Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

A spatially triggered dissipative resource distribution policy for SMT processors[*]

Hong-zhou CHEN[†], Xue-zeng PAN, Ling-di PING, Kui-jun LU, Xiao-ping CHEN
(*School of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China*)
[†]E-mail: honjoychan@hotmail.com

Received Nov. 26, 2007; revision accepted May 16, 2008

Abstract: Programs take on changing behavior at runtime in a simultaneous multithreading (SMT) environment. How reasonably common resources are distributed among the threads significantly determines the throughput and fairness performance in SMT processors. Existing resource distribution methods either mainly rely on the front-end fetch policy, or make distribution decisions according to the limited information from the pipeline. It is difficult for them to efficiently catch the various resource requirements of the threads. This work presents a spatially triggered dissipative resource distribution (SDRD) policy for SMT processors. Its two parts, the self-organization mechanism that is driven by the real-time instructions per cycle (IPC) performance and the introduction of chaos that tries to control the diversity of trial resource distributions, work together to supply sustaining resource distribution optimization for changing program behavior. Simulation results show that SDRD with fine-grained diversity controlling is more effective than that with a coarse-grained one. And SDRD benefits much from its two well-coordinated parts, providing potential fairness gains as well as good throughput gains. Meanings and settings of important SDRD parameters are also discussed.

Key words: Simultaneous multithreading (SMT), Resource distribution, Dynamic optimization, Dissipative structures
doi:10.1631/jzus.A0720083 **Document code:** A **CLC number:** TP302

INTRODUCTION

Simultaneous multithreading (SMT) (Hirata *et al*., 1992; Tullsen *et al*., 1995; 1996) is one microarchitectural innovation exploiting thread level parallelism. SMT processors can run instructions from different threads concurrently, and alleviate the suffering from insufficient instruction level parallelism (ILP). As a result the combined ILP from all threads produces high performance gains. In an SMT model, each thread possesses some private resources, and also shares some critical data-path resources such as the physical registers, the execution units, and the caches. However, threads compete for common resources more than they share them, so how reasonably common resources are distributed among threads mainly determines the throughput and fairness performance of SMT processors.

Most of the existing resource allocation methods only rely on the front-end fetch policy (Tullsen *et al*., 1996; Tullsen and Brown, 2001; El-Moursy and Albonesi, 2003; Cazorla *et al*., 2004a), which select threads to fetch according to some limited pipeline information, such as cache miss or issue queue occupancy counts, and allocate resources among threads implicitly by controlling the number of instructions that flow into the pipeline. It is difficult for this implicit method to reflect the real resource requirement of threads. Moreover, the fetch selection made by limited monitored information is somewhat uncertain, and cannot efficiently address the problem of resource monopolization or wastage.

[*] Project supported by the Hi-Tech Research and Development Program (863) of China (No. 2006AA01Z431) and the Key Science and Technology Program of Zhejiang Province (Nos. 2007C11068 and 2007C11088), China

Some other methods allocate resources to threads explicitly, and obtain a good effect. But the majority of them (Latorre *et al*., 2004; Cazorla *et al*., 2004b; Sharkey *et al*., 2006) still adjust resource distribution according to limited pipeline information, without targeting end performance directly, and they are unaware of the impact of their resource distribution on real performance. So it is also hard to meet the real resource requirement of threads, especially in a complicated SMT environment with changing program behavior. The learning-based policy in (Choi and Yeung, 2006) optimizes resource distribution directly targeting the end performance. However, when program behavior becomes complicated, it exhibits weak persistence in exploring new optimal distribution, and is apt to stagnate in the less optimal space of resource distribution. This indicates an imbalance between its exploitation of historical experiences and its self-exploration for new distribution.

This work presents a spatially triggered dissipative resource distribution (SDRD) policy for SMT processors. There are two parts in the distribution optimization procedure of SDRD, one is the self-organization mechanism that tries to exploit the best historical distribution solution, and the other is the introduction of chaos that tries to control the spatial diversity of trial distribution solutions. These two parts work together to supply persistent resource distribution optimization for changing program behavior. Both the fine- and coarse-grained diversity controlling of SDRD are studied. Simulation results show that SDRD with fine-grained diversity controlling is more effective than that with a coarse-grained one. And SDRD benefits much from its well-coordinated parts, providing good fairness gains as well as throughput increments. Meanings and settings of important SDRD parameters are also discussed.

The rest of this paper is structured as follows. Section 2 discusses related work, and Section 3 presents our SDRD policy. Then Section 4 describes our evaluation methodology, and Section 5 discusses the experimental results and important parameters of SDRD. Finally, Section 6 concludes the paper.

RELATED WORKS

Current resource distribution methods for SMT processors can be categorized into implicit policy and explicit policy.

Implicit policy performs resource distribution among threads implicitly by controlling the number of instructions flowing into the pipeline at the front-end, such as ICOUNT, STALL, DG (data gating), FLUSH, etc. ICOUNT (Tullsen *et al*., 1996) gives a higher fetch priority to the thread that has fewer instructions in pre-issue stages. It is unaware of the long latency problem of a cache miss which results in resource clogging. STALL (Tullsen and Brown, 2001) prevents a thread from fetching more instructions if it experienced an L2 cache miss. DG (El-Moursy and Albonesi, 2003) stops fetching from a thread if its cache miss count exceeds a threshold, and PDG (predictive data gating) (El-Moursy and Albonesi, 2003) stops fetching as soon as a cache miss is predicted. DWarn (Cazorla *et al*., 2004a) establishes different fetch priorities for the threads. When a cache miss happens, DWarn only degrades the fetch priority of the corresponding thread, instead of stopping fetching immediately. FLUSH (Tullsen and Brown, 2001) introduces a wiping mechanism to release the clogged resources by sweeping out the instructions of those threads that monopolize the resources, and then re-fetch those instructions if necessary. However, the re-fetching operation imposes extra pressure on fetch bandwidth and power consumption and, in addition, the problem of resource wastage still exists.

These implicit methods select the fetching thread according to monitored pipeline information such as the cache miss count, the occupancy of the issue queue, etc., and they implicitly allocate resources among threads by controlling the number of instructions flowing into the pipeline. It is difficult for these implicit methods to reflect the real resource requirements of threads. Moreover, their fetch selection or decision is made from the limited or uncertain information that cannot efficiently guide the resource distribution.

