P-Slice Based Efficient Speculative Multithreading

Rakesh Ranjan [1, 2], Pedro Marcuello [2], Fernando Latorre [2], Antonio González [1, 2]

[1] Computer Architecture Department
Universitat Politècnica de Catalunya
Barcelona, Spain
rranjan@ac.upc.edu

[2] Intel Barcelona Research Center
Intel Labs Barcelona – UPC
Barcelona, Spain
{pedro.marcuello, fernando.latorre,
antonio.gonzalez}@intel.com

***Abstract*—Microprocessor industry has recently shifted towards multi-core to take advantage of the ever increasing number of transistors provided by the new technologies. Unfortunately, the multi-core approach does not allow single threaded applications to benefit from the additional cores to improve their execution time. Speculative multithreading (SpMT) has been proposed in the past to boost performance of irregular applications in multi-core environments. In this work, we study the main bottlenecks of these architectures, such as the memory behavior and the pre-computation slices and propose two novel schemes that allow SpMT to get 25% average speedup over single threaded execution.**

We propose Selective Replication as a technique to improve the performance of the SpMT memory system. This technique does not introduce additional traffic in the bus and improves the performance of a conventional SpMT memory model by 6% on average and up to 21% for some applications. Also, we propose a scheme called Slice Specialization that reduces the number of instructions in the pre-computation slices by adapting the slice to every single speculative thread spawned. The later proposal outperforms previous schemes with slices by 15% and overall, both techniques combined achieve an improvement of 20% over a conventional SpMT processor.

Keywords-Multithreading; Speculation; TLS

I. INTRODUCTION

Using the ever-increasing number of transistors to improve instruction level parallelism nowadays shows diminishing returns at the expense of significant power increase. Thus, the industry and the academia are moving towards on-die multi-core systems where these additional transistors are used to exploit other types of parallelism like thread level parallelism.

Increasing the number of cores available on the chip augments the number of threads that can be run in parallel. Thus, parallel applications whose performance scale well with the number of threads can significantly benefit with these systems. However, non-parallel applications cannot take advantage of this multi-core approach because conventionally they execute in a single core.

Speculative Multithreading (SpMT) has been proposed in the past as a way to improve the performance of single threaded applications through speculative parallelization. In this paradigm, parallelization constraints are relaxed and the new parallel threads are data and control dependent among themselves. Speculative threads are spawned and executed in parallel as regular threads, but due to their speculative nature, they cannot modify the architectural state until the speculation is proved to be correct. Therefore, processors that are able to exploit speculative thread level parallelism include support for storing the speculative state until validation.

Previous proposals on speculative multithreading [2][14][18][20][21][29] mainly differ on how speculative threads are selected and the way inter-thread data dependences are managed. The speculative threads can be generated by the compiler [12] or detected at run-time [16]. Code regions that can be speculatively parallelized are loop iterations, loop continuations, subroutines, modules or more complex schemes based on profiling. To handle inter-thread dependences there are several proposals like assuming no dependences, use of hardware/software value prediction [13][15], synchronization [26] or speculative loop fission [27].

In spite of the promising potential of the SpMT architectures, the observed performance has been far from ideal. In this work, we focus on an architecture that assumes a simple spawning scheme, i.e. loop iterations using pre-computation slices to early compute the dependent values among concurrent threads. For this architecture, we found that the poor memory behavior and the significant cost of pre-computation slices are two main bottlenecks among others, which cause the unexpectedly low performance. In this work we propose two profile guided techniques to alleviate the effect of these two bottlenecks:

- *Selective Replication* reduces the impact of cold caches and loss of locality in SpMT memory hierarchies, by replicating those cache lines which are expected to be reused with no extra hardware or increase in bus traffic.
- *Slice Specialization* reduces the cost of the pre-computation slices by removing the predictable control flow computation.

Overall, the two techniques improve the performance by an average of 25% over single thread on a 4-core CMP.

The rest of the paper is organized as follows. Section II explains the spawning model and the supporting architecture whose bottlenecks we explore. In the same section we also define pre-computation slices and the method to build them. In Section III we describe our experimental framework. In Section IV we study the various bottlenecks of the SpMT

978-1-4244-4921-7/09/$25.00 ©2009 IEEE

architecture. In Section V we discuss the *Selective Replication* technique. In Section VI we define our proposed technique of *Slice Specialization* and the results obtained using the technique. Finally, Section VII discusses some related work and Section VIII summarizes the main conclusions of this work.

II. SPECULATIVE MULTITHREADED ARCHITECTURES

A. Speculative Threads

SpMT architectures ([2][13][20] among others) have been proposed as an execution model in which single threaded applications can be speculatively split into multiple threads. These threads can then be executed on multiple cores. By speculative execution we mean that the spawned threads may be data and/or control dependent on previous uncommitted threads. Hence, their execution may not be correct. In case of a dependence violation, the SpMT architecture squashes the violating thread. If the speculation turns out to be correct, the non-speculative thread validates its successor thread and then commits itself. After the commit the validated thread becomes the non speculative thread. The presence of one non-speculative thread in the system at any point in time guarantees forward progress of the overall system while correct thread speculation ensures added speedup.

A speculative thread in this model is identified by a *SP-CQIP* pair [14], where *SP* stands for the *Spawning Point*, i.e. the instruction in the execution stream where the speculative thread's execution is triggered. *CQIP* stands for *Control Quasi Independent Point*, i.e. the instruction from which the speculative thread begins execution. The choice of these pairs strongly affects the performance achieved by the system [14].

Figure 1(a) shows a single thread execution stream in which there is a data dependence between instructions I1 and I2. The same execution stream when executed in the SpMT execution model is shown in Figure 1(b). When the execution reaches instruction SP, it spawns a new thread on a free Thread Unit (TU1). The new thread first executes a chunk of code called the pre-computation slice (p-slice for short) which computes all those values that are produced by the parent thread (Region B) and are required by the new thread (Region C). The values generated by the p-slice are stored in a special buffer called the *slice buffer*. In the example shown in Figure 1(b), the p-slice produces the value needed by instruction I2. Once the p-slice finishes execution, the speculative thread starts executing instructions starting from CQIP. The speculative state of the thread is buffered in the local register file and the L1 data cache. For any input value that the speculative thread needs, it first checks if it has produced the value itself. Otherwise, it checks if it is available in the slice buffer. If not available in any of the local buffers (i.e. slice buffer, register-file or L1 Data cache), it then requests the value from its predecessor thread.

