Exploiting the Cache Capacity of a Single-Chip Multi-Core Processor with Execution Migration

Pierre Michaud
IRISA/INRIA
Campus de Beaulieu 35042 Rennes Cedex, France
pmichaud@irisa.fr

Abstract

We propose to modify a conventional single-chip multi-core so that a sequential program can migrate from one core to another automatically during execution. The goal of execution migration is to take advantage of the overall on-chip cache capacity. We introduce the affinity algorithm, a method for distributing cache lines automatically on several caches. We show that on working-sets exhibiting a property called "splittability", it is possible to trade cache misses for migrations. Our experimental results indicate that the proposed method has a potential for improving the performance of certain sequential programs, without degrading significantly the performance of others.

1. Introduction

In the recent years, the number of cache levels integrated on high-performance processor chips has increased from 1 level to 2 levels, now 3 levels of on-chip caches. By taking advantage of the density of integration and bringing a lot of cache capacity in the close vicinity of the execution core, it is possible to decrease the average memory access latency. However, for a given technology, there is a limit inherent to the two-dimensional chip layout. The cache levels closer to the core have a small capacity and a small access latency, whereas peripheral levels have a larger capacity but also a longer access latency. One could call this the "cache-wall" problem, by analogy with the so-called "memory-wall". We propose *execution migration* as a possible answer. Our approach is based on the assumption that future processors will be multi-cores [10, 12]. Some commercial multi-cores are already available, e.g., the IBM Power4 [19], featuring 2 execution cores on the same chip. As it becomes difficult to exploit higher degrees of instruction-level parallelism, high-performance processors now try to exploit task-level parallelism. Putting several execution cores on the same chip is a possible way to achieve this. However, it is important for a general purpose multi-core to offer good performance when a single sequential task is presented to it. From the point of view of a sequential program running on one core, the cache capacity available on other cores is wasted. We propose to modify a conventional multi-core to allow a sequential application to migrate from one core to another during execution, so that the application is able to take advantage of the overall cache capacity.

The remainder of this paper is organized as follows. Section 2 describes the machine model assumed in this study. Section 3 introduces the *affinity algorithm*, a method for deciding when and where to migrate. Section 4 presents simulation results on the SPEC2000 and Olden benchmarks. Finally, Section 5 relates our solution to prior work, and Section 6 concludes this study.

2. Machine model

We define *execution migration* as the possibility, on a multi-core processor, to migrate the execution of a sequential program quickly from one core to another. We assume that execution migration is possible only when the processor is in *migration mode*, i.e., only a single task is running on the machine. The migration mode could also be decided by the operating system based on process priority. However, this is a separate problem, which we do not address in this study. We assume that all the cores are available for one application. The multi-core configuration we consider in this study is depicted on Figure 1. Each core has level-1 (L1) data and instructions caches and a unified level-2 (L2) cache. The level-3 (L3) cache is shared by all the cores. We aim to distribute the working-set of an application onto the L2 caches so as to benefit from the overall L2 capacity, while keeping the latency of a local L2 hit. At a given time, only a single core, the *active* core, is executing instructions. It might be possible to save power on inactive cores by powering down parts that are not used (e.g., the instruction pipeline). However this study considers performance only, not power consumption. We assume that all inactive cores are powered on, in particular the retirement unit, architectural registers, TLBs, caches, branch prediction tables, and, more generally, all the tables holding architectural or microarchitectural program state.

2.1. Caches

We assume that the L1 data cache (DL1) is write-through (as in the IBM Power4 [19]), non write-allocate. We do not maintain the inclusion of L1 in L2. To facilitate snooping, one could maintain the inclusion of the L1 caches into all the L2 caches considered as a whole. However in this study, we assumed that the L1 tags were simply replicated, as in [4]. We assume a write-back, write-allocate L2. Because we

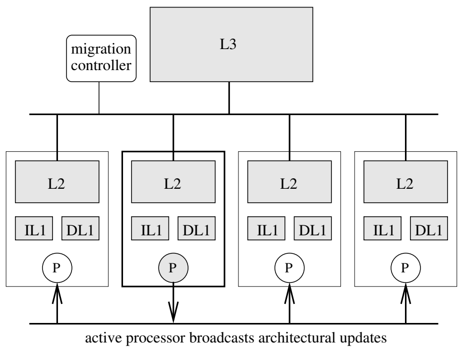

Figure 1: Example of 4-core processor with migration mode enabled.

do not force the inclusion of L1 into L2 when in migration mode, write allocation in L2 may be triggered even upon DL1 hits. Multiple copies of a line may be replicated on several cores. However, when in migration mode, cache coherence is maintained differently from normal mode. Upon writing a L2 line on the active core, the *modified* bit is set, as usual, but copies of that line on inactive cores ("inactive lines") are not invalidated. Instead, the *modified* bit on inactive L2 lines is reset, and inactive L2 lines remain valid. Consequently, at most a single copy of the line can be marked *modified* at any time. The content of inactive lines is maintained coherent via a dedicated *update bus* (see Section 2.3). There is a delay for updating the copies, and the most up-to-date copy is the one marked *modified*. When a line is evicted from L2, it is written back to L3 only if its *modified* bit is set. A line marked *modified* may be forwarded to another core upon request (L2-to-L2 miss). In that case, the line is simultaneously written back into L3, and the *modified* bit is reset. A L2 line which *modified* bit is not set can be used only by the local core. It cannot be forwarded to another core, and must be re-fetched from L3.

Throughout this study, we do not distinguish between L2-to-L2 misses and L3 hits. In other words, we assume that the L2-to-L2 miss penalty is roughly equivalent to a L2-miss/L3-hit penalty. Though this may be wrong for a particular technology or implementation, [1] we believe this is a fair assumption.

2.2. Performing a migration

The decision to migrate the execution is taken by the *migration controller*. The migration controller monitors all the L1-miss requests issued from the active processor, and it bases its decisions on current and past requests. When the controller decides to migrate the execution from core X1 to core X2, it sends an interrupt request to the I-fetch unit of X1. Upon receiving an interrupt request, the I-fetch unit stops fetching instructions and sets a transition point. Setting a transition point consists in choosing an instruction (preferably the one that was fetched the latest), marking this instruction as the transition instruction T, and recording the transition PC, i.e., the PC of the instruction following T. The I-fetch unit returns the transition PC to the migration controller. The migration controller forwards the transition PC to the I-fetch unit of X2, and X2 starts fetching instructions. While pipeline X1 is draining, instructions following T progress in pipeline X2. However, the issue stage of X2 remains blocked so that instructions in X2 cannot execute before T retires. This is for ensuring that, when instructions execute on X2, the architectural state is up-to-date. Once T is retired, X2 becomes the new active core.

