Anaphase: A Fine-Grain Thread Decomposition Scheme for Speculative Multithreading

C. Madriles, P. López, J. M. Codina, E. Gibert, F. Latorre, A. Martínez, R. Martínez and A. González
Intel Barcelona Research Center,
Intel Labs, Universitat Politècnica de Catalunya, Barcelona (Spain)
{carlos.madriles.gimeno, pedro.lopez, josep.m.codina, enric.gibert.codina, fernando.latorre, alejandro.martinez, raul.martinez, antonio.gonzalez}@intel.com

Abstract

Industry is moving towards multi-core designs as we have hit the memory and power walls. Multi-core designs are very effective to exploit thread-level parallelism (TLP) but do not provide benefits when executing serial code (applications with low TLP, serial parts of a parallel application and legacy code). In this paper we propose Anaphase, a novel approach for speculative multithreading to improve single-thread performance in a multi-core design. The proposed technique is based on a graph partitioning technique which performs a decomposition of applications into speculative threads at instruction granularity. Moreover, the proposed technique leverages communications and pre-computation slices to deal with inter-thread dependences.

Results presented in this paper show that this approach improves single-thread performance by 32% on average and up to 2.15x for some selected applications of the Spec2006 suite. In addition, the proposed technique outperforms by 21% on average schemes in which thread decomposition is performed at a coarser granularity.

1. Introduction

Single-threaded processors have shown significant performance improvements during the last decades by exploiting instruction level parallelism (ILP). However, this kind of parallelism is sometimes difficult to exploit, requiring complex hardware structures that lead to prohibitive power consumption and design complexity. Moreover, such complex techniques to further extract ILP are lately giving diminishing performance returns. In this scenario, chip multiprocessors (CMPs) have emerged as a promising alternative in order to provide further processor performance improvements under a reasonable power budget.

CMP processors comprise multiple cores where the applications are executed. These high-performance systems strongly rely on mechanisms to generate parallel workloads. Parallel applications take advantage of large amount of cores that efficiently exploit thread level parallelism (TLP). However there is a large set of applications which are difficult to parallelize and it is likely that this type of applications will still remain in the future. Therefore, in order to sustain high-performance on these applications, novel techniques to parallelize them must be developed.

Traditional parallelizing techniques decompose the applications using conservative dependence and control analysis to guarantee independence among threads. Hence, the performance improvements in hard to parallelize applications is limited due to the fact that correctness must be guaranteed. In order to overcome this limitation, speculative multithreading techniques can be used. When decomposing an application into speculative threads, the independence among threads is no longer required. Therefore, hardware support is provided to detect violations among threads, as well as to rollback to a previous correct state or checkpoint.

Previous approaches [1][3][32][7][13][11][20][23][2] to speculative multithreading decompose sequential codes into large chunks of consecutive instructions. The lack of flexibility imposed by the granularity of these decomposition techniques may severally limit the potential of the speculative multithreading paradigm:

- Coarse grain decomposition limits the opportunities for exploiting fine-grain TLP, which is key in order to improve performance in irregular applications such as SpecInt.
- Moreover, this coarse grain decomposition may incur in a large number of inter-thread dependences that may severally constraint the final performance of the speculative threads.
- Finally, coarse-grain decomposition limits the flexibility when distributing the workload among threads, which may end up creating imbalances among threads and may limit the opportunities for increasing memory level parallelism.

In this paper we propose a novel speculative multithreading technique in which the compiler is responsible for distributing instructions from a single-threaded application or a sequential region of a parallel application into threads, that can be executed in parallel in a multicore system with support for speculative multithreading. In order to overcome the limitations of previous schemes, we propose a paradigm in which the decomposition of the original application into speculative threads is performed at instruction granularity.

The main contributions of this work are the following:

- We propose a novel speculative multithreading paradigm in which code is shred into threads at the finest possible granularity: at instruction level. This has important implications on the hardware design to manage speculative state and on the software to obtain such threads.
- We develop a new algorithm named Anaphase to generate correct and good code to be executed on this platform. The proposed algorithm includes a subset of the heuristics we have tried. Due to space constraints and for clarity reasons, this paper mainly focuses on those heuristics that obtained more performance.

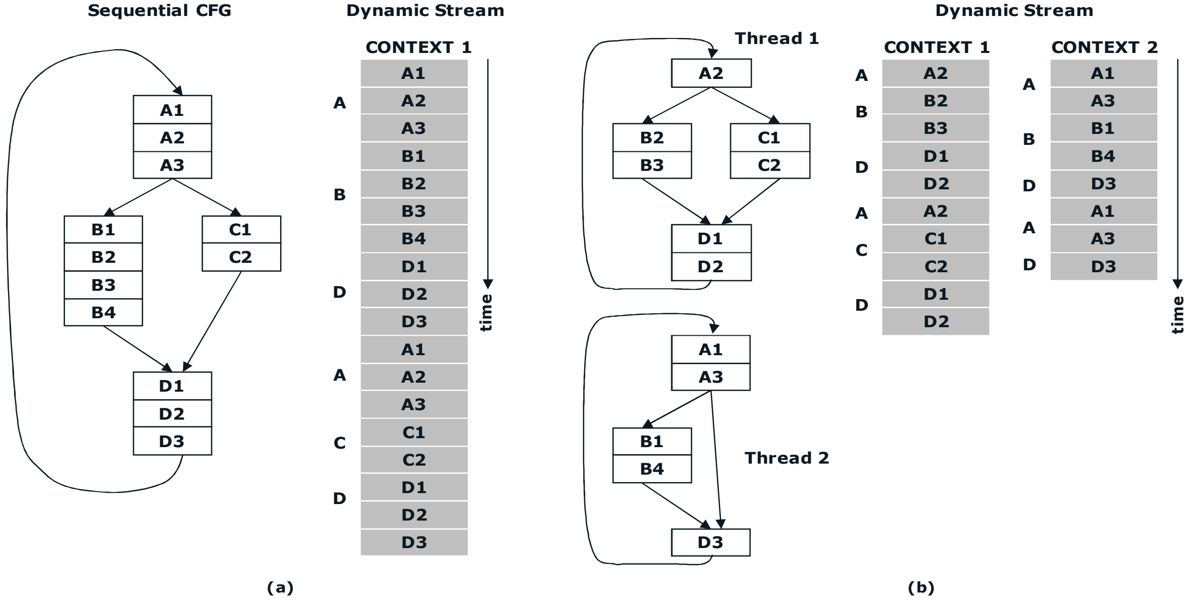

Figure 1. Conceptual view of the fine-grain decomposition into speculative threads. An example shows (a) a potential execution in a single core, and (b) a potential execution on a fine-grain speculative multithreading execution.

- Among the aforementioned heuristics, special emphasis must be put on the way we manage inter-thread dependences. In fact, inter-thread dependences can be either (i) ignored (relying in the hardware to detect them and recover from them), (ii) fulfilled through an explicit communication instruction pair or (iii) fulfilled by replicating its pre-computation slice (or part of it) [30][11].

