Performance Implications of Single Thread Migration on a Chip Multi-Core

Theofanis Constantinou*, Yiannakis Sazeides*, Pierre Michaud+, Damien Fetis+, and Andre Seznec+
*Department of Computer Science
University of Cyprus, Nicosia, Cyprus
+Irisa/Inria
Campus de Beaulieu 35042, Rennes Cedex, France

Abstract

High performance multi-core processors are becoming an industry reality. Although multi-cores are suited for multi-threaded and multi-programmed workloads, many applications are still mono-thread and multi-core performance with a single thread workload is an important issue. Furthermore, recent studies suggest that performance, power and temperature considerations of future multi-cores may necessitate activity-migration between cores.

Motivated by the above, this paper investigates the performance implications of single thread migration on a multi-core. Specifically, the study considers the influence on the performance of a single thread of the following migration and multi-core parameters: frequency of migration, core warm-up modes, subset of resources that are warmed-up, number of cores, and cache hierarchy organization. The results of this study can provide insight to architects on how to design performance-efficient power and thermal strategies for a multi-core chip.

The experimental results, for the benchmarks and microarchitectures used in this study, show that the performance loss due to activity migration on a multi-core with private L1s and a shared L2 can be minimized if: (a) a migrating thread continues its execution on a core that was previously visited by the thread, and (b) cores remember their predictor state since their previous activation (all other core resources can be cold). The data also show that the transfer of the register state between two cores can be slow*, latency of several 100s of cycles, without limiting performance.*

1 Introduction

For the past 40 years, technology advances have enabled continuous miniaturization of circuits and wires and decreases in clock cycle time. Unfortunately, the shrinking and speeding up of circuits has not been accompanied by a similar decrease of the on chip power consumption [26, 10]. In fact, with each technology generation processors are becoming increasingly power-inefficient. Power-inefficiency has at least two negative consequences, shorter battery life for mobile devices [7] and higher power–density [24, 2, 22]. A rise in power–density can produce thermal hot–spots and cause timing errors, and even physical damage to circuits. Consequently, power constraints may force to limit the clock frequency and, therefore, pose a major performance challenge for future microarchitectures.

The response from industry to this challenge are power and thermal management strategies [24, 11, 9, 7, 5, 21], and from academia a plethora of methods that facilitate power and temperature aware design.

At the same time, diminishing returns from increasing the issue width on superscalar processors have lead to the emergence of single-chip multi-core processors [20]. A two-way multi-core is already a reality [27, 14, 19, 1, 13] and with increasing on-chip capacity many more cores will soon be available on a single chip [8, 15]. Multi-cores offer the means to increase thread and program level parallelism but they can also be leveraged to overcome power limitations using activity–migration [18, 12] between cores.

Activity–migration in a heterogeneous multi-core [18, 17] can improve power-efficiency by transferring the execution of a thread to a core that better matches its power needs. On a homogeneous multi-core, activity-migration can help alleviate the power-density(temperature) problem by distributing power consumption more uniform over the entire chip.

Nonetheless, while multi-cores can be efficiently designed and built, many applications remain single-threaded. Therefore, multi-core performance for a single thread workload is a major issue. This paper investigates the performance implications of activity migration when a single thread executes on a multi-core.

A multi-core capable of thread migration may have additional requirements, as compared to a conventional multi-core. In particular, every migration requires, for correctness, the transfer of some state, such as architectural registers, from one core to the other. Furthermore, microarchitectural structures in the new core, such as caches and predictors, may provide better performance when they are warmed-up. A resource on an inactive core can be warmed-up by explicit updates from a currently executing core and/or can remain warm by retaining its state from its last activation.

We explore various dimensions of the design space of a multi-core that facilitate thread migration. In particular, we will consider the following parameters: latency and frequency of migration, core warm-up modes, subset of resources that are warmed-up, number of cores, and cache hierarchy organization. One of our main goals is to understand the effects of thread migration over a range of migration periods. We do not, therefore, consider migrations that are triggered by "real" criteria but rather we examine the behavior with various fixed interval sizes and for several random distributions assuming a constant clock frequency.

Our intention is to provide the designers of a multi-core processor, with insight on the impact of thread migration on performance and with information on how to best design a power and

thermal strategy for a multi-core microarchitecture that supports activity-migration.

The main findings of our work, is that the performance losses due to activity migration for a multi-core with private L1 and shared L2 caches can be minimized by remembering the predictor state between migrations while all other resources can be turned off. The results also show that the migration latency for transferring register state and flashing local data cache is not critical to performance.

The paper is organized as follows. In Section 2 we discuss related work. Section 3 describes the microarchitecture and presents the migration model used in this study and discusses its various parameters. Section 4 presents the simulation framework. The experimental results and their discussion are given in Section 5. Section 6 presents possible applications where the findings of this work can be useful for. Section 7 concludes the paper and provides direction for future work.

2 Related Work

To the best of our knowledge, the notion of activity migration as the means to reduce power–density was independently proposed by [18, 12]. In [18] a microprocessor with two pipelines is suggested, a primary high-power out-of-order superscalar pipeline and a low-power in-order pipeline. When thermal-sensors detect a hot-spot while executing in the primary pipeline, execution can migrate to the secondary pipeline to reduce temperature. Activity migration at regular time intervals was investigated in [12] to increase the sustainable power dissipation for a given constant peak temperature or to reduce temperature for a constant frequency. The experimental evaluation considered various configurations of a superscalar processor that have one or more duplicated resources. The best performance was obtained when having an entire pipeline duplicated and migrating execution across the two pipelines.

Activity migration techniques for reducing power–density were also considered in previous temperature work [25] where access to an overheated integer register file are directed to a duplicated register file until the primary cools down, and for cluster microarchitectures where activity migrates between back-end processing elements [4]. Another [23] recently proposed approach to reduce power-density is to leverage simultaneous-multithreaded (smt) cores in a multi-core to co-schedule processes that stress complementary resources and to migrate a thread from an overheated core to a core with an available smt context.

Activity migration for better power-efficiency was proposed by [18, 16]. In [18] migration to a low-power pipeline occurs when extended battery-life is required for executing a lightweight application. In [16] a heterogeneous chip-multiprocessor is proposed to facilitate power-efficiency by migrating, at the granularity of an operating-system interval, a process to the core that is expected to provide the best power-efficiency.