An event requiring to redirect the control flow, like a branch mispredict, may occur while X1 is draining (i.e., after setting the transition point). In this case, instructions in X1 after the branch mispredict are flushed, as usual, but the I-fetch unit of X1 remains blocked. The mispredicted branch becomes the new transition point. Pipeline X2 is flushed, and the new PC is sent to the I-fetch unit of X2.

2.3. Program state maintenance

In order to minimize the migration penalty, the architectural and microarchitectural program states are updated continuously. Each instruction retiring from the active core is broadcast to inactive cores :

- If the instruction writes in a register, we broadcast the logical register number and the value : each inactive core writes the value in its local copy of the architectural register file.
- If the instruction is a store, we broadcast the store address and value : each inactive core writes the value in its local L1 and L2 caches **if the cache line is present**.
- If the instruction explicitly modifies the TLB content we broadcast the TLB-modifying instruction so that the update is replicated on all inactive TLBs. For architectures with hardware-managed TLBs, the TLB engine must be able to update inactive TLBs.
- If the instruction is marked as a transition instruction, it also unlocks the issue stage.

We assume that there is a dedicated *update bus* for broadcasting retired instructions. At a given time, only the active core is writing on the bus. The retirement unit of each inactive core monitors the bus and "retires" the instructions (almost) as if they had been executed on the inactive core. The update bus is conceptually depicted on Figure 1. An actual implementation, for example, could be a pipelined ring.

It should be noted that we have implicitly assumed that there is an architectural register file that is written at retirement. This is the case for in-order issue processors (e.g., Sun Ultrasparc, Intel Itanium), or for certain out-of-order issue microarchitectures (e.g., HP PA-8x00, Intel P6). Some processors, on the other hand, merge the architectural and rename registers in a single physical register file (e.g., MIPS R10000, Alpha 21264) [17]. These processors should be modified so that the active retirement unit is able to broadcast register values. This can be done either by having extra read ports on the physical register file, or by storing values redundantly in a duplicate physical register file which read ports are accessed at retirement.

---

1 Accessing a remote cache may take longer than accessing a shared cache level [4]. On the other hand, if there is a direct access to remote caches, a cache-to-cache miss might be faster.

L1 caches. When a line is brought into the active L1 upon a L1 miss, the line is broadcast to all the inactive L1 caches through the shared L2-L3 bus. This way, when execution migrates to another core, the L1 miss frequency is the same as if execution had not migrated. Because DL1 caches are write-through, there is no write-back traffic from inactive L1 caches. [2]

Branch predictor. In order to train inactive branch predictors, branch instructions are broadcast on the update bus at retirement. The branch instruction at retirement consists mainly of the branch address and branch outcome (direction and target). It may require to implement a specific path from the retirement unit to the branch predictor, so that inactive cores can update their branch predictor (some processors update the branch predictor tables in the execute stage)

Update bus bandwidth requirement. The bandwidth requirement on the update bus is proportional to the retirement bandwidth. For example, let us assume that the active core can retire 4 instructions per cycle, but only a single store and a single branch. We should broadcast 4 register identifiers (e.g., 6-bit register identifiers), four 64-bit values, one 64-bit store address, one branch address, plus a few bits for identifying the instruction type. The branch address is used only for updating the branch predictor, so it does not need to be a full 64 bit address. Let us assume that we send only the 16 low-order bits of the branch address. With these assumptions, the bandwidth requirement is approximately 45 bytes per cycle.

2.4. Migration penalty

The basic idea explored in this paper consists in trading migrations for L2 hits. The trade-off depends on the relative penalties of migrations and L3 accesses. The penalty of a migration from X1 to X2 corresponds approximately to the number of cycles between the retirement of the transition instruction T on X1 and the retirement of the instruction following T on X2. We assume that T is the instruction that was fetched the latest on X1 when X1 receives the interrupt signal from the migration controller. For simplicity, we also assume that the delay for transmitting the transition PC from X1 to X2 equals the delay for broadcasting the transition instruction on the update bus. With these assumptions, the migration penalty corresponds to the number of cycles for broadcasting T on the update bus plus the number of pipeline stages from the issue stage to retirement.

The relative penalty P_{mig} of a migration compared with a L2-miss/L3-hit depends on many parameters : technology, microarchitecture, application, ... We make no assumption on the value of P_{mig} in this study, but $P_{mig} > 1$.

3. The migration controller

What constitutes the core of the problem can be stated as follows : how to distribute the working-set of an application automatically on several caches so as to take advantage of the overall cache capacity, while having as few migrations as possible ?

3.1. Partitioning a working-set

The problem we are trying to solve may be viewed as a graph partitioning problem [13]. Let us consider a graph which nodes are the static cache lines constituting the program working-set. An edge from line A to line B means that line B may be referenced just after line A, the edge being weighted with its frequency of occurrence. The 2-way partitioning problem (a.k.a. graph bisection or graph bipartitioning) consists in splitting the set of nodes in two subsets of the same size, while minimizing the cut size, which, in our problem, is the frequency of transitions from one subset to the other. This problem is NP-hard [9]. Several heuristics are known [1], but they are not suitable for an implementation in hardware. Our problem is specific. We want a hardware mechanism that learns the program behavior automatically. We are not relying on information from the programmer or the compiler. So we are looking for an online algorithm, simple enough to be implemented in hardware. However, unlike classical graph partitioning problems, we do not need an optimal partitioning, but one that is good enough that the frequency of transitions from one subset to the other is small, say less than 1 transition every 100 L1 misses. If the working-set behavior does not make it possible (e.g., because lines are referenced in random order), we do not care about finding an optimal partitioning.

3.2. The affinity algorithm

The algorithm we propose does not require maintaining a graph representation of the working-set. It is based on a quantity which we call *affinity*. At time t, the affinity $A_e(t)$ is a signed integer value associated with each element e in the working-set (i.e., each cache line). Our goal is to split a working-set in two subsets of equal size : elements with a positive affinity define one subset, elements with a negative affinity define the other subset.