Although in this paper we limit our focus on evaluating the use of Anaphase for the parallelization of loop regions, Anaphase is a general approach able to parallelize any code region: loops, routines, straight-line code, or any other code structure.

Results reported in this paper show that when Anaphase is applied over the hottest loops of the Spec2006 benchmark suite using 2 threads running on 2 cores, the overall performance is improved by 32% on average compared to a single-core execution, and up to 2.15x for some selected benchmarks. Moreover, it outperforms previous schemes that perform speculative parallelization at iteration granularity by 21% on average.

The rest of the paper is organized as follows. Section 2 gives an overview of the proposed speculative model and the underlying CMP architecture. Then, the algorithm for generating speculative threads is discussed in Section 3. After that, the evaluation is presented in Section 4 and related work is discussed in Section 5. Finally, conclusions are drawn in Section 6.

2. Speculative Multithreaded Model

The proposed scheme decomposes a sequential application into speculative threads (SpMT threads) at compile time. SpMT threads are generated for those regions that cover most of the execution time of the application. In this section we first describe the speculative threads considered in this model, the extensions added to a multicore architecture to execute them, and the proposed execution model.

2.1 Speculative Threads

The main feature of the proposed speculative multithreading scheme is that thread decomposition is performed at instruction granularity. An example of such fine-grain decomposition is shown in Figure 1. Figure 1 (a) depicts the static Control Flow Graph (CFG) of a loop and a possible dynamic execution of it consisting of the basic block stream {A, B, D, A, C, D}, while Figure 1 (b) shows a possible fine-grain decomposition into speculative threads.

When the compiler shreds a sequential piece of code into speculative threads, it decides how to handle inter-thread dependences. For each inter-thread dependence it may (i) ignore the dependence, (ii) satisfy it by an explicit communication, or (iii) satisfy it by replicating its pre-computation slice (p-slice). The p-slice of an instruction is defined as the set of instructions which this instruction depends upon traversing backwards the data dependence graph. Therefore, an instruction may be assigned to both threads (referred to as replicated instructions), as is the case of instruction D1 in Figure 1.

Finally, another feature of the proposed scheme is that each speculative thread must be self-contained from the point of view of the control flow. This means that each thread must have all the branches it needs to resolve its own execution. This is in line with the hardware design presented below in Section 2.2, in which the two cores fetch, execute and locally retire instructions in a decoupled fashion. Hence, the compiler is responsible for replicating all the necessary branches in order to satisfy this rule. Note, however, that replicating a branch into more than one thread may or may not imply replicating its p-slice, since the compiler may decide to communicate some of these dependences.

2.2 Multicore Architecture

Although the main focus of this paper is on the software-side of the proposed fine-grain speculative multithreading paradigm, this

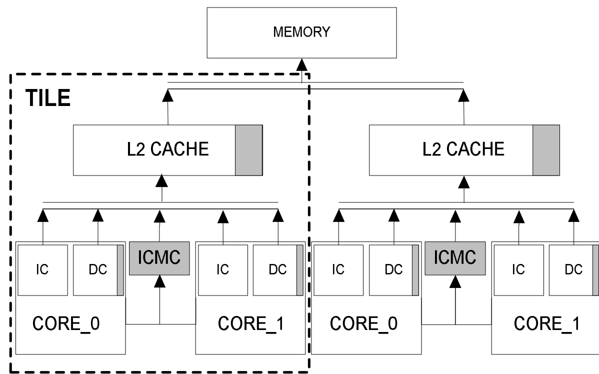

Figure 2. Multicore architecture overview

subsection gives an overview of how the hardware-side looks like. The detailed explanation of the hardware, its design decisions and its evaluation is described in another paper [16].

In this paper we assume a multi-core x86 architecture [19] divided in tiles as shown in Figure 2. Every tile implements two cores with private first level write-through data caches and instruction caches. The first level data cache includes a state-of-the-art stream hardware prefetcher. These caches are connected to a shared copy-back L2 cache through a split transactional bus. Finally, the L2 cache is connected through another interconnection network to main memory and to the rest of the tiles.

Tiles have two different operation modes: independent mode and cooperative mode. The cores in a tile execute conventional threads when the tile is in independent mode and they execute speculative threads (one in each core) from the same decomposed application when the tile is in cooperative mode.

In cooperative mode, a piece of hardware inside a tile referred to as the Inter-Core Memory Coherency Module (ICMC) is responsible for orchestrating the execution. The ICMC mainly consists of two FIFO queues (one for each core) with 1K entries in which instructions from each core are inserted as they retire locally. The ICMC globally retires instructions from these FIFO queues in the original program order specified by some marks associated with the instructions. Therefore, one of the duties of the ICMC is to reconstruct the original program order. When the ICMC detects a memory violation, it rollbacks to a previous consistent state and the software redirects execution towards the original sequential version of the code.

Hence, in cooperative mode, the cores fetch, execute and locally retire instructions from the speculative threads in a decoupled fashion most of the time. The only points where synchronization occurs between the two cores are: (i) when an inter-thread dependence is satisfied by an explicit communication instruction pair, and (ii) when a core fills up its FIFO queue in the ICMC. In the second case, the other core has still to locally retire the oldest instruction/s in the system; hence, the core must wait until this happens and the ICMC frees up some of its FIFO queue space.

The memory speculative state is kept in the local L1 data caches. The L2 always has the non-speculative correct state and is updated by the ICMC in the original program order using the information kept in the FIFO queues. Hence, the local L1 data caches may have multiple versions of the same datum since merging is performed correctly at the L2 by the ICMC. A consequence of such a design decision is that when transitioning from cooperative mode to independent mode, the contents of the L1 data caches must be invalidated. Furthermore, the ICMC detects a memory violation whenever a store and a load that are dependent are assigned to different cores and the dependence is not satisfied by an explicit communication or through a pre-computation slice. However, as will be shown in the evaluation section, the performance loss due to memory violations is negligible because: (i) profiling information is accurate enough to guarantee this hardly ever happens, and (ii) the regular checkpointing mechanism presented below guarantees that the amount of work to squash is small.

The non-speculative register state, on the other hand, is distributed between the two cores' physical register files. Register checkpointing is performed by hardware at the places decided by the compiler through CKP instructions. This instruction marks the place where the register checkpoint must be taken. In this paper, CKP instructions are inserted at the beginning of any loop belonging to optimize regions. Thanks to this mechanism each core takes a partial register checkpoint at regular intervals. From these partial checkpoints, the core that is retiring the oldest instructions in the system can recover a complete valid architectural register state. As previously pointed out, these regular checkpoints allow the system to normally throw away very little work when a rollback occurs.

Finally, the architecture provides a mechanism to communicate values through memory with explicit send / receive instruction pairs (a send is implemented through a special type of store, while a receive through a special type of load). Such a communication only blocks the receiver when the datum is not ready yet: the sender is never blocked. Although these communications happen through the L2 cache (with a roundtrip latency of 32 cycles), the decoupled nature of the cores requires that the sender is at the head of the ROB before sending the datum to memory. Hence, a communication is often quite expensive.

