Post-Pass Binary Adaptation for Software-Based Speculative Precomputation

Steve S.W. Liao, Perry H. Wang, Hong Wang, Gerolf Hoflehner[†], Daniel Lavery[†], John P. Shen
Microprocessor Research — Intel Labs
Intel Compiler[†] — Software and Solutions Group

{shih-wei.liao, perry.wang, hong.wang, gerolf.f.hoflehner, daniel.m.lavery, john.shen}@intel.com

ABSTRACT

Recently, a number of thread-based prefetching techniques have been proposed. These techniques aim at improving the latency of single-threaded applications by leveraging multithreading resources to perform memory prefetching via speculative prefetch threads. Software-based speculative precomputation (SSP) is one such technique, proposed for multithreaded Itanium models. SSP does not require expensive hardware support—instead it relies on the compiler to adapt binaries to perform prefetching on otherwise idle hardware thread contexts at run time. This paper presents a post-pass compilation tool for generating SSP-enhanced binaries. The tool is able to: (1) analyze a single-threaded application to generate prefetch threads; (2) identify and embed trigger points in the original binary; and (3) produce a new binary that has the prefetch threads attached. The execution of the new binary spawns the speculative prefetch threads, which are executed concurrently with the main thread. Our results indicate that for a set of pointer-intensive benchmarks, the prefetching performed by the speculative threads achieves an average of 87% speedup on an in-order processor and 5% speedup on an out-of-order processor.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *compiler, optimization, code generation, memory management.*

General Terms
Measurement, Performance, Design, Experimentation, Algorithms.

Keywords
Long-range thread-based prefetching, pointer, slicing, slack, chaining speculative precomputation, speculation, prediction, scheduling, post-pass, dependence reduction, loop rotation, delay minimization, triggering.

1. INTRODUCTION

Memory latency has become a critical bottleneck in achieving high performance on modern processors. Today, many large applications are memory intensive, as both their data working set and the complexity to predict their memory accesses increase. Despite continued advances in cache design and development of new prefetching techniques, the memory latency problem persists and escalates especially with *pointer-intensive* applications, which tend to defy conventional stride-based prefetching techniques. One solution is to overlap memory stalls in one program with the execution of useful instructions from another program, as done in emerging simultaneous multithreading (SMT) processor architectures [10][15][22][28]. In addition to improving multitasking throughput, SMT has also been used to improve the performance of *single-threaded* applications by leveraging speculative threads to perform cache prefetches on behalf of the main (or non-speculative) thread [25]. A speculative thread executes code to precompute memory addresses and issue prefetches. Instead of using a complex address pattern predictor, this pre-execution approach uses the program itself as a predictor to prefetch for a pointer-intensive program accurately and efficiently.

Various forms of such thread-based prefetching have been proposed recently. Examples include Collins et al.'s *speculative precomputation* [7], Luk's *software controlled pre-execution* [21], Roth and Sohi's *data driven multithreading* [25], and Zilles and Sohi's *speculative slices* [34]. These studies demonstrated the performance potential of thread-based prefetching by assuming the availability of hardware and/or compiler support. In this paper, we introduce an automated tool for transforming application code in order to attach prefetch threads in the binary. The aim of this paper is to demonstrate the feasibility of automatically generating binaries for thread-based prefetching and the effectiveness of the resulting binaries. To our knowledge, this work is the first to automate the entire process of extracting dependent instructions leading to target operations, identifying proper spawning points and managing inter-thread communication to ensure timely pre-execution.

Our tool is post-pass because it does not modify the normal compilation steps, but rather is invoked after the compilation process. The tool is based on the *speculative precomputation* (SP) paradigm for future Itanium^{TM} processors [16]. SP utilizes hardware thread contexts to execute *precomputation slices* (p-slices), which consist of instructions that compute the memory addresses for prefetching [7]. Speculative threads can be spawned by one of two events: a *basic trigger*, which occurs when a designated trigger instruction in the non-speculative thread is retired, or a *chaining trigger*, by which one speculative thread explicitly spawns another. Collins et al. demonstrated that long-range prefetching using chaining triggers is the key to high performance via speculative precomputation [7]. As a proof of concept, they manually find the chaining triggers in the binary. Collins et al. later proposed *dynamic speculative precomputation*, which shows the implementation of an all-hardware approach [6]. In contrast, our work uses the SMT model without expensive hardware support and relies on the post-pass compilation to generate p-slices and to place triggers judiciously. Instead of constructing p-slices dynamically, the post-pass tool examines code regions and extracts p-slices statically with profiling feedback. To maximize the concurrent usage of available memory bandwidth, the chaining triggers inside the p-slices are scheduled early across multiple threads. We also traverse the dependence graph to identify and embed basic triggers in the main thread's code.

We show that the tool is effective for a set of seven pointer-intensive benchmarks with exploitable parallelism among the prefetches in the speculative threads and memory accesses in the main thread. The algorithms employed in the tool effectively schedule the p-slices and triggers so that speculative threads can run ahead of the main thread to perform effective prefetches. The tool improves the performance by 87% on an in-order processor and by 5% on an out-of-order processor. SSP provides a greater benefit for the former, because the latter already hide some memory stall cycles. Finally, it is also important to examine whether the automated results match the results from hand adaptation in finding slices and locating triggers. We show that the former loses at most 20% of the performance on the in-order processor and 27% on the out-of-order processor, compared with the two hand-tuned binaries available from our previous work using the same machine model [31].