Explicit policies allocate resources to threads in a direct way. Among them, the STATIC method (Marr *et al*., 2002; Raasch and Reinhardt, 2003) partitions the resources among all threads statically, with each thread monopolizing equal resources. It suffers resource wastage because some thread may not take full advantage of the allocated resources, while other explicit methods dynamically make a decision on resource distribution based on some information such as the feedback information from the back-end in DBA (dynamic back-end assignment) (Latorre *et al*., 2004), the limited pipeline information monitored in DCRA (dynamically controlled resource allocation) (Cazorla *et al*., 2004b), and the program phase information in AROB (adaptive reorder buffers) (Sharkey *et al*., 2006). DBA collects statistical information from the back-end periodically, then feeds this back to the resource allocation decision and assigns clustered back-end resources to threads. It is a coarse-grained resource distribution policy. DCRA classifies the threads into 'fast type' and 'slow type' according to their cache miss rates, and determines the resource distribution according to their usage of resources. AROB dynamically identifies the issue- or commit-bound phase of each thread, and then allocates a new reorder buffer (ROB) partition to the thread that steps into a commit-bound phase and de-allocates a ROB partition of the thread that steps into an issue-bound phase.

Generally, these dynamic explicit policies have good effect. But they are guided by supervised indirect information instead of the real performance, and they have no idea about the impact of their distribution solutions on the actual performance. So it is difficult to figure out an optimal solution. In addition, such policies often focus on alleviating some specific bottlenecks in the SMT processors, making them not generally effective for those complicated SMT environments with changing program behavior.

HILL (Choi and Yeung, 2006) dynamically optimizes resource distribution via the hill-climbing technique established over several trial partitions. Although HILL directly targets performance and obtains impressive results, it fastens the exploration scope of the trial partitions when they try to explore new optimal partition in the resource distribution space, making the exploration less flexible. On the other hand, HILL always allows the trial partitions to be fully influenced by the best historical experience, placing too much emphasis on the exploitation of historical experience. Hence, when the program behavior becomes complicated, HILL exhibits weak persistence in exploring new optimal partitions, and is apt to fall into the trap of less optimal resource distribution space. Moreover, when the harmonic mean of weighted IPC (instructions per cycle) or average weighted IPC is taken as the function for evaluating the fitness of the trial partitions, the single mode IPC performance of each thread is needed. It needs to periodically run on only a single thread and stall all of the rest of the threads, and this wastes the processor's resources.

Our approach merely uses the throughput IPC performance as the feedback to drive a resource distribution decision. Inspired by the dissipative structure theory in the thermodynamic field, the spatially triggered chaos in our approach maintains the diversity of trial resource distributions at a certain level. It not only avoids an excessive exploitation on the best historical distributions but also provides extra chance for meeting the resource requirements of slow threads. Therefore both the throughput IPC and fairness performance are improved, especially in a complicated SMT environment.

SPATIALLY TRIGGERED DISSIPATIVE RESOURCE DISTRIBUTION

The emergence of order in the evolution process of biological and social systems has been a fundamental inspiration in the development of evolutionary theory. Open system structures of increasing complexity were developed into a general thermodynamic concept of dissipative structures by Prigogine (1967), which is frequently used as a generic dynamic concept to describe the evolvement of nonlinear systems (Prigogine, 1976; Nicolis and Prigogine, 1977). In the dissipative structure theory, a nonlinear open system, which is far from an equilibrium state, can transform from an original chaotic state to a spatiotemporally ordered state in a fluctuant process resulting from the synergy of the self-organization mechanism and continual entropy exchange with the outer environment.

When threads run in an SMT environment, their resource requirements change with varying workload behavior. It is necessary to adjust the resource distribution among threads in different program phases. In our resource distribution evolvement system, there are a number of distribution candidates representing different solutions for resource distribution. Each distribution candidate is applied to the system for a period of time (called 'applying period'). When all of the distribution candidates are applied to the system one after the other, a new generation of distribution candidates is figured out. The period between two adjacent generations is called a 'generation period'.

Our approach borrows the idea of dissipative structures from two angles: (1) it establishes a self-organization mechanism that tries to bring the distribution candidates to an ordered state of desirable convergence with optimal distribution for the program behavior; (2) it introduces some extra chaos on behalf of the entropy exchange with outer surroundings to keep the distribution candidates away from undesired convergence and launches a new evolvement stage. Then, with the cooperation of self-organization and extra chaos, the fluctuation comes into being to provide sustainable optimization in the resource distribution evolvement process.

Self-organization mechanism

The distribution of two kinds of critical common resources, the ROBs and the issue queue (IQ), is involved in our approach. For simplicity, only the distribution of ROB among threads is evolved. The distribution of IQ is kept in proportion to that of ROB in every applying period. The distribution candidate that yields the best IPC performance in a previous generation period directs the resource distribution evolvement in the succeeding generation period. In the iteration of this procedure, optimal resource distribution is supposed to be approached.

Let M denote the number of distribution candidates, N denote the number of threads, the ith distribution candidate $\boldsymbol{D}_i=(d_{i1}, d_{i2}, \cdots, d_{ij}, \cdots, d_{iN})$ denote the ith solution of ROB distribution among threads from 1 to N. According to the program locality principle, the local best historical distribution candidate $\boldsymbol{D}_l$ is taken to direct the self-adjustment activities of all candidates. The preliminary computational model for SDRD to calculate a new generation of the distribution candidates is established by

$$d_{ij} = d_{ij} + r \times (d_{lj} - d_{ij}), \qquad (1)$$

where r is a random number in $[B_l, B_u]$. The positive numbers B_l and B_u represent the exploitation scope to which extent the distribution candidates are attracted by the best historical experience. The randomicity introduced here brings more chances for those low ILP threads to obtain a share of extra resources, preventing pursuit of the throughput performance in disregard of the fairness performance.

Spatially triggered chaos

According to Eq.(1), however, the distribution candidates may become closer and closer to the local historical experience and more and more similar to each other, till they arrive at a stationary state of convergence. It is desirable if they converge in the optimal resource distribution space. Unfortunately if it is not a pleasing convergence, but a trapped stagnation in a less optimal distribution space, then the similar candidates will lose their diversity and it is hard for the evolvement to find a better distribution solution. Moreover, when a new program phase arrives the distribution candidates may converge in a distribution space that is optimal for the previous phase but not for the succeeding one, and cannot escape from the stationary equilibrium state to capture the emergence of the new resource requirement agilely. To handle this problem of diversity loss, according to the dissipative structures theory, extra chaos is expected to break the undesired equilibrium and maintain the diversity of the distribution candidates.