When the spawner thread reaches the CQIP, it validates the next speculative thread. For validation, the spawner thread compares the values read by the spawnee thread from the slice buffer to the ones produced by the spawner thread.

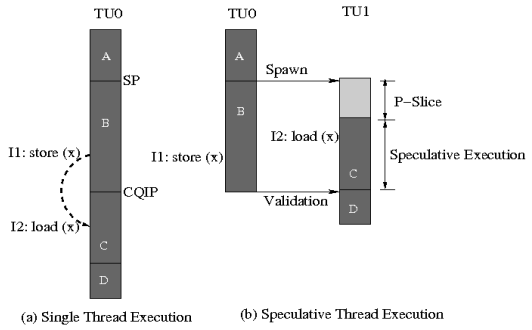

Figure 1: P-slice based SpMT Execution Model

If the validation is correct, the spawner thread commits and the validated speculative thread becomes non-speculative. If the validation is wrong, the speculative thread and its successors are squashed and the spawner thread continues execution beyond the CQIP. If the spawner thread reaches the validation point while the spawnee thread is still executing the p-slice, then the spawnee thread is squashed and the spawner thread continues execution beyond the CQIP. In Figure 1(b) when the non-speculative thread running on TU0 commits correctly, we see that the Region C has executed in parallel with the parent thread and hence improves the overall performance.

In the SpMT execution model, efficient handling of interthread dependences is very crucial for performance. In our terminology, we call those output values of a thread as live-ins which are used as inputs for threads that are more speculative than itself. In the example of Figure 1(b), the value stored at address(x) is a *live-in* and instruction I1 is a *Live Instruction*. Previous works have proposed other schemes to deal with these dependences, e.g. synchronization mechanisms [21], optimistic execution assuming no dependence, hardware value prediction [15] and speculative loop fission [27]. Synchronization can have a significant overhead if dependences are frequent as is the case for many irregular programs in the workloads presented here. Hardware Value Predictors (HVP) [19] exploit context information, e.g. value history, branch history, etc. to predict values. For this reason their accuracy suffers significantly when the immediate context information is missing as is the case with large speculative threads. Although HVPs perform relatively well for exploiting instruction level parallelism, their performance is severely limited in the context of speculative multithreading when threads can be large in size.

B. Pre-Computation Slice

A p-slice is a piece of code that is executed before a speculative thread in order to generate the live-ins needed by the thread. Next we describe how to construct a p-slice.

1) *P-Slice Construction*: The p-slice is built by the compiler from the *Program Dependence Graph (PDG)* [9] using profile information. There are two steps in constructing the p-slice: (a) Identifying *the thread live-ins*, and (b) *Generating the p-slice*. To identify the thread live-ins, the PDG is traversed top-down starting at the CQIP and marking all those register and memory values which are read without

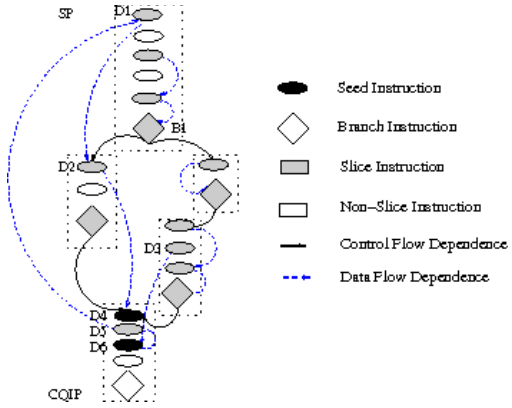

Figure 2: Dependence Graph for a typical thread

being written first. These values are the live-ins for the speculative thread. This is done for a certain length, which is same as the number of instructions that is expected to execute in parallel with the previous thread. As this number depends on the size of the p-slice, for simplicity we assume to be same as the number of instructions between the SP and CQIP. After the live-ins have been identified, the PDG is traversed bottom-up starting at the instructions which produce the live-ins directly. All the ancestor instructions are recursively traversed following the data and control dependence edges, as long as the instructions are below the SP in the program order. All the instructions traversed in this fashion comprise the p-slice. The process of p-slice construction is explained in the example shown in Figure 2. In Figure 2, instructions D4 and D6 (black nodes) produce the live-ins for the next thread starting at CQIP. To build the p-slice for the thread, beginning at CQIP, we include D4 and D6 into the p-slice and follow recursively their control and data dependences (denoted by the solid and dotted lines). Every instruction traversed is included in the p-slice. The instructions comprising the complete p-slice are denoted using the grey and black nodes.

2) *P-slice Advantages*: The P-slices offer many advantages over other proposals for handling inter-thread dependences. Since a p-slice is a subset of instructions from the original program, they compute the live-ins with a much higher accuracy than HVPs. It is useful at this point to compare p-slice model to speculative fission [27], where the instructions which compute the dependencies are pre-computed before spawning the speculative threads. In p-slice model, the slice does not update the architectural state. Hence, it is a pure overhead. On the other hand, in speculative fission, the instructions computing the dependencies are not re-computed and they directly update the architectural state.

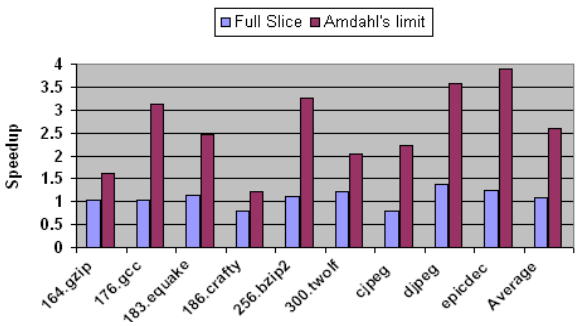

Figure 3: Speedup with Full Slices

At the first sight, p-slices seem very promising. However, we will see later, when we separate the set of instructions that compute the dependencies from those which do not, the left over thread size is usually too small to achieve any significant performance gain. In the case of p-slice based execution, since the slice is a speculative piece of code, we can apply very aggressive set of optimizations to reduce its size. In speculative fission, aggressive optimizations cannot be applied to the dependence computation as correctness has to be ensured. Using p-slices, in many cases we are able to extract parallelism out of threads where speculative fission does not provide any significant parallelism. The reason why a large part of the thread appears in the slice, as we will study more exhaustively in later sections, arises from the fact that integer programs have very complex control flows.

