

Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices

Carlos García Quiñones[†], Carlos Madriles[†], Jesús Sánchez[†],
Pedro Marcuello[†], Antonio González[†] and Dean M. Tullsen^{‡1}

[†] Intel Barcelona Research Center
Intel Labs, Universitat Politècnica de Catalunya, Barcelona

[‡] Dept. of Computer Science and Engineering
University of California, San Diego

e-mail: {carlos.garcia.quinones, carlos.madriles.gimeno, f.jesus.sanchez,
pedro.marcuello, antonio.gonzalez}@intel.com, tullsen@cs.ucsd.edu

Abstract

Speculative parallelization can provide significant sources of additional thread-level parallelism, especially for irregular applications that are hard to parallelize by conventional approaches. In this paper, we present the Mitosis compiler, which partitions applications into speculative threads, with special emphasis on applications for which conventional parallelizing approaches fail.

The management of inter-thread data dependences is crucial for the performance of the system. The Mitosis framework uses a pure software approach to predict/compute the thread's input values. This software approach is based on the use of pre-computation slices (p-slices), which are built by the Mitosis compiler and added at the beginning of the speculative thread. P-slices must compute thread input values accurately but they do not need to guarantee correctness, since the underlying architecture can detect and recover from misspeculations. This allows the compiler to use aggressive/unsafe optimizations to significantly reduce their overhead. The most important optimizations included in the Mitosis compiler and presented in this paper are branch pruning, memory and register dependence speculation, and early thread squashing.

Performance evaluation of Mitosis compiler/architecture shows an average speedup of 2.2.

Categories and Subject Descriptors C.1.4 [Processor Architectures]: Parallel Architectures, D.3.4 [Programming Languages] Processors – compilers, code generation, optimization.

General Terms Performance, Design

Keywords Speculative multithreading; thread-level parallelism; automatic parallelization; pre-computation slices.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.
Copyright 2004 ACM 1-59593-056-6/05/0006...\$5.00.

1. Introduction

Several microprocessor vendors have recently introduced single chip architectures that can execute multiple threads in parallel, exploiting *thread-level parallelism*. Two different approaches have been used to architect these systems: simultaneous multithreading [25][7] and multiple cores [24][22][16]. These architectures increase throughput by executing independent jobs in parallel, or reduce execution time by parallelizing applications. This latter case has proved to be successful for regular numerical applications, but less so for non-numerical, irregular applications, for which the compiler usually fails to discover a significant amount of thread-level parallelism.

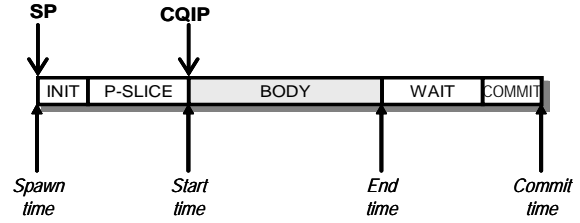
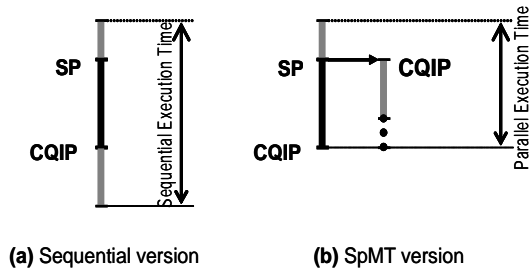
Speculative multithreading (SpMT for short) attempts to speed up the execution of applications through speculative thread-level parallelism. Threads are speculative in the sense that they may be data and control dependent on previous threads (that have not completed) and their execution may be incorrect.

There are two main strategies for speculative thread-level parallelism: (1) use helper threads to reduce the execution time of high-latency instructions/events through side effects, and (2) parallelize applications into speculative parallel threads, each of which contributes by executing a part of the original application.

Helper Threads [6][5][18][27] attempt to reduce the execution time of the application by using speculative threads to reduce the cost of high-latency operations (such as load misses and branch mispredicts). For instance, in [5][27] this is done by executing a subset of instructions from the original code to pre-compute load addresses or branch directions. Instructions executed by speculative threads do not compute/modify any architectural state of the processor, and thus, all architectural state must still be computed by the main, conventional thread.

With speculative parallelization ([10][1][13] among others), each of the speculative threads executes a different part of the program. This partitioning is based on relaxing the parallelization constraints and allowing the spawning of speculative threads even when the compiler cannot guarantee correct execution. When a speculative thread finishes, the speculation is verified. Unlike helper threads, the values produced by a speculative thread are

¹ This work was done while he was a visiting professor at Intel Barcelona Research Center



committed. In case of a misspeculation (control or data), the work done by the speculative thread is discarded. The Mitosis architecture used in this work follows this approach.

A potential speculative thread is defined by a *spawning pair*, which consists of the point in which the spawning instruction is inserted and the point where the speculative thread will start execution when it is spawned. Prior work has demonstrated that well chosen speculative threads (or spawning pairs) can result in significant speedups [15]. Conversely, a poorly chosen pair can hurt performance.

A key point in any SpMT system is how to deal with inter-thread data dependences. Two mechanisms have been studied so far: (1) synchronization mechanisms and (2) value prediction. The synchronization approach imposes a high overhead when dependences are frequent, as in the workload presented here. Value prediction has more potential – if the values that are computed by one thread and consumed by another can be predicted, the consumer thread can be executed in parallel with the producer thread since these values are only needed for validation at a later stage. It is typically assumed that these value predictions are computed in hardware. The Mitosis system presents a novel approach, which adds code (derived from the original program) to predict in software the live-ins (values consumed, but not produced by, the thread) for each speculative thread. Because mechanisms for recovery of incorrect threads are already in place, the code to produce the values need not always be correct, and can be highly optimized. We refer to this code as pre-computation slices (p-slices). The main advantages of p-slices are: (1) they are potentially more accurate in the prediction of live-ins than a hardware-based predictor, since it is derived from the original code, (2) they can encapsulate multiple control flows that contribute to the prediction of live-ins, and (3) they can accelerate the detection of incorrectly spawned threads.