Principle. Let S be the working-set. Let $R \in S$ be the n elements of S that have been referenced the most recently, $n = |R|$ being a fixed parameter. We define the affinity A_R of R as

$$A_R(t) = \sum_{e \in R} A_e(t)$$

Initially, $A_e(t_e) = 0$ at time t_e when an element e is first referenced. Time t is incremented on each reference. We update the affinity as follows. For all $e \in S$,

$$A_e(t+1) = \begin{cases} A_e(t) + \mathrm{sign}(A_R(t)) & \text{if } e \in R \\ A_e(t) - \mathrm{sign}(A_R(t)) & \text{if } e \notin R \end{cases} \quad (1)$$

with the $sign$ function being defined as

$$\mathrm{sign}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Expression 1 defines what we call the *affinity algorithm*.

2 Although we maintained a strict L1 mirroring for our simulations, we believe there are possibilities to release this constraint, which purpose is only to minimize the migration penalty.

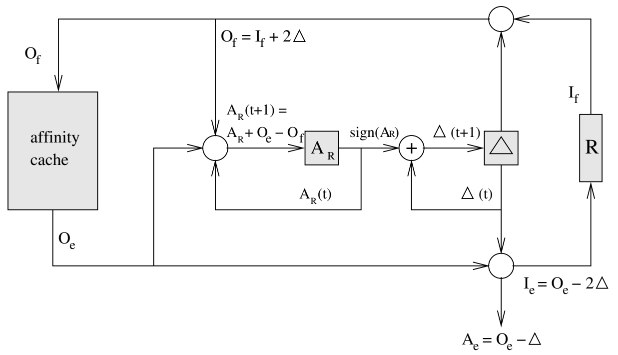

Figure 2: Practical implementation of the affinity algorithm for 2-way working-set splitting. At time t, e is referenced and f is evicted from the R-window.

Negative and positive feedback effects. One can have an intuitive understanding of how the algorithm works by considering a working-set which elements are referenced with equal frequency, and which size is much larger than $|R|$. Let us assume that the distribution of affinity values is not balanced. For instance, a majority of elements have a positive affinity. It means that A_R is positive most of the time. From Definition 1, and because each element spends a majority of time out of R, the affinity of each element decreases, though not monotonically. Symmetrically, if a majority of elements had a negative affinity, the affinity of all elements would increase. This negative feedback makes the affinity values converge toward a balanced distribution, with approximately as many elements with positive and negative affinity. Moreover, the splitting is not arbitrary. Let us consider a group of m synchronous elements, i.e., elements that are often in R at the same time. For simplicity, let us assume $|R| = m$. If the sum $\mathcal{A}$ of their affinity values is positive, the affinity of each of the m elements is incremented when the m elements are simultaneously in R, i.e., $\mathcal{A}$ is increased by m. Otherwise if $\mathcal{A}$ is negative, the affinity of each of the m elements is decreased, i.e., $\mathcal{A}$ is decreased by m. This positive feedback tends to give an identically-signed affinity to synchronous elements. Positive feedback manifests also with $|R| \neq m$. However, R should not be much larger than m, otherwise the contribution of $\mathcal{A}$ to A_R is no longer preponderant, and the positive feedback effect is lost in noise.

The two effects, a local positive feedback and a global negative feedback, act together to achieve working-set splitting. The interaction between them is sometimes complex. We show in Section 3.3 how the affinity algorithm behaves on some examples.

Postponed update. In a practical hardware implementation, it is difficult to update the affinity of all the elements of the working-set simultaneously. However, it is possible to postpone the update. We will update the affinity of an element only when this element enters or leaves R. We define the signed quantity $\Delta(t)$ as follows :

$$\Delta(t+1) = \Delta(t) + \text{sign}(A_R(t))$$

with Δ being initialized as $\Delta(0) = 0$. In addition, we define the following quantities I_e and O_e :

$$I_e(t) = A_e(t) - \Delta(t)$$
$$O_e(t) = A_e(t) + \Delta(t)$$

From Definition 1 of the algorithm, quantity $I_e(t)$ remains unchanged while $e \in R$, whereas $O_e(t)$ remains unchanged while $e \notin R$. Hence it is sufficient to record I_e when e enters R and O_e when e leaves R. When e enters R, we obtain I_e from $I_e = O_e - 2\Delta$, and when e leaves R, we obtain O_e from $O_e = I_e + 2\Delta$. One can recover A_e from $A_e = O_e - \Delta$.

We assume that there is a dedicated storage, the **affinity cache**, that records the value O_e for all cache lines e in the working-set. The value I_e is recorded in the **R-window**, which is a $|R|$-entry table representing R. If we want all the elements in R to be distinct, we need to implement a fully associative memory with LRU replacement, which can be costly to implement. In practice, it is possible to release this constraint, which is not an essential feature of the algorithm. Instead, we implement the R-window as a FIFO (first-in first-out), i.e., a memory array and a circular pointer on that array. When a cache line e is referenced at time t, its address is pushed in the R-window, along with the value $I_e(t)$.

The total affinity A_R in the R-window can be obtained incrementally. At a given time t, cache line e is brought into the R-window and line f is evicted from it. The affinity cache provides $O_e(t)$ and the R-window provides $I_f(t)$, from which $O_f(t)$ is derived. The total affinity in the R-window is obtained as

$$\begin{aligned} A_R(t+1) &= A_R(t) + A_e(t) - A_f(t) \\ &= A_R(t) + O_e(t) - O_f(t) \end{aligned}$$

Figure 2 gives a summary of the affinity algorithm implementation.

Limited number of affinity bits. In practice, O_e is coded with a limited number of bits. Consequently, the affinity algorithm works with saturating addition. Throughout this study, we assume 16 bits are used for coding the affinity. The other parameters are dimensioned accordingly :

- bits[I_e] = bits[O_e] = 16
- bits[A_R] = bits[O_e] + $\log_2(|R|)$
- bits[Δ] = bits[O_e] + 1

3.3. Behavior on examples

This section shows the behavior of the affinity algorithm on some examples. The version of the algorithm we implemented is the one described on Figure 2. We assume a working-set of N elements, each being identified by a number between 0 and $N-1$. We work with two examples of working-set behavior, which we call respectively $Circular$ and $HalfRandom(m)$ for convenience.