2.3 Execution Model

Speculative threads are executed in cooperative mode on the multi-core processor described in Section 2.2. In Figure 3 an overview of the overall scheme is presented, assuming two cores. The compiler detects that a particular region B is suitable for applying speculative multithreading. Hence it decomposes B into two speculative threads that are mapped somewhere else in the application address space. We refer to this version of B as the optimized version.

A spawn instruction is inserted in the original code before entering region B. Such a spawn operation creates a new thread, and both, the spawner and the spawnee speculative threads, start executing the optimized version of the code. Both threads execute in cooperative-mode within a tile. For simplicity, we assume that when a spawn instruction is executed, the other core in the same tile is idle.

Violations, exceptions and/or interrupts may occur while in cooperative mode, and the speculative threads may need to be rollbacked. In order to properly handle these scenarios in cooperative-mode, each speculative thread performs partial checkpoints regularly as discussed in Section 2.2 and complete

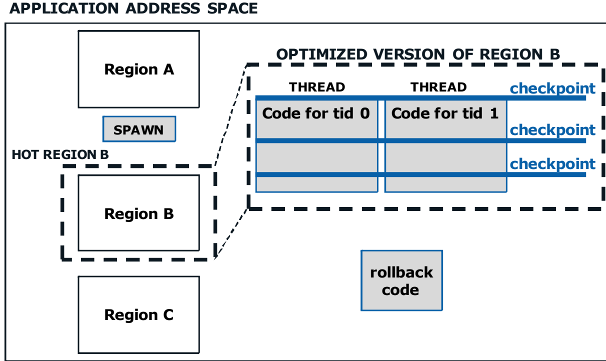

Figure 3. Execution model overview

checkpoints are performed by the thread executing the oldest instructions. When the hardware detects a violation, an exception or an interrupt, the execution is redirected to a code sequence referred to as the rollback code (see Figure 3). This code is responsible to roll back the state to the last completed checkpoint and to resume execution from that point on in independent mode by redirecting the spawner thread to the appropriate location in the original version of the code. Then, cooperative mode will restart when a new spawn instruction is encountered.

Although checkpoints are drawn as synchronization points in Figure 3, this is just for clarity purposes. A speculative thread that goes through a checkpoint does not wait to the other speculative thread to arrive to that point, as briefly discussed in Section 2.2.

When both threads complete, they synchronize to exit the optimized region, the speculative state becomes non-speculative, the execution continues with one single thread, and the tile resumes to independent mode.

3. Anaphase

Speculative threads are generated at compile time. The compiler is responsible for: (1) profiling the application, (2) analyzing the code and detecting the most convenient regions of code for parallelization, (3) decomposing the selected region in speculative threads, and finally, (4) generating optimized code and rollback code.

Although the proposed fine-grained speculative multithreading paradigm can be applied to any code structure, in this paper we have concentrated on applying the partitioning algorithm to loops. In particular, we limit our focus to outer loops which are frequently executed according to the profiling. In addition, such loops are unrolled and frequently executed routines are inlined in order to enlarge the scope for the proposed thread decomposition technique.

Once the hot loops are detected, they are passed to the Anaphase algorithm that decomposes each loop into speculative threads. Although Anaphase can decompose a loop into any number of speculative threads, in this paper we limit our study to partitioning each loop into two threads. For each individual loop, Anaphase performs the steps depicted in Figure 4. First, the Data Dependence Graph (DDG) and the Control Flow Graph (CFG) of the loop are built. Such graphs are complemented by adding profiling information such as how many times each node

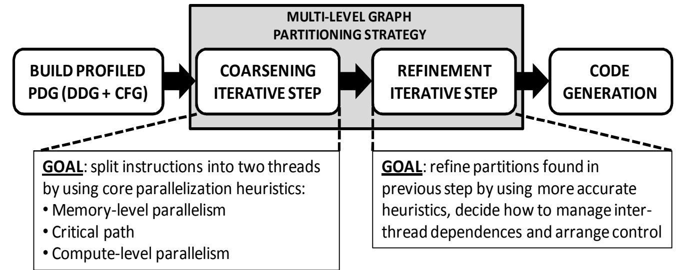

Figure 4. Anaphase partitioning flow for a given loop

(instruction) is executed and how many times occurs data dependence (reg or memory) between a pair of instructions occurs. Both graphs are collapsed into the Program Dependence Graph (PDG) [9].

These profiled graphs are then passed to the core of the decomposition algorithm. In order to shred the code into two threads a multi-level graph partitioning algorithm [14] is used. The first part of the multi-level graph partitioning strategy is referred to as the coarsening step. In such a step, nodes (instructions) are iteratively collapsed into bigger nodes until there are as many nodes as desired number of partitions (in our case two, since we generate two threads). The goal of this step is to find relatively good partitions using simple but effective parallelization heuristics.

After the coarsening step, a refinement process begins. In this step, the partitions found during the coarsening phase are iteratively revaluated using more fine-grained heuristics. Furthermore, it is during this phase that inter-thread dependences and control replication are managed. This step finishes when no more benefits are obtained by moving nodes (instructions) from one partition to the other based on the heuristics.

Finally, once a partition is computed, Anaphase generates the appropriate code to represent it. This implies mapping the optimized version of the loop somewhere in the address space of the application, placing the corresponding spawn instructions and adding rollback code.

The following sections describe the heuristics we have used during the coarsening and the refinement steps of the algorithm, which are the key components of Anaphase. Figure 5 shows an overview of how a multi-level graph partitioning algorithm works. Although the forthcoming sections give a few more insights, we refer the reader to [14] for further details on how this kind of algorithms works.

3.1 Coarsening Step

As previously mentioned, the coarsening step is an iterative process in which nodes (instructions) are collapsed into bigger nodes until there are as many nodes as the number of desired partitions. At each iteration a new level is created and nodes are collapsed in pairs. A graphic view of how this pass works is shown in Figure 5.

The goal of this pass is to generate relatively good intermediate partitions. Since this pass is backed up by a refinement step, the partitions must contain nodes that it makes sense to assign to the

same thread but without paying much attention to the partition details, such as how to manage inter-thread dependences and how to replicate the control. It is for this reason that we use core heuristics that we believe are simple but very effective to assign "dependent" code sequences to the same partition and "independent" code sequences to different threads.

These core heuristics are based on three concepts:

- *Memory-level parallelism*. Parallelizing miss accesses to the most expensive levels of the memory hierarchy (last levels of cache or main memory) is an effective technique to hide their big latencies. Hence, it is often good to separate the code into memory and computation components for memory-bound sequences of code, and assign each component to a different thread. The memory component includes the miss accesses and their pre-computation slice, while the compute component includes the rest of the code: code that does not depend on the misses and code that depends on the misses but does not generate more misses. The rationale behind this is that the memory component often has a higher density of misses per instruction and hence it normally does a better usage of the processor capabilities to parallelize misses (e.g. miss status holding registers) when assigned to the same core. For example, a core will not parallelize misses if they are at a distance (in dynamic number of instructions) greater than the ROB size. However, the distance may be reduced if some of the instructions in between are assigned to another core. This may allow servicing some misses in parallel.
- *Critical-path*. It is often good to assign instructions on the critical path to the same thread and off-loading non-critical instructions to other threads, in order not to incur in any delay in the execution of the former.
- *Compute-level parallelism*. It is often good to assign independent pieces of code to different threads and dependent pieces of code to the same thread in order to exploit the additional computation resources and to avoid unnecessary synchronization points between the two threads.