In this work, we present the Mitosis compiler. One objective of the Mitosis compiler is to identify the most effective points in any program to spawn speculative threads. This entails, among other tasks, locating regions of code where the live-ins can be accurately predicted through p-slices with a low overhead.

The Mitosis compiler provides the following features: (1) it identifies effective spawning pairs, (2) generates pre-computation slices, (3) optimizes pre-computation slices to minimize their overhead, and (4) maximizes the accuracy of pre-computation slices. The main contributions of this work are:

- A general compilation framework to analyze and insert spawning pairs at any point of any program.
- The use of pre-computation slices to predict values corresponding to inter-thread dependences.

- A mechanism to build and optimize (in terms of both accuracy and overhead) pre-computation slices.
- A model to estimate the benefit of any set of spawning pairs for a given SpMT configuration, and a scheme to select the most effective set.

Evaluation of the proposed compilation technique shows very encouraging results. Performance results reported for the Mitosis compiler show a speedup of about 2.2 for a subset of the Olden benchmark suite. This is code that state-of-the-art parallelizing compilers/architectures *cannot parallelize*.

The rest of the paper is organized as follows. Section 2 describes basic concepts of the Mitosis SpMT architecture that are relevant to the Mitosis compiler. In Section 3, the Mitosis compiler infrastructure is presented. The scheme to build and optimize speculative pre-computation slices is further detailed in Section 4. Section 5 evaluates the Mitosis compiler. Finally, Section 6 discusses some related work and Section 7 summarizes the main conclusions of this work.

2. Overview of the Mitosis Architecture

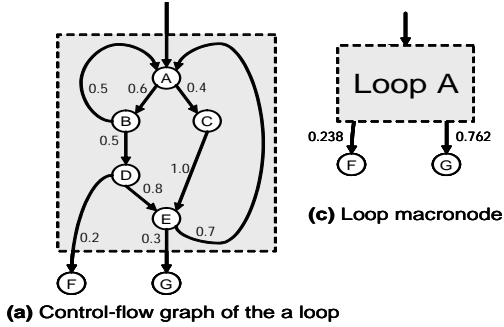
This section presents an overview of the underlying speculative multithreaded architecture, with special emphasis on those features that are relevant to the compiler for generating effective speculative threads.

2.1. Features

The Mitosis SpMT architecture is composed of several *thread units* (TUs), each able to execute a thread. A thread unit contains its own register file, instruction/data caches, functional units and program counter. This means that the compiler can see a thread unit as an independent entity of execution. The thread units can be organized in various ways. For instance, one can implement this architecture in a SMT (Simultaneous Multithreaded) or a CMP (Chip Multiprocessor) fashion. For this study, we assume a CMP-like design for simplicity and scalability.

2.2. Execution Model

In the Mitosis architecture there is always one (and only one) non-speculative thread, which is the only one allowed to commit its results. All other threads are speculative. A speculative thread is created when a `spawn` instruction is found. A `spawn` instruction basically describes a spawning pair. A spawning pair is defined as a set of two points in the program (each at the beginning of a basic block). The former is called the *spawning point* (SP), marked by the `spawn` instruction, and identifies *when* a new speculative thread is created. The latter is called the *control quasi-independent point* (CQIP) and represents *where* the speculative thread starts executing (after some initialization), and is identified as an operand of the `spawn` instruction. Figure 1 shows the



TYPE	PATH	PROB	NEXT
Continue	{A B}	0.300	A
Continue	{A B D E}	0.168	A
Break	{A B D E}	0.072	G
Break	{A B D}	0.060	F
Continue	{A C E}	0.280	A
Break	{A C E}	0.120	G

(b) Loop paths

difference between sequential and SpMT parallel execution of code.

Any thread can spawn a new thread. The requirements to spawn a thread are: (i) there is a free thread unit, or (ii) there is at least one running thread more speculative (further in sequential time) than the thread is to be created. In the latter case, the most speculative thread is cancelled and the freed thread unit is assigned to the spawned thread.

Correct execution of the code following the CQIP requires that the future state (memory and register values) of the processor at the CQIP is correctly predicted. Our SpMT model differs from prior proposals in that we assume that each speculative thread includes a *pre-computation slice* (p-slice for short) that computes the live-ins of the thread. A speculative thread has two operation modes, depending on whether it is executing code from the pre-computation slice or the body of the thread. In particular, data produced while in p-slice mode is stored in a special buffer (called the *slice buffer*) and will be used as input for the body of the speculative thread. Data produced by the speculative thread is kept in the regular structures of the thread unit (register file and memory) and will be committed once the thread becomes non-speculative. This distinction arises from the fact that the p-slice only predicts machine state, while the speculative thread body (following the p-slice) calculates actual machine state; thus, data produced by the p-slice must be confirmed, but never committed.

Threads commit in sequential order. This means that a thread must wait to become the oldest one (*i.e.*, the non-speculative thread) to commit. A thread finishes when it reaches the start (CQIP) of another thread that is active. Then, the latter thread is validated. If the validation is correct, the latter thread is allowed to commit once it is the oldest thread. Otherwise, it is squashed (as well as its successors) and the former thread proceeds to execute the instructions beyond the CQIP. Figure 2 shows a scheme with the different stages in the life of a speculative thread.

3. Selecting Spawning Pairs

The Mitosis compiler for this architecture performs the following tasks: (1) generate the p-slices for each pair, (2) optimize the p-

slices to minimize overhead, and (3) select the best candidate spawning pairs. These tasks are heavily inter-related; however, we will discuss each separately. This section describes the selection of spawning pairs. Generation and optimization of p-slices are described in Section 4. The proposed scheme has been implemented in the code generation phase (after optimizations) of the ORC compiler [11]. The compiler makes use of the information provided by an edge profile. This information includes the probability of going from any basic block to each of its successors and the execution count of each basic block.

Selecting the best set of spawning pairs requires assessing the benefit of any given candidate pair. However, determining the benefits of a particular spawning pair is not straightforward. The effectiveness of a pair depends on the control flow between the spawning point and the start of the thread, the control flow after the start of the thread, the accuracy of the p-slice, the overhead of the p-slice, the number of hardware contexts available to execute speculative threads, and interactions with other speculative threads running at the same time.