However, it is hard to differentiate an undesired stagnation (which wants disturbance) from a desirable convergence (which should be preserved). In SDRD this difficulty is addressed by a random control on the spatial diversity of the distribution candidates. That is, if the spatial diversity metric of the candidates is detected as below some kind of diversity threshold θ, then it is suspected that an undesirable stagnation might be happening, and extra chaos is triggered at a chaos probability of p by resetting some candidates to an arbitrary distribution to hold the whole diversity at a certain level.

Depending on whether the extra chaos is triggered in an intra-generation or an inter-generation manner, two modes are designed for SDRD: the local spatially triggered mode (L-SDRD) and the global spatially triggered mode (G-SDRD). L-SDRD measures the diversity between $\boldsymbol{D}_i$ and its predecessor $\boldsymbol{D}_{i-1}$ before applying each candidate $\boldsymbol{D}_i$ in every generation. The local diversity metric between $\boldsymbol{D}_i$ and $\boldsymbol{D}_{i-1}$ is defined as the sum of the differences of their corresponding components (the Manhattan distance), shown by Eq.(2):

$$DIV_i = \sum_{j=1}^{N} \left| d_{ij} - d_{(i-1)j} \right|, \quad i = 2,3,\ldots,M, \tag{2}$$

If DIV_i is below the diversity threshold θ, then $\boldsymbol{D}_i$ is disturbed at the chaos probability p with an arbitrary distribution.

In the global mode, the diversity of the whole of the candidates is measured one time when each new generation is figured out by Eq.(1). The global diversity metric is defined as the sum of the Manhattan distance between each candidate and the center $\boldsymbol{C}=(c_1, c_2, \cdots, c_j, \cdots, c_N)$ of the whole of the candidates, shown by Eq.(3):

$$DIV_{\mathrm{G}} = \sum_{i=1}^{M} \sum_{j=1}^{N} \left| d_{ij} - c_j \right|, \tag{3}$$

where

$$c_j = \frac{1}{M} \sum_{i=1}^{M} d_{ij}. \tag{4}$$

If DIV_{G} is below the diversity threshold θ, then one random candidate is disturbed at the chaos probability p with an arbitrary distribution. Apparently L-SDRD controls the diversity of the distribution candidates in a finer-grained manner than G-SDRD does.

The introduced chaos is supposed to contribute to the resource distribution evolvement in three ways: (1) It generally maintains the spatial diversity of the candidate distributions, keeping the evolving process continuing with changing program behavior; (2) It takes account of the prevention of a desirable convergence from disturbance in a speculative manner; (3) The additional disturbance may bring more fairness for low ILP threads in obtaining extra desired resources.

Algorithm description

To summarize, Eq.(1) provides the self-organization mechanism, and the spatially triggered chaos introduces negative entropy flow. As in an evolutionary system, the distribution candidates not only refer to the historical experiences, but are also influenced by the environment. The extra chaos carries negative entropy from outer surroundings, bringing the system into a nonequilibrium state, and then the self-organization mechanism takes the system back again into a new equilibrium state. In such a fluctuation resulting from the inherent nonlinear interactions among the distribution candidates, a dissipative structure comes into being and leads to a durable resource distribution evolvement.

According to the description above, SDRD works as follows.

At the start the distribution candidates are initialized with random distributions of the ROB resources; the IQ resources are always allocated proportionately to the distribution of ROB among threads in the following steps. The IPC performance of every candidate is set to zero. Then let the system begin to apply the distribution candidates from the first one.

At the beginning of every applying period (finish applying a previous candidate $\boldsymbol{D}_{i-1}$ and begin to apply the next candidate $\boldsymbol{D}_i$), the IPC performance produced by $\boldsymbol{D}_{i-1}$ is evaluated. If it is in the L-SDRD, DIV_i is measured according to Eq.(2), and if it is less than θ, then $\boldsymbol{D}_i$ is reset to an arbitrary distribution of ROB at probability p, otherwise keep $\boldsymbol{D}_i$ unchanged. And then $\boldsymbol{D}_i$ is applied to the system.

When a generation period arrives (finish applying all of the distribution candidates in a round), $\boldsymbol{D}_l$ that yields the best IPC performance is picked out, and a new generation of the candidates is calculated according to Eq.(1). If it is in the global mode, the center and DIV_{G} of the whole of the candidates are calculated by Eq.(4) and Eq.(3), respectively; if DIV_{G} is under θ, then one random candidate is reset to an arbitrary distribution of ROB resources at probability p, otherwise keep all candidates untouched.

The operations in the applying period and the generation period are iterated until the finishing of the workloads.

Implementation

For a possible implementation of our SDRD policy, additional hardware is needed on the basis of SMT processors, as shown in Fig.1.

First, a set of resource distribution registers is needed to store the resource shares of two kinds of resources allocated to each thread. It is updated by SDRD algorithm in every applying period.

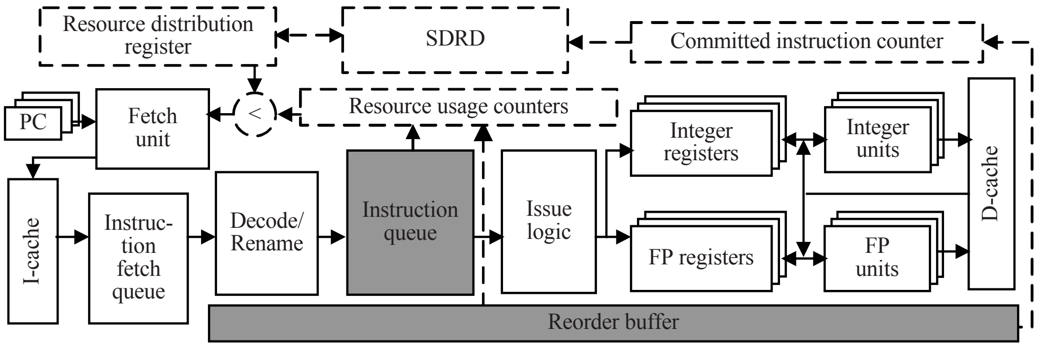

Fig.1 Diagram of SDRD on SMT model. Dashed parts are the additional hardware needed for implementation of SDRD, and shaded parts are the shared resources involved in SDRD

To calculate the in-flight IPC performance, a committed instruction counter is required. It increases automatically by 1 as soon as an instruction from any of the threads is committed from the ROB. Usage counters of ROB and IQ are also demanded by each thread for comparison with the resource shares stored in the distribution registers. For each thread, whenever it obtains a resource item, its corresponding usage counter will increase by 1 automatically; its ROB usage counter will decrease by 1 automatically when one of its instructions is committed; its IQ usage counter will decrease by 1 when one of its instructions is issued.