The pseudo-code of the coarsening step is shown in Figure 6. The following subsections describe the process in detail.

3.1.1.1 Heuristics Used by Anaphase

In order to exploit the aforementioned criteria a matrix M is built to describe the relationship between node pairs (see routine *create_and_fill_matrix_at_current_level* in Figure 6). In particular, the matrix position M[*i,j*] describes how good is to assign nodes *i* and *j* to the same thread and M[*i,j*] = M[*j,i*]. Each matrix element is a value that ranges between 0 (worst ratio) and 2 (best ratio): the higher the ratio, the more related two nodes are. The matrix is initialized to all zeros, and cells are filled based on the next three heuristics following the order described below:

Delinquent loads. As mentioned before, exploiting memory-level parallelism is very important to achieve good performance in memory-bound code sequences. In order to do so, the algorithm detects delinquent loads [6], which are those load instructions that will likely miss in cache often and therefore impact performance. After using different thresholds, we have observed that a good trade-off is achieved by marking as delinquent all loads that have a miss rate higher than 10% in the L2 using profiling.

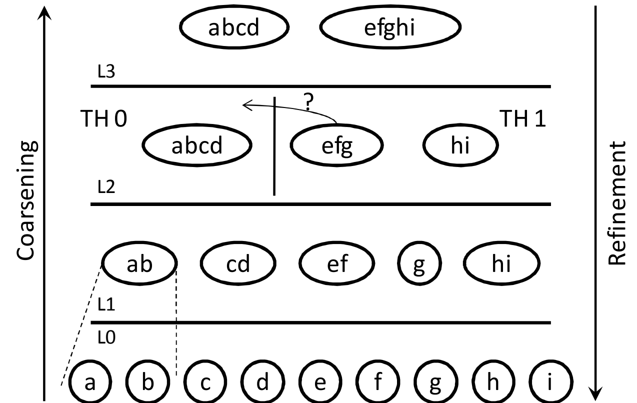

Figure 5. Multi-level graph partitioning simple example that requires four levels (L0…L3) to achieve an initial partition of nodes into two sets

By using this heuristic we want to favor the formation of nodes with delinquent loads and their pre-computation slices, in order to allow the refinement stage to model these loads separated from their consumers. Therefore, the data edge that connects a delinquent load with a consumer instruction is given very low priority. In order to achieve this effect, the ratio for these two nodes is fixed to 0.1 in matrix M (a very low priority). The rest of the cells of M are filled with the following two heuristics.

Slack. As discussed above, grouping together critical instructions to the same thread is an obvious way to avoid delaying their execution unnecessarily. Unfortunately, computing at compile time how critical an instruction will be is impossible. However, since Anaphase has a posterior pass that refines the decisions performed at this step, a simple estimation works fine at this point.

For the purpose of estimating how critical an instruction is we compute its local slack [10], defined as its freedom to delay its execution without impacting total execution time. Slacks are assigned to edges. In order to compute such slack, first, the algorithm computes the earliest dispatch time for each instruction considering only data dependences in the PDG and ignoring cross-iteration dependences. After this, the latest dispatch time of each instruction is computed in the same manner. The slack of each edge is defined as the difference between the earliest and the latest dispatch times of the consumer and the producer nodes respectively.

Two nodes *i* and *j* that are connected by an edge with very low slack are considered part of the critical path and will be collapsed with higher priority. We have considered critical edges those with a slack of 0. Therefore, the ratios M[*i,j*] and M[*j,i*] for these nodes are set to 2.0 (the best ratio). The rest of the matrix cells are filled with the following last heuristic.

Common predecessors. Finally, in order to assign dependent instructions to the same thread and independent instructions to different threads, we compute how many predecessor instructions each node pair (*i,j*) share by traversing edges backwards. In particular, we compute the predecessor relationship of every pair of nodes as the ratio between the intersection of their predecessors

```
Routine coarsening step()
current_level = 0
while num_partitions > 2 do
  call create_and_fill_matrix_at_current_level()
  current_level++
  call collapse_nodes()
done


Routine collapse nodes()
for each node pair (i,j) in any order
  collapse them if all the three conditions are met:
    (i) neither node i nor node j have been
        collapsed from previous level to current_level
    (ii) M[i][j] ≥ 0.95*M[i][k] for all nodes k
    (iii) M[i][j] ≥ 0.95*M[k][j] for all nodes k
endfor
```

```
Routine create and fill matrix at current level()
initialize matrix M to all zeroes
for each node i identified as a delinquent load
  for each consumer node j of i based on data deps
    M[i][j] = M[j][i] = 0.1
  endfor
endfor

compute slack of each edge
for all edges with a slack of 0 connecting nodes i & j
  if M[i][j] = 0
    M[i][j] = M[j][i] = 2.0
  endif
endfor

compute common pred. ratio F for all node pairs (i,j)
for each node pair (i,j)
  if M[i][j] = 0
    M[i][j] = M[j][i] = F(i,j)
  endif
endfor
```

Figure 6. Pseudo-code of the coarsening step of the algorithm.

and the union of their predecessors. The following equation computes the ratio (R) between nodes *i* and *j*:

$$R(i,j) = \frac{P(i) \cap P(j)}{P(i) \cup P(j)}$$

Where *P(i)* denotes the set of predecessors of *i*, including itself. It is also worthwhile to mention that each predecessor instruction in *P(i)* is weighted by its profiled execution frequency in order to give more importance to the instructions that have a deeper impact on the dynamic instruction stream.

The ratio *R(i,j)* describes to some extent how related two nodes are. If two nodes share an important amount of nodes when traversing the graph backwards, it means that they share a lot of the computation and hence it makes sense to map them together. They should have a big relationship ratio in matrix M. On the other hand, if two nodes do not have common predecessors, they are independent and are good candidates to be mapped into different threads. Note that R(*i*,*j*) = R(*j*,*i*).

In the presence of recurrences, we found many nodes having a ratio *R(i,j)* of 1.0 (they share all antecessors). To solve this, we compute the ratio twice, one as usual, and a second time ignoring cross-iteration dependences. The final ratio F(*i*,*j*) is the sum of these two. We have observed that computing this ratio twice as explained here improves the quality of the obtained threading and increases performance consequently. This final ratio is the value that we use to fill the rest of cells in M[*i*,*j*] and this is why the matrix values range between 0 and 2 and not between 0 and 1.

