# A Swarm-Inspired Resource Distribution for SMT Processors

Hongzhou Chen, Lingdi Ping, Xuezeng Pan, Kuijun Lu
College of Computer Science, Zhejiang University
Hangzhou, China
honjoychan@gmail.com

Xiaoning Jiang
Sunyard System Engineering Co. Ltd.
Hangzhou, China
jxn@sunyard.com

## ABSTRACT
The performance in Simultaneous Multi-Threading (SMT) processors is mainly determined by the distribution of the common resources among the threads. However, the threads exhibit dynamically complicated behavior while they compete for resources at runtime. It is a challenge to meet the changing resource requirements of the threads. This work proposes a Swarm-inspired Resource Distribution (SRD) policy to address the dynamic optimization problem of resource distribution for SMT processors, which uses the runtime performance to guide the generating of trial distributions. A computational model is established by adaptation of swarm intelligence to direct the social exploitation and self exploration activities of the trial distributions in the dynamic optimization environment. Results from simulation show that, benefiting from the good cooperation between SRD's social exploitation on historical experience and self exploration of new solutions, SRD obtains satisfying improvements of both throughput and fairness performance, especially in complicated SMT environment.

## Keywords
Simultaneous multithreading, resource distribution, bio-inspired model, optimization method, swarm intelligence.

## 1. INTRODUCTION
Simultaneous Multi-Threading (SMT) is one paradigm of processor design exploiting Thread Level Parallelism (TLP). SMT processors can run instructions from different threads concurrently, reducing the wastage of instruction slots in the pipeline due to insufficient Instruction Level Parallelism (ILP); the combined ILP from all threads therefore provides high performance gains [1, 2]. In an SMT model, each thread owns private program counter, rename table, load/store queues, branch predictor, etc., while shares some critical data-path resources such as the physical registers, the issue queues, and the execution units. However, the threads change their resource requirement as their program behavior changes, and they compete for the common resources more than share them. So how reasonably the common resources are distributed among the threads mainly determines the throughput and fairness performance of SMT processors.

Most of the existing resource distribution methods, such as ICOUNT [2], STALL [3], Data Gating (DG) [4], rely mainly on the fetch policy at the front-end, which select threads to fetch according to some monitored information in the pipeline such as the cache miss count or the occupancy of issue queues. The common resources are implicitly distributed among the threads by controlling the number of instructions that flow into the pipeline. Unfortunately, the fetching selection that is made from this limited information is somewhat uncertain. It is difficult for this implicit way to reflect the real resource requirement of threads.

As another type of resource distribution methods, explicit policies allocate resource to the threads in a direct way. For example, the STATIC [5, 6] method partitions common resources among all threads statically with each thread monopolizing equal shares of resources, pitifully suffering resource wastage because some thread may not take full advantage of the allocated resources. Other explicit policies makes dynamic allocation decision based on information such as the limited pipeline information monitored in Dynamically Controlled Resource Allocation (DCRA) [7] and the program phase information in Adaptive Reorder Buffers (AROB) [8]. These dynamic distribution policies—though generally attaining good effect—are guided by the supervised indirect information instead of the real performance, and unaware of the impact of their distribution solutions on the real performance, it is not easy to figure out the optimal distribution solution efficiently. In addition, their more focus on alleviating specific bottlenecks in SMT processors decreases their generality for complicated SMT environments with changing program behavior.

Taking advantage of swarm intelligence, this work proposes a Swarm-inspired Resource Distribution (SRD) policy to solve the nonlinear optimization problem of resource distribution for SMT processors. In SRD, there is a resource distribution colony consisting of several trial resource distributions on behalf of different resource distribution solutions. The runtime performance produced by applying each of the trial distributions for a period helps to judge how well the trial distributions are, and it is used to guide the generating of new colony of trial distributions. A computational model for generating the colony is adapted from swarm intelligence to direct the social exploitation and self exploration activities of the trial distributions in the dynamic optimization environment. Results from simulation show that


*Bionetics'08*, November 25-28, 2008, Hyogo, Japan.