The rest of the paper is organized as follows. Section 2 describes the SSP paradigm and highlights the design of two research Itanium machine models and a high-level overview of our tool. We describe the post-pass compilation algorithm and evaluate the resulting binaries in Sections 3 and 4, respectively. Section 5 discusses the related work and Section 6 concludes the paper.

2. SOFTWARE-BASED SPECULATIVE PRECOMPUTATION

In speculative precomputation, a run-time event (either a basic or a chaining trigger) invokes the execution of a p-slice as a speculative thread. The thread precomputes and prefetches the address accessed by a load that misses the cache frequently, hereafter called a *delinquent* load. Once the trigger is reached, the load is expected to appear later in the instruction stream of the main thread, hence the speculatively executed p-slice can reduce the cache misses in the main thread. In general, any long-latency operation can be viewed as a delinquent operation and become a potential candidate to benefit from speculative precomputation. SSP uses the otherwise idle hardware thread contexts to execute p-slices. The post-pass tool ensures that no store instructions are included in the precomputation. The speculative execution of p-slices does not alter the architecture state of the main thread. Consequently, the integration [25] of results from speculative threads into the main thread becomes unnecessary. Although prefetching wrong addresses may hurt performance, the SSP paradigm does not require p-slice computation to satisfy the correctness constraints, since the precomputation is prevented from modifying the main thread's architecture states. Thus, SSP enables many optimizations in the speculative threads by separating the performance issue from the correctness issue.

SSP assumes no expensive hardware support. It uses (1) existing lightweight exception-recovery mechanisms in Itanium [26] to spawn a thread and (2) existing memory buffers in the processor to transfer the live-in values from a parent thread to its child thread, as detailed in Section 3.4.2. Thus, the machine does not need to have special flash-copying hardware between register files of the two thread contexts. SSP relies on the software tool to generate p-slices and code for transferring live-in values from the main thread. In the following sections, we first describe our Itanium machine models and then the post-pass compilation tool that adapts binaries for SSP.

2.1 Research Itanium Models

We investigate both in-order execution and out-of-order execution Itanium machine models. The in-order model uses a 12-stage pipeline that resembles a two-bundle wide Itanium [26], with each bundle consisting of three instructions. Both models provide an SMT mechanism with four hardware thread contexts. Compared to the baseline in-order model, the out-of-order (OOO) model assumes four additional front-end pipe stages to account for the extra OOO complexity, such as register renaming and instruction scheduling stages. The expansion queue in the in-order machine is per-thread 16-bundle long. In contrast, the OOO model has a per-thread 255-entry reorder buffer and an 18-entry reservation station. Table 1 specifies cache and memory latency information, along with processor details. To account for future processor generations, the models are designed with higher memory latencies than current product generations.

Table 1. Modeled Research Itanium Processor

Threading	SMT processor with 4 hardware thread contexts.
Pipelining	In-order: 12-stage pipeline. OOO: 16-stage pipeline.
Fetch per cycle	2 bundles from 1 thread or 1 bundle each from 2 threads
Branch predict.	2k-entry GSHARE. 256-entry 4-way associative BTB.
Issue per cycle	2 bundles from 1 thread or 1 bundle each from 2 threads
Function units	4 int. units, 2 FP units, 3 branch units, 2 memory port
Register files per thread	128 integer registers, 128 FP registers, 64 predicate registers, 8 branch registers, 128 control registers.
Cache structure	L1 (separate I & D): 16KB each. 4-way. 2-cycle latency. L2 (shared cache): 256KB. 4-way. 14-cycle latency. L3 (shared cache): 3072KB. 12-way. 30-cycle latency. Fill buffer: 16 entries. All caches have 64-byte lines.
Memory	230-cycle latency. TLB Miss Penalty: 30 cycles.

Spawning a speculative thread involves allocating a hardware thread context for the thread, and providing to the thread context the address of the first instruction of the speculative thread. If a free hardware context is not available, then the spawn request is ignored. We use the lightweight exception-recovery mechanism to raise an exception when a free hardware context is available for thread spawning. To implement live-in buffers for inter-thread communication, we utilize the backing store of the Register Stack Engine on the Itanium processor family [26]. Our software tool adds code to copy necessary live-in values into this on-chip buffer, which is shared by both threads. The values can then be copied to the register file in the newly spawned thread, eliminating the possibility of inter-thread hazards where a register may be overwritten before a child thread has read it. In our experiment, the number of live-in values is relatively small as described in Section 4.

2.2 Post-Pass Compilation Tool

In contrast to *dynamic speculative precomputation* [6], SSP uses software to identify p-slices and trigger points prior to run time. The tool adapts the binaries to enhance them to achieve performance speedups. Figure 1 shows the tool flow of the system. To minimize changes to the normal compilation process, we encapsulate the tool in a post-pass phase. Partly because of this strategy, this encapsulation allows us to reuse the same tool in a future binary translation tool when the source code is not available.