In addition, in the front-end a comparing logic should be added to execute the comparison between the resource usage counters and the distribution registers. The output of the comparing logic is fed to the fetch unit, which will stall fetching from a thread if any of its usage counters is beyond its corresponding allocated resource shares.

Finally, hardware that performs the SDRD algorithm is required. It applies the distribution candidates to the system by outputting the current distribution candidate to the distribution registers, and evaluates the runtime IPC performance according to the committed instruction counter. Also, it calculates the new generation of the distribution candidates via the self-organization mechanism and the spatially triggered chaos.

METHODOLOGY

To evaluate the performance of this policy, we extended M-Sim (Sharkey *et al*., 2005), which is a modified version of Simplescalar 3.0 (Burger *et al*., 1996) and supports both the SMT model and the superscalar model, to support a shared ROB model. Details of the simulator configuration are shown in Table 1.

Table 1 SMT simulator configuration

Parameters	Configuration
Bandwidth	8-wide fetch, issue and commit
Queue size	512-entry ROB, 256-entry LSQ, 160-entry IQ
Physical registers	256 integer and 256 floating-point
Fetch policy	ICOUNT2.8 (Tullsen *et al*., 1996)
Function unit and Lat. (total/issue)	6 Int Add (1/1), 3 Int Mult (3/1) / Div (20/19), 4 Mem Port (1/1), 3 FP Add (2/1), 3 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
Branch predictor	2k-entry gshare, 2k-entry 4-way set-associative BTB
L1 I-cache	64 kB, 2-way set-associative, 32-byte line, 1-cycle hit time
L1 D-cache	64 kB, 4-way set-associative, 32-byte line, 1-cycle hit time
Unified L2 cache	2 MB, 8-way set-associative, 64-byte line, 10-cycle hit time
Memory	64-bit width, 200-cycle access latency

The workloads from SPEC CPU2000 benchmarks (Henning, 2000) simulated in our experiment are composed of the precompiled Alpha binaries available at the Simplescalar website (Burger *et al*., 1996). The reference input sets of those benchmarks were used. For each thread, the simulator fast skipped to the earliest representative point among the single simulation points, the early single simulation point and the initialization end point (Sherwood *et al*., 2002;

Perelman *et al*., 2003), and then simulated the following 100M instructions. In SMT mode we stopped the simulation when any thread committed 100M instructions.

In creating the multithreaded workloads, we first classified all benchmarks into two types according to the results obtained in a single-threaded superscalar environment. One type has high ILP, and the other is memory intensive, labeled 'ILP' and 'MEM' respectively. Then we organized 30 multithreaded workloads in total: five 2-threaded and five 4-threaded for each of the ILP, MEM and MIX (mixture of ILP and MEM) types. Details of the workloads are shown in Table 2. In the following expression, let 'WL2' and 'WL4' indicate all of the 2-threaded and the 4-threaded workloads respectively, 'ILP2' and 'ILP4' indicate all of the 2-threaded and the 4-threaded workloads from the ILP type respectively, and so on for the MEM and MIX type workloads.

Table 2 Simulated multithreaded workloads

Type	2 threads	4 threads
ILP	bzip2, crafty	mesa, fma3d, eon, bzip2
	apsi, fma3d	fma3d, apsi, bzip2, crafty
	galgel, vortex	apsi, perlbmk, galgel, vortex
	wupwise, gzip	mesa, apsi, wupwise, perlbmk
	apsi, perlbmk	wupwise, gzip, mesa, gcc
MIX	art, gzip	art, mcf, wupwise, gzip
	mesa, twolf	lucas, perlbmk, vortex, bzip2
	swim, vortex	lucas, vpr, apsi, perlbmk
	lucas, perlbmk	swim, twolf, mesa, gcc
	wupwise, mcf	twolf, art, eon, gcc
MEM	art, mcf	art, swim, mcf, vpr
	equake, twolf	art, twolf, equake, mcf
	lucas, vpr	vpr, parser, applu, mgrid
	mcf, vpr	lucas, vpr, equake, twolf
	art, swim	art, mcf, swim, twolf

We used two metrics for evaluating the performance in the multithreaded workloads, the first one is the total throughput in terms of the commit IPC rate (IPC), and the second is the harmonic mean of individual thread speedups (Hmean) (Luo *et al*., 2001), which takes account of the fairness among threads, in case of favoring a thread with high IPC at the expense of restraining a thread with low IPC.

RESULTS AND DISCUSSION

Among existing methods, AROB and HILL obtain better results, so we compared SDRD with AROB, HILL, and also two widely known methods ICOUNT and STATIC. The optimal parameter settings for AROB and HILL are referred to (Sharkey *et al*., 2006; Choi and Yeung, 2006). From our experimental experiences, the number of SDRD's distribution candidates is set to 6; other settings are shown in Table 3 for L-SDRD and G-SDRD. IPC performance is used as the feedback for both HILL and SDRD. The evaluation or applying period of AROB, HILL and SDRD is set to 32k machine cycles. The settings and meaning of SDRD's important parameters are also discussed in this section.

Table 3 Parameter settings of SDRD

Mode	B_l	B_u	θ	p
L-SDRD	0.2	1.8	8	0.05
G-SDRD	0.4	1.6	12	0.20

Fig.2 compares SDRD (in both local mode and global mode) against ICOUNT, STATIC, AROB and HILL in our 30 workloads. The comparison is shown separately for the 2-threaded and 4-threaded workloads using both the IPC and Hmean metrics.

L-SDRD vs. G-SDRD

From Fig.2, we see that L-SDRD outperforms or matches G-SDRD in all but 3 of our 30 workloads using the IPC metric, and in all but 2 under the Hmean metric. While in the workload twolf-art-eon-gcc (MIX4 type) where G-SDRD has a little better IPC performance than L-SDRD, G-SDRD provides a Hmean performance obviously outshone by that of L-SDRD. This shows G-SDRD pursues the throughput performance at the expense of sacrificing fairness in this case.