While p-slices are very promising, the performance achieved using p-slices built from the first principles (described in Section 2.2.1), which we call conservative or *full slices*, tend to be very poor. Figure 3 shows the speedups obtained from a SpMT execution on a 4-core CMP using conservative slices, over a single thread execution. It also shows the ideal case speedups using Amdahl's law for the same threads. Note that in some benchmarks (*186.crafty* and *cjpeg*) we even observe a slowdown over single thread execution. The ideal limit is less than 4 for some benchmarks because, due to limitations of the spawning scheme and the infrastructure (discussed later), we are unable to cover 100% of the execution in the selected threads. Nevertheless, Figure3 shows that on an average the performance of SpMT obtained using conservative p-slices is 150% less than the ideal limit. We identify and evaluate the different bottlenecks causing this performance gap in Section 4 and propose some techniques and future directions to address them.

III. EXPERIMENTAL FRAMEWORK

A. *Thread Selection*

In the spawning model assumed in our framework, we only consider program loops as potential thread candidates. We do this precisely because loops make up for most of the dynamic execution in a program. Since different iterations of the same loop mostly do similar work, they also tend to be more balanced in their execution length. Further, loops are well defined program structures and hence they simplify the thread analysis and the architecture design. We assume an in-order spawn model, which means the threads are spawned in the program order. To maintain the in-order spawn, we spawn the thread only at a single nesting level when the loops are nested. We choose the nesting level according to a cost benefit model based on the metric that we call Weight, Wl for the loop *l*, which is defined as follows:

$$W_l = ThreadInstCount_l - pSliceInstCount_l$$

Where $ThreadInstCount_l$ and $pSliceInstCount_l$ denote the number of dynamic instructions executed in l and in the p-slice of l respectively, across all iterations. Additionally, for every loop we also define another metric Resultant Weight, RWl which is defined as follows:

$$RW_l = W_l - \sum W_l o - \sum W_l i$$

Where $\sum W_l o$ and $\sum W_l i$ refer to the sum of Weights of all the outer and inner loops of l respectively. Once we have the Resultant Weights of all the loops, we use the greedy algorithm shown in Figure 4 to select the final loop candidates. The algorithm in Figure 4 takes as input, the set of all loops L(N), where N is the total number of loops in the program and selects a subset. The inner loop in Figure 4 chooses the loop *SL* that has the maximum RW_l and marks it as selected. Once a loop is selected, all its nested outer and inner loops including itself are marked as visited. If all the loops in the set L[N] are marked visited, the program exits or else it calls itself recursively. All the selected loop candidates have the selected field set. Once we have the loops selected, the sum of Weights of all the selected loops gives us an estimate of the benefit of the chosen loops. We call this the *Program Weight*:

$$WP = \sum Wl$$

We repeat this process several times with a different starting loop (L[0]) and choose the set which maximizes the WP . When the p-slices are optimized using the *Slice Specialization*, discussed later, we re-evaluate the selection procedure as different loops might have different amount of reduction in the size of their p-slices. The selection procedure aims to maximize the overall Program Weight. In our current spawning scheme we do not consider loops which include function recursivity. For some of the benchmarks, this limitation reflects the low coverage of the loops and hence the low Amdahl's limit in Figure 3.

B. Evaluation Infrastructure

This section describes the infrastructure framework used for this work. For the purpose of this study we modified an SMTSIM [23], an alpha ISA based execution driven simulator to run the SpMT execution model. Henceforth, we call the simulator as SpMTSIM. SpMTSIM simulates a 4-core CMP, where each core is a 4-issue out-of-order processor. The architectural configuration of the processor is described in Table 1. We also developed a trace analyzer tool to build the Static *Program Dependence Graph (PDG)* of the whole program. This PDG is the same that a compiler would build using the profile data. From the information in the PDG, it generates the p-slice for each of the loop using the method described in Section 2.2.1. Using the algorithm described in the previous section a set of loops is selected. This set of loops with their slices is fed to SpMTSIM. We selected a set of programs from the SPEC2000 benchmark suite and Mediabench. The selected benchmarks are highly control-intensive single threaded programs. Conventional parallelizing compilers fail to discover any thread level parallelism in these programs. In *cjpeg*, *djpeg* and *epicdec* we simulate the complete program execution, whereas in others we fast forward 1 billion instructions before simulating 100 million instructions. The simulator executes Alpha binaries compiled with DEC-C Compiler using full (-O3) optimizations.

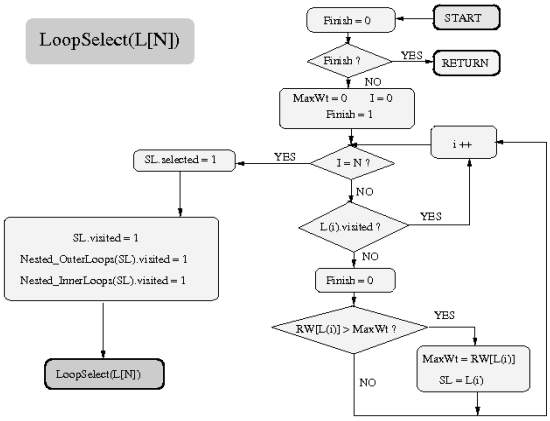

Figure 4: Selection Algorithm for Loop Nesting Level

Num. of Cores	4	Fetch width/core	4
ROB Size	128	GShare Table Size	2K
IQueue Size	64	FQueue Size	64
CacheLine Size	64bytes	Slice Buffer	1K
DCache Size	64 K	DCache Assoc	2
DCache Hit Lat.	1	DCache Miss Lat.	8
ICache Size	64 K	ICache Assoc	2
ICache Hit Lat.	1	ICache Miss Lat.	8
Shared L2	512 K	L2 Miss Latency	18
Shared L3	4M	L3 Miss Latency	92

Table 1: Processor Configuration

IV. CHARACTERIZATION OF THE SPMT MODEL

In this section we evaluate the aspects of the SpMT architecture and spawning scheme which explains the low performance of the p-slice based SpMT paradigm. These are *Memory*, *P-Slice Size* and others that include *Workload Imbalance* and *Spawning Scheme*