What distinguishes our work, from earlier activity-migration research, is the problem parameters we consider and the analysis of their combined effects.

The work that most closely resembles ours is the seminal work on activity migration [12]. The study in [12] considers various types of resource replication to address the power-density problem when a single thread is executing on a chip. One of the migration scenarios examined was effectively for a dual multi-core. However, the study in [12] did not quantify the effect and interaction of several parameters, for example the impact of changing migration frequency, increasing the number of cores, the subset of resources that are warmed-up, the warm-up modes, cache hierarchy etc.

3 Migration Model

In this section we present first the microarchitecture of the multi-cores used in the study and then discuss the model for migration in these multi-cores.

3.1 Multi-core Architecture

This work considers a multi-core architecture, shown in Fig. 1.a, with private L1 data and instruction caches and a shared L2 cache. Both the L1 data and L2 caches are write-back and write-allocate. Next we introduce the model for describing migration scenarios and trade-offs.

3.2 Thread Migration Model

The various phases required in our model for a thread migration between two cores are illustrated in Fig. 1.b. During the execution of a thread on a core, *P1*, a temperature threshold has been reached, and it is decided that the execution should be transferred to an inactive core, *P2*. As soon as this decision is made, a migration signal is send to the two cores and *P2* is activated. When *P2* is activated, it enters first the initialization-phase where its resources are turned-on. Some resources on inactive cores may be already on to preserve their state between activations for performance reasons. However, it may be undesirable to keep all resources on since the power consumption of inactive cores may limit the potential of activity-migration. During the initialization phase of *P2*, core *P1* continues in normal execution and therefore performance is not hurt. However, the length of the initialization phase places a limit as to how often we can migrate. According to [16] bringing up a core that is completely shut down requires about one thousand cycles. We also assume that our worst case initialization latency will require one thousand cycles and, therefore, the smallest migration interval we consider is larger than that.

After the initialization-phase, we may need to enter the train phase where some of the resources of *P2*, such as caches and predictors, are trained based on the outcome of instructions executing on core *P1*. Training may require a dedicated update-bus for communication between cores. The length of a train phase should not be very long because training occurs when a temperature or power threshold is exceeded on *P1*, and together with the initialization phase place a limit to the frequency of migration. Furthermore, the train phase incurs its own energy overhead due

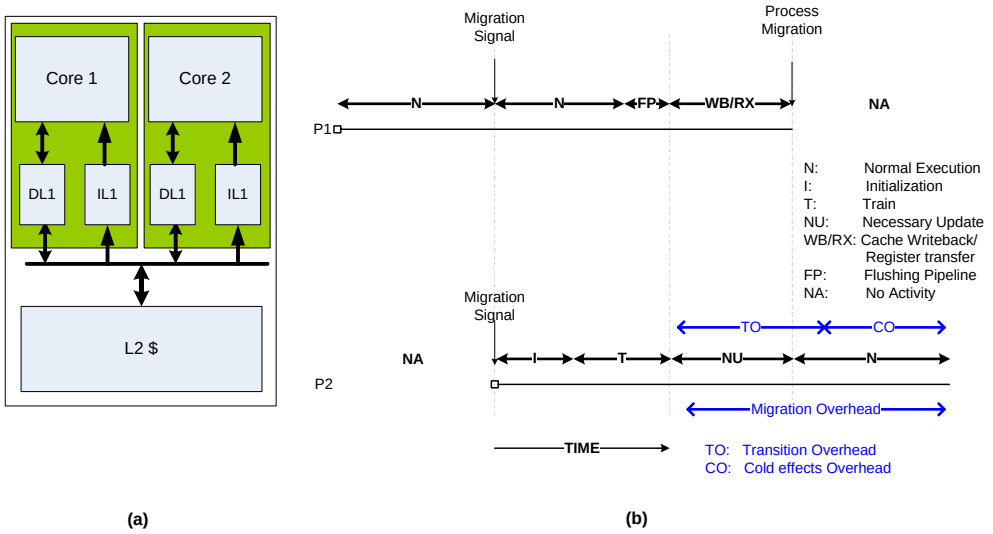

Figure 1. (a) Multi-core with private L1 caches and shared L2, and (b) Different Phases for a Thread Migration from core P1 to core P2

to the update-bus activity. So a balance must be reached between the expected performance benefit and the length of the train phase. At the tail of the train phase the pipeline of core *P1* is flushed. In the experimentation we will investigate whether it is necessary to have a train phase.

At this point the necessary-update phase can commence. This is the phase where state essential for correctness is updated. During this phase the architectural register state is transferred from *P1* to *P2*, and the dirty cache blocks, in a private cache to be turned off in *P1*, are written in a lower level cache. The write-back is necessary so that the loads from the new core get the correct memory data. The length of this phase depends on the number of registers, latency and bandwidth of register transfer, number of dirty blocks and the write-back latency. If the register transfer and the cache flushing occur in parallel, then whichever takes longer to complete determines the latency of a necessary-update phase. Note that the latency to transfer the registers is constant whereas the latency to flush dirty blocks is not as it depends on the number of dirty blocks.

The transfer of registers and the writeback can take place in parallel to minimize the length of the necessary-update phase. But that introduces complexity due to the need for a bus to communicate the registers between cores. An alternative is to perform first the writeback and then the register transfer. In such case the register transfer can be implemented using microcode or a software trap. This routine will store each register in shared memory from where the target core can load them. This can be carried out using existing resources and without the need for a dedicated update bus. We will investigate both serial and parallel necessary-update phase for different values of latency for the register transfer to decide whether the parallel transfer and the update bus are really needed.

When the necessary-update phase completes, core *P1* can become inactive and core *P2* starts to execute normally.

Thread migrations can cause a performance degradation because during migration there are phases without normal execution. This occurs during the flushing of the pipeline and during the necessary update-phase(Transition Overhead). Therefore, it may be important to keep the latency of these events small. Furthermore, performance can suffer even after a thread's migration to a new core due to cold resources. Cold start effects are mainly dominated by cold cache misses and branch mispredictions. It is for this reason that having warmed-up resources may be useful. This can be accomplished using the train phase prior to migration and/or having resources in inactive cores preserving their state between migrations.