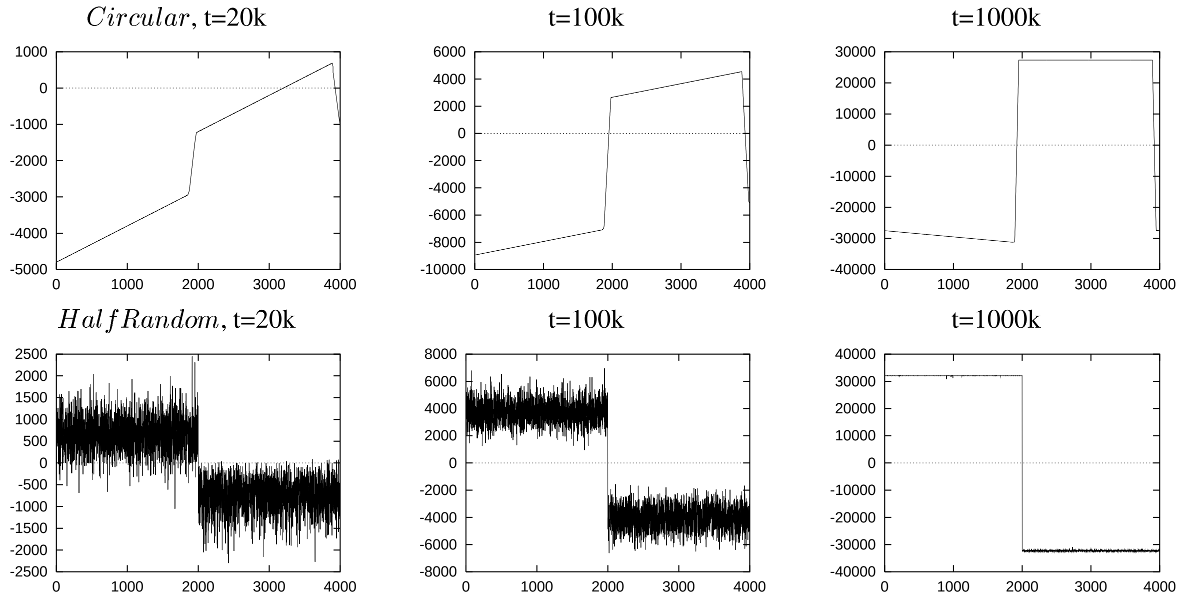

Figure 3: Affinity A_e for each $e \in [0..3999]$ on $\mathrm{Circular}$ (upper graphs) and $\mathrm{HalfRandom}(300)$ (lower graphs) with $|R| = 100$, after 20k, 100k, and 1000k references.

The $Circular$ working-set behavior corresponds to the infinite reference stream 0,1,2,..,N-1, 0,1,2,..,N-1, 0,1,2,..,N-1,... The $HalfRandom(m)$ working-set behavior corresponds to the infinite reference stream which consists in taking m random numbers in $[0, \frac{N}{2} - 1]$, then m random numbers in $[\frac{N}{2}, N - 1]$, then again m new random numbers in $[0, \frac{N}{2} - 1]$, m new random numbers in $[\frac{N}{2}, N - 1]$, and so on. In fact, $Circular$ is an important case, as many applications exhibit this kind of working-set behavior, especially after filtering by a L1 cache. On the other hand, $HalfRandom$ is artificial. Nevertheless, it shows the behavior of the affinity algorithm in a more difficult situation.

We consider a working-set of $N = 4000$ elements. We assume a R-window size $|R| = 100$. Figure 3 shows the affinity A_e for each $e \in [0..N - 1]$ on $Circular$ and $HalfRandom(300)$ after t=20k, t=100k and t=1000k references. On this example, after some time, the elements are split in two equally-sized subsets according to the sign of their affinity. At t=100k on this example, the splitting is optimal, with only one transition every 2000 references for $Circular$, and one transition every 300 references for $HalfRandom(300)$.

Impact of the R-window size. The ability of the affinity algorithm to split a working-set is somewhat independent from the R-window size. However, this is true to a certain extent only. We observed that the algorithm is able to split a $Circular$ working-set if $N > 2|R|$, but not if $N \leq 2|R|$. This can be understood intuitively from the discussion in Section 3.2 on the negative feedback effect. For the negative feedback to be effective, elements must spend more time out of R than in R. Hence the R-window should be smaller than half the working-set size. The size of the R-window has also an impact on the positive feedback effect. As noted in Section 3.2, the positive feedback is stronger when the R-window size equals m, i.e., the size of a group of synchronous elements. We verified experimentally that, in order to split $HalfRandom(m)$ correctly, one should not take $|R|$ much larger than m.

Initial affinity. In the definition of Section 3.2, we force $A_e(t_e) = 0$ at time t_e when e is first referenced. However, if the behavior of a working-set changes suddenly, the affinity algorithm will be able to adapt, but it will evolve from different initial conditions. We ran the algorithm on a $Circular$ behavior with different initialization methods (non-null constant, random value, $O_e(t_e) = 0$) and with different values for $|R|$. Unlike on Figure 3, the splitting for $Circular$ was not optimal, which is not a problem as long as transitions do not happen too often. Actually, we observed that the R-window acts as a sort of low-pass filter with respect to affinity frequencies on $Circular$: after enough time, the transition frequency never exceeded one transition every $2|R|$ references. Hence we made sure to take a R-window that is not too small.

3.4. Filtering transitions

One may define "splittability" as the existence of a balanced partitioning with a transition frequency smaller than, say, one transition every 10 references on average. For example, let us consider a working-set which elements are accessed in a completely random fashion, with equal probability. Such working-set is not "splittable" : however we split the set in two parts of equal size, the transition frequency equals $1/2$. There exists applications with random-like reference streams (we observed such behavior on *164.gzip* and *175.vpr* for instance). For these applications, there is nothing to expect from the affinity algorithm. Nevertheless, we want the transition frequency to be as small as possible in this case, because the penalty of migrations is not compensated by a reduction of L2 misses. Our goal is to limit transitions on working-sets that are not "splittable", while allowing fast transitions on working-sets that are "splittable". To this aim, we define a **transition filter** F. The transition filter is an up-down saturating counter updated on each reference : for a reference e at time t, $F(t+1) = F(t) + A_e(t)$. Instead of looking at the sign of A_e for determining which subset e belongs to, we look at the sign of F. The optimal saturation level for the counter depends on the migration penalty. If we double the saturation level (i.e., we add one bit to F), we roughy divide by two the transition frequency on working-sets with a random behavior, but we also double the delay between the potential and actual transitions on working-sets that are "splittable". For example, with 16 bits of affinity and a 20-bit transition filter, the transition frequency is $1/2^{(1+20-16)} \approx 3\%$ on a working-set with no "splittability" and for which the affinity appears to be saturated positive or negative with probability $1/2$. On the other hand, there is a delay of 16 references between the potential and actual transition on working-sets that are "splittable".

L2 filtering. When a working-set fits in a single L2 cache, migrations are useless. All the requests resulting from L1 misses are processed by the mechanism described on Figure 2, but it is possible to decrease unnecessary migrations by updating the transition filter only on L2 misses. It should be noted that, with such L2 filtering, a migration can happen only upon a L2 miss.