To make further observation, Fig.3 shows the IPC and Hmean increments of L-SDRD and G-SDRD over ICOUNT. Obviously they obtain comparable IPC performance for all types of workloads, with average gains of 32.7% and 30.0% over ICOUNT, respectively. But using the Hmean metric, L-SDRD and G-SDRD have a distinct performance, with average gains of 24.1% and 16.1% respectively. The

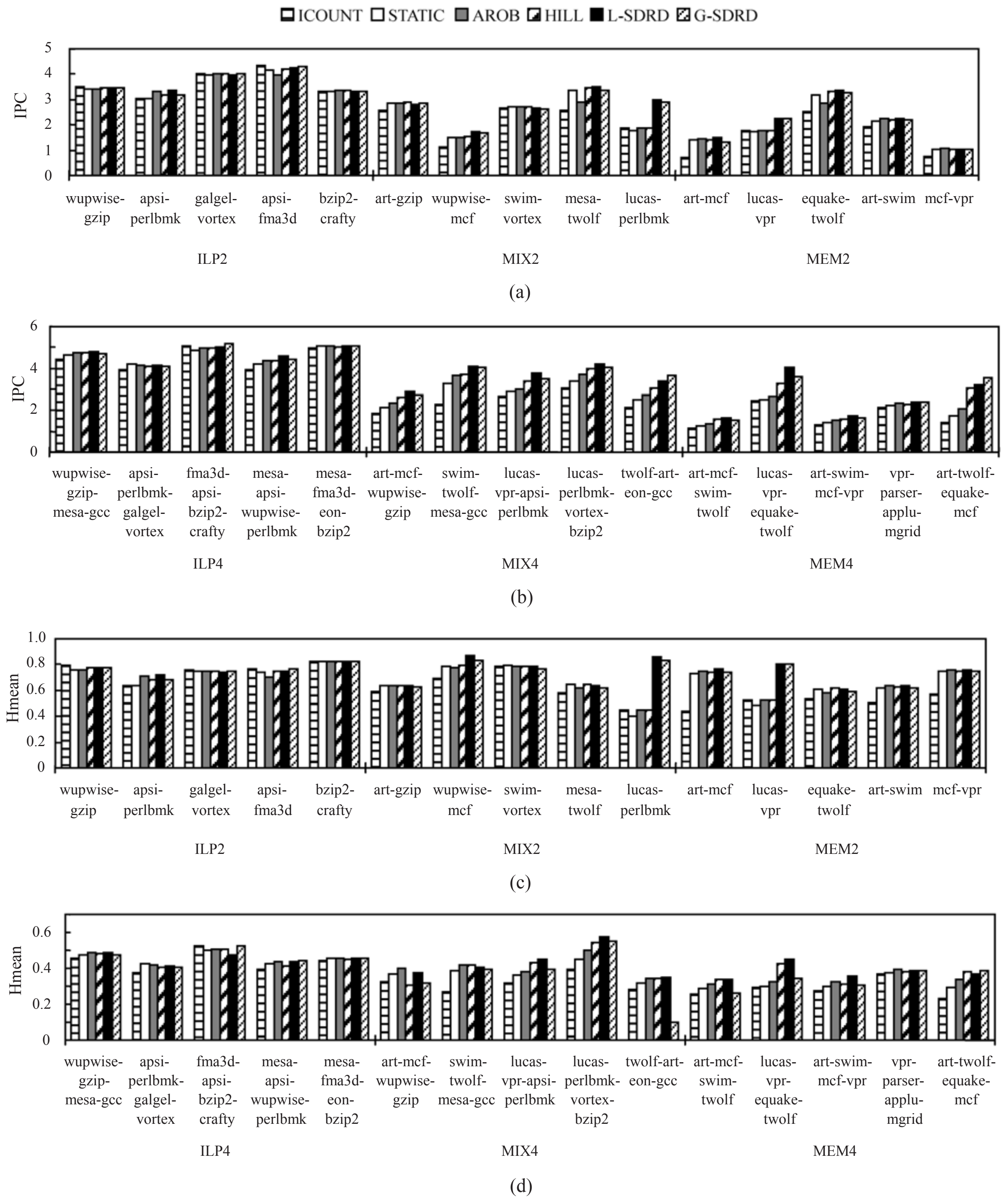

Fig.2 Per-workload performance of ICOUNT, STATIC, AROB, HILL, L-SDRD and G-SDRD. (a) 2-threaded workload, IPC; (b) 4-threaded workload, IPC; (c) 2-threaded workload, Hmean; (d) 4-threaded workload, Hmean

contrast between G-SDRD's considerable IPC gains in the WL4, MIX and MEM workloads (1.8%, 2.8% and 4.5% less than L-SDRD, respectively) and the poor Hmean gains in those type workloads (13.6%, 15.7% and 8.8% less than L-SDRD, respectively) indicates again that G-SDRD favors fast threads at the expense of sacrificing the fairness of slow threads.

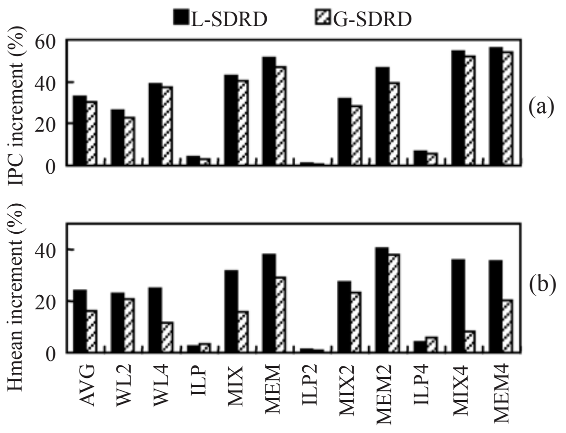

Fig.3 IPC (a) and Hmean (b) increments of L-SDRD and G-SDRD over ICOUNT

Since G-SDRD deals with diversity controlling in a coarser granularity than L-SDRD does, making the effect of the chaos weak, the IPC-directed self-organization part defined in Eq.(1) dominates the evolvement process. G-SDRD has to raise the chaos probability (shown in Table 3) to balance the two parts. However, this cannot provide a sufficient contribution to the Hmean metric.

In summary, fine-grained diversity controlling endues L-SDRD with a better balance between the IPC-directed self-organization part and the chaos part, providing good potential Hmean gains as well as good IPC gains. It is difficult for G-SDRD to makeup for the weak chaos part so as to take both IPC and Hmean performance into account.