This analysis requires a model of program execution. To avoid capturing and repeatedly traversing a full path trace of the program, we generate a (much smaller) synthetic trace of execution that captures the dynamic behavior. The key idea is to traverse this trace while keeping track of the threads that are active at any time. For each thread, its state (see Figure 2) is maintained to emulate its evolution during its lifetime. This analysis emulates the timing behavior of the speculative threads, assuming a simple model where each instruction takes a fixed time. Based on this, the compiler can estimate the expected benefits of any set of spawning pairs, and select those that are expected to minimize total execution time.

3.1. Building the Synthetic Trace

We build a synthetic trace of the program to translate the edge profile into path information, without having to capture and maintain a full path profile. The synthetic trace is built based on edge profiling information at the basic block level. The analysis performs a reverse topological traversal of the call graph. This means that callee routines are analyzed before callers.

For each routine, we compute a set of paths. A routine path is defined as a list of connected nodes in the CFG from the entry node to an exit node. We assume here that a routine has only one entry. A path node can be one of these types: basic block (BB), loop, or call. Loop and call nodes are macronodes in the sense that they include more than a single basic block. A loop node consists of the header of a given loop and contains all the nodes (basic blocks, inner loops or calls) that belong to that loop. A call node is just a basic block that ends with a call to a particular function. The exit node of a routine path could be either a return (node with no successor) or a call to the exit function. Thus, there are two types of routine paths: return paths and exit paths. Each routine path is characterized by its total length (in instructions) and its probability (using the edge profiling information). This is summarized in the following expressions:

$$\begin{aligned}
 \text{ROUTINE} &= \text{SET OF } \{ \text{ROUTINE PATH} \} \\
 \text{ROUTINE PATH} &= \text{LIST OF } \{ \text{NODE} \} + \text{TYPE} + \text{LENGTH} + \text{PROB} \\
 \text{ROUTINE PATH TYPE} &= \{ \text{RETURN} \mid \text{EXIT} \} \\
 \text{NODE} &= \{ \text{BB} \mid \text{LOOP} \mid \text{CALL} \}
 \end{aligned}$$

A loop node requires more analysis. For each loop, we compute a set of loop paths. A loop path is defined as a sequence of nodes in the CFG of the loop from the head of the loop to a possible loop

```

[ 1] t_exec = SeqExecTime;
[ 2] Selected_Pairs = ∅;
[ 3] exit = FALSE;
[ 4] while (!exit) {
[ 5]   select = NULL;
[ 6]   for (cand=First_Cand(Candidate_Pairs); cand; cand = Next_Cand(cand)) {
[ 7]     Analyzed_Pairs = Selected_Pairs + cand;
[ 8]     t_exec_tmp = Model_Set_of_Pairs(Analyzed_Pairs, Trace, N_TUs);
[ 9]     if (t_exec_tmp < t_exec) {
[10]       t_exec = t_exec_tmp;
[11]       select = cand;
[12]     }
[13]   }
[14]   if (select == NULL)
[15]     exit = TRUE;
[16]   else {
[17]     Candidate_Pairs = Candidate_Pairs - select;
[18]     Selected_Pairs = Selected_Pairs + select;
[19]   }
[20] }

```

exit node. A loop exit node can be a node with an edge to the loop head (continue path), an edge outside the loop (break path) or a call to a function that may call the exit function (exit path). We are assuming here that a loop has a single header. As in the case of routines, each loop path has its length (in instructions) and probability. This is summarized with the following expressions:

$LOOP\ PATH = LIST\ OF\ \{ NODE \} + TYPE + LENGTH + PROB$
 $LOOP\ PATH\ TYPE = \{ CONTINUE \mid BREAK \mid EXIT \}$

Next, a set of synthetic traces are built for each loop. A loop synthetic trace consists of selecting $NITER - 1$ loop paths of type continue and 1 loop path of type break or exit, where $NITER$ is the average trip count. For large loops, we build the trace assuming a fixed maximum value for the trip count. The rest of the iterations are considered to have the same behavior as the analyzed ones. The selection of the paths inside the loop consists of a random weighted model according to each path probability.

The number of loop synthetic traces that are built per loop is a parameter of the tool and can be adjusted depending on time and memory space requirements. We call each loop's synthetic trace a loop instance. For each loop node the compiler keeps a list of instances and a pointer to one of them (initially the first one), that is updated in a circular fashion.

$LOOP = CIRCULAR\ LIST\ OF\ \{ LOOP\ INSTANCE \} + POINTER + TRIP\ COUNT + LENGTH$
 $LOOP\ INSTANCE = LIST\ OF\ \{ LOOP\ PATH \}$

The length of a loop node is computed as the weighted average length of the loop instances. Figure 3 shows an example of how a given loop is split in its different paths and the shape of the resultant loop macronode.

A call node also has attached a list of instances. A call instance refers to a possible path in the callee routine. The compiler builds this list of instances, similarly to loops, by randomly selecting (based on the probabilities) routine paths from the callee routine. In the same way, a pointer is also attached.

$CALL = CIRCULAR\ LIST\ OF\ \{ CALL\ INSTANCE \} + POINTER + LENGTH$
 $CALL\ INSTANCE = ROUTINE\ PATH$

The length of a call node is computed as the weighted length of routine paths in the callee function.

The synthetic trace will be traversed by starting the path at the main routine of the program. As the main routine is called only

once, there is only one routine path for that function. When a loop node is found, the traversal proceeds through the loop instance pointed by the loop instance pointer and the pointer is set to the next instance. In the case of a call node, the traversal proceeds through the path in the callee routine described by the call pointer and the pointer is set to the next one in the same way as for loops. When an exit path (or the only path in the main routine) is completely traversed, the program traversal finishes.