SRD obtains potential increments of fairness performance as well as good increments of throughput performance by just using the real-time throughput performance as the feedback information. This can be attributed to the good cooperation between SRD's social exploitation part and self exploration part. Some important parameters of SRD are discussed also.

In the remaining sections of this paper, we introduce our SRD policy for SMT processors in Section 2 and present the experimental methodology in Section 3. Then after the simulation results and some related discussions in Section 4, we conclude this paper in Section 5.

# 2. SWARM-INSPIRED RESOURCE DISTRIBUTION

## 2.1 Adoption of Swarm Intelligence

The main idea of our policy is derived from Particle Swarm Optimization (PSO) [9, 10], which is one of the swarm intelligence methods and developed from researches on the behavior of bird colony in prey. In PSO, each individual particle is treated as a point in a *D*-dimension space, representing a potential solution to the problem. The *i*th particle is represented by $\boldsymbol{X}_i = (x_{il}, x_{i2}, \ldots, x_{iD})$, its flying velocity is represented by $\boldsymbol{V}_i = (v_{il}, v_{i2}, \ldots, v_{iD})$, and its best historical position is recorded as $\boldsymbol{P}_i = (p_{i1}, p_{i2}, \ldots, p_{iD})$. The index of the best particle among the whole colony is represented by symbol *g*. At the end of every generation *t*, particles evolve their velocity and position into the next generation *t+1* according to the following iterative velocity and position equations [9]:

$$v_{id}(t+1) = v_{id}(t) + \mathrm{c}_1 \times r_1 \times (p_{id}(t) - x_{id}(t)) + \mathrm{c}_2 \times r_2 \times (p_{gd}(t) - x_{id}(t)) \quad (1)$$

$$x_{id}(t+1) = x_{id}(t) + v_{id}(t+1) \quad (2)$$

where $\mathrm{c}_1$, $\mathrm{c}_2$ are two positive constants, $r_1$ and $r_2$ are two random numbers in range [0,1]. The first part of Eq.(1) represents the former velocity of particle. The second part of Eq.(1) is the "cognition" part, representing particle's self exploration of the solution space. The third part is the "social" part, representing the information sharing (exploitation on historical experience) or collaboration among particles [10]. Compared with other algorithms in the evolutionary computation field, PSO can both avoid high gene flow via implicit weak selection mechanism in early iterations, and converge quickly at reasonable solution.

When PSO is introduced into the optimization problem of resource distribution, each particle represents a trial resource distribution among the threads. Each trial distribution is applied to the threads for a period of time (named Allocation Period, AP). At the end of every AP, the processor performance in that AP is evaluated. When all of the trial distributions are applied one after the other, a new generation of colony is figured out according to the velocity and position equations. And then a new iteration begins. The period between two adjacent generations is called a generation period.

In the SMT environment, however, threads' resource requirement changes with their program behavior. All the three parts in Eq.(1) need adaptation for the dynamic optimization of resource distribution for SMT processors.

1. In dynamic SMT environment, the optimal resource distribution often changes with program behavior. According to the program locality principle, the local optimal distribution plays a more significant role than the global one does in generating a new distribution colony. Therefore, the best local historical position $\boldsymbol{P}_l$, instead of the best global historical position $\boldsymbol{P}_g$, should be used to direct the social exploitation part in Eq.(1).

2. For the self part in Eq.(1), the individual global experience $\boldsymbol{P}_i$ is of less significance in the same sense as $\boldsymbol{P}_g$ does. Instead, the trial distributions are designed to take random explorations in the resource distribution space as their self activities (see Section 2.2).

3. The first part in Eq.(1) originally means particle's memory of former velocity, representing the global exploration ability. For optimization problem that features a large solution space, the velocity part can keep large exploration scope in early iteration stages, preventing individual particle from being excessively influenced by local optimal solutions. While in SMT environment, the limited quantity of common resources provides a small discrete solution space of resource distribution. It is not difficult for particles to cover the entire resource distribution space even without the velocity part. So the first part has no more obvious meaning and can be omitted.