3.5. Working-set sampling

The affinity cache size should be proportional to the total on-chip L2 capacity. For instance, let us assume a 4-core processor with a 512-Kbyte L2 cache on each core. The total on-chip L2 capacity is 2 Mbytes, hence we are interested in splitting working sets which size does not exceed 2 Mbytes. Assuming 64-byte cache lines, we need a 32k-entry affinity cache. Each entry holds a tag (some bits from the line address e), an affinity value O_e, plus a few bits for age-based replacement, say 2 bits. If we assume 20 tag bits and 16 affinity bits, the total affinity cache size is 152 Kbytes, i.e., 7.4% of the total on-chip L2 data.

It is possible to decrease the size of the affinity cache by sampling the working-set. The lines are sampled by applying a hashing function $\mathcal{H}$ on the line address e. There are many possible ways to implement $\mathcal{H}$. For this study, we used $\mathcal{H}(e) = e \mod 31$. Choosing a prime number like 31 limits the occurrence of pathological behaviors under constant-stride reference streams, which are frequent. Moreover, it can be implemented in hardware by splitting the bits of e into blocks e_i of 5 consecutive bits, i.e., $e = \sum_i 2^{5i} e_i$. We have $\mathcal{H}(e) = \sum_i e_i \mod 31$, which can be obtained with a carry-save adder and a ROM. It is possible to reduce a 32k-entry/152-Kbyte affinity cache to 8k-entry/38-Kbyte by applying the affinity algorithm to only one fourth of the working-set, for instance lines e such that $\mathcal{H}(e) < 8$. For a line e such that $\mathcal{H}(e) \geq 8$, we do not allocate any entry in the affinity cache. We simply rely on the transition filter to tell us which subset e belongs to. It should be noted that working-set sampling decreases the frequency of migrations. Hence the transition filter should be dimensioned accordingly. For example, if only 25% of references update the transition filter, the transition filter can be 2 bits shorter.

benchmark	input	instr.	16KB i-miss	16KB d-miss
SPEC2000				
164.gzip	train	1000	0.00	15.41
171.swim	train	1000	0.27	21.32
172.mgrid	train	1000	0.08	6.50
175.vpr	train / place	1000	0.01	14.41
176.gcc	train	1000	41.61	8.27
179.art	train	1000	0.00	97.74
181.mcf	train	1000	0.00	68.80
186.crafty	train	1000	83.46	5.57
188.ammp	train	1000	0.03	127.12
197.parser	train	1000	0.16	9.81
255.vortex	train	1000	41.83	6.65
256.bzip2	train	1000	0.00	7.20
300.twolf	train	1000	2.71	20.04
Olden				
bh	2k bodies	1000	0.00	4.60
bisort	250000 num	419	0.00	2.06
em3d	2000 nodes	47	0.00	3.54
health	5 lev,500 iter	154	0.00	10.36
mst	1024 nodes	156	0.00	3.06

Table 1: Benchmarks, inputs, dynamic instruction count, number of IL1 and DL1 misses. All numbers are in millions.

3.6. 4-way working-set splitting

So far we have considered 2-way working-set splitting. However, it is possible to achieve 4-way splitting by applying 2-way splitting recursively. First we split the working-set in two subsets, then we split each subset. Let X be the 2-way splitting mechanism for the whole working-set. This mechanism X comprises a R-window R_X, the associated A_{R_X}, Δ_X, and a transition filter F_X. Similarly, we define two 2-way splitting mechanisms $Y[+1]$ and $Y[-1]$. In the remaining, we assume $|R_{Y[+1]}| = |R_{Y[-1]}| = |R_X|/2$. The three mechanisms X, $Y[+1]$ and $Y[-1]$ share the affinity cache. Instead of defining two affinity values per element in the working-set, we take advantage of working-set sampling. First we read O_e from the affinity cache, as depicted on Figure 2. Meanwhile, we compute $\mathcal{H}(e)$. If $\mathcal{H}(e)$ is odd, O_e is processed by mechanism X. Otherwise if $\mathcal{H}(e)$ is even, O_e is processed by mechanism $Y[\text{sign}(F_X)]$. Which of 4 subsets an element belongs to is given by $(\text{sign}(F_X), \text{sign}(F_{Y[\text{sign}(F_X)]}))$.

4. Simulations

All simulation results in this study are obtained with the SimpleScalar infrastructure [3]. Our simulator is built on top of the PISA functional simulator. We show results for 13 benchmarks from the SPEC CPU2000 suite [18], and 5 benchmarks from the Olden suite [7] (we used the sequential version by Amir Roth [16]). Benchmarks from these