3.2. Candidate Pairs

A key feature of the proposed compilation tool is its generality, in the sense that it can discover speculative thread-level parallelism in any region of the program. The tool is not constrained to analyze potential spawning pairs at loop or subroutine boundary, but practically any pair of basic blocks is considered a candidate spawning pair. To reduce the search space we first apply the following filters to eliminate candidate pairs that are likely to have little potential:

- 1) Spawning pairs in routines whose contribution to the total execution of the program is lower than a threshold are discarded.
- 2) Both basic blocks of the spawning pair must be located in the same routine and at the same loop level.
- 3) The length of the spawning pair (as the average length of all the paths from the SP to the CQIP) must be higher than a certain minimum size in order to overcome the initialization overhead when a speculative thread is created. It must also be lower than a certain maximum size in order to avoid very large speculative threads and avoid stalls due to the lack of space to store speculative state.
- 4) The probability of reaching the CQIP from the SP must be higher than a certain threshold.
- 5) Finally, the ratio between the length of the p-slice and the estimated length of the speculative thread must be lower than a threshold. This ratio is a key factor for the benefits of the thread. In Section 4 we describe in detail how the pre-computation slices are built and optimized to reduce this overhead.

This step analyzes the different routines in the program one by one. For each routine, all combinations of basic blocks are considered and passed through the different filters. The result of

CQIP is reached); otherwise, we mark it as a `NORMAL` thread. In either case we look for a thread unit. If a thread unit is free we assign it to the new thread. Otherwise, we check if the most speculative thread (the one whose next thread is `NULL`) is further in program order (more speculative) than the new thread. If so, the most speculative thread is cancelled and the freed unit is allocated to the new thread. Otherwise, the spawn is discarded.

When a basic block that corresponds to a CQIP in the set of pairs is found, it is checked whether any more speculative active thread was started at this basic block instance. If this is the case, the current thread is terminated and current thread and current time variables are updated accordingly.

Finally, the last basic block of the trace just terminates the current thread. The commit time of this last thread represents the SpMT execution time of the program.

Spawning a new thread requires the following actions:

- 1) Identify the order of this new thread with respect to the current ones. Its previous thread is the thread that contains the CQIP of the new thread (i.e., the linked list of threads starting at the current thread is traversed until the first that contains the CQIP is found). Its next thread is the thread that was the successor (before spawning) of its previous thread.
- 2) Decide whether this is a `CANCEL` thread or a `NORMAL` one: this is randomly selected based on the cancel probability of each type for this particular thread.
- 3) Record the start and end basic blocks of the thread. The former is the current CQIP and the later is the start of the next thread.

Finally, canceling a thread requires the following actions: (1) identify previous and next threads, and (2) update links and end information of the previous thread.

4. Speculative P-Slices

This architecture handles inter-thread dependences through the execution of a pre-computation slice inserted at the beginning of every speculative thread. The goal of the p-slice is to calculate the live-ins of the new speculative thread very quickly. Regardless of the code we generate for a given p-slice, the architecture guarantees a functionally correct execution of the program. Thus accuracy of p-slices only affects performance, not correctness. This is a key observation, since it allows the compiler to perform aggressive, unsafe optimizations when generating these p-slices.

The steps to build a p-slice for a given spawning pair are:

- 1) Identify live-ins.
- 2) Generate conservative p-slice.
- 3) Optimize the p-slice.

4.1. Identifying Live-ins

Identifying the live-ins of a speculative thread requires a top-down traversal of its control-flow graph starting at the CQIP to identify register and memory values read before being written by the speculative thread. Each path is explored until a certain length. This length represents the time that previous threads take to compute and commit these values. This is because once the previous thread commits, the speculative thread need no longer rely on predicted values, but can read committed values. This time is estimated as the time it takes to sequentially execute all the code between the SP and CQIP minus the thread spawn overhead.

4.2. Generating Conservative P-Slices

The p-slice for a spawning pair is built by traversing the control-flow graph backwards from the CQIP until the SP. The input to that step is the set of live-ins, both register and memory values. The first instructions included in the slice are those that directly produce the thread live-ins. Then the process inserts ancestors of these instructions, taking into account both data and control dependences, provided that they are below the SP.

Initially, all dependences among instruction as given by the compiler in the conventional, conservative way are considered. This means that the only reason why a p-slice may be incorrect is if not all the live-ins of the thread are being considered (e.g. the length of the thread, as described in the previous section, may have been incorrectly estimated).

Upon finding a call to a subroutine, the side-effects of that subroutine as well as the use of the returned value(s) are analyzed, and if there is any dependence, the call instruction to the subroutine is included into the slice. This means that the slice includes the whole subroutine, although the full code of that subroutine may not be needed. A possible optimization (not considered in the results presented in this paper) would be the inlining or specialization of some functions.

4.2.1. Early Cancellation

A safe optimization that we have implemented is called *early cancellation*. Starting from the SP, we can analyze whether a path in the control-flow graph will reach the CQIP or not. In the latter case, the thread would keep executing useless instructions (wasting power and keeping a thread unit busy) until it is squashed by a less speculative thread. A reachability analysis of the CFG from the SP to the CQIP is used to identify the points in the program where we can guarantee that the CQIP will never be reached. The compiler inserts a `cancel` instruction in each of these points, which will squash the thread when executed.

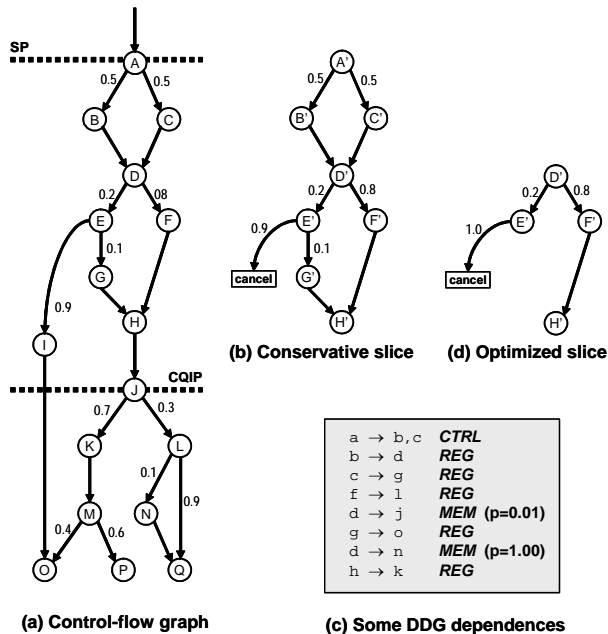
4.3. Speculative Optimizations

The p-slices built using the conservative assumptions of the compiler, as described above, are normally very large. Large p-slices significantly constrain the benefits of speculative threads. However, a key feature of the Mitosis SpMT architecture is that it can detect and recover from misspeculations. This opens the door to new types of aggressive/unsafe optimizations that otherwise could not be applied by the compiler, and which have the potential to significantly reduce the overhead of p-slices. In the following subsections, we describe the set of speculative optimizations currently included in the Mitosis compiler.