In the following experiments and discussion, SDRD is defaulted to the local mode if not otherwise stated.

SDRD vs. others

According to Fig.2, we took count of workloads in which SDRD provides a different rank of gains when compared with ICOUNT, STATIC, AROB and HILL using both the IPC and Hmean metrics, as shown in Table 4. For the three gain ranks, 'outperform' means SDRD provides positive gains over other methods, 'close to' means SDRD provides gains within (−3.0%, 0), and 'underperform' means SDRD provides gains that do not exceed −3.0%.

We see that using the IPC metric, SDRD outperforms all the four other methods in a large majority of the 30 workloads, and obtains a performance close to them in several workloads. Using the Hmean metric, SDRD outperforms the four methods in most of the workloads, and obtains a performance close to them in some workloads. And SDRD underperforms in few or none of the workloads when compared with the four methods using both the IPC and the Hmean metrics.

For further evaluation, Fig.4 shows the statistical IPC and Hmean improvements of SDRD over the other four methods. In Fig.4a, we observe that SDRD outperforms ICOUNT, STATIC, AROB and HILL by 32.7%, 17.3%, 13.5% and 7.1% respectively on average using the IPC metric. And in Fig.4b, SDRD outperforms the four methods by 24.1%, 13.2%, 8.5% and 7.2% respectively on average using the Hmean metric. Comparing the gains in the ILP, MIX and MEM type workloads, both the IPC and the Hmean gains in the MIX and MEM type workloads contribute most to the general average of gains, while the gains in ILP workloads are not so outstanding or even sometimes slightly poor.

Table 4 Workload counts for SDRD using IPC and Hmean metrics with different gain ranks against other methods

Method	IPC			Hmean		
	Outperform	Close to	Underperform	Outperform	Close to	Underperform
ICOUNT	25	5	0	24	5	1
STATIC	25	5	0	22	7	1
AROB	23	7	0	17	11	2
HILL	25	4	1	19	10	1

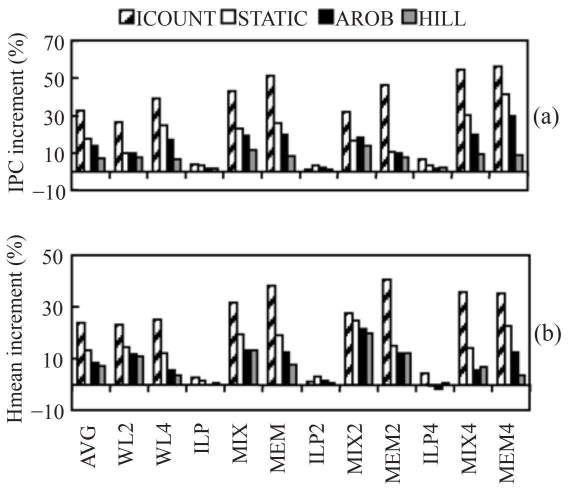

Fig.4 IPC (a) and Hmean (b) increments of SDRD over ICOUNT, STATIC, AROB and HILL

Since the programs in the ILP type workloads have a similarly high ILP level, those resource distribution policies that are biased to the high ILP thread seem to favor each of the threads to the same extent. Therefore the ILP-favored policies can produce good fairness as well as can take full advantage of the high ILP of fast threads to obtain good throughput performance. The mechanisms of ICOUNT, AROB and HILL do favor more the high ILP thread than SDRD, and so does STATIC, which is a static resource sharing version of ICOUNT. So the four methods have a desirable performance in ILP type workloads, leaving little potential gain space to SDRD.

Whereas in the MEM workloads (where a cache miss happens frequently) and MIX workloads (which are composed of ILP and MEM workloads), threads have distinct ILP levels and exhibit complicated behavior when they compete for resources. The ILP-favored policies are apt to favor high ILP threads by allocating more resources to them, with little consideration for the resource requirements of low ILP threads. This may result in performance degradation for both the whole throughput metric and the fairness metric, while SDRD keeps a good balance between fast threads and slow threads via its well-geared self-organization part and chaos part, providing outstanding gains in a more competitive resource environment.

Exploitation scope

Fig.5 illustrates the impact of different (B_l, B_u) pairs on IPC and Hmean performance. To make it clear, the improvements of SDRD, with various (B_l, B_u) settings, over ICOUNT are presented. Other parameters of SDRD are kept unchanged. On the abscissa axis, B_l and B_u move away from 1 in decreasing and ascending direction respectively by a step of 0.2 from left to right. This means the scope of exploitation of the historical experience becomes larger and larger.

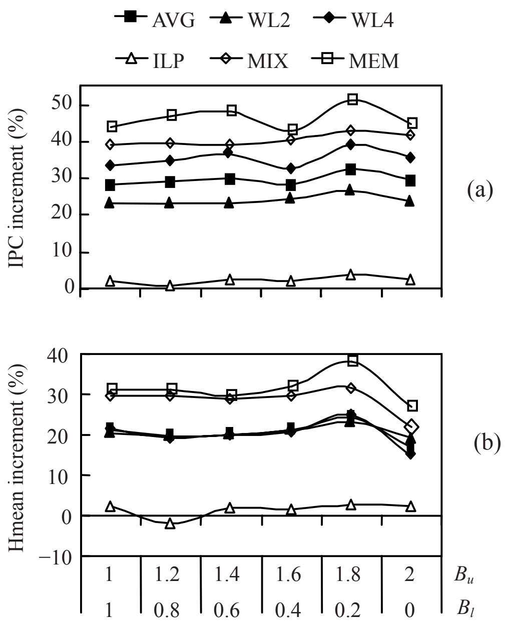

Fig.5 Impact of exploitation scope on IPC (a) and Hmean (b) performance

Fig.5 shows that the IPC gains are insensitive to the variation of the exploitation scope for more workload types than the Hmean gains. This can be explained by Eq.(1). No matter how the exploitation scope changes, the distribution candidates are always organized around the best historical distribution. This means the IPC performance is influenced slightly by the exploitation scope. While Hmean is the harmonic mean of the individual thread speedups (Luo *et al.*, 2001), different IPC combinations from the threads may keep the total throughput IPC steady, but can lead to various changes in the Hmean metric.