## 2.2 SRD Policy

Our policy involves the distribution of three kinds of important common resources: the reorder buffers (ROB), the physical register files (RF) and the issue queues (IQ). For simplicity, only the distribution of ROB among threads is taken into account, and the distribution of RF and IQ are just kept in proportion to that of ROB in every allocation period, since a given thread often utilizes roughly similar fractions of the three common resources. Moreover, we only care about the distribution of the integer register files because it can indirectly control the distribution of the float point register files. The trial distribution that produces the best IPC (Instructions Per Cycle, a throughput metric) performance in a previous generation period is chosen to direct the resource distribution optimization in the succeeding generation period.

Let *Colony_size* denote the colony scale, *N* denote the number of threads, the *i*th trial distribution $\boldsymbol{X}_i = (x_{i1}, x_{i2}, \cdots, x_{id}, \cdots, x_{iN})$ represent the distribution of ROB among threads from 1 to *N*. The best local historical distribution is represented by $\boldsymbol{P}_l = (p_{l1}, p_{l2}, \cdots, p_{ld}, \cdots, p_{lN})$, the self exploration variation $\boldsymbol{\Delta_i} = (\delta_{i1}, \delta_{i2}, \cdots, \delta_{id}, \cdots, \delta_{iN})$ represents the variation of ROB distribution that is made by the self exploration activity of the *i*th trial distribution, it is generated by borrowing *Delta* (named exploration step) entries of ROB for a random thread *m* from each of the rest *N-1* threads, defined as Eq.(3):

$$\delta_{id} = \begin{cases} -Delta, & if\,(d \neq m) \\ (N-1)Delta, & if\,(d = m) \end{cases} \quad (3)$$

According to the main idea described above, the computational model for SRD to update trial distributions can be simply established as Eq.(4):

$$x_{id}(t+1) = x_{id}(t) + self \times r_1 \times \delta_{id} + social \times r_2 \times (p_{ld}(t) - x_{id}(t)) \quad (4)$$

where *self* is self exploration factor, indicating the capability of exploration of new distribution solutions; *social* is social exploitation factor, indicating the capability of exploitation on historical experience. $r_1$ and $r_2$ are random numbers in range [0,1]. The introduced randomicity is supposed to avoid excessive influence by the historical experience, and also it brings chance to low ILP threads to obtain extra resources for fairness.

Finally, SRD works as follows. In the initialization stage, the trial distributions are set with even distribution of ROB resources. The IPC performances of all trial distributions are set to zero. Let the system begin to apply the trial distributions from the first trial distribution $\boldsymbol{X}_0$. At the outset of every allocation period (finish applying the previous trial distribution $\boldsymbol{X}_{i-1}$ and begin to apply the next trial distribution $\boldsymbol{X}_i$), the IPC performance produced by $\boldsymbol{X}_{i-1}$ is evaluated. If $\boldsymbol{X}_{i-1}$ is the last trial distribution $\boldsymbol{X}_{Colony\_size-1}$ (namely, a generation period also comes at the outset of the current allocation period), then $\boldsymbol{P}_1$ is updated with the trial distribution that yields the best IPC performance, and the self exploration variation is calculated for every trial distribution by Eq.(3), and then a new generation of trial distributions are figured out according to Eq.(4). If $\boldsymbol{X}_{i-1}$ is not $\boldsymbol{X}_{Colony\_size-1}$, then apply $\boldsymbol{X}_i$ to the threads; else let the system start another applying round from the first trial distribution $\boldsymbol{X}_0$. The RF and the IQ resources are always allocated proportionately to the ROB distribution in every allocation period.

## 2.3 Implementation

For a possible implementation of the SRD policy, additional hardware is needed by SMT processors for supporting SRD.

First, a set of allocation registers is needed to store the distributions of the three kinds of common resources among threads, which are updated by SRD algorithm in every allocation period.