The above implies that the migration overhead is both application and microarchitectural dependent. To assess its performance impact we perform an empirical study using simulation that is discussed in Section 5.

3.3 Other Migration Issues

Below we elaborate on migration issues related to predictors, caches, and migration order.

Branch Predictor: High branch predictor accuracies are very important for the performance of a modern processor. With the increase of pipeline stages branch misprediction penalty is also increasing. Thus, maybe important for a branch predictor after migration to have a level of accuracy as if no migration has occurred. If the predictor is turned-off between migrations then the train phase can be useful to train the branch predictor based on the outcomes of branches committed in *P1*. by transferring for each branch instruction executed in P1 its address, its direction and its train phase the better the predictor is trained. But the train phase should be minimized because it incurs, as mentioned previously, energy cost and can limit migration frequency. So we need a branch predictor with both high accuracy and fast training. We believe that a good solution is to employ a hybrid predictor with high accuracy that includes a "simple" component, such as bimodal, that can be trained fast. An alternative to training is to preserve the state of the predictor between migrations into a low

leakage drowsy state [12]. Deciding which of the two approaches or their combination is better depends on program characteristics and, therefore, will be established experimentally. In the experimentation we will consider the following predictor warm-up modes: *cold*, *train* (updated during train-phase), *drowsy* (preserve state from previous activation), the combination of train and drowsy, and *ideal*. The ideal predictor warm-up corresponds to the case where the entire predictor state is transferred between cores without a penalty. The experimentation also investigates which of the following predictor components are more important to warm-up: conditional direction tables, branch-target-buffers (BTB), return-address-stack (ras) and target-cache.

Caches: with activity-migration there are at least two options as far as what to do with the state of private caches when a core becomes inactive. The one option, referred to as *cold*, is to have the private write-back caches flushing their contents to a lower level shared cache and then turn them off. The other option, called *coherent*, is to keep the private write-back caches active, maintain their tags coherent and allow data transfers between private caches in inactive cores to the active core. This effectively corresponds to an invalidation coherence protocol.

When the coherent cache mode is employed and the active core has a miss in its private cache, it will first probe the caches in the other cores for the missing block and only if its not found it will probe for the block lower in the memory hierarchy. For the cold mode, when a core has a miss in its private cache it does not check the caches in the other cores but probes directly for the block from lower in memory hierarchy.

In the experimentation we explore the performance impact of two cache warm-up alternatives which are denoted as cold and ideal. The *ideal* scenario is represented by the case where the entire private cache content is transferred between cores without any penalty.

In the results section we are going to study several combinations of cache and predictor warm-up modes and consider the warming-up of different combination of core resources. We will also quantify the significance of having a train phase by considering its impact with different lengths of training time.

Rotation Policy: Another dimension of the design space to consider is the rotation order between cores. One policy, refer to as *Mod*, is to rotate to core (i+1)%N, where i is the present core and N is the total number of cores on a multi-core. For the remaining paper we assume the *Mod*, however, we note that alternative policies exists

4 Experimental Framework

To establish the importance of the migration model parameters presented in Section 3, a simulation study was performed. The study was based on a modified version of the Simplescalar 3.0 simulator [3] that implements the various multi-core configurations, warm-up and migration modes. The runs used complete runs of SPEC95 integer benchmarks and selected regions of a subset of SPEC00 benchmarks (shown in Table 1).

The experimentation compares the performance of perfect migration with the performance of several migration scenarios

Benchmark	Skip(mil)	Dynamic Instr (mil
compress95 INT	-	443
gcc95 INT	-	177
go95 INT	-	133
ijpeg95 INT	-	553
li95 INT	-	202
M88ksim INT	-	241
perl95 INT	-	40
vortex95 INT	-	101
bzip00 INT	315	100
gcc00 INT	700	100
gzip00 INT	300	100
mcf00 INT	2000	100
parser00 INT	400	100
vortex00 INT	100	100
ammp00 FP	2000	100
art00 FP	50	100
equake00FP	1300	100
mesa00 FP	350	100

Table 1. Benchmarks and Regions Simulated

Fetch/Issue/Commit	4 instructions
Pipeline Stages	10,15,**20**
Direction Predictor	8,16,**32**,64KB Hybrid Bimodal/Gshare
BTB/Target$/RAS	4K/0.5K/16,8K/1K/32,**8K/2K/64**,16K/4K/128
Instr. Window Size	128
LSQ Size	64
L1 I$	1,**2**,3 cycle 8,**16**,32,64KB,64B blocks,2-way
L1 D$	1,**2**,3 cycle **8**,16,32,64KB, 64B blocks,4-way
ALU/Cache Ports	4/2
L2$	**7**,9,12,14,20,28 cycles 2MB,256B blocks,8-way
Memory	200 cycles

Table 2. Microarchitectural Parameters

discussed in Section 3 with varying number of cores. The effects of migration period is examined by considering runs with fixed migration period and with random migration periods. The various microarchitectural parameters are shown in Table 2 (when there is more than one option the default value is shown in bold).

The default policy for the register transfer is to be performed in parallel with the writeback phase. The default latency for transferring the registers during the necessary-update phase is assumed to be 100 cycles, 30 cycles for the initiation of the transfer and a cycle thereafter to transmit each of the 70 architectural registers. Unless indicated otherwise, when reporting the normalized IPC for a given configuration it is computed with respect to the same hardware configuration but with *perfect-migration*. In perfect-migration the migration-penalty is zero and the resources are warmed-up ideally. Finally, the default number of cores is two.

5 Results

In this section we present our simulation results. We report results for a multi-core with private L1 caches and shared L2. Due to space considerations in sections 5.1.1-4 we only show results for some representative benchmarks.