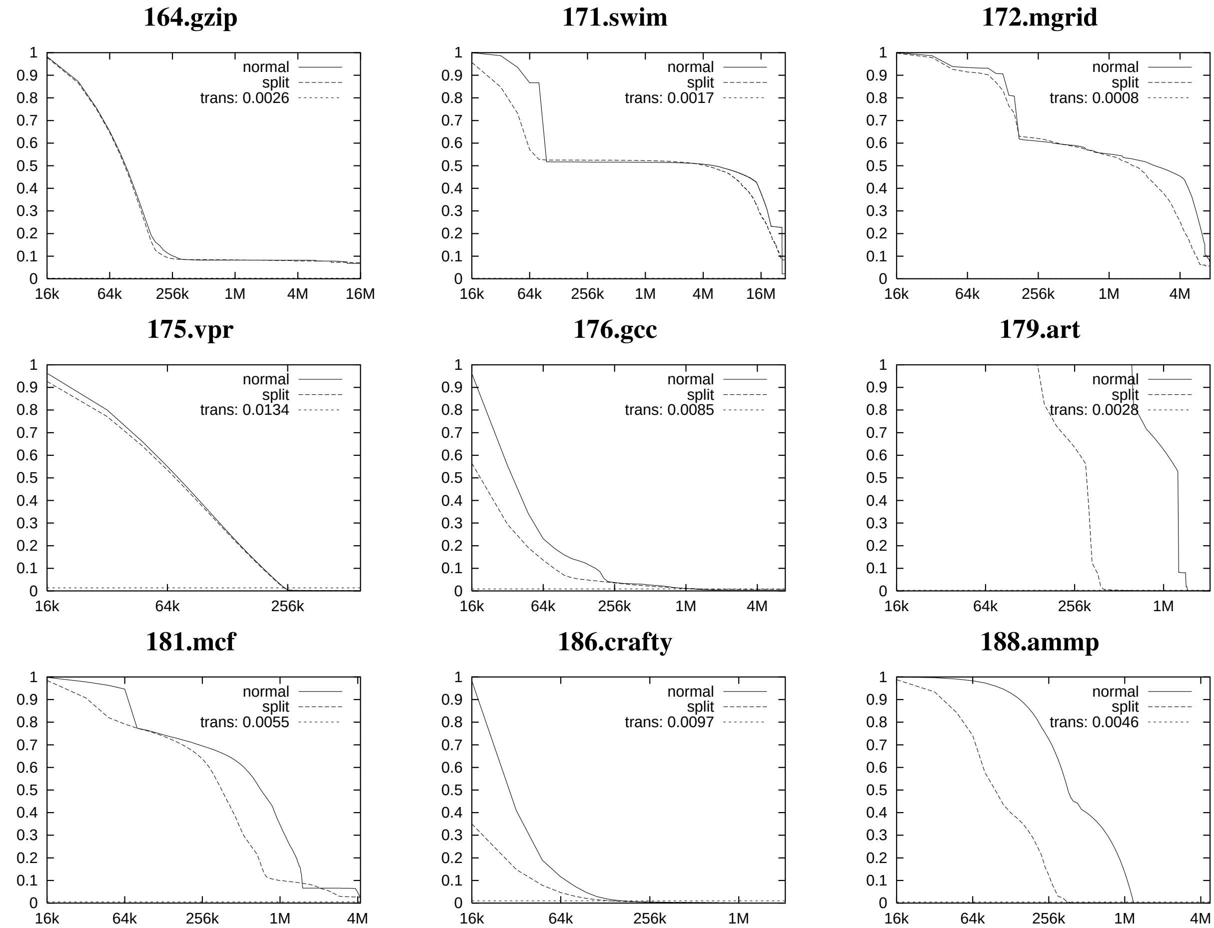

Figure 4: Fraction of L1-filtered references with a LRU-stack depth greater than a given cache size (from 16 Kbytes to 16 Mbytes). Comparison between the single stack ("normal") and the 4-way stack ("split"). The transition frequency is indicated on each graph.

suites that we did not select either have few capacity misses or could not be compiled with the PISA gcc compiler. The third column of Table 1 gives the number of dynamic instructions simulated for each benchmark. We stop simulation after 1 billion instructions.

4.1. LRU stack profiles

In this section, we simulate LRU stacks [15] and show that "splittability" is a rather common property of programs. We work with a stream of references that is filtered by a 16-Kbyte DL1 cache and a 16-Kbyte IL1 cache, both fully-associative with LRU replacement. Each reference consists of a cache line address, assuming 64-byte lines. The fourth and fifth columns of Table 1 give the number of DL1 and IL1 misses respectively. In this experiment, we do not distinguish between loads and stores. The first part of the experiment consists in determining the LRU stack profile $p_1(x)$, which gives the fraction of dynamic references (i.e., L1 misses) with a LRU stack depth greater than x, considering that a reference which is encountered for the first time has an infinite LRU stack depth. In other words, it is like simulating a fully-associative cache with LRU replacement and giving the cache miss ratio as a function of the cache size. The second part of the experiment consists in simulating the affinity algorithm and four LRU stacks. The address of each cache line missing the L1 is sent to only one of the four LRU stacks, which is determined as described in Section 3.6. After accessing the appropriate LRU stack, we update the migration controller state (affinity cache, R-window, A_R, Δ, transition filter). We assume 20-bit transition filters, $|R_X| = 128$, and $|R_{Y[+1]}| = |R_{Y[-1]}| = 64$. Moreover, we assume an unlimited affinity cache size. We do not apply L2 filtering in this experiment, as the L2 is not defined. At the end of the simulation, we report the global LRU stack profile $p_4(x)$, i.e., the fraction of all dynamic references with a LRU stack depth greater than x. Then we compare $p_1(x)$ and $p_4(x)$, keeping an eye on the transition frequency. Results are presented on Figures 4 and 5. Each graph shows $p_1(x)$ and $p_4(x)$ between 16 Kbytes and 16 Mbytes (i.e., for x between 256 and 256k lines). Also indicated on each graph is the transition frequency, that is, the ratio of the number of transitions to the total number of references. It is represented by a horizontal line, which,