Speculative optimizations require a new factor in the analysis: the *misspeculation probability*. This factor represents the probability that a given p-slice is incorrect. This happens when some live-ins are not computed or they are incorrect. This probability is attached to each candidate pair and used by the model described in Section 3.3.1 when deciding whether a spawned pair is `NORMAL` or `CANCEL`.

4.3.1. Memory Dependence Speculation

Modern compilers often fail to parallelize applications because of ambiguous memory dependences. Many memory dependences are only included because the compiler cannot prove that the corresponding instructions are independent, but in fact they are. In many other cases, two static instructions do have a memory



dependence, but this dependence only happens for a very few dynamic instances of these instructions.

We have implemented a memory dependence profile to minimize the number of unnecessary dependences considered when generating p-slices. The profiler computes the dependence frequency between any pair of store-load, store-call, call-load or call-call instructions for each routine (the SP and CQIP of a spawning pair are in the same routine). Dependences for calls refer to dependences due to any memory reference inside the called routine.

The compiler only considers that two instructions have a dependence whenever this dependence has happened with a frequency above a given threshold. In other words, dependences that never occur in practice, or occur very infrequently, are discarded.

4.3.2. Branch Pruning

Branch pruning consists of ignoring those paths that exhibit low probability of being taken when generating p-slices. These paths may belong to either the body of a speculative thread or its p-slice, with different consequences in each case.

Pruning branches of the body of speculative threads is done during the process of identifying the live-ins of the threads (Section 4.1). A pruned branch is still included in the thread body code, but any live-in in the pruned path is ignored when the p-slice is generated, which reduces the size of the p-slice.

On the other hand, pruning a branch in the p-slice removes all the instructions of the pruned path from the p-slice. Additionally, predecessors of these removed instructions are also removed if their output is not used elsewhere. In the place of a pruned path, a `cancel` instruction is inserted; if this path happens to be taken, the thread input values will likely be miscomputed, and it is preferable to cancel the thread and free this hardware context for

another thread. We call this optimization *speculative early cancellation*.

Examples of both types of branch pruning are shown in Figure 6. On the left, (a) shows a control-flow graph that includes a SP and a CQIP. Each edge is annotated with its probability of being taken (edges without label have probability 1.0). On the right (b), the control-flow graph of the conservative p-slice is shown. Basic blocks are labeled with prime letters to indicate that they contain just a subset of the instructions of the original basic blocks. Some data dependences among instructions in some basic blocks are also listed in Figure 6(c) (lower case letters represent instructions in basic blocks with the corresponding capital letter, e.g., instruction ‘a’ is in basic block ‘A’). A possible edge (i.e., branch) to be pruned in the speculative thread is L→N, which will remove a live-in (data dependence d→n) and then some instructions in the slice (dependence b→d is not needed). On the other hand, an example of pruning in the p-slice would be for edge E→G, which will remove the data dependence g→o. This will remove in turn the need for dependence c→g in the p-slice. In this example, as no instructions are needed from basic blocks B and C (since their dependences have been removed), the control dependence a→b,c can also be removed from the slice. The resulting optimized p-slice is shown in Figure 6(d).

4.3.3. Dependence Pruning

Data dependences that are infrequent can also be ignored. For memory dependences, the profiler described in Section 4.3.1 is used for this purpose. In the case of register dependences, the probability of reaching the producer once the spawn has been executed is computed and multiplied by the probability of reaching the consumer after executing the producer. Note that a consumer can be located either in the slice or the speculative thread body. As in the case of memory dependences, if this probability is lower than a threshold, the dependence is ignored for the purpose of generating the final p-slice.

4.3.4. Cancel Elimination

As previously discussed, cancel instructions are inserted at points where the compiler can guarantee that the speculative thread is incorrect or the flow cannot reach the CQIP. This allows the processor to squash early a speculative thread in order to free the thread unit for other threads. However, this means that the branch instruction leading to the pruned code must be preserved (and all its ancestors in the dependence graph). This overhead may be large in some cases, which significantly impacts the effectiveness of the speculative thread. In these cases, it may be more effective just to remove the cancel operation and the associated branch instruction (which will also remove some of its ancestors). This will make the slice always follow the frequent path, which can be incorrect in some infrequent cases. The architecture will still detect these misspeculations, and squash the thread.

5. Experimental Evaluation

5.1. Framework

The Mitosis compiler has been implemented on top of the ORC compiler [11] to generate IPF code. The performance of the Mitosis compiler/architecture has been evaluated through a detailed, execution-driven microarchitectural simulator built on top of SMTSIM [25]. The modeled Mitosis processor is a research Itanium® CMP processor with 4 hardware contexts. Each hardware context is a 6-way issue, in-order core. The main

parameters of the processor configuration are shown in Table 1. The figures in the table are per thread unit.

Table 1. Mitosis processor configuration

Fetch, in-order issue and commit bandwidth	2 bundles (6 instructions)
I-Cache	64KB
L0-Cache	4-way associative 16KB – hit latency: 1 cycle
L1-Cache	4-way associative 1MB – hit latency: 4 cycles
L2-Cache (share)	4 way associative 8 MB – hit latency: 8; miss latency: 250
Local Register File	Latency = 1 cycle
Remote Register File	Latency = 6 cycles
Spawn overhead	5 cycles
Validation overhead	15 cycles
Commit overhead	5 cycles

To evaluate the potential performance of the Mitosis architecture, a set of non-automatic parallelizable codes have been used. These benchmarks correspond to a subset of the Olden benchmark suite. The benchmarks used are *bh*, *em3d*, *health*, *mst* and *perimeter*. We have used a train input for profiling of around 10M instructions per benchmarks, and a different input set that on average executes around 300M instructions for the simulation. Statistics in the next section correspond to the whole execution of these programs. The rest of the suite has not been considered due to the recursive nature of the programs. The Mitosis compiler is not currently able to extract speculative thread-level parallelism in recursive routines. This feature will be targeted in future work.