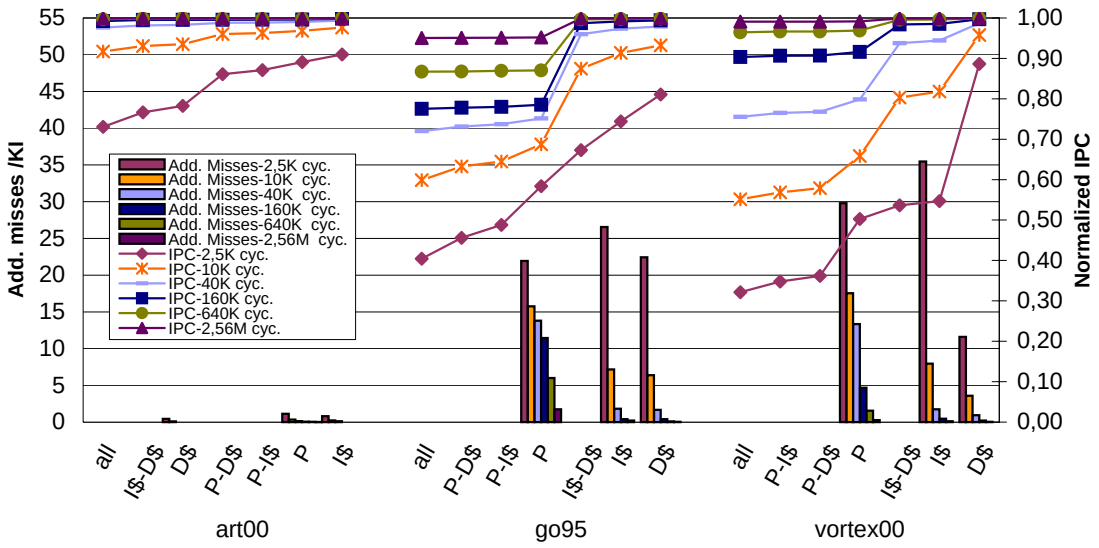

Figure 2. Performance Impact when Ideally Warming-Up a Subset of Resources

5.1 Results with Private L1s and Shared L2

5.1.1 Performance Impact of Ideally Warming-Up a Subset of Resources

In this section we present data that show the importance of warming-up different combinations of resources on a dual multi-core. This is examined for various migration frequencies, at every 2.5K, 10K, 40K, 160K, 640K and 2.5M cycles. For these experiments, after each thread migration a core resource is either cold or ideally warmed-up (there is no train or coherent warm-up). The experiments examined the influence of the i-cache (including i-tlb), the d-cache (including d-tlb), and the predictor (all predictor components: direction, ras, BTB and target-cache).

Fig. 2 shows these results for *art00*, *go95* and *vortex00*. For each benchmark the x-axis represents the seven combination of cold resources we considered (in each case the missing resources are ideally warmed-up), and the y-axis represents IPC normalized to perfect migration. For better data presentation, the resource combinations on the x-axis are listed in reversed order of importance for each benchmark. The graph also shows the additional misses per thousand instructions (misses/KI) as compared to perfect-migration. This is useful to illustrate the main causes of any performance degradation. For clarity, we only show the additional misses when a single resource is cold, the contribution from multiple cold resources is roughly additive. When a branch predictor is cold the additional misses/KI represent the additional branch mispredictions/KI and in the other cases correspond to additional cache misses/KI.

We can observe that for high migration frequencies, every 2.5K and 10K cycles, there is a large performance loss when migrating to a core with any of its resources cold (points *all* for 2.5K and 10K migration period). For *go95* and *vortex95* this appears to be caused by a large number of additional misses in all resources. However, *art00* has very few additional misses and we still observe a significant degradation. This implies that part of the observed loss, at 2.5K and 10K migration periods, is due to the migration penalty. Therefore, with high migration frequency all resources seem to be important to be warmed-up and the migration penalty can not be ignored.

On the other hand, when migration occurs every 2.5M cycles, performance loss is usually very small even when all resources are cold (points *all* for 2.5M migration period). The additional misses, with migration period of 2.5M cycles, are very small and the migration penalty must be negligible as compared to the migration period. The exception is *go95* that incurs a small performance loss when the predictor is cold (points with *P* and 2.5M).

The behavior for medium migration frequency, between every 40K–640K cycles, appear to have more unpredictable behavior than those just discussed and also seems to be application and microarchitectural dependent. Below we focus the analysis on the behavior with migration frequency every 40K–640K cycles.

The behavior of *art00* indicates that for some benchmarks may not be important to warm-up any resource. With all resources cold the loss in performance is small. The largest reduction in IPC is around 3% when all resources are cold and the interval size is 40K. The number of additional misses/KI when resources are cold is insignificant and that is why the performance loss is small.

However, the above is not valid for all benchmarks. As we can see for *go95* with all resources cold there is a reduction in IPC around 30%, 25% and 15% for migration at every 40K, 160K and 640K cycles respectively. Similar behavior is observed for *vortex00*. A notable observation from the results for *go95* and *vortex00* is that the branch predictor is by far the most important resource to have warm on a core, and that there is no benchmark for which it is important to have warm either the instruction and/or data cache. An implication of this observation is that when migrating every 40K cycles or more the migration overhead is mainly dominated by cold predictor effects and not from cold caches or the penalty to flush the pipeline and perform the necessary update phase.

One reason for the tolerance of thread migration to cold L1 caches is that the working set they can fit is very small (256 unique blocks for the instruction and 128 blocks for the data cache used in this study). As a result the penalty to warm them up

and to flush dirty blocks is very small as compared to the period of migration we consider here (40K cycles or more). Analysis of the fraction of blocks that need to flushed for 40K intervals has shown that typically 40% (50 blocks) of the blocks are dirty. On the other hand the direction prediction table for conditional branches can hold predictions for many unique combinations or branch histories (the predictor used in this study can hold 98204 unique combinations). Consequently, a benchmark with many unique branch history patterns, like *go95* and *vortex00*, will incur significant performance penalty when the predictor is cold. For *go95* and *vortex00* there are 14 and 5 additional misses/KI respectively when migrating at every 40K cycles.

An important difference between *go95* and *vortex00* with cold predictor, point *P* in the graph, is that the performance(mispredictions) of *vortex00* is improving(decreasing) faster as the migration frequency decreases. This may indicate that the working set of the predictor for *vortex00* is smaller and with longer migration periods the negative effects of having a cold predictor are diminishing.

Overall, we can observe that to avoid performance degradations with high-migration frequencies, every 10K cycles or less, all resources need to be warmed-up and migration penalty must be small, whereas when migration is infrequent, every 2.5M cycles or more, resources can be left cold.