A. Memory

1) SpMT Memory Model:

The Speculative Multithreading Execution model has very significant consequences for the memory architecture of a CMP. In this work we model a memory hierarchy similar to the Speculative Versioning Cache (SVC) design [11]. Each cache line has two pointers, pointing to the less and more speculative versions of the data contained in the cache line. The list consisting of cache-lines, connected by these pointers is called the *Version Ordering List (VOL)*. Each cache line is also augmented with additional state bits called *Load (L)* bits for read-after-write conflict detection. Conflicts are detected at word boundaries and hence a cache line has

as many *L* bits as it has words. As in SVC, each cache line also has as many *Dirty (D)* bits as number of words. When a thread reads a cache line without updating it, the L bit for the touched words are set. While, on a write to a cache line, the *D* bits of the touched words are set. Also, write-update message is sent to the successor threads. When a thread snoops a write-update message from a predecessor, it checks if it has a cache line corresponding to the VOL of the updated cache line. If it has, then it checks if the *L* bits for those words are set. If they are set, it implies the thread has read a value which has been later updated by a predecessor thread, implying a read-after-write dependence violation. In this case the violating thread and its successors are squashed.

2) Effect of SpMT Model on Memory Behavior:

When a single thread program transitions from a single threaded execution to a SpMT execution model it loses some of the temporal locality between memory accesses to the local L1 cache. Going back to Figure1(a), if the Store instruction I1 and Load instruction I2 are close enough in their execution so that the cache line containing address(x) is not replaced between their execution, then I1 and I2 will cause a miss and hit in L1 respectively. In the SpMT model as in Figure 1(b), since both instructions are executed in different cores, both of them will lead to L1 misses.

The loss of temporal locality is further exacerbated by the thread *Commit* and *Squash* events, unique to SpMT execution model. In a naive SpMT design, for every thread commit, the dirty cache lines are written to the next level cache, after which all the cache lines are invalidated. Similarly on a thread squash all the cache lines are invalidated as well. This gives rise to the problem of *Cold Caches*. This means that when a new thread is spawned, it starts with an empty cache and suffers lots of cache misses. To deal with this problem in the SVC design, additional state bits were added to the cache lines, namely the *Commit(C)*, the *Architectural (A)* and the *sTale (T)* bits, one per cache line. When a thread commits, it sets the C bit in all the valid cache lines. This avoids the need to write the dirty lines to the next level cache. When a thread is squashed, the cache might contain data which is same as the non-speculative version of the data e.g. the squashed thread might have read the value from the L2 cache. On squash, the A bit is set in all the cache lines containing non-speculative data. This is done to indicate that the cache line has valid architectural version of the data. When a new thread is spawned the data in these lines are valid. The T bit is used to mark whether it is safe to use a committed line or not, since it might have become stale because a new version of the data has been produced. More details of this model are explained in the SVC design [11].

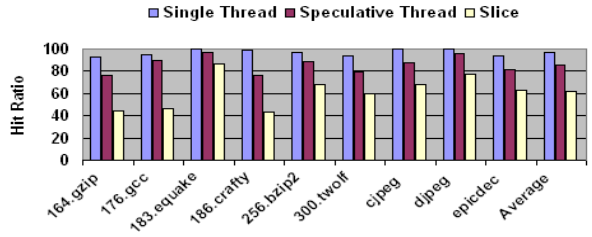

Figure 5: Cache-Hit Ratios with SVC Cache

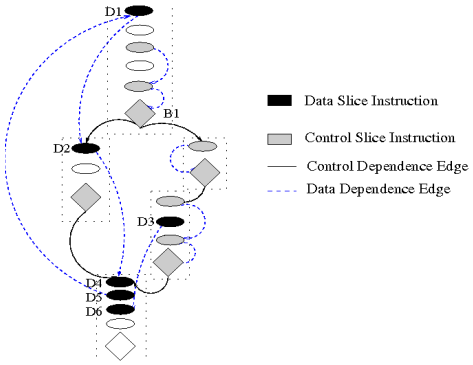

Figure 6: P-Slice Instruction Categories

Figure 5 shows the cache hit ratios for the speculative threads and their p-slices for the memory model described in Section 4.1.1 (including the A and T bits), and the corresponding single thread execution. The cache hit-ratio for the p-slices is much lower than the speculative threads as the p-slice prefetches some of the cache lines which are needed by the speculative thread. It is important to note that the p-slice consists of data-flow chains of dependent instructions and hence there is very little ILP available in it. Since, the time the p-slice takes to execute determines the overall performance, it is important to explore mechanisms to speedup its execution. As Figure 5 shows, the hit ratio for the speculative threads and the p-slices is significantly lower than that of the single thread. This is primarily due to two reasons: (1) *Cold caches* (2) *Loss of locality* (both temporal and spatial) in speculative threads. Cold caches refer to the problem of empty caches with which speculative threads begin execution. Locality loss happens when the data accessed by successive threads access say the same cache line. In a single thread execution, such an access would cause a single cache miss at the first access, whereas in speculative multithreading, it will lead to cache misses in all the threads.

B. P-Slice Size

A p-slice for a *live instruction* is built by a backward traversal of the Program Dependence Graph (PDG) of the static program starting from the *live instruction* up to the Spawning Point instruction as explained in section 2.2.1.

Figure 6 shows the PDG of a typical p-slice. The instructions in a p-slice can be categorized into two types: (a) *those which actually participate in the actual flow of data leading to the live instruction*, and (b) *those which help in determining the control flow*. Note that these two categories are not necessarily mutually exclusive. However, for ease of understanding we include the instructions which are common to both categories as only part of the former type. In the Figure 6, the nodes colored in black actually compute the live-ins needed by the thread. The ones in grey are used for computing the control flow within the slice. In an ideal scenario, where the control flow of the program can be correctly determined, the instructions needed for control flow computation can be safely removed without affecting the correctness of the slice outcome. We refer to the slice without the control-flow instructions as the Pure *Data Slice*.