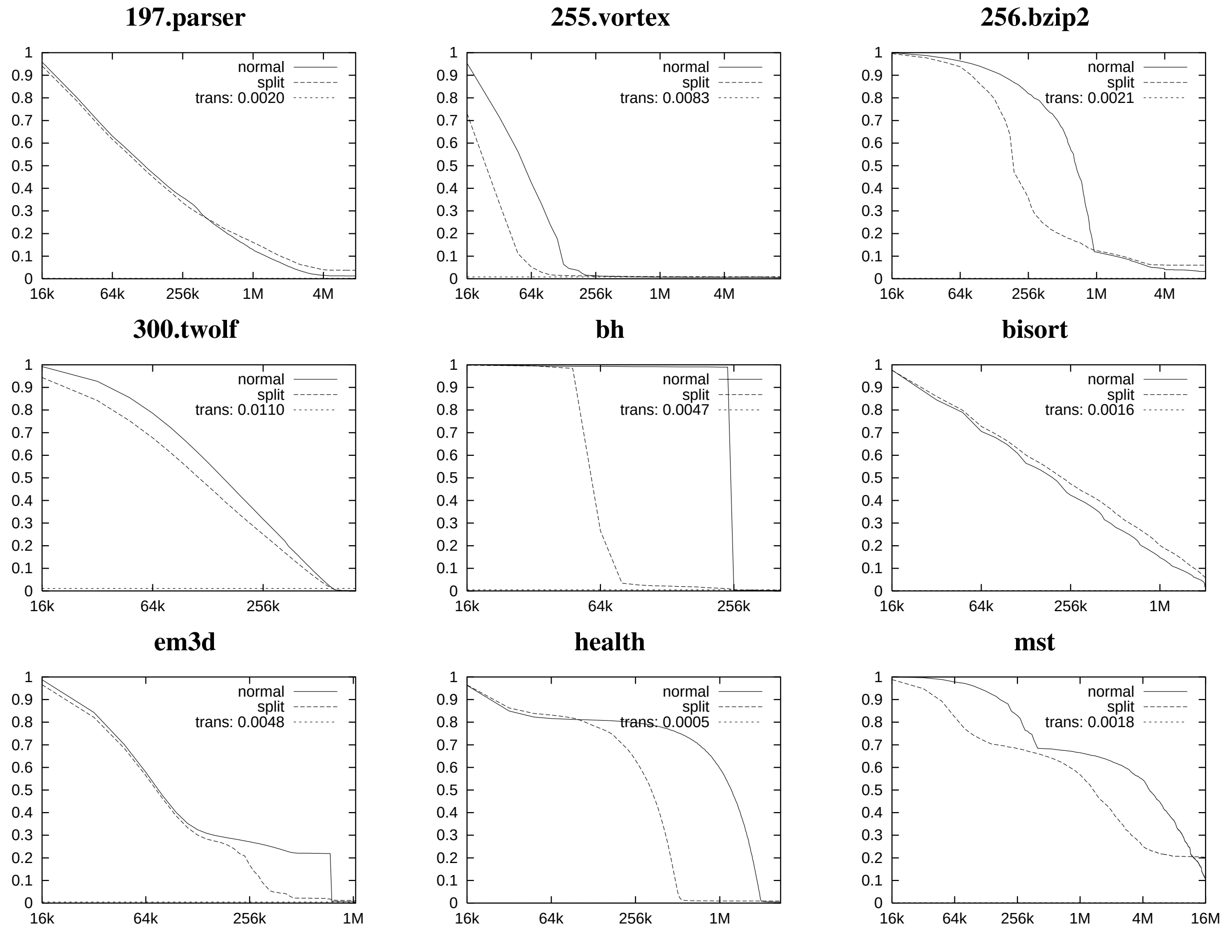

Figure 5: Cf. Figure 4

for many benchmarks, is difficult to distinguish from the zero line. It can be observed that, on many benchmarks, the curves for p_1 and p_4 are quite distinct. This is the case, for instance, on *179.art*, *188.ammp*, *bh*, *health*, and several others. These benchmarks exhibit "splittability". On the other hand, on *164.gzip*,*175.vpr*, *197.parser* and *bisort*, $p_1(x)$ and $p_4(x)$ are very close whatever value of x, which indicates either a lack of "splittability" or not enough working-set reuse. It can be observed that, in all cases, the transition frequency remains low. Among our benchmarks, the one with the highest transition frequency is *175.vpr*, with 1.34% of transitions per stack access.

We ran similar experiments with different cache line sizes, and observed that "splittability" is less pronounced with larger lines. Going back to the graph partitioning problem, using larger lines is like merging nodes, or equivalently, adding the constraint that merged nodes must be in the same subset. This constraint can only increase the minimum cut size.

4.2. Four-core processor with 512-Kbyte L2 caches

In the following experiment, we assume a four-core processor with a 512-Kbyte cache on each core, hence a total 2-Mbyte on-chip L2 capacity. The cache line size is still 64 bytes. The IL1 and DL1 caches are both 16 Kbytes, 4-way set-associative. The L2 cache is 512-Kbyte, 4-way skewed-associative [5]. The DL1 is write-through, non write-allocate, while the L2 is write-back, write-allocate. The affinity cache has 8k entries and is 4-way skewed-associative. We apply 25% working-set sampling, as described in Section 3.5. Upon a miss for line e in the affinity cache, we force $A_e = 0$ by setting $O_e = \Delta$ (cf. Figure 2). The other parameters of the migration controller are : 18-bit transition filters, $|R_X| = 128$, and $|R_{Y[+1]}| = |R_{Y[-1]}| = 64$. We apply L2 filtering as described in Section 3.4 : the affinity cache, R-window, A_R and Δ are updated on each L1 miss request, but the transition filter is updated only on L2 misses. Results are shown on Table 2. Instead of giving raw numbers for a given event (cache miss or migration), we give the average number of instructions per event (higher is better). Raw numbers can be recovered from the number of instructions given on table 1. The column denoted "ratio" shows the reduction in L2 misses. When this number is smaller than 1, it means that execution migration is able to remove L2 misses. The benchmarks for which this is the case are *179.art*, *181.mcf*, *188.ammp*, *256.bzip2*, *em3d* and

	L1 miss	L2 miss	4xL2 miss	ratio	migration
SPEC2000					
gzip	64	775	770	1.01	2.2×10^6
swim	46	89	89	1.00	1.4×10^7
mgrid	152	259	259	1.00	3.0×10^6
vpr	54	90424	56363	1.60	2.0×10^8
gcc	19	823	869	0.95	2.1×10^5
art	10	11	341	**0.03**	2.0×10^4
mcf	14	24	36	**0.67**	4.5×10^3
crafty	11	4451	3942	1.13	2.2×10^6
ammp	7	16	99	**0.17**	1.0×10^4
parser	94	436	437	1.00	1.9×10^5
vortex	17	2215	2018	1.10	1.6×10^6
bzip2	138	253	724	**0.35**	5.0×10^5
twolf	30	612	613	1.00	1.7×10^5
Olden					
bh	216	138197	63979	2.16	6.9×10^6
bisort	188	676	625	1.08	2.6×10^5
em3d	13	65	476	**0.14**	1.3×10^5
health	13	20	147	**0.14**	5.8×10^4
mst	50	72	72	1.00	1.8×10^5

Table 2: L1 misses, L2 misses, L2 misses with migrations enabled ("4xL2") and number of migrations are given in number of instructions per event (higher is better).

health. In all cases, the frequency of migrations is kept under control. This is due to the transition filter, but not only. L2 filtering is very effective at limiting unnecessary migrations, for instance on benchmarks with a small working-set already fitting in a single 512-Kbyte L2 cache (e.g., *bh*, *255.vortex*, *186.crafty*). Moreover, for benchmarks with a large working-set not fitting in a 2-Mbyte 4xL2 cache (e.g., *171.swim*, *172.mgrid*, *mst*), migrations are reduced thanks to the limited size affinity cache : as we set $A_e = 0$ when there is a miss in the affinity cache, the transition filter value does not change. Benchmarks with the highest migration frequency are those on which migrations permit decreasing L2 misses. For example, on *181.mcf*, a migration occurs every 4500 instructions on average. For this benchmark, the number of L2 misses removed per migration is $4500/24 - 4500/36 \approx 60$. It means that as long as the migration penalty is less than 60 times the L2-miss/L3-hit penalty, i.e., $P_{mig} < 60$, we will observe performance gains on *181.mcf*.

5. Related work

The "cache-wall" problem has been pointed out in a recent study [14]. It was observed that by breaking a large L2 cache into several banks, it is possible to access some parts of the cache faster. This leads to a non-uniform cache structure which tries to approximate a continuum of access latencies. Cache lines move automatically so that lines that are accessed more often are located in banks closer to the processing core.