Secondly, a committed instruction counter is demanded to evaluate the runtime IPC performance. When an allocation period starts, the committed instruction counter is reset to zero, and then increases automatically by 1 as soon as any instruction is committed from the ROB. Occupancy counters of the ROB, RF and IQ resource are also demanded by each thread for comparing with the resource quotas stored in the allocation registers. Whenever an instruction obtains a resource item, its corresponding occupancy counter increases by 1 automatically. The ROB and RF occupancy counters will decrease by 1 automatically when an instruction is committed. The IQ occupancy counter will decrease by 1 automatically when an instruction is issued.

And in the front end, a comparing logic should be implemented to perform the comparison between the occupancy counters and the allocation registers. The output of the comparing logic is fed to the fetch logic, which will stall fetching from a thread if any of its occupancy counters exceeds its corresponding allocation register.

Finally, hardware that implements SRD algorithm is required. At each allocation period, it evaluates the runtime IPC of the prior allocation period via dividing the committed instruction counter by AP, and applies the current trial distribution to threads by writing it into the allocation registers. At each generation period, it generates new trial distributions.

In consideration of efficiency, hardware that performs SRD algorithm may work asynchronously with the critical pipeline in the new SRD-supported SMT processor, there is no block or wait operation between them except two kinds of mutex between: the read operation on the allocation registers by the comparing logic and the write operation on them by the SRD hardware; the write operation on the committed instruction counter by the instruction committing logic and the read operation on them by the SRD hardware. Since the two mutexes happen only once for every allocation period, the SRD hardware will entail a little expense on SMT processor.

# 3. SIMULATION METHOD

To evaluate the performance of our policy, we used M-Sim [11], which is a modified version of Simplescalar 3.0 [12] and supports both the SMT model and the superscalar model. Details in simulator configuration are shown in Table 1.

**Table 1. SMT simulator configuration**

| Parameters | Configuration |
|---|---|
| Bandwidth | 8-wide fetch, issue and commit |
| Queue size | 256 entry ROB, 128 entry LSQ, 160 entry IQ |
| Phys. registers | 160 integer and 160 float-point |
| Fetch policy | ICOUNT |
| 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 | 64KB, 2-way set-associative, 32 byte line, 1 cycle hit time |
| L1 D-cache | 64KB, 4-way set-associative, 32 byte line, 1 cycle hit time |
| Unified L2 cache | 2MB, 8-way set-associative, 64 byte line, 10 cycles hit time |
| Memory | 64-bit width, 200 cycle access latency |

The workloads of SPEC CPU2000 benchmarks [13] simulated in our experiment were composed of the precompiled Alpha binaries available at the Simplescalar website [12]. The reference input sets of those benchmarks were used. For each thread, the simulator fast skipped to the earliest simulation point among the single simulation point, the early single simulation point and the initialization end point [14, 15], and then simulated the sub-sequent 100M instructions. In the SMT mode, we stopped the simulation as soon as 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 features high instruction level parallelism, and the other is of memory-intensive, labeled “ILP” and “MEM” respectively. Then we totally organized 36 multithreaded workloads: six 2-threaded and six 4-threaded for each of the ILP, MEM and MIX (mixture of ILP and MEM) type (see Table 2 for details). In the following expression, “WL2” and “WL4” denote all the 2-threaded and 4-threaded workloads respectively, “ILP2” and “ILP4” denote the 2-threaded

and 4-threaded workloads of ILP type, and so on for the MEM and MIX type workloads.

**Table 2. Simulated multithreaded workloads**

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

We used two metrics for performance evaluation in these multithreaded workloads, the first one is the total throughput in terms of commit IPC rate (IPC), and the second one is the harmonic mean of individual speedups (Hmean) [16], taking account of fairness among threads in case of favoring a thread with high IPC at the expense of restraining a thread with low IPC.

# 4. RESULTS AND DISCUSSIONS

We compared SRD with three other methods: AROB (a better one among existing methods), the common method ICOUNT and STATIC. The optimal parameter settings for AROB are referred to in work [8]. From our experimental experience, SRD's *Colony_size* is set to 6; both *social* and *self* are set to 1.6. The decision or allocation period of AROB and SRD are set to 32K machine cycles. Also, the settings and meaning of SRD's important parameters are discussed in this section.