Following the same terminology, the first slice that contains all the necessary instructions (data and control) is referred to as the *Full Slice*. For our purposes we define *Slice Quality (SlcQual)* as the p-slice length relative to the thread length, where length can be defined in terms of number of instructions or number of execution cycles. Figure 7 shows the *Slice Quality* in terms of instruction count for the selected programs. Each bar represents the size of the full slice relative to the size of the thread. Also each *full slice* size is broken into the fraction which consists of the *data slice* and those which are needed only to compute the control flow. As can be seen in Figure 7 the fraction of slice instructions which are needed to compute only the control flow is very significant. We propose a technique called *Slice Specialization* to exploit this characteristic of the slices in order to reduce the size of the slice. We discuss the technique in more detail in the next section.

C. Other Factors

Figure 8 shows the speedup over single thread of the SpMT execution with full slices and the SVC model (leftmost bar) and the Amdahl's limit (rightmost bar). The bar in the middle shows the performance potential assuming no dependences between threads and perfect memory for speculative threads. As we can see there is a significant performance gap between the obtained with full slices and the potential benefit. Memory and how dependences are handled can save almost half of the way. The other half of this loss can be attributed to other factors such *Workload Imbalance* and *Spawning Scheme* among others.

In SpMT execution, threads executing on different cores take different amount of execution time. When this happens, a speculative thread might reach its CQIP before its parent thread has validated it. Hence, the speculative thread has to wait till its parent validates it.

In several benchmarks, *free cycles* constitute a large fraction of the total cycles, i.e. the Thread Units are not running any thread at all. This is surprising given that the selected loops have a high coverage for some benchmarks, e.g. the loops in *epicdec* account for 99% of the total dynamic instructions. This can be attributed to the loops having a low trip count (number of iterations per invocation). Further, during the time when the sequential portion of the program is being executed by the non-speculative Thread Unit, the other cores remain idle. This accounts for a substantial fraction of the free cycles in some benchmarks where our loop candidates have a low coverage due to the limitations of our spawning scheme. Also, our spawning scheme does not handle loops calling themselves through recursive routines and those which are non-natural (loops with multiple entries) [17].

To address the problem of free cycles due to low trip count loops, the simple spawning scheme that we have employed needs modification. For loops where the iteration size is significant but have low trip count, loops could be chunked [18] into smaller threads, instead of spawning threads at loop boundaries. However, to do this automatically it needs more sophisticated compiler analysis. We intend to explore the techniques to solve workload imbalance and improving the spawning scheme in our future work.

V. Selective Replication

Previously proposed techniques to improve the memory locality in SpMT architectures can be classified into two categories: the ones that propose additional state bits in the cache lines to identify data that is retained across thread executions [11]; and those which attempt to snoop the bus for data used by other threads and replicate them in special local buffers [13].

We propose a new technique called *Selective Replication (SR)* that is built over the SVC architecture. It belongs to the latter category as mentioned above. Unlike [13], instead of replicating every data available on the bus, the compiler is responsible for selectively identifying the memory operations for which replication is done. Using profile information, the compiler inserts a hint in all those static memory instructions whose data, if replicated is likely to benefit the memory locality across the threads. We call these instructions as *load_all* and *store_all* instructions or as *mem_all* collectively.

If the execution of a *load_all* instruction causes a L1 miss, the cache-request encodes the information that the requesting instruction is a *load_all*. When the requested data is available on the bus, the active as well as the idle cores snoop the bus and replicate the cache line in their respective local L1-caches. In the case of *store_all* instructions causing a L1 miss, only the threads which are more speculative than the requesting thread replicate the cache line. If the line to be replaced, in order to store the replicated data is dirty, then no replication is done. In the case of a L1 hit, no replication is done as well. Since, replication is only done when lines are brought from L2 and we use snoop-based bus architecture, selective replication does not increase the bus traffic or the bus contention.

Figure 9 shows the overall Cache Hit Ratios for different

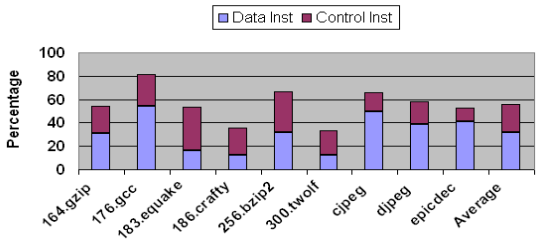

Figure 7: Quality and Composition of Full P-Slices

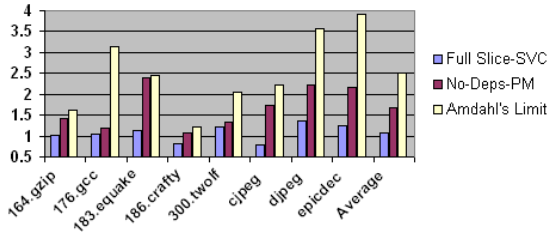

Figure 8: Speedups assuming Perfect Memory and No Dependences

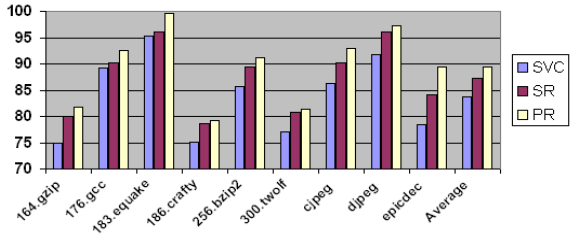

Figure 9: Cache Hit Ratios with various Replication Types

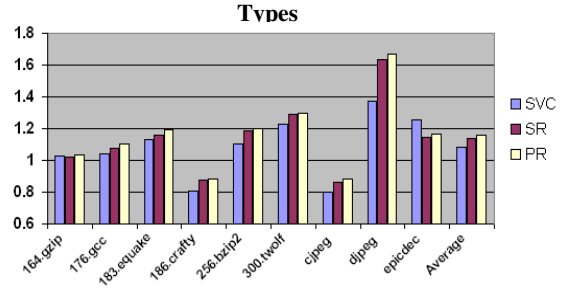

Figure 10: Speedups with various Replication Types

cache types. The left bar shows the hit ratio for the baseline SVC cache. The middle bar shows the hit ratios with our SR scheme and the rightmost bar shows the hit ratios with perfect replication, (i.e. using an unbounded sized structure for replicate everything). We see that with our SR scheme the hit ratio improves by an average of 4.1%. Figure 10 shows the speedups of SpMT using SVC, SR and PR memory types over single thread execution, using an oracle branch predictor (to remove the effect of branch prediction) and full slices. SR outperforms by an average of 6% the SVC model while a perfect replication would help improve the performance by an average of 7%. Improvement in memory affects the performance of speculative thread more than the p-slice, since the thread has higher ILP than the p-slice. This increases the ratio of execution time of the p-slice vs. the thread. In the case that the non-speculative thread reaches the CQIP while the spawnee thread is still executing the p-slice, the spawnee thread is squashed. This increase in the number of squash explains the fall in performance in *epicdec* in spite of the increase in cache-hits due to replication.