For medium migration frequencies, every 40K-640K cycles, the migration overhead is highly dependent on the benchmark and when there is a significant overhead it is dominated by cold predictor effects. I.e. we have found that there is little to gain by keeping L1 caches warm, however, depending on the benchmark it may be very important to have the branch predictor warm.

For the remaining of Section 5.1 we focus on migration with medium frequency because that is where benchmarks exhibited the most unpredictable behavior.

5.1.2 Performance Impact of Ideally Warming-Up a Subset of Resources for Different Table Sizes

In the previous section, we have established that to avoid performance degradation for medium migration frequencies it is important to have the branch predictor warmed-up but not the caches. However, this was obtained for one particular configuration. In this section we investigate, for a dual multi-core, whether by changing the size of the L1 caches and the predictor, we reach a different conclusion.

Fig. 3 shows the normalized IPC for I-cache sizes of 8KB, 16KB, 32KB and 64KB for three different migration frequencies, when all resources after migration are cold and when all resources are ideally warmed-up except the i-cache which is cold. For clarity we have omitted the sizes in the graph, for each migration period there are four bars each corresponding to one of the sizes in increasing order from left to right.

These results are shown for the benchmarks *go95*, *vortex00* and *art00*. Analogous study was performed for the d-cache and the predictor, and these results are also shown in Fig. 3. For the branch predictor the size are for the direction prediction tables, however, the size of the other predictor components is scaled by the same factor the direction table size is changed as compared to its default size.

Fig. 3 indicates that for the three benchmarks and across different i-cache sizes it is not very important to have a warm i-cache. The only exception is *vortex00* when migrating every 40K cycles, where the performance loss is considerable, about 12%, for a 64KB I-cache size. For *art00* the performance loss is small even when all resources are cold.

The data clearly indicate that a warm d-cache is not important for any cache size, migration frequency, and benchmark. This may appear unintuitive especially for large caches, but recall that the underlying core microarchitecture provides some tolerance to cold-caches through out-of-order scheduling and long misprediction penalties.

The results suggest that for *go95* and *vortex00* a cold branch predictor is the major cause of the performance loss with thread migration. For *go95* the performance loss increases with increasing predictor size, whereas for *vortex00* the performance is not sensitive to the branch predictor size. The results for these two benchmarks support the observation made in the previous section about little benefit when migrating with warm i-cache and d-cache. This is evident by the small difference between the cold-all and rest-ideal results when the predictor is cold. The behavior for *art00* is almost insensitive to any combination of cold resources.

Overall, the data in Fig. 3 show that even with different table sizes still the most important resource to have a warm is the branch predictor and that cold instruction and data caches have insignificant effect on performance. Consequently, for the remaining of Section 5.1, unless indicated otherwise, the d-cache and i-cache start cold after each migration and d-caches write-back their dirty blocks before migrating out of a core.

We like to note that we have also performed sensitivity analysis for numerous combinations for L1 caches latencies (1-3 cycles), L2 cache latencies (5-28 cycles), pipeline depths (10,15 and 20 stages) and register transfer latencies(150, 220 and 420 cycles). The main conclusion was that in most cases the branch predictor is by far the most important resource to warm-up. The exceptions occur when the L2 latencies are more than 20 cycles. In this case the two L1 caches and especially the L1 data cache may be also important to have warm.

The data also suggest that overall migration overhead is mainly determined by cold effects rather than transition overhead. Transition overhead has significant contribution on migration latency only for small migration periods(less than 100K cycles).

5.1.3 Which Branch Predictor Components are Important to Warm-Up

In this section we present data that attempt to quantify for a dual multi-core the importance of warming different branch predictor components (direction tables, ras, BTB and target cache). In Fig. 4 we show the performance for *gcc95* and *vortex00* for various interval sizes, when different combinations of branch predictor resources are cold – for each case the resources missing are ideally warmed-up. For this experiment we assumed that the caches are ideally-warmed up to isolate the effects of branch

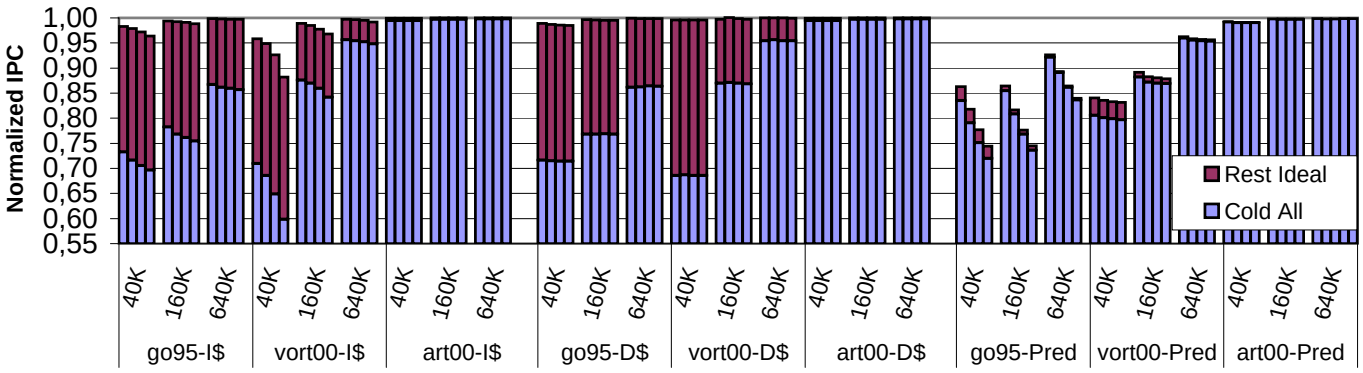

Figure 3. Performance Impact with different i-cache,d-cache and branch-predictor sizes