## 4.1 IPC and Hmean Performance

Figure 1 shows the IPC and Hmean improvements of SRD over ICOUNT, STATIC and AROB. It is observed that, under the IPC metric (Figure 1(a)), SRD outperforms ICOUNT, STATIC and AROB by 29.1%, 24.1% and 16% on average respectively, and under the Hmean metric (Figure 1(b)), SRD outperforms the three methods by 18.2%, 14.1% and 9.6% on average respectively. Comparing the gains in the ILP, MIX, MEM type workloads, both the IPC and the Hmean gains in the MIX and MEM type workloads contribute most to the whole average gains, whereas the gains in ILP workloads are not so outstanding or sometimes slightly poor. And regarding the gains in the WL2 and WL4 workloads, SRD provides more IPC and Hmean gains in the WL4 workloads than in the WL2 workloads.

Since the programs in the ILP type workloads own similarly high ILP level, those resource distribution policies that are biased to 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 take full advantage of the high ILP of fast threads to obtain considerable throughput performance. The mechanism of ICOUNT and AROB does more favor to high ILP thread, and so does STATIC—a static resource distribution version of ICOUNT. Thus the three methods have desirable performance in ILP type workloads, leaving little potential gain space to SRD.

However, in the MEM workloads where cache miss happens frequently and the MIX workloads that are composed of ILP and MEM workloads, threads have distinct ILP levels and exhibit complicated behavior when competing for resource. The ILP-favored policies are prone to favor high ILP threads with more resources, with little consideration of the resource requirement of low ILP threads. This may result in degradation of both the whole throughput and the fairness performance. SRD however keeps good balance among fast threads and slow threads through its well-cooperated social exploitation part and self exploration part, providing outstanding gains in more complicated SMT environment.

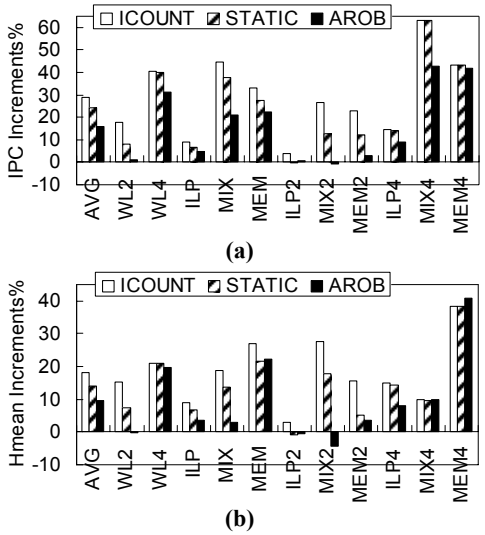
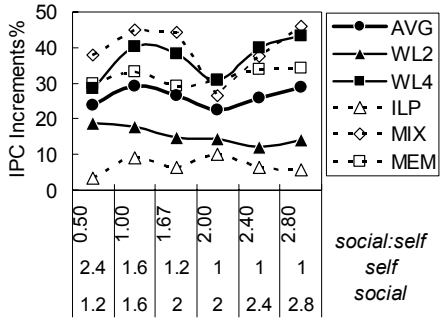



**Figure 1. Increments of SRD over ICOUNT, STATIC and AROB.**

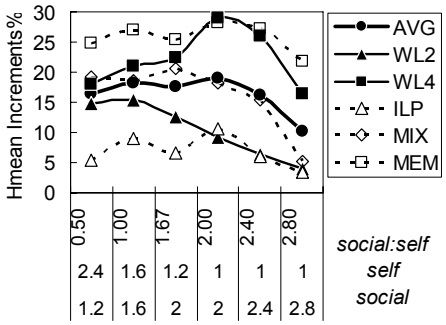
As to the contrast between the gains in WL2 and WL4 workloads, the common resources in 2-threaded environment are comparatively sufficient. The ILP-favored policies help to exert the potential performance of the fast thread and do not excessively restrain the slow one, consequently producing good IPC and Hmean performance and leaving little gain space to SRD. While in the WL4 workloads where resources are comparatively scarce, such favor to high ILP threads will ignore the resource requirement of low ILP threads and result in a worse performance. While SRD keeps a good balance among fast threads and slow threads, obtaining more gains in environment with more competition for the common resources.