Benchmark	Static	Dynamic
164.gzip	29.96	10.51
176.gcc	24.58	3.08
183.equake	17.38	0.35
186.crafty	34.8	16.52
256.bzip2	45.37	9.84
300.twolf	34.82	15.05
cjpeg	22.09	4.69
djpeg	16.09	2.75
epicdec	19.33	4.88

Table 2: Percentage of Mem Insts converted to mem_all

In Table 2, we show the percentage of memory instructions which are converted into *load_all* and *store_all* instructions. The second column shows the percentage of all

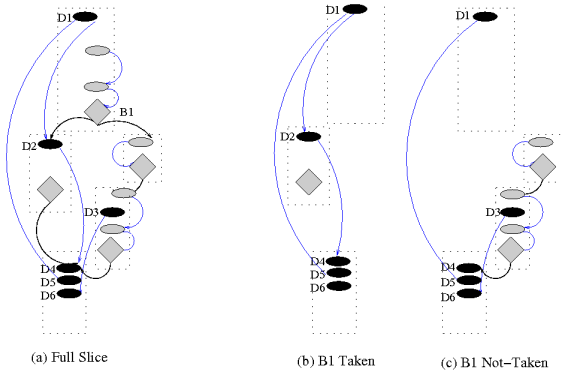

Figure 11: (a) The Full Slice extracted from Thread (b) Specialized Slice when B1 is predicted taken (c) Specialized Slice when B1 is predicted not-taken

static memory instructions which are load all or store all i.e. in which the extra hint bit is encoded. The third column shows the corresponding percentage of all dynamic memory instructions which act as *load_all* or *store_all* i.e. those which cause replication

VI. SLICE SPECIALIZATION

We plan to reduce the slice size by removing the instructions that compute part of the control flow which is easily predictable by hardware control flow predictors like branch predictors. A p-slice is a speculative piece of code, so if the control flow prediction is incorrect, the thread can be squashed without affecting correctness. We propose *Slice Specialization* as a technique to create specialized slices for the different possible control flows inside the slice. We define a branch as *prunable* when its predictability is above a certain threshold, determined using profiling. *Slice Specialization* aims to remove these branches and the instructions needed to compute them. By predicting the *prunable* branches we split the slice along the taken and not-taken paths. The slices obtained in this manner are called *Specialized Slices*. Consequently, for every thread there are multiple specialized slices instead of one *full slice*. At run time using a hardware control flow predictor, we select the specialized slice to fetch the slice instructions from.

By predicting the control flow taken in the slice, not only the instructions needed for control flow computation can be removed but also those live instructions and their slices which do not appear in the selected path. In preparing specialized slices, there might be multiple *prunable* branches, which form a binary-tree like structure with the first *prunable* branch of the thread at its root. In our terminology we refer the specialized slices as *Pruned Slices*. In this work we only take into account those slice branches for pruning which are executed only once per execution of the slice i.e. branches which are part of inner loops or recursive routines are not included. Techniques to prune multiple dynamic instances of the same static branch will be explored in our future work

Figure 11 illustrates the *Slice Specialization* technique used over the slice that was earlier shown in Figure 6. Figure

11(a) shows the *full slice*. Assume that B1 is a *prunable* branch; we can safely remove B1 and all those instructions which are only used for its computation. This gives rise to two specialized slices as shown in Figure 11 (b) and (c). In the original *full slice* depending on whether B1 is taken or not-taken, we need to execute 9 and 13 instructions respectively. On the other hand, if we execute the *specialized slices*, we execute 6 and 10 instructions respectively.

A. Slice Predictor

As a consequence of *Slice Specialization*, when a new thread is spawned, a hardware slice predictor predicts the most likely to be executed slice from a pool of *pruned slices*. One possible way to implement the *Slice Predictor* is by predicting the set of *prunable* branches in succession. The outcomes of these branches are used to generate a bitvector (*Uid*) which uniquely identifies the specialized slice. The *Uid* is then used to generate an address in memory where the corresponding specialized slice is stored. Unlike for the thread, for which we use the gshare branch predictor, we use a local Bi-modal branch predictor for predicting the slice. The reason for this selection is that the gshare predictor works by exploiting the correlation between consecutive branches preceding the branch, but the branches which are predicted by the Slice Predictor are not consecutive and hence not necessarily strongly correlated. Figure 12 shows that a 2K entry Bi-modal predictor outperforms a gshare predictor with the same size by an average of almost 60% when it is used to predict the *pruned slices*. We assume a cost of one cycle per branch predicted.

The spawner thread records the final outcome of the predictable branches while executing the thread into a new bitvector. This is communicated to the spawnee thread at the validation time for updating the branch-predictor table used by the p-slice predictor. When the spawner thread commits, it checks those values that have been read from the *Slice Buffer*. If the values read were correct, the spawner commits or else it squashes the spawnee thread and its successors..

B. Results

In this section we present the evaluation of our slice specialization technique. Using branch profiling we identify the *prunable* branches. This is done by selecting all those branches whose hit ratio is above a certain threshold. For the experiments in this work, a threshold of 80% was enough to predict the slices correctly. As for thread candidates, in our current spawning scheme we are unable to handle non-natural loops and those which execute themselves recursively. This limits the coverage we are able to attain for some of the benchmarks (as evident from the low Amdahl's limit in Figure 3 compared to the ideal speedup value of 4). Nevertheless, the benefits obtained in the part of the program we are able to parallelize shows the effectiveness of our scheme. Extending our scheme to work with recursive loops will be part of our future work.