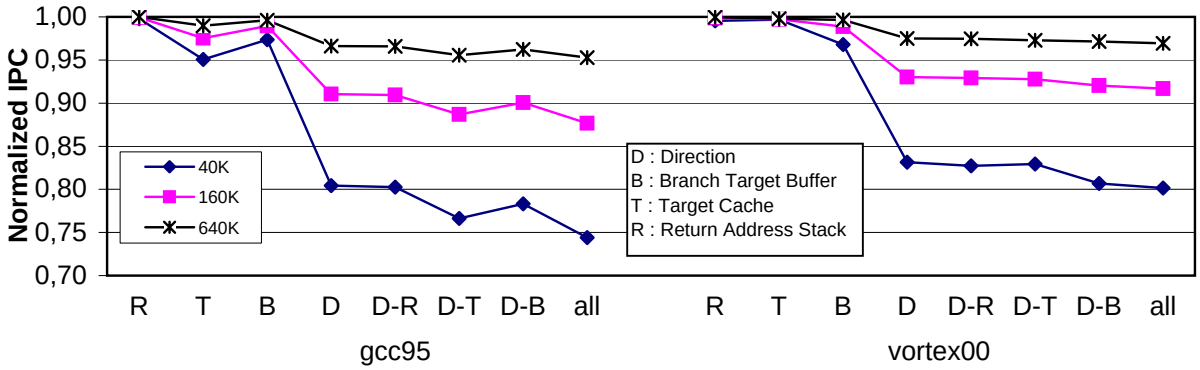

Figure 4. Performance Impact of different predictor components

mispredictions.

The data across all interval sizes suggest that the most important resource to warm-up is the direction prediction table and that the return-address-stack is not useful to warm-up. A cold branch target buffer, when migration every 40K cycles, causes a performance loss around 5% for *vortex00* and around 3% for *gcc95*. This indicates that a branch target buffer may be also important to warm-up. Finally, the target cache also seems to be useful for *gcc95*, but not for *vortex00*. When it is cold causes *gcc95* to loose 5% of its performance when migrating every 40K cycles.

Overall, the data suggest that the importance for a warm branch predictor, observed in previous sections, is primarily due to the direction tables and secondary due to the target cache and/or branch target buffer. A cold return address stack has insignificant impact on performance. In the remaining paper we consider warming up all branch predictor components except the return-address-stack.

5.1.4 The importance of Train and Drowsy Warm-up modes for the Branch predictor

This section investigates the performance implications of different branch predictor warm-up modes. The objective is to determine how close to ideal are practical warm-up modes. In particular, we consider how the *train*, *drowsy* and *train* combined with *drowsy* warm-up modes compare to perfect migration. When the warm-up mode includes train, we consider three different train lengths that occur for 12%, 25% and 50% of the previous interval. Fig. 5 compares the normalized performance of the various predictor warm-up modes for *gcc95*, *go95*, *vortex00* and *art00*.

The results show that the train warm-up mode provides a modest performance improvement over cold mode and that with increasing train phase length there is an increasing improvement in performance (particularly for *gcc95* and *vortex00*). Nevertheless, across all migration periods, for three out of the four benchmarks, the train mode incurs large performance loss as compared to perfect migration.

On the other hand, the results show that the drowsy warm-up mode can recover most of the performance loss due to cold resources. The largest loss of 6% is for *go95* when migrating every 40K cycles. The performance loss gets close to 3% or less when migrating every 160K cycles and to 1.5% or less when migrating every 640K cycles.

The low performance with the train warm-up mode suggests that the working set of a branch predictor can not be learned by considering only the branch outcomes from the previous interval. The drowsy mode performs significantly better than train because a predictor can accumulate and remember knowledge from all previous intervals its core was activated.

The combination of drowsy and train warm-up modes, offers a minimal performance increase over only drowsy. Considering, the potential energy overhead of the train phase, the possible limit it places on migration frequency and its minimal benefit, it is reasonable to conclude that is not an attractive warm-up mode. This means that the migration policy does not need to include a train phase and the update bus that was needed for it.

Fig. 6 compares the normalized performance of cold and drowsy branch predictor warm-up modes for various interval sizes for all benchmarks. The drowsy branch predictor reduces

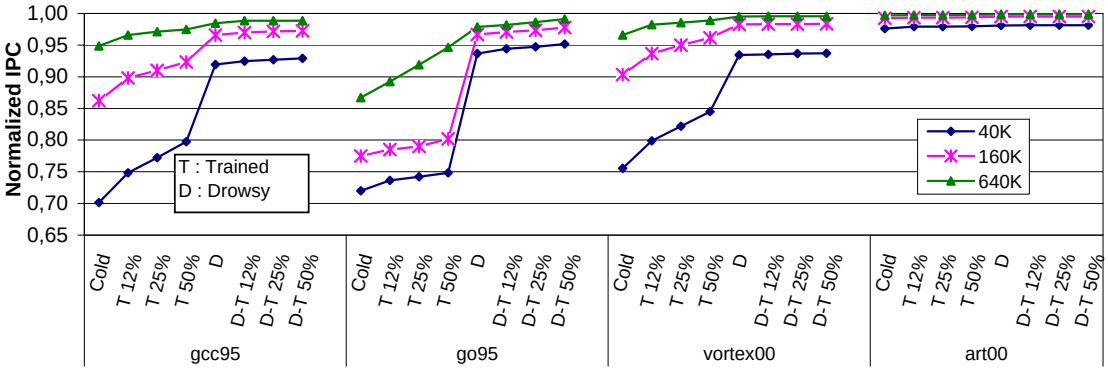

Figure 5. Performance Impact of different branch predictor warm-up modes

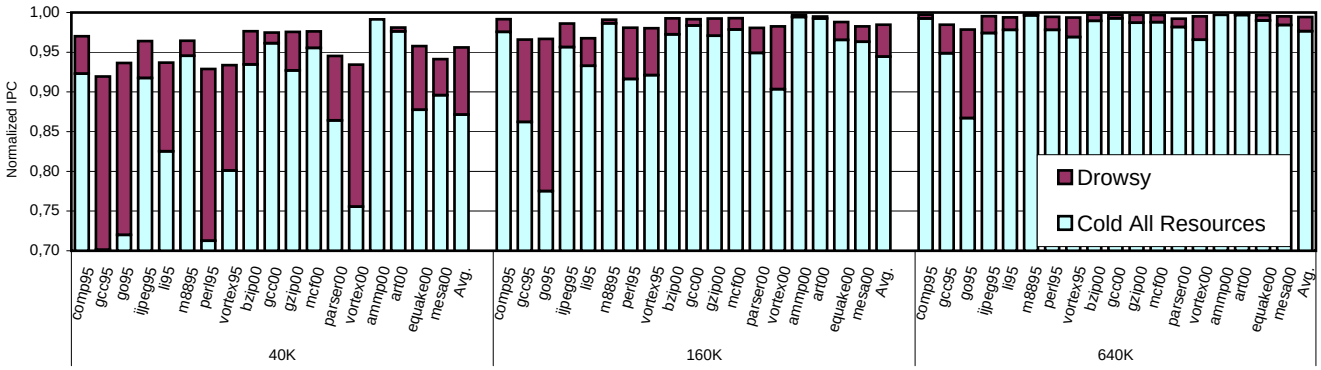

Figure 6. Performance Impact of drowsy branch predictor warm-up mode for all benchmarks