The ORC compiler has been used with full optimizations enabled (-O3) except software pipelining and if-conversion. For the Mitosis optimizations, we have considered a 5% threshold for dependence pruning and 15% for branch pruning. We have not paid special attention to the compilation time. Our first attempt at using filters to trim the search space is very promising, but timing aspects require further work in the future.

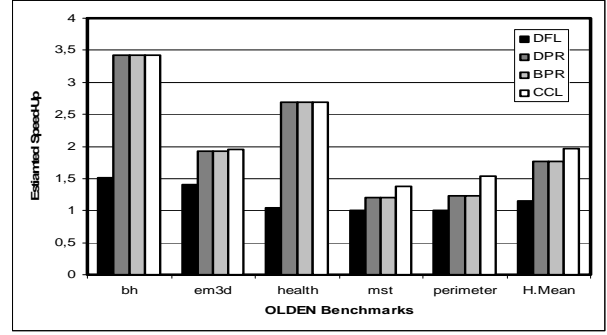
Olden benchmarks have been chosen since they are pointer intensive programs for which automatic parallel compilers are unable to extract thread-level parallelism. To corroborate this, we have compiled the Olden suite with the Intel® C++ production compiler which produces parallel code. Almost no part of the code was parallelized for any benchmark.

5.2. Results

The first results we will show focus on the benefit of the proposed optimization for the p-slices. For that, we will use a metric that we call *average benefit per pair*. It is an approximation of the number of parallelized instructions by each instance of the pair. The expected benefit of a single pair is computed as follows:

$$\begin{aligned} \text{Overlap} &= \text{PairLength} - (\text{SliceLength} + \text{Init}) \\ \text{ProbCorrect} &= (1 - \text{Cancel}) * (1 - \text{Misspec}) \\ \text{Benefit} &= \text{Overlap} * \text{Count} * \text{ProbCorrect} \end{aligned}$$

PairLength and *SliceLength* show the average length of the pair and the slice, respectively. *Init*, as shown in Figure 2, represents the latency of the spawn instruction. *Cancel* shows the probability that the slice is cancelled, and *Misspec* is the probability that the slice is incorrect due to speculative optimizations. Finally, *Count*



is the number of times the spawning instruction is executed. From those expressions, the average benefit per pair is computed as:

$$\text{AvgBenefitPerPair} = \text{SUM}(\text{Benefit}_i) / \text{SUM}(\text{Count}_i), \text{ for all } \text{PAIR}_i$$

To quantify the effect of the different optimizations applied to p-slices, Table 2 shows the average benefit per pair for all pairs after filtering (that is, the set of candidate pairs that will be later considered by the pair selection scheme). We show in the different columns the proposed metric without any speculative optimization (DFL), after dependence pruning (DPR), after branch pruning (BPR) and finally after cancel elimination (CCL). Each optimization is added on top of previous ones.

Table 2. Benefit of p-slice optimizations on all candidate pairs

Default	Dependence pruning	Branch pruning	Cancel elimination
1.9	106.5	106.5	287.6

We can observe that all optimizations significantly improve the quality of p-slices, with the exception of branch pruning. However branch pruning is necessary for cancel elimination, which is shown to be quite effective.

Figure 7 shows the estimated speedup (using the model proposed in Section 3.3.1) for the different optimizations. As in the previous table, each optimization is applied on top of the previous one. We can observe that the improvement in the p-slice overheads shown in Table 2 actually translate into speedup. We can observe that, on average, the expected speedup grows from 1.15x without optimizations up to almost 2x when all optimizations are set.

Table 3 presents some statistics of the speculative threads generated by the Mitosis system with fully optimized p-slices. The last row shows the arithmetic mean for the evaluated benchmarks. The second column shows the number of spawned threads by benchmark and the second column the average number of speculative instructions executed by speculative threads. It can be observed that *bh* spawns the fewest threads but their average size is about 30 times larger than for the rest of the benchmarks. On the other hand, *mst* spawns the most but the average size of its speculative threads is the lowest. The fourth column shows the average dynamic size of the p-slices and the fourth column the relationship between the sizes of the speculative threads and their corresponding p-slices. This percentage is consistently quite low for all the studied benchmarks and on average represents less than 3%. The sixth column shows the average number of thread input

values that are computed by the p-slice, which is only three values on average. The seventh column represents the percentage of threads that are squashed. This percentage is rather low for all the benchmarks except for *health*, for which about one out of every four threads is squashed. We have observed that for this particular benchmark, memory dependences for the profiling and simulated inputs are significantly different, which result in many memory dependence misspeculations.

Finally, the right-most column shows the degree of speculative thread-level parallelism that is exploited by Mitosis. This column represents the average number per cycle of active threads that are executing correct code. It can be observed that, even though parallelizing compilers cannot find parallelism in these benchmarks, there is still a high degree of thread-level parallelism that is exploited by the Mitosis compiler/architecture. On average, the number of active and correct threads per cycle is slightly higher than 2.5.