The ESP MMM model [8] introduced the key idea that execution resources can be replicated not only for exploiting execution parallelism, but also for dealing with a large amount of memory. This idea was re-visited for the DataScalar architecture [6]. The concept of execution migration is somewhat related to DataScalar. However, there are significant differences. DataScalar works at the physical memory level, with a static binding between physical memory regions and processors, whereas execution migration works at the L2-cache level. Moreover, DataScalar relies on spatial locality for minimizing the frequency of *lead changes* (equivalent of migrations), whereas execution migration relies on "splittability".

Execution migration makes sense only if the latency between cores is small and the communication bandwidth high, which is plausible on a single-chip multi-core. We believe this approach is not viable on a traditional parallel machine. The term "computation migration" has been used in the context of parallel machines (e.g., [7]). The goal was to speed-up parallel programs. Execution migration, on the other hand, is a microarchitectural technique that concerns sequential programs.

An approach related to ours is the *cascaded execution* model proposed in [2]. Cascaded execution was proposed as a compiler optimization for speeding-up sequential execution, more specifically the execution of loops that are difficult to parallelize. Iterations of the loop are distributed among the different processors. At any time, only a single processor is executing the loop. Execution switches to another processor after a certain number of iterations. Meanwhile, other processors execute a *helper phase* which goal is to prepare the data for future iterations on these processors. The helper phase can prefetch data into the caches by executing a shadow version of the loop body, or it can reorganize data to improve spatial locality.

6. Conclusion

We proposed to modify a multi-core processor so that the execution can migrate quickly from one core to another. The goal is to allow a sequential application to take advantage of the overall on-chip L2 capacity. We introduced the affinity algorithm, a method for splitting automatically a working-set. We have shown that this method works on 4-core configurations. However, it works also on 2-core configurations, and we believe it is possible to adapt it to a larger number of cores. An application can take advantage of execution migration if its working-set exhibits a property which we call "splittability". We have shown that "splittability" is rather common among popular benchmarks like SPEC CPU2000 and Olden. This study has exhibited a potential for significant performance gains on certain sequential applications. However, the viability of execution migration needs to be confirmed.

Although we did not consider any particular value for the migration penalty P_{mig}, an implicit assumption was that P_{mig} does not exceed a few tens of L2 misses. The optimal tuning of parameters (R-window size, transition filter, working-set sampling ratio) depends on P_{mig}. If P_{mig} is small, e.g., $P_{mig} < 10$, it is possible to tolerate migration frequencies higher than those reported on Table 2. Moreover, it may be useful to distinguish low-penalty and high-penalty L2 misses. For instance, pointer loads found in applications using linked data structures generally have a high miss penalty. One could decide to restrict the class of applications triggering migrations by having the transition filter updated only on requests coming from pointer loads.

Further research is required to better understand "splittability" and ways to expose it. In particular, there is a connection with cache prefetching. Execution migration is not intended to replace prefetching. Future research should determine how to best combine prefetching and execution migration. However, it is not clear to what extent execution migration is orthogonal to prefetching. In theory, there is more to "splittability" than predictability (e.g., *HalfRandom*). However in practice, much of the "splittability" we observed seems to come from *circular* working-set behaviors on which prefetching is likely to succeed. It is possible that execution migration, as a way to decrease L2 misses, is mostly interesting on applications using linked data structures. However, prefetching into a "larger" cache leaves more room for the unpredictable portion of the working-set, and execution migration may be useful on other applications as well.

As for the complexity of implementing execution migration, most difficulties lie not in the migration controller, but in the hardware support for allowing fast migrations. In particular, a straightforward implementation of the update bus, as described in Section 2.3, may incur a significant power consumption. Several directions are possible for decreasing the update bus bandwidth without overly increasing the migration penalty. In particular, register updates consume most bandwidth. One could decide to broadcast register updates only when the transition filter absolute value falls below a certain threshold, as it indicates a possible migration. Upon a migration, it would be necessary to broadcast missing register updates. One may also filter register updates with a small register-update cache. A register update would be sent only upon evicting an entry from the register-update cache. Upon a migration, the content of the register-update cache would be spilled on the update bus.

Although execution migration offers a way to tackle the cache-wall problem on working-sets exhibiting "splittability", we do not think that execution migration alone justifies implementing a multi-core. The main motivation for building a multi-core remains increasing the execution throughput on multiprogrammed workloads and parallel applications. Then, execution migration comes as a bonus. We believe the hardware cost of execution migration will be better accepted if one can find other advantages to migrating the execution quickly to another core. For instance, it has been suggested that migrating periodically the activity to different parts of the chip permits a higher heat dissipation [11]. Another interesting direction to explore is the use of execution migration to exploit branch prediction tables.

References

[1] C. Alpert and A. Kahng. Recent directions in netlist partitioning : a survey. *Integration, the VLSI Journal*, 19(1-2):1–81, 1995.

[2] R. Anderson, T. Nguyen, and J. Zahorjan. Cascaded execution: speeding up unparallelized execution on shared-memory multiprocessors. In *Proceedings of the 13th International Parallel Processing Symposium*, 1999.

[3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. *IEEE Computer*, 35(2), Feb. 2002.

[4] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[5] F. Bodin and A. Seznec. Skewed associativity improves program performance and enhances predictability. *IEEE Transactions on Computers*, 46(5):530–544, May 1997.

[6] D. Burger, S. Kaxiras, and J. Goodman. Datascalar architectures. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[7] M. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1995.

[8] H. Garcia-Molina, R. Lipton, and J. Valdes. A massive memory machine. *IEEE Transactions on Computers*, C-33(5), May 1984.

[9] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[10] L. Hammond, B. Hubert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, Mar. 2000.

[11] S. Heo, K. Barr, and K. Asanović. Reducing power density through activity migration. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2003.

[12] J. Huh, S. Keckler, and D. Burger. Exploring the design space of future CMPs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[13] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, Feb. 1970.

[14] C. Kim, D. Burger, and S. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[15] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[16] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structure. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[17] D. Sima. The design space of register renaming techniques. *IEEE Micro*, 20(5):70–83, Sept. 2000.

[18] Standard Performance Evaluation Corporation. http://www.spec.org.

[19] J. Tendler, J. Dodson, J. Fields, H. Le, and B. Sinharoy. POWER4: system microarchitecture. *IBM Journal of Research and Development*, 46(1), Jan. 2002.