the average performance loss from around 13% to 4% when migrating every 40K cycles, from around 6% to 1.5% when migration every 160K cycles, and from around 2.5% to 0.5% when migrating every 640K cycles. We remind the reader that for the drowsy results all other resources, except the predictor, start cold after migration.

The experimental results also show that average migration period is a strong indicator of how well activity-migration is performing. .

To summarize, the observations in Sections 5.1.1-5.1.4 suggest that keeping the branch predictor drowsy between activations of the same core while the other resources are cold appears to recover most of the performance loss due to thread migration on a dual multi-core. The data also show that there is no need for a train phase or an update bus. We adopt this migration policy for the remaining paper unless indicated otherwise.

5.1.5 The Impact on Thread Migration with increasing number of Cores

This section explores the effect of thread migration on performance with increasing number of cores. Fig. 7 presents results for 2, 4 and 8 cores. Data are shown with all resource cold and with the predictor in drowsy mode. In cold mode a benchmark has the same performance irrespective of the number of cores thus Fig. 7 shows it only once for each benchmark. Recall that the rotation policy used for these experiments is *Mod*.

The data suggest that for most benchmarks the performance with drowsy predictor remains unchanged as the number of cores increase irrespective of the migration frequency.

However, for some benchmarks(*gcc95*, *go95*, *parser00*) performance gets worse as the number of cores increases. This indicates that with drowsy mode and the *Mod* migration policy, the branch predictor state is getting stale as the number of cores increases.

The data can also be interpreted in a different way by assuming that with increasing number of cores the migration interval increases. For example, if with two cores we migrate every 40K cycles then with four cores we migrate every 160K cycles and with eight every 640K cycles The data show that most of the benefits of migrating over more cores are captured with four cores in most benchmarks.

Overall, with increasing numbers of cores a drowsy predictor combined with the *Mod* migration policy provides an effective migration strategy with good performance.

5.1.6 Random Migration frequencies

The results presented so far were for constant migration frequency and, therefore, someone may claim have limited scope. In an attempt to extend the generality of our observations we conducted several simulations where the migration frequency is determined using random distributions.

The simulations were done by randomly selecting during a run, the next interval size from the following migration periods: 10K, 40K, 80K, 120K, .., 640K cycles. Different random distributions were considered: a) same probability for all interval sizes, b) increased-probability for the shortest-one migration period size (10K), shortest-two (10K and 40K), shortest-three (10K-80K) and shortest-four (10K-120K),and c) increase probability for largest-one interval size (640K), largest-two (600K and 640K), largest-three (560K-640K), and largest-four (520K-

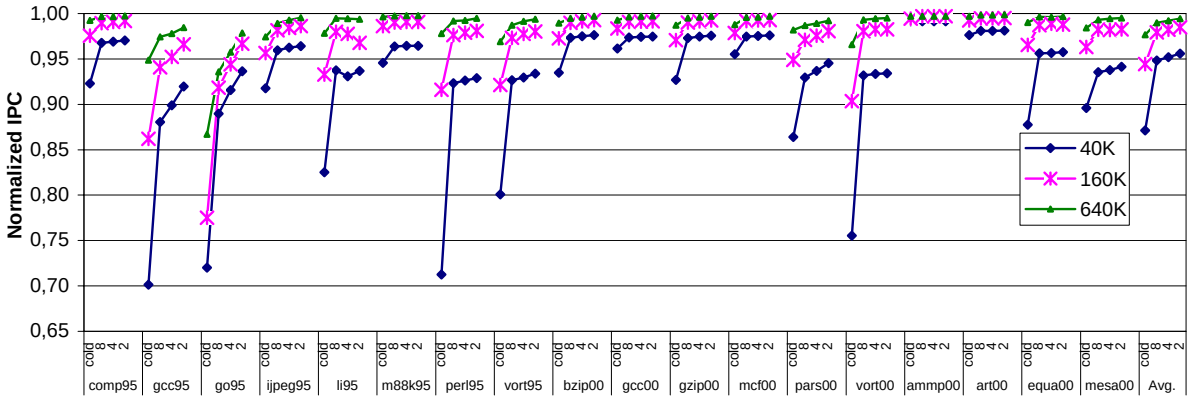

Figure 7. Performance Impact of increasing number of cores

640K). Three different increased-probabilities were considered for 25%, 50% and 75%. Each random simulation was repeated five times and here we report the results using averages of these five runs. This study was performed for a dual multi-core for all benchmarks.

The results show that the drowsy predictor migration policy can provide good performance irrespective of the distribution of the migration-period. For all benchmarks and for all distributions considered, the performance was within 3.2% of perfect migration. The largest performance loss is observed with the *shortest* distributions which they have high migration frequency.

For each of the above random runs we determined its effective migration frequency, by averaging for each run its migration periods, and performed another run with a constant migration frequency equal to the effective migration frequency of the random run. The data reveal for all cases that there is very little variation - at most 1.3% - between the runs. This may suggest that a run with a constant migration frequency may be indicative of many runs that have different distribution of migration periods but the same effective migration frequency. *This also indicates that the average migration period is a good indicator of the performance loss to be incurred by activity migration.*

The data for the random experiments are not shown because all points have similar value which is close to 1.

5.1.7 Is there a need for an update bus?

In Section 3, we have argued that an update bus may be useful for two reasons: (a) training the predictor and caches prior to a migration to a new core, and (b) for transferring the register state in parallel with the writeback of dirty blocks.