Once matrix M has been filled with the three heuristics described above, the coarsening step of Anaphase uses it to collapse pairs of nodes into bigger nodes (see routine *collapse_nodes* in Figure 6). In particular, the algorithm collapses node pairs in any order (we have observed that the ordering in this case is not important) as long as the ratio M[*i*,*j*] of the pair to be collapsed is at most 5% worst than the best collapsing option for node *i* and than the best collapsing option for node *j*. This is so because a multi-level graph partitioning algorithm requires a node to be collapsed just once per level [14]. Hence, as we proceed collapsing, there are fewer collapsing options at that level and we want to avoid collapsing two nodes if their ratio M is not good enough.

Finally, it is important to remark that matrix M is filled in the same manner when internal levels of the graph partitioning algorithm are considered. The main difference is that the size of the matrix is smaller at each level. In these cases, since a node may contain more than one node from level 0 (where the original nodes reside), all dependences at level 0 are projected to the rest of the levels. For example, node *ab* at level 1 in Figure 5 will be connected to node *cd* by all dependences at level 0 between nodes *a* and *b* and nodes *b* and *c*. Hence, matrix M is filled naturally at all levels as described in this section.

When the multi-level graph is done collapsing, the algorithm proceeds to the next step, the refinement phase.

3.1.1.2 Final Remarks

Our final proposal for the coarsening step of the algorithm uses the aforementioned three heuristics. We previously tried several other techniques. For instance, we observed that applying each of them individually offers less speedup for most of the benchmarks when the threading is complete. We also tried to collapse nodes based on the weight of the edges connecting them, or to collapse them based on workload balance. However, the overall performance was far from that obtained with the above heuristics. We even tried a random coarsening step and, although some performance was achieved by the refinement step, the overall result was poor. Hence, it is the conjunction of the three heuristics presented here that achieved the best performance, which is later reported in Section 4.2.

3.2 Refinement Step

The refinement step is also an iterative process that walks all the levels created during the coarsening step from the topmost levels to the bottom-most levels and, at each level, it tries to find a better partition by moving one node to another partition. An example of a movement can be seen in Figure 5: at level 2, the algorithm decides if node *efg* should be at thread 0 or thread 1.

The purpose of the refinement step is to find better partitions by refining the already "good" partitions found during the coarsening process. Due to the fact that the search at this point of the algorithm is not blind, finer heuristics can be employed. Furthermore, it is at this moment that Anaphase decides how to manage inter-thread dependences and replicate the control as required.

We have used the classical Kernighan-Lin (K-L) algorithm [15] to build our refinement step. Thus, at a given level, we compute the benefits of moving each node *n* to the other thread by using an objective function and choose the movement that maximizes such objective function. Note that we try to move all nodes even if the new solution is worse than the previous one based on the objective function. This allows the K-L algorithm to overcome local optimal solutions. However, after trying all possible movements at the current level, if the solution is not better than the current one, we discard such a solution and jump down to the next level.

The following sections describe in more detail how we apply some filtering in order to reduce the cost of the algorithm, how inter-thread dependences are managed and how the objective function works.

3.2.1.1 Movement Filtering

Trying to move all nodes at a given level is very costly, especially when there are many nodes in the PDG. This is alleviated in Anaphase by focusing the movements to the subset of the nodes that seem more promising. In particular we focus on those nodes that if moved may have higher impact in terms of (i) improving workload balance among threads and (ii) reducing inter-thread dependences.

For improving workload balance, we focus on the top K nodes that may help to get close to a perfect workload balance between the two threads. Workload balance is computed by dividing the biggest estimated number of dynamic instructions assigned to a given thread by the total number of estimated dynamic instructions. A perfect balance when two threads are considered is 0.5. On the other hand, Anaphase picks the top L nodes that may reduce the number of inter-thread dependences. The reduction on the number of inter-thread dependences is simply estimated by using the occurrence profiling of the edges that are in the cuts of the partition [14].

After some experiments trying different K and L values, we have seen that a good trade-off between improving the quality of the generated threads (that is, improving performance) and the compiling cost to generate them is achieved by K=L=10. Hence, we reduce the amount of movements to 20.

3.2.1.2 Inter-Thread Deps. and Control Replication

As mentioned before, the refinement step tries to move a node from one thread to another thread and computes the benefits of such a movement based on an objective function. Before evaluating the partition derived by one movement, the algorithm decides how to manage inter-thread dependences and arranges the control flow in order to guarantee that both threads are self-contained as explained in Section 2.2.

Given an inter-thread dependence, Anaphase may decide to:

- Fulfill it by using explicit inter-thread communications, which in our current approach are implemented through the regular memory hierarchy.
- Fulfill it by using pre-computation slices to locally satisfy these dependences. A pre-computation slice consists of the minimum instructions necessary to satisfy the dependence locally. These instructions can be replicated into the other thread in order to avoid the communication.
- Ignore it, speculating no dependence if it barely occurs.

Our current heuristic does not ignore any dependence (apart, of course, from those not observed during profiling). Hence, the main decision is whether to communicate or pre-compute an inter-thread dependence.

Communicating a dependence is relatively expensive, since the communicated value goes through the shared L2 cache when the producer reaches the head of the ROB of its corresponding core. On the other hand, an excess of replicated instructions may end up delaying the execution of the speculative threads and hence impacting performance as well. Therefore, the selection of the most suitable alternative for each inter-thread dependence may have a significant impact on the performance achieved by Anaphase.

The core idea used in Anaphase is that the larger the replication needed to satisfy a dependence locally, the lower the performance. By leveraging this idea, when the profiled-weighted amount of instructions in a pre-computation slice exceeds a particular threshold the dependence is satisfied by an explicit communication. Otherwise, the dependence is satisfied by means of the pre-computation slice.

We have experimentally observed that by appropriately selecting the threshold that defines the amount of supported replication, this naïve algorithm achieves better performance than other schemes that we have tried that give more importance to the criticality of the instructions (recall that criticality in fact is an estimated approximation).

3.2.1.3 Objective Function

At the refinement stage, each partition has to be evaluated and compared with other partitions. The objective function estimates execution time for this partition when running on a tile of the multicore processor.

In order to estimate the execution time of a partition, a 20K dynamic instruction stream of the region obtained by profiling is used. Using this sequence of instructions, the execution time is estimated as the longest thread based on a simple performance model that takes into account data dependences, communications among threads, issue width resources and the size of the ROB of the target core.

3.2.1.4 Final Remarks

We have observed that the refinement step of the algorithm is also crucial to achieve good performance. In fact, if we used the final partition obtained by the coarsening phase, the overall performance was very poor. This does not mean that the coarsening step is useless, since other coarsening heuristics explained in Section 3.1.1.2 did not work out even with the same refinement pass. Recall that the main goal of the coarsening step

is not to provide a final partition, but to provide meaningful nodes in all levels for the refinement to play with.

We also tried other heuristics in order to decide when a partition is better than another during the refinement phase. These heuristics included maximizing workload balance, maximizing miss density in order to exploit memory-level parallelism, minimizing inter-thread dependences, among others. Overall, such heuristics tended to favor only a subset of the programs and never reached the performance of the heuristic presented in this paper.