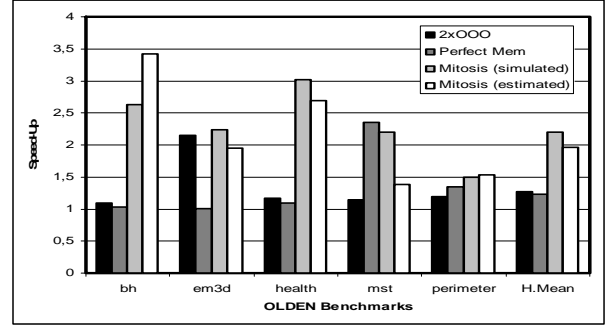
Table 3. Characterization of the Olden benchmarks

OLDEN	Spawned Threads	Thread Size	Slice Size	Slice / Thread	Thread Liveness	Squash Pctg	Active Threads/cycle
<i>bh</i>	422	15543	196	1.3%	4.4	0.7%	2.68
<i>em3d</i>	396638	422	9	2.1%	1.0	0.3%	2.87
<i>health</i>	198497	1112	41	3.7%	2.7	26.9%	2.35
<i>mst</i>	1367114	271	5	2.1%	2.3	0.8%	2.62
<i>perimeter</i>	493725	576	24	4.2%	3.6	1.0%	2.08
MEAN	491279	3585	55	2.7%	2.8	6.0%	2.52

Figure 8 shows the performance of the Mitosis processor compared to other single- and multi-threaded architectures. Performance is reported as speedup over execution on a single Mitosis thread unit. The compared architectures are: a) an out-of-order superscalar processor, with twice the resources of an in-order Mitosis thread unit, and b) a single thread unit with perfect memory. This represents an upper bound on the performance that can be achieved by helper threads that target memory latency [5].

The main conclusion of this study is that the Mitosis system is very effective at exploiting thread-level parallelism for irregular applications. An average speedup of 2.2 is observed, and significant speedup is achieved for all benchmarks. It can be observed that the Mitosis processor clearly outperforms the other architectures. Average speedups for the big out-of-order core and perfect memory are 1.26 and 1.23 respectively.

The last bar in each group of bars in Figure 8 shows the speedup estimated by our model with full optimizations for the selected pairs. In three of the five benchmarks (*em3d*, *health* and *perimeter*) the speedup predicted by the model is relatively close to that of the simulation. In the case of *mst* we have observed that the difference is due to many high-latency instructions. Note that for *mst* the Perfect Memory scheme performs better than Mitosis. We expect that including a more accurate latency for each instruction in the model (instead of the fixed 1-cycle currently assumed) will significantly improve performance in these cases. In the case of *bh*, the main source of discrepancy between simulated and estimated speedups are due to the use of average lengths to estimate the timing of the p-slices and speculative



threads. We have observed that for this program, these lengths experience a significant variability, and thus, the selected threading scheme is not optimal for the cases that significantly depart from the average.

Looking at particular benchmarks, we can observe that the big out-of-order core is comparable to Mitosis only for *em3d*. This is due to the fact that this program has abundant ILP, which could also benefit more aggressive configurations of Mitosis, for instance based on out-of-order cores. Perfect memory is comparable to Mitosis only for *mst*. For this program, the performance of the memory system is rather poor; for a single-threaded execution, the L0 and L1 miss ratios are around 50% and 70% respectively. This clearly points out that memory is the main bottleneck for this program, and any technique that tries to accelerate it should focus on memory. Obviously, perfect memory attacks this problem but the results show that Mitosis solves it effectively too.

To summarize, we find the Mitosis architecture and compiler to represent a highly flexible parallel architecture. Whether the code contains traditional thread-level parallelism (not shown in these benchmarks, but easily handled by this system), instruction-level parallelism (*em3d*), or memory-level parallelism (*mst*), Mitosis exploits it effectively. Additionally, codes that exhibit none of the above also experience high speedups.

6. Related Work

Several speculative multithreaded architectures have been proposed, along with hardware and compiler techniques to extract speculative threads. In this section we review the main works, with regard to the schemes used to identify speculative threads and to manage inter-thread data dependences, which are the topic of this paper.

The Expandable Split Window Paradigm [10] and the follow-up work, the Multiscalar processor [19][26] were pioneering works in the area of SpMT. Speculative threads (called tasks) are created by the compiler based on several heuristics that tried to minimize the data dependences among threads as well as maximize the workload balance, among other compiler criteria. The process consists of walking the control-flow graph and accumulating basic blocks into tasks. Inter-thread data dependences are managed differently depending on whether they are through memory or registers. For register dependences, the compiler is responsible for detecting the instruction that performs the last write on this register in order to bypass the value to the consumer thread. Memory dependences are handled through the ARB mechanism.

Several studies propose architectures and schemes to create speculative threads based on well-known program constructs such as loop iterations, loop continuations and subroutine continuations ([23][9][1][4][13][2][20][17] among others) either through hardware or software mechanisms. The Superthreaded [23] and the SPSM [9] system are two examples where the loop parallelization task is performed by the compiler.

These schemes differ in the mechanism used to deal with inter-thread data dependences. Most detect memory dependence violations based on modifications of traditional snoop-based cache coherence protocols. Memory data value prediction has also been proposed, but these values usually show lower predictability [3][21]. Register dependent values are either synchronized or hardware predicted. Some compiler-based schemes, such as the Superthreaded architecture [23], reorder the code in order to compute the dependent value earlier.

Du *et al.* [8] have recently proposed a cost-driven compilation framework that statically determines which loops are good candidates to parallelize. They compute a cost graph from the CFG and DDG and estimate the probability of misspeculations. Inter-thread data dependences are handled by moving producer instructions before the spawn of the next iteration.

A more complex scheme to partition the program into speculative threads is presented in a recent work [15]. That scheme is based on profiling information. As in Mitosis, any combination of basic blocks is considered a candidate spawning pair. In that work, inter-thread data dependences are handled by means of hardware value prediction. Our work differs from that in the fact that we use a software approach to predict thread inputs, which implies a significantly different microarchitecture, and the compiler support presented in this paper.

The use of Helper Threads, which speculatively execute a subset of the original code to reduce the latency of high-cost instructions, has been thoroughly studied [5][12][18][27]. This research borrows some concepts from that body of work to create the pre-computation slices for thread live-ins. However, the need of Mitosis to pre-compute a set of values accurately (as opposed to a single load address or branch result), and an increased cost of misspeculation, requires significantly more careful creation of slices, and the inclusion of more accurate control flow in the slice – previous work on helper threads typically followed only a single control flow path in a slice.

Finally, Zilles's *et al.* Master/Slave Speculative Parallelization (MSSP) [28] represents a different scheme to exploit speculative thread-level parallelism via distilled programs. Distilled programs are a small subset of instructions of a given program that compute the input values of the speculative threads. In that execution model, the distilled program runs as a master thread and when all the input values for a speculative thread are computed, it is spawned on an idle context while the master starts computing new input values for the next thread. Our execution model differs from that previous work in the fact that the computation of the thread live-in values are done by speculative threads, which allows the processor to spawn threads out of the program order, and to often compute the live-ins for speculative threads in parallel.