In Fig.5 we see the low level IPC and Hmean gains in the ILP workloads again. The reason is given in the previous discussion on ILP-favored policy (in the subsection “SDRD vs. others”). ICOUNT leaves little potential space in the ILP workloads for SDRD to improve performance.

As the exploitation scope increases, both the IPC and Hmean gains in all workload types touch their peaks at (0.2, 1.8), and then keep falling down. When (B_l, B_u) is set to (0, 2), which means the exploitation scope is very large, the distribution candidates are organized at some small remove from the historical experience from a statistical point of view. This degrades the significance of the historical experience, making the self-organization part of SDRD weak. So less IPC and Hmean gains are seen.

According to the discussion above we can find an optimal (B_l, B_u) setting for SDRD at around (0.2, 1.8).

Chaos probability

In this subsection, the role of chaos probability *p* will be revealed. To make the illustration clear, the IPC and Hmean improvements of SDRD with different chaos probabilities compared with ICOUNT are presented in Fig.6, while other parameters of SDRD keep their original settings.

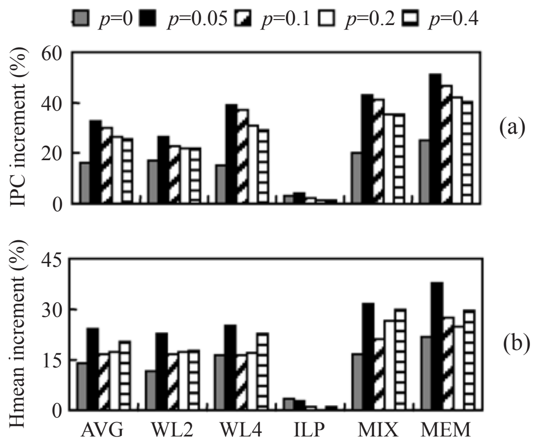

Fig.6 Impact of chaos probability on IPC (a) and Hmean (b) performance

Obviously p=0.05 and p=0.1 perform alike in all types of workloads using the IPC metric, with outstanding average gains of 32.7% and 30%, respectively. But using the Hmean metric, performance at p=0.05 surpasses that at p=0.1, with an average gain of 24.1% for p=0.05 and 16.6% for p=0.1.

When *p* increases from 0.1 to 0.2 and 0.4, we see a general degradation of IPC gains and a general upgrade of Hmean gains in all types of workloads. As the chaos probability becomes bigger, the distribution candidates are more likely to be disturbed as soon as a low diversity is detected, and the desirable distribution convergence brought by the self-organization part is more likely to be ruined. This makes the IPC-targeted historical experience less productive, resulting in less IPC performance, whereas the likelihood of more disturbance provides more of a chance to meet the resource requirements of those low ILP threads, leading to a fairer performance.

When *p* is set to 0, SDRD provides both low IPC gains and low Hmean gains. This is because when the distribution candidates converge in an uncomfortable distribution space, there is no chaos introduced to the distribution candidates to control the diversity. It is difficult to sustain the evolvement to capture a better optimal distribution space.

By comparison, chaos probability of around 0.05 shows outstanding IPC and Hmean performance. It endues SDRD with a good balance between disturbing uncomfortable convergence and preserving reasonable convergence in the evolvement process.

Diversity threshold

Fig.7 illustrates the impact of different diversity threshold θ on IPC and Hmean performance. To make it clear, the improvements of SDRD with various θ settings over ICOUNT are presented, while other parameters of SDRD are kept unchanged.

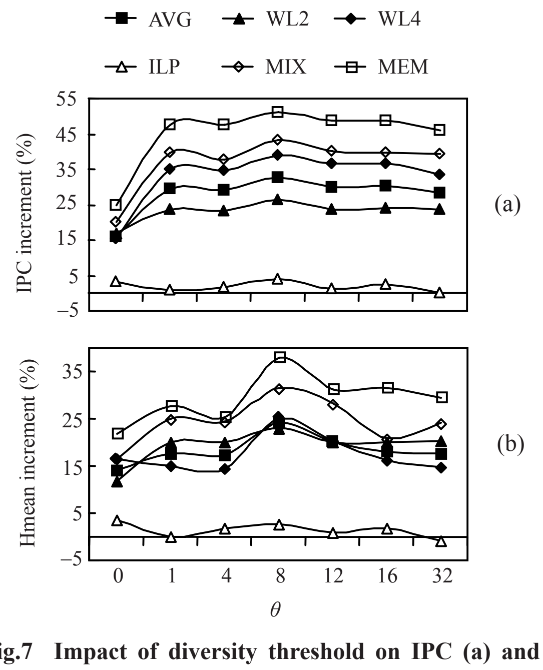

Fig.7 Impact of diversity threshold on IPC (a) and Hmean (b) performance

From Fig.7, we see that the diversity threshold has less influence on the IPC gains than on the Hmean gains, especially in the WL4, MEM and MIX type workloads where it produces only a small change in IPC gains but obviously a fluctuation in Hmean gains. This is because in whatever manner the diversity threshold changes, the distribution candidates always self-adjust around the historical distribution with the best IPC performance in the evolvement process, thus making IPC performance slightly influenced by θ. While with various diversity thresholds, SDRD maintains the diversity of the distribution candidates at a different level, which directly concerns fairness among the threads, thus making more fluctuations using the Hmean metric.

When θ=0, SDRD has no chance of introducing extra chaos, since the diversity of two distribution candidates will never be below zero according to Eq.(2). This situation is the same as that of p=0, which is discussed in the subsection "Chaos probability". As the diversity threshold increases, the IPC gains climb quickly to a high level, and then plateau off. The IPC gains appear a little better at θ=8. The Hmean gains climb up to their peak at θ=8 and then show an obvious decline. This indicates that a diversity threshold around 8 is optimal for SDRD. A big threshold will damage reasonable convergence. A small diversity threshold only allows SDRD to introduce chaos to very similar distribution candidates; it is not enough to maintain the diversity of the distribution candidates and supply persistent resource distribution evolvement. Therefore neither a big diversity threshold nor a small one will make SDRD produce a pleasing performance.

CONCLUSION AND FUTURE WORK

This work presents a spatially triggered dissipative policy for resource distribution in SMT processors, which is composed of the self-organization mechanism that is driven by runtime IPC performance, and the introduction of extra chaos that is triggered to control the spatial diversity of the resource distribution candidates, to supply sustaining resource distribution optimization for complicated program behavior. The simulation results show that:

(1) SDRD with a fine-grained diversity controlling (L-SDRD) is more effective than that with a coarse one (G-SDRD), with 2.7% more IPC gains and 8% more Hmean gains than G-SDRD on average over ICOUNT.