The results so far had shown that there is no need for an update bus for training predictors and caches. Consequently, to answer the question whether an update bus is needed for activity-migration, we performed experiments comparing the performance of serial and parallel register transfer with increasing register transfer latency. Our data - not shown due to space limitations - suggest that for migration periods between 40K and 100K cycles an update bus may be useful but for larger migration periods the serialization overhead has very little impact and thus there is no need for it.

5.2 Results for a multi-core with private L1s and L2s and a shared L3

We have also investigated the performance impact of thread migration on a multi-core with private L1 and L2 caches and a shared L3. Due to limited space we briefly summarize the main findings without presenting data. The interested reader can check a more extended version of this report [6].

The results revealed that for migration with cold resources the performance penalty is very high and that both the predictor and the L2 cache are crucial to be warm. The data also shown that activity-migration that combines drowsy predictor with coherent L2 cache can recover most of the performance loss due to cold resources. L2 coherence implemented with an invalidation protocol works well(i.e. maintain the tags of the L2 caches coherent and allows L2-L2 data transfers from inactive cores to the active core).

6 How to interpret the results of this paper?

The findings of this work may be useful for several applications. Below we offer some discussion on few such applications.

Improving Power-Efficiency on a Heterogeneous Multi-core: Previous work [17], on activity migration for power efficiency, considered migration at a very coarse scale (during context switch) and thus may have missed opportunities to improve power-efficiency between context switches. The data in Section 5 suggest that fine grain activity migration with low performance overhead is feasible. Therefore, future work should consider the potential of a strategy as outlined by these findings.

Reducing the Temperature on a Homogeneous Multicore: If we assume that as long as there is no temperature problem a thread is keep executing on the same core at peak frequency. However, when a temperature reaches a trigger-threshold we need to engage a dynamic thermal management (DTM) technique to reduce temperature. If the DTM technique fails to decrease temperature and the temperature emergency-threshold is reached then the processor is put in a *halt* state until the temperature drops below the trigger-threshold.

If we assume that the only DTM technique employed, in addition to halt, is activity-migration. Then the results from

the previous section suggest that for the time the processor is not halted, the overhead of migration is usually small and thus activity-migration is a performance efficient DTM technique to consider to address the temperature problem.

The results also show that average migration periods can be useful to decide whether or not to switch from activity-migration to a different DTM technique. Specifically, the data suggest that for average performance loss of less than 1% migration should occur every 160K cycles or more.

7 Conclusions

This paper investigates the performance implications of single thread migration on a multi-core.

The main conclusions of our work, is that the performance losses due to activity migration for a multi-core with private L1 and shared L2 caches can be minimized by remembering the predictor state between migrations. The experimental results also show that average migration period is a strong indicator of how well activity-migration is performing.

The results suggest that an effective migration strategy can possibly be implemented without using a dedicated update bus, for transferring state between cores, and without a train phase, for warming up caches and predictors prior to a migration on a new core.

This work has assumed several idealizations that need to be addressed in future work. For example, uniform cache latencies, and infinite write buffers. More importantly, the migrations were not triggered by actual events but rather occurred at regular or random intervals. Future work, should combine thread-migration with power and temperature modeling and evaluate the overall effectiveness of the strategy proposed in this paper. Another direction of work is to establish the performance impact of multiple thread migration on a multi-core.

References

[1] AMD. Multi-core processors the next evolution in computing. In *AMD Multi-Core Technology Whitepaper*, 2005.

[2] S. Borkar. Design Challenges of technology scaling. *IEEE Micro*, 19(4):23–29, Jul. 1999.

[3] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996.

[4] P. Chaparro, J. Gonzalez, and A. Gonzalez. Thermal-effective clustered microarchitectures. In *First Workshop on Temperature-Aware Computer Systems (TACS-1)*, 2004.

[5] Clabes et al. Design and implementation of the power5 microprocessor. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 670–672, June 2004.

[6] F. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec. Initial Results on the Performance Implications of Single Thread Migration on a Chip Multi-Core. Technical Report CS-TR-05-03, University of Cyprus, January 2005.

[7] G. et al. The Intel Pentium M Processor: Microarchitecture and Performance. *Intel Technology Journal*, 7(Q2), May 2003.

[8] Flachs et al. The microarchitecture of the streaming processor for a cell processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, February 2005.

[9] M. Fleischmann. Crusoe longrun power management. In *Transmeta Corporation Whitepaper*, 2001.

[10] D. J. Frank. Power-constrained CMOS scaling limits. *IBM Journal of Research and Development*, 46(2/3):235–244, 2002.

[11] S. Gunther, F. Binns, D. Carmean, and J. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, 5(Q1), Feb 2001.

[12] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. In *ISLPED '03: Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 217–222, 2003.

[13] Intel. Intel multi-core processor architecture development backgrounder. In *Intel Whitepaper*, 2005.

[14] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, Mar./Apr. 2004.

[15] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, Mar./Apr. 2005.

[16] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures. *Computer Architecture Letters*, 2, April 2003.

[17] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *36th International Symposium on Microarchitecture*, pages 81–92, December 2003.

[18] C. H. Lim, W. R. Daasch, and G. Cai. A thermal-aware superscalar microprocessor. In *Proceedings of the 2002 international Symposium on Quality Electronics*, pages 517–522, 2002.

[19] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2):10–20, 2005.

[20] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11, 1996.

[21] C. Poirier, R. McGowen, C. Bostak, and S. Naffziger. Power and temperature control on a 90nm itanium-family processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, February 2005.

[22] F. Pollack. New microarchitecture challenges in the coming generations of cmos process technologies. In *Micro32 conference key note*, 1999.

[23] M. D. Powell, M. Gomaa, and T. N. Vijaykumar. Heat-and-run: leveraging smt and cmp to manage power density through the operating system. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 260–270, 2004.

[24] Sanchez et al. Thermal management system for high performance powerpc microprocessors. In *Proceedings of COMPCON 97*, pages 325–330, 1997.

[25] Skadron et al. Temperature-aware microarchitecture: Modeling and implementation. *ACM Transactions on Architecture and Code Optimization*, 1(1):94–125, Mar. 2004.

[26] Y. Taur. CMOS design near to the Limit of Scaling. *IBM Journal of Research and Development*, 46(2/3):213–222, Mar./May 2002.

[27] Tendler et al. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, Jan. 2002.