## 4.2 Factor of Social Exploitation and Self Exploration

Figure 2 illustrates the impact of different (*social*, *self*) pairs on performance. To make it clear, performance increments of SRD with various (*social*, *self*) over ICOUNT are presented. On the parameter setting axis, the ratio of *social* to *self*, *self* and *social* are listed from top to bottom, and from left to right the value of *social*:*self* is kept ascending with generally descending *self* and ascending *social*.




**(a)**




**(b)**

**Figure 2. Impact of (*social*, *self*) pair on performance.**

Figure 2 demonstrates that (*social, self*) has less influence on the IPC performance than on the Hmean performance. Since the IPC metric is simply the sum of IPCs of all the threads, while Hmean is the harmonic mean of the individual thread speedups [16]. As SRD's capability of social exploitation and self exploration changes along with the (*social, self*) pair, each thread contributes different individual IPC performance. Different combination of IPCs from the individual threads often keeps the total IPC steady, while it is prone to result in varieties of Hmean metric.

As *social* grows (means the trial distributions are more attracted by historical experiences) and *self* shrinks (means the exploration scope of trial distributions becomes more limited), several representative settings are displayed in Figure 2.

At (1.2, 2.4), SRD's exploration capability surpasses its exploitation capability, the trial distributions often search blindly across the whole resource distribution space and make the historical experience $\boldsymbol{P}_l$ almost out of use. At (1.6, 1.6), the IPC and Hmean gains in most of workloads reach their crest values, indicating a balanced capability of social exploitation and self exploration. At (2, 1), most of IPC gains touch their bottom while most Hmean gains attain their peak values. As viewed from statistics, when *social* is set to 2, $social \times r_2$ in Eq.(4) is about 1 in statistical sense, then Eq.(4) transforms to:

$$x_{id}(t+1) \approx p_{ld}(t) + self \times r_1 \times \delta_{id} \tag{5}$$

Eq.(5) implies that the trial distributions do their exploration just away from the historical IPC-best experience $\boldsymbol{P}_l$, there is little chance for SRD to approach to the IPC-best distribution, but many chances for those low ILP threads to obtain extra resources. Therefore, SRD exhibits degraded IPC performance but excellent Hmean performance at (2, 1). After (2, 1), SRD's exploitation capability exceeds its exploration capability, the trial distributions are excessively attracted by historical experience while do tiny exploration. SRD inclines to pursuing IPC performance at the expense of fairness.

According to the above discussion, we can find optimal (*social*, *self*) settings for SRD around (1.6, 1.6).

## 4.3 Exploration Step

The exploration step *Delta* represents the searching scope of the trial distributions, which has a direct impact on SRD's self exploration part. This subsection analyses several illustrative and evident optimization procedures of SRD with different *Delta* (4, 8 and 12) in several workloads. To make it clear, average IPC of every 10 allocation periods is presented.

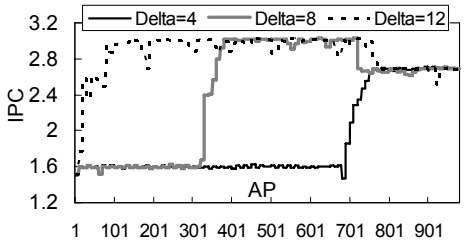



**Figure 3. Impact of *Delta* on optimization procedure in workload {lucas, perlbmk}.**

Figure 3 implies that the three optimization procedures come into different resource distribution spaces. *Delta*=12 is the earliest one to find the better optimal distribution space, but there are frequent drops on its IPC line. These indicate that a bigger exploration step provides a better searching ability, but it is not easy to make the trial distributions converge at a steady state. *Delta*=4 does not escape from the inferior optimal distribution space until terminal periods, suggesting that a small exploration step provides a weak searching ability and makes the trial distributions apt to stagnate in inferior optimal distribution space.

