we can schedule an iteration chunk without violating the active processor count limit and the threshold temperature constraint. After a new iteration chunk is scheduled, $NextSlot$ is updated to reflect the change in the schedule (line 12 in Algorithm 2).

It might happen under this scheduling algorithm that some processors have many more iteration chunks scheduled on them than other processors, since a pure data locality-driven loop scheduling algorithm does not guarantee load balance among the processors. Such a

Algorithm 2 Data locality driven scheduling.

```

1:  $\forall$  processor  $p$ :  $NextSlot[p] \leftarrow 0$ ;
2:  $Bound \leftarrow \lceil (\text{number of iteration chunks})/A \rceil$ ;
3: while exist nonscheduled iteration chunk do
4:   if none of the processors is schedulable then
5:     increase  $Bound$  by  $\lceil (\text{number of nonscheduled iteration chunks})/A \rceil$ ;
6:     for each processor  $p$ ,  $p$  is schedulable if  $NextSlot[p] < Bound$ ;
7:   end if
8:   for all nonscheduled iteration chunks and all schedulable processors do
9:     find the best iteration chunk and processor pair  $(\mathcal{I}, p)$  such that data reuse is maximum
10:  end for
11:  schedule iteration chunk  $\mathcal{I}$  on processor  $p$  at time slot  $NextSlot[p]$ ;
12:  update  $NextSlot[]$  for all processors;
13:  if  $NextSlot[p] \geq Bound$  then
14:     $p$  is not schedulable;
15:  end if
16: end while

```

Slot	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
1	0	1	2	3	4			
2	5	6				7	8	9
3			10	11	12	13	14	
4	15	16	17	18				19
5					20	21	22	23
6	24	25	26	27	28			
7						29		

(a)

Slot	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
1	0	4	8	12	16			
2	1	5	9	13	17			
3						20	22	24
4	2	6	10	14	18			
5						21	23	25
6	3	7	11	15	19			
7						27	28	26
8							29	

(b)

Figure 4. Example schedule (a) using algorithm 1, (b) using algorithm 2.

scenario should be avoided since the total number of time slots to finish all the iteration chunks in this scenario can be much higher than that of a load-balanced schedule. The term $Bound$ in Algorithm 2 addresses this scenario.

As one can observe from Figure 4(b), Algorithm 2 can generate a schedule that exhibits much better data locality than the schedule in Figure 4(a) generated by the temperature-driven algorithm (Algorithm 2). However, we also observe that the schedule given in Figure 4(b) requires one more slot to finish than the schedule given in Figure 4(a). Therefore, we can conclude that these two scheduling algorithms have their own advantages and disadvantages. In the next subsection, we propose a loop scheduling algorithm that combines these two algorithms and has the advantages of both of them.

4.3 A combined approach to scheduling

The idea behind our combined approach is straightforward. Since the temperature-driven approach is good in terms of load balancing, we first use the temperature-driven approach to determine for each processor its state (i.e., running or idle) at each time slot. After that, we use a locality-driven approach to determine the iteration chunks to be executed on each processor so that data locality is optimized. Algorithm 3 gives the algorithm for our combined approach. In the first part of this algorithm (lines 1–11), we determine the time slots at which each processor should be running an iteration chunk. This information

Algorithm 3 *Algorithm for the combined approach.*

```

1: mark all processors as idle for all time slots;
2:  $TimeSlot \leftarrow 0$ ;
3: while exist nonscheduled iteration chunk do
4:   for all processors do
5:      $\mathcal{P} \leftarrow$  the  $A$  coolest processors;
6:     delete from  $\mathcal{P}$  the processors with threshold temperature;
7:   end for
8:   mark each processor in  $\mathcal{P}$  as running at  $TimeSlot$ ;
9:   remove these iteration chunks;
10:   $TimeSlot \leftarrow TimeSlot + 1$ ;
11: end while
12:  $\forall$  processor  $p$ :  $NextSlot[p] \leftarrow$  the first running time slot for  $p$ ;
13: while exist nonscheduled iteration chunk do
14:   for all nonscheduled iteration chunks and all schedulable processors do
15:     find the best iteration chunk and processor pair  $(\mathcal{I}, p)$  such that data
       reuse is maximum
16:   end for
17:   schedule iteration chunk  $\mathcal{I}$  on processor  $p$  at time slot  $NextSlot[p]$ ;
18:    $NextSlot[p] \leftarrow$  the next running time slot for  $p$ ;
19:   if no next running time slot for  $p$  then
20:      $p$  is not schedulable;
21:   end if
22: end while