Figure 13 shows the speedups obtained using different kinds of slices, namely, the *Full Slice*, *Pruned Slice* and *Data Slice* over a single threaded execution, using SVC Cache. On an average, the *Slice Specialization* technique achieves 18%

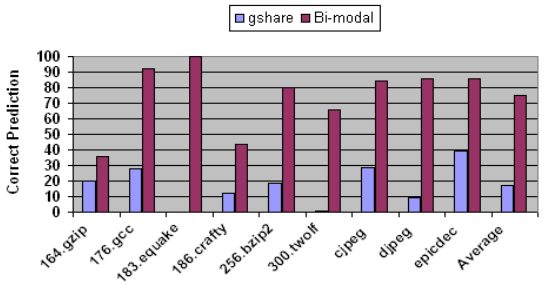

Figure 12: Percentage of Specialized Slices Predicted Correctly

performance improvement over single thread execution, with some benchmarks benefiting up to 40%. Compared to statically build full slices, our scheme achieves an average of 9% performance gain with some benchmarks benefiting up to 12% (*epicdec*). In *186.crafty* we suffer a slowdown because less than 25% of the total execution is in our selected loops (since our spawn model does not handle recursion and non-natural loops). Further the selected loops in *186.crafty* have a very low trip-count, deteriorating any improvement that could be achieved by SpMT execution. Even in *186.crafty* we see an improvement by 10% in the speedup of the pruned slice over the full slice execution. The speedups for the *Data Slice* show the ideal performance that could be achieved if all the control flow computation in the p-slice could be removed. In our future work we plan to extend *Slice Specialization* to work for those predictable branches of the p-slice which appear in inner loops and recursive routines. This would further reduce the gap between our current performance benefits and the performance limits using *Data slice*.

We compare the effectiveness of our technique with Mitosis [10] which is the state of the art scheme for speculative multithreading using p-slices. In Mitosis every dynamic instance of a static thread executes the same static slice which is obtained by pruning the highly biased branches from the conservative (*full*) slice. Unlike Mitosis, our scheme is capable of pruning predictable branches which already include the biased branches as they are always highly predictable. It is important to note that though in Mitosis the thread selection is not limited to loop structures, the Slice Specialization technique is orthogonal to the selection scheme. The technique can be employed for any static thread

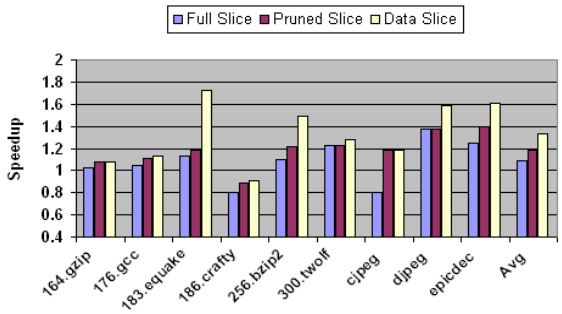

Figure 13: Speedups for Different Slice Types

selection scheme.

Slice generated from Mitosis scheme performs well only when the slice has a significant number of highly biased branches and the single pruned static slice obtained by pruning those branches covers a high percentage of the total threads. For every thread, the single most used pruned slice ($slice_h$) is a close approximation to the corresponding Mitosis slice. Figure 14 shows the number of distinct specialized slices needed to correctly generate the live-ins for all the threads. We note that for some of the very control intensive benchmarks like *176.gcc* and *186.crafty* the $slice_h$ is correct only for 60% and 53% of the total threads respectively. Our scheme also adapts very well when the branches are not only completely biased but highly predictable. Hence, it generates smaller slices even when the program is very control intensive. We observe that with 5 *pruned slices* on average we are able to cover more than 90% of the threads across all benchmarks.

Figure 15 shows the speedup gains of Slice Specialization over Mitosis, assuming a SVC based cache and a bimodal slice predictor. From Figure 15 we expect significant improvements in speedup using Slice Specialization in *176.gcc*, *186.crafty* and *epicdec*, since these benchmarks have low coverage using $slice_h$. For *186.crafty*, *164.gzip* and *epicdec* the performance improvements are up to 12%. By contrast, we do not see significant improvement for *176.gcc* (2.19%), because even after pruning, the p-slices are significantly large. Hence, the threads get squashed as the spawner thread catches up with the spawnee thread while it is still executing the p-slice. This fact is observed in Figure 7 where the pure data slice for *176.gcc* is still about 55% of the thread size. Finally, Figure 16 shows the speedups of our combined proposal of *Slice Specialization* and *Selective Replication* over full slices using SVC cache. We see that our combined techniques improve the performance by an average of 15% over full slices using SVC, and an average of 24% over single thread execution.

VII. Related Work

Speculative Multithreading has been the subject of research for many years and several architecture and compiler techniques have been proposed to support this execution paradigm. Multiscalar [20] was one of the earliest and pioneering proposals. Speculative threads were extracted statically by analyzing the Control Flow Graph and taking care of the dependencies with the help of the compiler [24].

Many of the works have specifically focused on extracting the speculative threads based on various program structures like loops [12], subroutines [3] and loop and subroutine continuations [5][8][22][25]. These proposed techniques differ in the method they employ to handle thread dependencies. Gopal et al. [11] use the snoop based cache coherence protocol to communicate memory values and to detect memory violations. In [16] the authors have proposed to walk through the Control Flow Graph (CFG) of the program and spawn threads at points which are very likely control independent. To deal with the dependencies, they propose a hardware value predictor [15].

Program slices have been used earlier in the context of helper threads [4][6][7][28]. Helper threads are built in similar fashion to p-slices for speculative threads but they differ in their purpose. Helper threads are used to prefetch long latency load data or predict hard to predict branches. Due to the potentially higher impact of a misspeculation in case of speculative threads, the p-slices are built to deliver higher accuracy than helper threads.

Agarwal et al [1] make use of postdominators [17] in the program CFG to spawn speculative threads Program Demultiplexing(PD) [3] is a recent proposal that employs program slices (handlers) to predict some of the input values and the control flow between the spawning point and the spawned thread, while waiting for the rest of the inputs to be produced, at which point the thread is spawned. Mitosis [10] on the other hand uses program slice for producing all the input values and the control flow. As we saw in our evaluation that the control flow computation comprises a major fraction of the slice. Slices used in PD would be prohibitably large when they intend to exploit distant parallelism. Further, since threads used in PD are subroutines they also tend to be inherently load imbalanced than loops. Mitosis proposal is similar to our proposal but differs in the way the slices are generated and executed.