In Figure 4, the periodically emerging of IPC peaks reflects the periodically changing of program phase and optimal resource distribution. Performance in this kind of workloads is determined by how quickly the transient optimal distributions are found and for how long the exploitation on the optimal distributions will persist. It challenges both the social and self part of SRD.

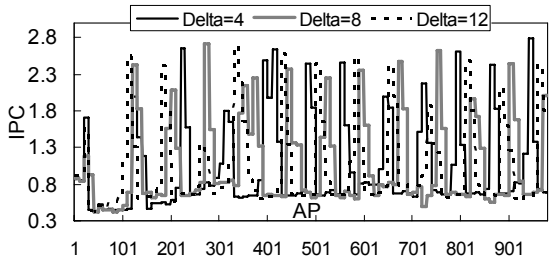



**Figure 4. Impact of *Delta* on optimization procedure in workload {art, mcf}.**

Among the eleven situations in APs numbered from 100 to 950 where complete IPC peaks emerge, *Delta*=12 always achieves peaks earlier than the other two, since *Delta*=12 owns better searching capability. However, in three situations (near 275, 750 and 900), *Delta*=12 misses the better peaks while *Delta*=8 catches them; in six situations (near 120, 200, 350, 450, 520 and 830), peaks of *Delta*=12 are narrower than those of *Delta*=8 although *Delta*=12 obtains better peak values; only in two (near 590 and 670) of the eleven situations, *Delta*=12 attains performance comparable to *Delta*=8. These phenomena show that for most of the situations *Delta*=8 can make persistent exploitation on optimal distributions for a longer time than *Delta*=12 does.

*Delta*=4 is always the latest one to catch the new performance peaks because of its weak exploration ability, it is difficult to capture transient program phases.

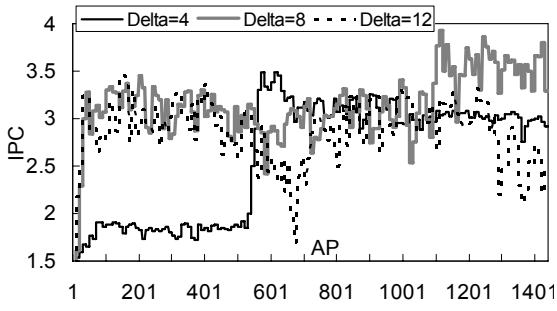



**Figure 5. Impact of *Delta* on optimization procedure in workload {lucas, equake, applu, vpr}.**

Figure 5 shows that *Delta*=12 and *Delta*=8 find the better optimal distribution space earlier than *Delta*=4 in APs from 40 to 550, but *Delta*=12 obviously loses the optimal distributions near 680, 800 and in APs from 1180 to 1400 while *Delta*=8 holds them.

In summary, a large *Delta* provides a big exploration scope, it is easy to capture transient performance peaks, but also easy to lose them. For a small *Delta* supplying weak exploration ability, it is too slow to capture transient program phases, stagnation in the inferior optimal distribution space often happens. Only a moderate *Delta* can help to do both necessary exploration and persistent exploitation.

## 4.4 Colony Scale and Allocation Period

In the simulation, we found that *Colony_size* from 4 to 8 is reasonable for SMT processors with a max thread number of 4. It is not easy for a small scale colony to discover the optimal allocation in resource distribution space. While a large colony scale results in a long generation period, which makes the historical experience lose its directive significance in the next generation period in an environment with changing program behavior. Hence it is difficult to catch the optimal distribution also.

Since the generation period is the product of the allocation period and *Colony_size*. A small allocation period will produce a short generation period and many generations of colony in a given cycles of simulation. It can perform fine-grained optimization, but needs more additional computation. While a large allocation period makes a long generation period, consequently the historical experience loses its directive significance according to the program locality principle, and it is difficult to find the optimal distribution. Therefore, the allocation period is a tradeoff between the optimization effect and additional computation.