(2) SDRD benefits much from its two well-coordinated parts, obtaining potential Hmean gains as well as good IPC gains. Using the IPC metric, SDRD outperforms ICOUNT, STATIC, AROB and HILL by 32.7%, 17.3%, 13.5% and 7.1% respectively on average, and using the Hmean metric, SDRD outperforms the four methods by 24.1%, 13.2%, 8.5% and 7.2% respectively on average. And SDRD provides more IPC and Hmean gains in a complicated SMT environment, such as the MIX and MEM type workloads, than in the ILP type workloads.

Experiments and discussion of SDRD's important parameters, including the exploitation scope, the chaos probability and the diversity threshold, reveal their meaning for SDRD and their impact on both the throughput and fairness performance.

Further work will attempt some research on other diversity controlling mechanisms, in which the extra chaos is triggered by some supervised information from the pipeline, and also where the chaos can consist of a planned resource distribution rather than an arbitrary one. Adaptive adjustment of several important parameters is also meaningful to make SDRD generally smarter for specific program phases that emerge at runtime.

References

Burger, D., Austin, T.M., Bennett, S., 1996. Evaluating Future Microprocessors: The Simplescalar Tool Set. Technical Report 1308. Computer Science Department, University of Wisconsin-Madison, Madison. Http://www.cs.wisc.edu/techreports/viewreport.php?report=1308

Cazorla, F., Ramirez, A., Valero, M., Fernández, E., 2004a. Dcache Warn: An I-fetch Policy to Increase SMT Efficiency. Proc. 18th Int. Parallel and Distributed Processing Symp., Santa Fe, NM, p.74-83. [doi:10.1109/IPDPS.2004.1303005]

Cazorla, F., Ramirez, A., Valero, M., Fernández, E., 2004b. Dynamically Controlled Resource Allocation in SMT Processors. Proc. 37th Int. Symp. on Microarchitecture, Portland, OR, p.171-182. [doi:10.1109/MICRO.2004.17]

Choi, S., Yeung, D., 2006. Learning-based SMT Processor Resource Distribution via Hill-climbing. Proc. 33rd Annual Int. Symp. on Computer Architecture, Boston, MA, p.239-251. [doi:10.1109/ISCA.2006.25]

El-Moursy, A., Albonesi, D.H., 2003. Front-end Policies for Improved Issue Efficiency in SMT Processors. Proc. 9th Int. Conf. on High Performance Computer Architecture, Anaheim, CA, p.31-42. [doi:10.1109/HPCA.2003.1183522]

Henning, J., 2000. SPEC CPU2000: measuring CPU performance in the new millennium. *IEEE Computer*, **33**(7):28-35.

Hirata, H., Kimura, K., Nagamine, S., Mochizuki, Y., Nishimura, A., Nakase, Y., Nishizawa, T., 1992. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. Proc. 19th Annual Int. Symp. on Computer Architecture, Gold Coast, Australia, p.136-145. [doi:10.1109/ISCA.1992.753311]

Latorre, F., González, J., González, A., 2004. Back-end Assignment Schemes for Clustered Multithreaded Processors. Proc. 18th Annual Int. Conf. on Supercomputing, Malo, France, p.316-325. [doi:10.1145/1006209.1006254]

Luo, K., Gummaraju, J., Franklin, M., 2001. Balancing throughput and fairness in SMT processors. Proc. Int. Symp. on Performance Analysis of Systems and Software, Tucson, AZ, p.164-171.

Marr, D.T., Binns, F., Hill, D.L., Hinton, G., Koufaty, D.A., Miller, J.A., Upton, M., 2002. Hyper-threading technology architecture and microarchitecture. *Intel Technol. J.*, **6**(1):4-15.

Nicolis, G., Prigogine, I., 1977. Self-organization in Nonequilibrium Systems: From Dissipative Structures to Order through Fluctuations. John Wiley, New York. p.55-60, 429-474.

Perelman, E., Hamerly, G., Calder, B., 2003. Picking Statistically Valid and Early Simulation Points. Proc. 12th Int. Conf. on Parallel Architectures and Compilation Techniques, New Orleans, LA, p.244-255. [doi:10.1109/PACT.2003.1238020]

Prigogine, I., 1967. Introduction to Thermodynamics of Irreversible Processes (3rd Ed.). Interscience Publisher, New York, p.124-134.

Prigogine, I., 1976. Order through Fluctuation: Self-organization and Social System. *In*: Jantsch, E., Waddington, C. (Eds.), Evolution and Consciousness: Human Systems in Transition. Addison-Wesley, London, p.93-134.

Raasch, S.E., Reinhardt, S.K., 2003. The Impact of Resource Partitioning on SMT Processors. Proc. 12th Int. Conf. on Parallel Architectures and Compilation Techniques, New Orleans, LA, p.15-25. [doi:10.1109/PACT.2003.1237998]

Sharkey, J., Ponomarev, D., Ghose, K., 2005. M-Sim: A Flexible, Multithreaded Architectural Simulation Environment. Technical Report CS-TR-05-DP01. Department of Computer Science, State University of New York at Binghamton. Http://www.cs.binghamton.edu/~jsharke/m-sim

Sharkey, J., Balkan, D., Ponomarev, D., 2006. Adaptive Reorder Buffers for SMT Processors. Proc. 15th Int. Conf. on Parallel Architectures and Compilation Techniques, Seattle, WA, p.244-253. [doi:10.1145/1152154.1152192]

Sherwood, T., Perelman, E., Hamerly, G., Calder, B., 2002. Automatically Characterizing Large Scale Program Behavior. Proc. 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, p.45-57. [doi:10.1145/605397.605403]

Tullsen, D.M., Brown, J.A., 2001. Handling Long-latency Loads in a Simultaneous Multithreading Processor. Proc. 34th Annual Int. Symp. on Microarchitecture, Austin, TX, p.318-327.

Tullsen, D.M., Eggers, S.J., Levy, H.M., 1995. Simultaneous Multithreading: Maximizing On-chip Parallelism. Proc. 22nd Annual Int. Symp. on Computer Architecture, Santa Margherita Ligure, Italy, p.392-403.

Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., 1996. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. Proc. 23th Annual Int. Symp. on Computer Architecture, Philadelphia, PA, p.191-202.