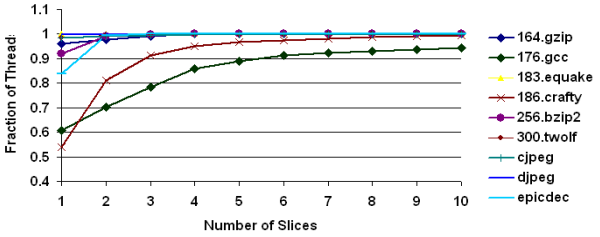

Figure 14: Number of Specialized Slices needed vs Fraction of Total Threads

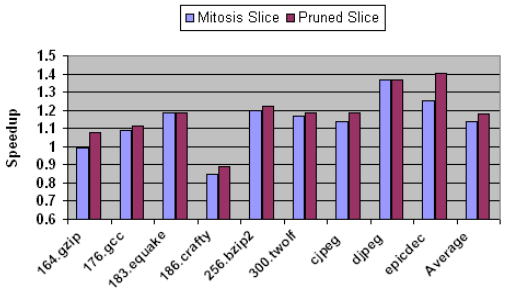

Figure 15: Speedups for Mitosis vs Slice Specialization assuming SVC Cache, over Single-Thread.

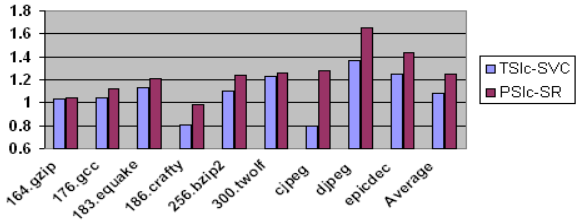

Figure 16: Speedups for Full Slice with SVC Cache vs Pruned Slice with SR Cache, over Single-Thread

VIII. CONCLUSIONS

We have identified the main bottlenecks of the p-slice based SpMT architecture, such as Cold Caches, loss of locality, p-slice size, workload imbalance, and spawning scheme limitations.In this paper we proposed two techniques to alleviate the negative impact of two most important bottlenecks i.e. loss of memory locality and large p-slice sizes. Selective Replication improves the memory locality using simple profiling by the compiler and achieves performance close to that of a perfect replication. In addition, we proposed Slice Specialization reduces the overhead of p-slices used by the architecture, by selectively removing predictable branches from the p-slice and the instructions which are needed to compute these branches. This technique generates different slices for all the possible paths determined by the predictable branches. Using the specialized slices and selective replication, we notice a 25% average improvement in performance over a single thread execution. The benefits over conservative slices using the conventional SVC cache are on average 15%. Also, the slice specialization technique gives a performance benefit of 4% on average over static branch pruning based on biased branches. Besides, the various other bottlenecks discussed in this paper point to possible directions of future research on improving performance of speculative architectures

ACKNOWLEDGMENT

This work has been supported by the Spanish Ministry of Education and Sc. under grants TIN2007-61763 and TIN2004-03072 &Catalan Govt. under grant 2009SGR1250.

REFERENCES

[1] M. Agarwal, K. Malik, K. Woley, S. Stone and M. Frank, "Exploiting Postdominance for Speculative Parallelization", in Procs. Of the Int. Symp. on High-Performance Computing Architecture, pp. 295-205, 2007.

[2] H. Akkary, M.A. Driscoll, "A Dynamic Multithreading Processor", in Proc. of the 31st Int. Symp. on Microarchitecture, 1998.

[3] 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.

[4] 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.

[5] 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.

[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] E. Courses and T. Surveys, "Tolerating Memory Latency through Software-controlled Pre-execution in Simultaneous Multithreading Processors", in Procs. of the 28th Int. Symp. on Computer Architecture, 2001.

[8] P. Dubey et al. "Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grained multithreading". In Procs. of the Int. Conf. on Parallel Architectures and Compilation Techniques, 1995

[9] J. Ferrante, K. Ottenstein and J. Warren, "The program dependence graph and its use in optimization", ACM Transactions on Programming Languages and Systems (TOPLAS), 9(3), 1987.

[10] 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.

[11] S. Gopal, T. Vijaykumar, J. Smith and G. Sohi, "Speculative Versioning Cache", In 4th Int. Symp. on, High-Performance Computing Architecture 1998.

[12] W. Liu et al., "POSH: a TLS Compiler that exploits program structure", in 11th Symp. on Principles and Practice of Parallel Programming, 2006.

[13] C. Madriles et al., "Mitosis: Speculative Multithreaded Processor based on Pre-Computation Slices" In IEEE Transactions on Parallel Distributed Systems, 2008.

[14] P. Marcuello and A. González, "Clustered Speculative Multithreaded Processors. In Procs. of the 13th Int. Conf. on Supercomputing, 1999.

[15] P. Marcuello, J. Tubella and A. González, "Value Prediction for Speculative Multithreaded Architectures", In Procs. of the Int. Symp. On Microarchitecture, 1999.

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

[17] S. Muchnick. "Advanced Compiler Design and Implementation", Morgan Kaufmann, 1997.

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

[19] Y. Sazeides and J. Smith, "Implementations of Context Based Value Predictors" Univ. of Wisconsin Technical Report ECE97-8, 1997.

[20] G. Sohi, S. Breach and T. Vijaykumar, "Multiscalar processors", in 25th Int. Symp. on Computer Architecture ,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, 1998

[22] 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

[23] D. Tullsen, "Simulation and Modeling of a simultaneous multithreading processor", in Procs. of the 22nd Int. Conference for the Resource Management Performance Evaluation of Enterprise Computing Systems, CMG. Part 2(of 2), 1996.

[24] T. Vijaykumar, "Compiling for the Multiscalar Architecture", PhD thesis, University Of Wisconsin, 1998.

[25] F. Warg and P. Stenstrom, "Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms", in Procs. of the 10th Int. Conference on Parallel Architectures and Compilation Techniques, 2001.

[26] A. Zhai, C. Colohan, J. Steffan, and T. Mowry, "Compiler optimization of memory-resident value communication between speculative threads", in Procs. of the Int. Symp. On Code Generation and Optimization, 2004.

[27] H. Zhong, M. Mehrara, S. A. Lieberman, and S. A. Mahlke, "Uncovering Hidden Loop Level Parallelism in Sequential Applications", in Procs of the 14th Int. Symp. on High-Performance Computing Architecture, 2008.

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

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