7. Conclusions

In this work we have presented and evaluated the Mitosis compiler for exploiting speculative thread-level parallelism. This compiler includes a mechanism to build a synthetic trace, a

scheme to generate and speculatively optimize pre-computation slices, a model to estimate the benefits of any set of spawning pairs for a given SpMT configuration, and a greedy algorithm to select the best set of pairs. The two major novelties of the proposal are: (1) the use of pre-computation slices (i.e., software value prediction) to handle inter-thread data dependences, and (2) a model of the whole system that helps the compiler to identify which parts of the program will provide the highest benefit when parallelized through speculative threads. This model takes into account possible misspeculations, overheads, and load balancing.

A key contribution of this work is a set of compiler optimizations that reduce the length (and thus the overhead) of pre-computation slices. Branch pruning, memory and register dependence speculation, and early thread squashing are the main techniques proposed in this paper.

The results obtained by the Mitosis compiler/architecture for a subset of the Olden benchmarks are impressive. It outperforms single-threaded execution by 2.2x. When compared with a big out-of-order core, the speedup is close to 2x. We have also shown that the benefits of Mitosis do not come only from reducing memory latency since it outperforms an ideal system with perfect memory by about 60%.

Overall, this work shows that significant amounts of thread-level parallelism can be exploited in irregular codes, with a rather low overhead in terms of extra/wasted activity.

Acknowledgments

We would like to thank Peter Rundberg (currently at Gridcore, Sweden) for his valuable collaboration at the first stages of this work. Also, we would like to thank John Shen and Hong Wang (from MRL, Santa Clara) for their collaboration in the definition of the Mitosis architecture and the ORC team for their support in the compiler implementation. This work has been partially supported by the Spanish Ministry of Education and Science under contract TIN2004-03072 and Feder funds. Finally, we would like to thank the reviewers for their helpful and constructive comments.

References

- [1] H. Akkary and M.A. Driscoll, "A Dynamic Multithreading Processor", in Proc. of the 31st Int. Symp. on Microarchitecture, 1998
- [2] 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
- [3] M. Cintra and J.Torrellas, "Eliminating Squashes through Learning Cross-thread Violations in Speculative Parallelization for Multiprocessors", in Proc. of the 8th Int. Symp. on High Performance Computer Architecture, 2002
- [4] L. Codrescu and D. Wills, "On Dynamic Speculative Thread Partitioning and the MEM-Slicing Algorithm", in Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 40-46, 1999
- [5] 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
- [6] R.S. Chapel, J. Stark, S.P. Kim, S.K. Reinhardt and Y.N. Patt, "Simultaneous Subordinate Microthreading (SSMT)",

- in Procs. of the 26th Int. Symp. on Computer Architecture, pp. 186-195, 1999
- [7] K. Diekendorff, "Compaq Chooses SMT for Alpha", Microprocessor Report, December, 1999
- [8] 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
- [9] P.K. Dubey, K. O'Brien, K.M. O'Brien and C. Barton, "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading", in Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, 1995
- [10] M. Franklin and G.S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine Grain Parallelism", in Proc. of the 19th Int. Symp. on Computer Architecture, 1992
- [11] R. Ju, S. Chan and C. Wu, "Open Research Compiler for the ItaniumTM Family", in Tutorial in the 34th Int. Symp. on Microarchitecture, 2001
- [12] C. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors", in Proc. of the 28th Int. Symp. on Computer Architecture, pp. 40-51, 2001
- [13] P. Marcuello and A. González, "Clustered Speculative Multithreaded Processors", in Proc. of the 13th Int. Conf. on Supercomputing, pp. 365-372, 1999
- [14] P. Marcuello, J. Tubella and A. González, "Value Prediction for Speculative Multithreaded Architectures", in Proc. of the 32nd Int. Conf. on Microarchitecture, pp. 203-236., 1999
- [15] P. Marcuello and A. González, "Thread-Spawning Schemes for Speculative Multithreaded Architectures", in Proc. of the 8th Int. Symp. on High Performance Computer Architectures, 2002
- [16] T. Marr et al., "Hyper-threading Technology Architecture and Microarchitecture", Intel technology Journal, 6(1), 2002
- [17] J. Oplinger et. al., "Software and Hardware for Exploiting Speculative Parallelism in Multiprocessors", Technical Report CSL-TR-97-715, Stanford University, 1997
- [18] Roth and G.S. Sohi, "Speculative Data-Driven Multithreading", in Proc. of the 7th. Int. Symp. On High Performance Computer Architecture, pp. 37-48, 2001
- [19] G.S. Sohi, S.E. Breach and T.N. Vijaykumar, "Multiscalar Processors", in Proc. of the 22nd Int. Symp. on Computer Architecture, pp.414-425, 1995
- [20] J. Steffan and T. Mowry, "The Potential of Using Thread-level Data Speculation to Facilitate Automatic Parallelization", in Proc. of the 4th Int. Symp. on High Performance Computer Architecture, pp. 2-13, 1998
- [21] 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, pp. 58-62, 1998
- [22] S. Storino and J. Borkenhagen, "A Multithreaded 64-bit PowerPC Commercial RISC Processor Design", in Proc. Of the 11th Int. Conf. on High Performance Chips, 1999
- [23] 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
- [24] M. Tremblay et al., "The MAJC Architecture, a synthesis of of Parallelism and Scalability", IEEE Micro, 20(6), 2000
- [25] 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
- [26] T.N. Vijaykumar, "Compiling for the Multiscalar Architecture", Ph.D. Thesis, Univ. of Wisconsin-Madison, 1998
- [27] C.B. Zilles and G.S. Sohi, "Execution-Based Prediction Using Speculative Slices", in Proc. of the 28th Int. Symp. on Computer Architecture, 2001
- [28] C.B. Zilles and G.S. Sohi, "Master/Slave Speculative Parallelization", in Proc. of the 35th Int. Symp. on Microarchitecture, 2002