4. Experimental Evaluation

4.1 Framework

The Anaphase speculative thread decomposition scheme has been implemented on top of the Intel® production compiler (*icc*). For evaluating the proposed technique, we have selected the SPEC2006 benchmark suite compiled with *icc –O3*. In total 12 SpecFp and 12 SpecInt benchmarks have been optimized with Anaphase using the train input set for profiling information.

Representative loops of the train execution have been selected to be speculative parallelized with Anaphase based on 20M instruction traces generated with the PinPoint tool. In these traces, those outer loops that account for more than 500K dynamic instructions have been selected for thread decomposition.

For each of the selected loops, the Anaphase thread decomposition technique generates a partition for different replication thresholds and different loop unrolling factors, as described in section 3. In particular, we have considered up to 7 different thresholds for limiting the replication: 0 (meaning that all inter-thread dependences must be communicated), 48, 96, 128, 256, 512 and an unbounded threshold (meaning that all the inter-thread dependences must be pre-computed). Furthermore two unrolling factors have been considered: (a) no unrolling; and, (b) unroll by the number of thread (i.e. 2 in our experiments). Hence, we generate 14 different versions for each loop and choose the best one based on the objective function outlined in Section 3.2.1.3.

The performance of the optimized benchmarks has been evaluated through a detailed cycle accurate simulator using the ref input set. The simulator models the x86 CMP architecture described in section 2.2, in which each core is an out-of-order processor with the configuration parameters shown in Table 1. The shadowed fields on the table are per each core.

Parameter	Value
Fetch, Retire and Issue width	4
Pipeline stages	7
Branch Predictor GShare history bits/table entries	12/4096
Sizes ROB/Issue Queue/MOB	96/32/48
Miss Status Holding Registers	16
L1 and ICache size/ ways/ line size/ latency	32KB/4/32B/2
L2 size/ ways/ line size/ latency (round trip)	4MB/8/32B/32
Memory latency	328
Explicit communication penalty	32
Overhead spawn / join (cycles)	64/64

Table 1. Architecture configuration parameters

SpecFP	**Static**	**Dynamic**	**SpecINT**	**Static**	**Dynamic**
bwaves	526	2,409,648	**astar**	682	1,216,009
gamess	11,575	28,180	**bzip2**	946	6,283,643
GemsFDTD	331	361,529	**gcc**	10,984	13,910
lbm	467	2,041,565	**gobmk**	17,599	2,909
leslie3d	10,66	3,477,574	**h264ref**	6,036	4,298
milc	345	3,935	**hmmer**	775	25,944
namd	1,188	3,853,555	**libquantum**	207	25,523,330
povray	16,652	94,150,139	**mcf**	1,154	595,392
soplex	6,468	896,501	**omnetpp**	13,616	13,701
sphinx3	228	1,480,891	**perlbench**	8,343	217,821
tonto	5,002	8,442	**sjeng**	7,746	11,337
wrf	1,781	80,240	zeusmp	1,180	3,247,909

Table 2. Average static and dynamic size of optimized regions

In order to perform the performance simulations, for each studied benchmark we have randomly selected traces of 100M instructions of the ref execution beginning with the head of each of its optimized loops. Results for each benchmark are then reported as the addition of all the simulated traces. On average, about 10 traces have been generated for each benchmark. We have measured that on average the optimized loops found in these traces cover more than 90% of the whole ref execution.

4.2 Results

Table 2 shows the average size of the regions decomposed with Anaphase expressed in instructions excluding extra instructions added through replication. Note that the average static size of the regions ranges from a few hundreds of instructions to thousands of instructions. This large number of instructions in the PDG advocates for the use of smart algorithms and heuristics to shred the code into speculative threads. This explains some of the decisions exposed in Section 3. In addition, the dynamic size of a region is the amount of dynamic instructions committed between the time the threads are spawned and the time the threads finish (either correctly or through a squash). As we can see, this number tends to be very large and may tolerate bigger overheads for entering and exiting an optimized region (we currently assume 64 cycles in each case). However, although regions are dynamically big, the amount of work that is thrown away when a region is rolled back (in number of dynamic instructions) is much smaller because the hardware takes regular checkpoints, as we discuss later in this section.

Figure 7 shows the performance of the Anaphase fine-grain decomposition scheme compared to a coarse-grain speculative loop parallelization scheme that assigns odd iterations to one thread and even iterations to the other. Performance is reported as speedup over execution on a single core. For this comparison, the same loops have been parallelized with both decomposition schemes and inter-thread data dependences have been equally handled with the techniques proposed in our decomposition algorithm.

As can be seen, Anaphase clearly outperforms the loop parallelization scheme. An average speedup of 32% is observed, whereas loop parallelization just achieves 9%. For very regular

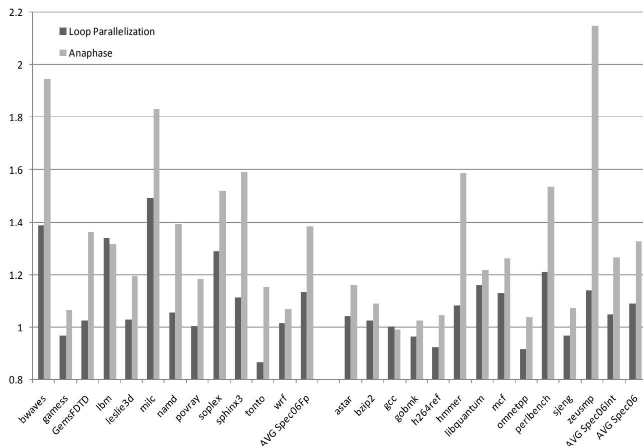

Figure 7. Anaphase performance compared to coarse-grain loop parallelization

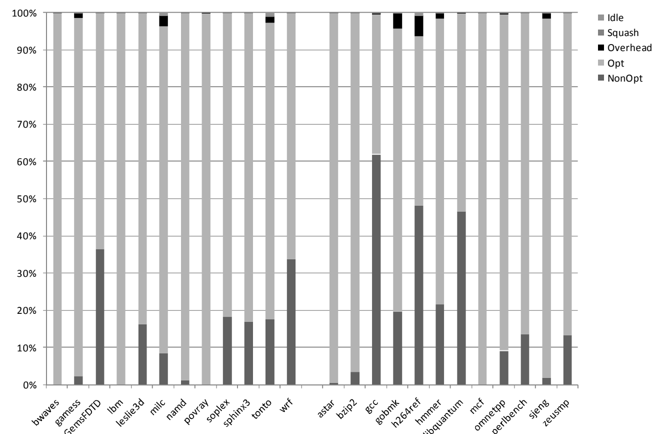

Figure 8. Anaphase activity breakdown

benchmarks, like bwaves, lbm, milc, sphinx3, perlbench, and zeusmp, where loop parallelization performs well, Anaphase significantly benefits from exploiting more MLP thanks to its fine-grain decomposition and the delinquent load heuristic. Moreover, Anaphase is able to exploit TLP on those irregular and hard to parallelize benchmarks, like astar, bzip2, gobmk, sjeng, among others, where conventional coarse-grain decomposition schemes fail. On the other hand, results for gcc show a slight slowdown with respect to loop parallelization and single-thread execution. This is mainly due to a very low coverage as will be explained later in this section. Furthermore, note that in a particular case (zeusmp) super-linear speedup is achieved by a combination of doubling computation resources with respect to single core and a better explotation of memory-level parallelism. Overall, results show that Anaphase is a very effective technique to speed-up single thread execution on regular and irregular applications.