# 5. CONCLUSIONS

This work presents a swarm-inspired resource distribution policy for SMT processors, which adjusts the trial resource distributions directly targeting the runtime performance. By adaptation of the particle swarm optimization, a computational model is established for SRD to steer the social exploitation and self exploration activities of the trial distributions on behalf of potential solutions of resource distribution.

Simulation results show that SRD obtains potential Hmean increment as well as good IPC increment over other three methods. Under the IPC metric, SRD outperforms ICOUNT, STATIC and AROB by 29.1%, 24.1% and 16% on average respectively, and under the Hmean metric SRD outperforms the three methods by 18.2%, 14.1% and 9.6% on average respectively. Generally, SRD provides more IPC and Hmean gains in a more complicated SMT environment such as the WL4, MIX and MEM type workloads.

Experiments and discussions on SRD's important parameters demonstrate that, the (*social*, *self*) pair functions as a lever between the exploitation on experience distribution and the exploration of new potential distribution, a balanced pair can take both throughput and fairness metrics into account; the exploration step impacts on the capability of searching for new solutions, a moderate step will help to timely find a better optimal distribution space and stick to the optimal space for enough time; and dealing with the colony scale and the allocation period is a tradeoff between the optimization quality of resource distribution and additional computation.

Further work will try to design and simulate the asynchronous interaction between the hardware that performs SRD algorithm and the new SRD-supported SMT processor (mentioned at the end of Subsection 2.3). And also do some studies on the auto-adjustment of important parameters of SRD, such as the (*social*, *self*) pair and *Delta*, to make SRD more generally smart for specific program phases that emerge in flight.

## 6. ACKNOWLEDGEMENT

This work was supported by the National Hi-Tech Research and Development Program (863) of China (No. 2006AA01Z431), and the Key Science and Technology Program of Zhejiang Province of China (No. 2007C11068, 2007C11088).

## 7. REFERENCES

[1] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, IEEE CS, Washington, DC, 1995, 392-403.

[2] D. M. Tullsen, S. J. Eggers, J. S. Emer, and H. M. Levy. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, IEEE CS, Washington, DC, 1996, 191-202.

[3] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, IEEE CS, Washington, DC, 2001, 318–327.

[4] A. El-Moursy and D. H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *Proceedings of the 9th International Conference on High Performance Computer Architecture*, IEEE CS, Washington, DC, 2003, 31-42.

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

[6] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *Proceedings of the 12th Int'l Conf. on Parallel Architectures and Compilation Techniques*, IEEE CS, Washington, DC, 2003, 15-25.

[7] F. Cazorla, A. Ramirez, M. Valero, and E. Fernández. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th International Symposium on Microarchitecture*, IEEE CS, Washington, DC, 2004, 171-182.

[8] J. Sharkey, D. Balkan, and D. Ponomarev. Adaptive reorder buffers for SMT processors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, ACM Press, New York, NY, 2006, 244-253.

[9] R. Eberhart and J. Kennedy. A new optimizer using particles swarm theory. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, IEEE Press, Piscataway, NJ, 1995, 39-43.

[10] J. Kennedy. The particle swarm: social adaptation of knowledge. In *Proceedings of the International Conference on Evolutionary Computation*, IEEE Press, Piscataway, NJ, 1997, 303~308.

[11] J. Sharkey, D. Ponomarev, and K. Ghose. *M-Sim: a Flexible, Multi-threaded Simulation Environment*. Tech Report CS-TR-05-DP01. Department of Computer Science, SUNY Binghamton, 2005.

[12] D. Burger, T. M. Austin, and S. Bennett. *Evaluating Future Microprocessors: the Simplescalar Tool Set*. Technical Report 1308, Computer Science Department, University of Wisconsin-Madison, Madison, USA, 1996.

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

[14] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, NY, 2002, 45-57.

[15] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, IEEE CS, Washington, DC, 2003, p.244.

[16] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, IEEE CS, Washington, DC, 2001, 164-171.