```

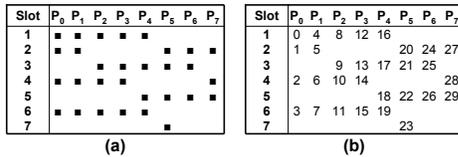


Figure 5. Example schedule using algorithm 3.

is exploited in the second part of the algorithm (lines 12–22) to determine the next schedulable time slot for each processor. In the second part, a data locality-driven approach is used to assign the iteration chunks to each processor’s running time slots for data locality optimization. Note that determining $NextSlot$ and schedulable processors is much simpler in Algorithm 3 compared to Algorithm 2, since we can utilize the information from a temperature-driven scheduling. We can conclude from the above discussion that Algorithm 3 combines the advantages of both a temperature-driven algorithm and a locality-driven algorithm. Therefore, this combined approach is expected to perform well in terms of both load-balancing and data locality.

Figure 5 gives a sample schedule obtained after applying our combined approach on the code presented in Figure 1. We first use the temperature-driven approach to obtain the active time slots for each processor. The results are given in Figure 5(a). Note that Figure 5(a) is the same as Figure 4(a) except that the entries in Figure 5(a) are represented as black boxes (which means

Table 1. Architectural details.

Parameter	Brief Explanation
Processor	300MHz single issue
Chip Area	$5.6mm^2$
L1 Data & Instr. Cache	Private, write through, 64 lines, 4 way, lru
L2 Cache	Shared, write back, 64 lines, 4 way, lru

Table 2. Benchmarks used.

Bench-mark	Cycles (Million)	Energy (μ J)
3step-log	1487	1894686.2
adi	438	1239551.1
btrix	1351	1802476.2
efflux	56	80918.1
tsf	1799	2548001.6

the corresponding iteration chunk to be executed has not been determined yet) rather than using iteration chunk numbers. After that, we apply the data locality-driven scheduling approach to the schedule table in Figure 5(a), and we obtain the final schedule shown in Figure 5(b). Compared with Figure 4(a), the schedule given in Figure 5(b) has much better data locality since most of the iteration chunks that have data reuse are scheduled successively (on the same processor) in Figure 5(b). The schedule given in Figure 4(b) also has good data locality, but it needs one more time slot to finish execution compared to the schedule in Figure 5(b).

5 Experimental results

We evaluate our temperature-sensitive loop parallelization scheme using five array-intensive codes. Their important characteristics are given in Table 2. All the values listed in this table are obtained by executing the applications without any temperature-sensitive loop scheduling. In the rest of our discussion, we refer to this version of an application as the *base version*).

Figure 6(a) gives the peak temperature curves for the execution of *adi*. The dashed line represents the threshold temperature. In Figure 6(a), using the original temperature-insensitive parallelization strategy, the peak temperature of the processors increases and exceeds the threshold temperature shortly after the start of execution. On the other hand, after using our temperature-sensitive loop parallelization strategy (the combined approach, referred to as “temperature sensitive”), we can effectively reduce the peak temperature of the processors. Specifically, the peak temperature of the processors is always kept below the threshold temperature as can be observed from Figure 6(a). Table 3 gives the experimental results of our temperature-sensitive loop parallelization approach. As we can observe from Table 3, our approach reduced the peak temperature and average temperature by 20.9 degrees and 4.3 degrees, respectively, when averaged over all the application tested (with respective the the base version). In addition, our approach achieves the significant tem-

Table 3. Results of our combined approach.