Figure 8 shows the active time breakdown for the execution of the different benchmarks optimized with Anaphase. The first thing to notice is that the overhead of the Anaphase execution model is extremely low for all benchmarks, even though we conservatively assume 64 cycles for entering and exiting an optimized region, due to the spawn and join operations. Benchmarks gobmk and h264ref present the larger overhead, 4% and 6% respectively, because regions are smaller and more frequently spawned than for the rest of benchmarks.

Thanks to the workload balance heuristic and the checkpointing support, the Anaphase scheme is also very effective on reducing the idle time of the cores and the amount of work that is thrown away on a squash. For all benchmarks both sources of overhead represent less than 1% of the active cycles. It is worth to point out that even though on average less than 2% of the optimized regions turn out to be squashed due to a memory misspeculation at some point during their execution, the percentage of squashed regions is close to 95% for some benchmarks like bwaves and lbm. However, in these cases speedup is still achieved. This is so because such squashes occur long time after entering the region (the loop) and the proposed fine-grain speculative paradigm takes checkpoints regularly. Hence, although for these benchmarks there are very high chances that a dependence not observed during profiling that requires a squash arises at some point during the execution of a loop, the last valid checkpoint is close enough to have negligible effects on performance.

As expected, the time spent in the optimized loops is very high in almost all the benchmarks. However, for some benchmarks like gcc, h264ref, and libquantum, the time spent in non-optimized code is greater than 40% due to the low coverage of the optimized code. Coverage is mainly a caveat of our research infrastructure. Since our analysis works with traces and not with the complete binaries, the hottest loops chosen with the train input set do not sometimes correspond with the hottest parts of the traces used with the ref input set. This explains the poor performance of these benchmarks.

Finally, the amount of extra instructions introduced by the Anaphase decomposition scheme is shown in Figure 9. These extra instructions are divided into two groups: replicated instructions and communications. The former includes all instructions that are replicated in order to satisfy a dependence locally (p-slice), plus all instructions that are replicated to manage the control. The latter, on the other hand, are additional instructions because of explicit communications. On average, about 30% of additional instructions compared to single-thread execution are introduced. However, for some benchmarks like povray, gobmk, and sjeng, the Anaphase decomposition scheme introduces more than 70% of replicated instructions. Although, this large amount of replication code does not imply a slowdown in performance, it may imply an increase in energy. One important thing to notice is that in Anaphase p-slices are conservative and do not include any speculative optimization. Previous work has shown that p-slices can be significantly reduced through speculative optimizations with a slight impact on accuracy [29][11]. On the other hand, explicit communications, only account for 3% of additional instructions on average. This short amount of extra instructions has proven to be a very effective technique to handle inter-thread dependences. We have verified that with a scheme that does not allow explicit communications. In this case, the performance speedup of Anaphase for the studied benchmarks drops from 32% to 20% on average.

These results strongly validate the effectiveness of the fine-grain decomposition scheme introduced in this paper in meeting the Anaphase design goals: high performance, resulting from better

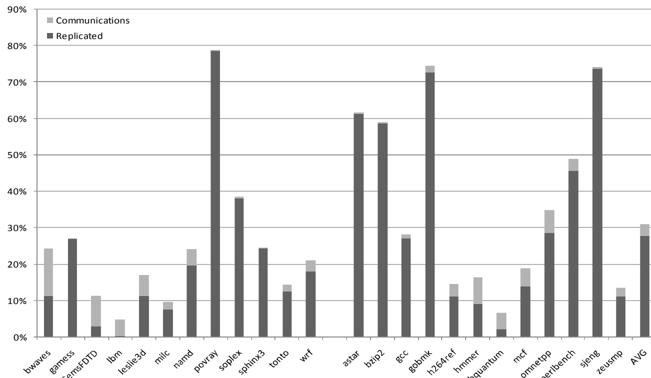

Figure 9. Anaphase instruction overheads

exploit of TLP and MLP, high accuracy (low squash rates), and low overheads.

5. Related Work

Traditional speculative multithreading schemes decompose sequential codes into large chunks of consecutive instructions. Three main different scopes for decomposition have been so far considered: loops [4][7][27], function calls [2] and quasi-independent points [18]. When partitioning loops, most of the previous schemes spread the iterations into different threads, i.e. one or more loop iterations are included in the same thread. A limited number of works consider the decomposition of the code inside the iterations by assigning several basic blocks into the same thread, while others assign strongly connected components to different threads. On the other hand, all previous works considering the decomposition at function calls shred the code in such a way that consecutive functions are dynamically executed in parallel. Finally, all works considering general control flows with the objective of decomposing the program in quasi-independent points considers the shredding of the whole application in such a way that consecutive chunks of basic blocks are assigned to different threads for its parallel execution.

The coarse grain decomposition featured by all previous speculative multithreading approaches may constraint the benefits of this paradigm. This is particularly true when these techniques must face hard to parallelize codes. When these codes are decomposed in a coarse grain fashion, it may be the case that too many dependences appear among threads. This may end up limiting the exploitable TLP for this codes and harming performance. In order to deal with this limitation, Anaphase parallelizes applications at instruction granularity. Therefore, the new model proposed experiences a larger flexibility because it will choose the granularity that best fits a particular loop. Thus it has more potential for exploiting further TLP than previous schemes and it can be seen as a superset of all previous threading paradigms.

On the other hand, four main mechanisms have been considered so far to manage the data dependences among speculative threads: (a) speculation; (b) communication [25][24][28][22]; (c) pre-computation slices [32][11], and (d) value prediction [1][21][17]. Each of these mechanisms has its benefits and drawbacks, and in each particular situation (i.e. dependence in a piece of code) one mechanism may be more appropriate than another. In order to overcome this limitation, Anaphase considers the possibility of using three of these mechanisms and selecting the most appropriate solution for each dependence.

Finally, other alternative mechanisms have been considered to improve single-thread performance. In particular, techniques such as enlarging the size of the cores, helper threads [6] or techniques pursuing the idea of fusing cores [5][8][12] have been proposed. Note however that, appropriate algorithms to decompose the application based on the proposed speculative multithreading paradigm can emulate the effects produced by all these techniques. Therefore the proposed paradigm can be also considered a superset of these techniques which try to tackle single-thread performance.