Benchmark Name	Peak Temperature		Average Temperature		Extra Energy Consumption	Extra Execution Cycles
	Original	Optimized	Original	Optimized		
3step-log	95.5	80.7	80.7	78.7	2.4%	1.8%
adi	146.1	86.8	100.5	85.0	2.4%	9.1%
btrix	84.9	78.9	74.1	73.9	0.8%	0.6%
eflux	84.9	74.2	76.4	73.7	7.4%	4.0%
tsf	87.6	74.2	80.0	73.0	1.6%	1.2%
average	99.8	78.9	81.2	76.9	2.9%	3.3%

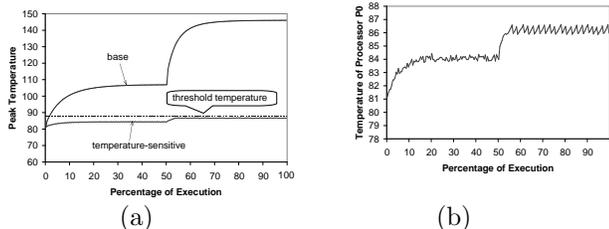


Figure 6. (a) Peak temperature of hottest processor at anytime for *adi*. The dashed line represents the threshold temperature. (b) Temperature curves of processor P_0 for *adi*.

perature reduction with only a small amount of overhead in terms of energy consumption and execution cycles. Specifically, the energy consumption and execution cycles increase by 2.9% and 3.3% respectively. Figure 6(b) gives the temperature curve of processor P_0 for *adi*. Using the original temperature-insensitive parallelization strategy, P_0 's temperature quickly goes beyond the threshold temperature (not shown). Using our temperature-sensitive approach, as processor P_0 's temperatures approaches the threshold temperature, it will not be scheduled since its temperature is too high. That is, P_0 is kept idle in order for it to cool down. The decreasing slopes indicate the idle periods. Note that, during these periods, other processors that are cooler may be scheduled to execute iteration chunks. When P_0 's temperature is low enough, it can execute iteration chunks (i.e., becomes active). The increasing slopes indicate the active periods. We can observe from Figure 6(b) that our approach can effectively prevent a processor from going beyond the threshold temperature.

In obtaining the results presented above, we set the threshold temperature parameter used in our compiler algorithm according to the physical temperature constraint, so that the peak temperature of the processors using our scheme will not exceed the physical temperature constraint. It is possible for us to lower the peak temperature further by lowering the threshold temperature parameter used in our scheme. We do not go into this further due to space constraints. Overall, we can conclude that our approach can achieve significant temperature reduction without incurring too much performance/power penalties.

6 Concluding remarks

Many advances in computer technology have been made possible by increases in the packaging density of electronics. However, with ever-increasing levels of power consumption, power density is starting to become a serious issue before computer architects and software writers alike. As against the prior hardware-based solutions to this problem, this paper focuses on compilers, and proposes three temperature-aware loop parallelization schemes for chip multiprocessors. The *combined* approach reduces the peak (average) temperature by 20.9 °C (4.3 °C) when averaged over all the applications tested.

References

- [1] L. A. Barroso et al. Piranha: A scalable architecture based on single-chip multiprocessing. *ISCA*, June 2000.
- [2] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. *HPCA*, 2001.
- [3] J. Donald and M. Martonosi. Temperature-aware design issues for SMT and CMP architectures. *WCED-5*, June 2004.
- [4] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single chip multiprocessor. In *IEEE Computer*, pp. 79–85, 1997.
- [5] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. *ISLPED*, 2003.
- [6] I. Kadayif, M. Kandemir, and M. Karakoy. An energy saving strategy based on adaptive loop parallelization. *DAC*, 2002.
- [7] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Sheipman, and D. Wonnacott. The Omega calculator and library, version 1.1.0. November 1996.
- [8] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. *MICRO*, 2001.
- [9] W. Pugh. Counting solutions to Presburger formulas: how and why. *PLDI*, 1994.
- [10] E. Rohou and M. Smith. Dynamically managing processor temperature and power. *FDDO-2*, 1999.
- [11] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. *HPCA*, 2002.
- [12] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. *ISCA*, June 2003.
- [13] M. B. Taylor et al. The RAW microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, March 2002.
- [14] http://www.hpl.hp.com/research/dca/smart_cooling/.