6. Conclusions

In this paper we have presented Anaphase, a novel speculative multithreading technique. The main novelty of the proposed technique is the instruction-grain decomposition of applications into speculative threads. Results reported in this paper show that fine-grain decomposition of the application into speculative threads provides a flexible environment for leveraging the performance benefits of the speculative multithreading paradigm. Moreover, we have shown that Anaphase results in a high accuracy with low squash rates, and low overheads.

While previous schemes that apply speculative multithreading to a coarse-grain granularity fail to exploit TLP on hard to parallelize applications, Anaphase is able to extract a larger amount of the TLP and MLP available in the applications. In particular, Anaphase is able to improve single-thread performance by 32% on average for the Spec2006 and up to 2.15x for some selected applications, while the average speedup of a coarse-grain decomposition working at iteration level is 9%.

7. Acknowledgments

This work has been partially supported by the Spanish Ministry of Science and Innovation under contract TIN 2007-61763 and the Generalitat de Catalunya under grant 2005SGR00950. We thank the reviewers for their helpful and constructive comments.

8. References

[1] H. Akkary and M.A. Driscoll, “A Dynamic Multithreading Processor”, in Proc. of the 31st Int. Symp. on Microarchitecture, 1998

[2] S. Balakrishnan, G. Sohi, “Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs”, Proc. of International Symposium on Computer Architecture, p. 302-313, 2006

[3] R.S. Chappel, J. Stark, S.P. Kim, S.K. Reinhardt and Y.N. Patt, “Simultaneous Subordinate Microthreading (SSMT)”, in Proc. Of the 26th Int. Symp. On Computer Architecture, pp. 186-195, 1999

[4] M. Cintra, J.F. Martinez and J. Torrellas, “Architectural Support for Scalable Speculative Parallelization in Shared-Memory Systems”, in Proc. of the 27th Int. Symp. on Computer Architecture, 2000

[5] J. D. Collins and D. M. Tullsen, “Clustered multithreaded architectures - pursuing both ipc and cycle time”. In Intl. Parallel and Distributed Processing Symp., April 2004

[6] J.D. Collins, H. Wang, D.M. Tullsen, C. Hughes, Y-F. Lee, D. Lavery and J.P. Shen, ”Speculative Precomputation: Long Range Prefetching of Delinquent Loads”, in Proc. of the 28th Int. Symp. on Computer Architecture, 2001

[7] Z.-H. Du, C-Ch. Lim, X.-F. Li, Q. Zhao and T.-F. Ngai, “A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs”, in Procs. of the Conf. on Programming Language Design and Implementation, June 2004

[8] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, “The Multicluster architecture: Reducing cycle time through partitioning”. In Intl. Symp. on Microarchitecture, December 1997

[9] J. Ferrante, K. Ottenstein, J. D. Warren, “The program dependence graph and its use in optimization”, in ACM Transactions on Programming Languages and Systems (TOPLAS), 1987

[10] B. Fields, R. Bodík, M. D. Hill, “Slack: maximizing performance under technological constraints”, in Procs. of the 29th Int. Symp. on Computer Architecture, 2002

[11] C. García, C. Madriles, J. Sánchez, P. Marcuello, A. González, D. Tullsen, “Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices”, Procs. of Conf. on Programming Language Design and Implementation, 2005

[12] E. Ipek, M. Kirman, N. Kirman, J.F. Martinez, “Core fusion: accommodating software diversity in chip multiprocessors”, in Proc. of the 34th Int. Symp. on Computer Architecture, 2007

[13] T. Johnson, R. Eigenmann, T. Vijaykumar, “Min-Cut Program Decomposition for Thread-Level Speculation”, Procs. of Conf. on Programming Language Design and Implementation, 2004

[14] G. Karypis, V. Kumar, “Analysis of Multilevel Graph Partitioning”, Procs. of 7th Supercomputing Conference, 1995

[15] B. Kernighan, S. Lin. “An Efficient Heuristic Procedure for Partitioning of Electrical Circuits”. In Bell System Technical Journal (1970)

[16] C. Madriles, P. López, J. M. Codina, E. Gibert, F. Latorre, A. Martínez, R. Martínez and A. González, “Boosting Single-thread Performance in Multi-core Systems through Fine-grain Multi-Threading”, in Proc. of the 36th Int. Symp. on Computer Architecture, 2009

[17] P. Marcuello, J. Tubella and A. Gonzalez, “Value Prediction for Speculative Multithreaded Architectures”, in Proc. of the 32nd Int. Conf. on Microarchitecture, 1999

[18] P. Marcuello, A. González, “Thread-Spawning Schemes for Speculative Multithreaded Architectures”, Procs. of Symp. on High Performance Computer Architectures, 2002

[19] A. Mendelson et al., "CMP Implementation in the Intel® CoreTM Duo Processor", in Intel Technology Journal, Volume 10, Issue 2, 2006

[20] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita, “Pinot: Speculative Muti-threading Processor Architecture Exploiting Parallelism over a wide Range of Granularities”, in Proc. of the 38th Int. Symp. on Microarchitecture, 2005

[21] J. Oplinger, D. Heine, M. Lam, “In search of speculative thread-level parallelism”, in Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, 1999

[22] G. Ottoni, D. August, “Communication optimizations for global multi-threaded instruction scheduling”, in Procs. of the 13th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 2008

[23] M. Prabhu, K. Olukotun, “Exposing Speculative Thread Parallelism in SPEC2000”, Proc. of Symposium on Principles and Practice of Parallel Programming, p. 142-152, 2005

[24] J. Steffan, C. Colohan, A. Zhai and T. Mowry, “Improving Value Communication for Thread-Level Speculation”, in Proc. of the 8th Int. Symp. on High Performance Computer Architecture, 1998

[25] J.Y. Tsai and P-C. Yew, “The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation”, in Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, 1995

[26] D. M. Tullsen, S.J. Eggers and H.M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism”, in Proc. of the 22nd Int. Symp. on Computer Architecture, pp. 392-403, 1995

[27] N. Vachharajani, R. Rangan, E. Raman, M. Bridges, G. Ottoni, D. August, “Speculative Decoupled Software Pipelining”, Procs. of Conference on Parallel Architecture and Compilation Techniques, p. 49-59, 2007

[28] A. Zhai, C. B. Colohan, J. G. Steffan, T. C. Mowry, “Compiler Optimization of Scalar Value Communication Between Speculative Threads”, in Procs. of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 2002

[29] H. Zhong, S. A. Lieberman, and S. A. Mahlke, “Extending multicore architectures to exploit hybrid parallelism in single-thread applications”. In Intl. Symp. On High-Performance Computer Architecture, Phoenix, Arizona, February 2007

[30] C.B. Zilles and G.S. Sohi, “Understanding the backward slices of performance degrading instructions”, in Proc. of the 27th Int. Symp. on Computer Architecture, 2000

[31] C.B. Zilles and G.S. Sohi, “Execution-Based Prediction Using Speculative Slices”, in Proc. of the 28th Int. Symp. on Computer Architecture, 2001

[32] C.B. Zilles and G.S. Sohi, “Master/Slave Speculative Parallelization”, in Proc. of the 35th Int. Symp. on Microarchitecture, 2002