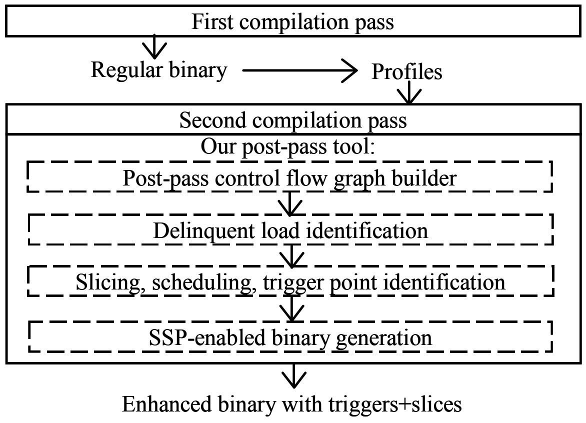

First compilation pass
Regular binary
Profiles
Second compilation pass
Our post-pass tool:
Post-pass control flow graph builder
Delinquent load identification
Slicing, scheduling, trigger point identification
SSP-enabled binary generation
Enhanced binary with triggers+slices

Figure 1. Tool flow for generating SSP-enhanced binaries

The first compilation pass generates the regular binary. In the second pass, we use the profiling information collected from running the original binary to enhance the binary for SSP. The post-pass tool first reads in the compiler intermediate representation (IR) and the control flow graph (CFG). This representation is also used by the code generator at the code emission stage where the IR exactly matches the hardware instructions in the binary [4]. Each CFG node is also annotated with run-time frequency information.

Since we target only delinquent loads, our tool needs to first identify them in the program. For many programs, only a small number of static loads are responsible for the vast majority of cache misses. The tool uses the cache profiles from the simulator to identify the top delinquent loads that contribute to at least 90% of the cache misses. Figure 2 shows that the tool successfully locates the loads that greatly impact performance for both in-order and OOO research models. These programs are from the SPEC CPU2000 [14], and Olden benchmarks suites [5]. In Figure 2, the first bar in each category shows the speedup assuming a perfect memory subsystem where all loads hit in the L1 cache. The phenomenal speedups confirm that these are memory-intensive programs. The second bar in each category represents the speedup when the delinquent loads are assumed to always hit in the L1 cache. This information also provides us the upper bound on what the post-pass tool can achieve. In most cases, eliminating performance losses from only the delinquent loads yields much of the speedup achievable by zero-miss-latency memory. Figure 2 also shows that, compared with the in-order model, the OOO model has less room for improvement via SSP.

After delinquent load identification, the tool computes the *program slice* of identified loads' memory addresses, defined as the set of instructions that contribute to the computation of the address for the speculative thread to execute before issuing the memory prefetch. Slicing can reduce the code to only the instructions relevant to the computation of an address. Using slicing and profiling, the tool extracts a minimal sequence of instructions to produce the addresses of delinquent loads, and schedules the slice to ensure timely prefetches.

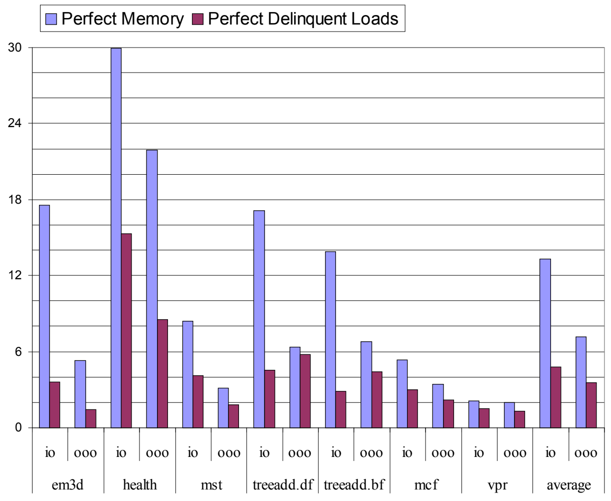


Figure 2. Speedup when assuming perfect memory vs. when assuming delinquent loads always hit the cache.

Finally, the tool locates the desirable triggering points in the main thread for spawning the p-slices in order to minimize the communication (i.e., the number of live-in values) between the threads, while ensuring enough prefetching distance. During code generation, the tool inserts code in the original binary that invokes the p-slices. Also, once the p-slices are built, they are compiled with live-in copying code and appended after the function in which the trigger resides in the program binary. Since our SSP tool targets hardware without the flash-copy mechanism, it also performs live-in analysis to generate code to copy live-in values. The detailed post-pass algorithm for SSP is presented in the following section.

3. POST-PASS ALGORITHMS FOR SSP

The post-pass tool adapts the binary so that the speculative thread prefetches the data of the delinquent load with large enough *slack,* which is defined as the execution distance between the main thread and the speculative thread. Specifically, the slack of a thread-based prefetch is defined as:

$$slack(load,prefetch)=timestamp_{main}(load)-timestamp_{spec}(prefetch),$$

where $timestamp_{main}(load)$ and $timestamp_{spec}(prefetch)$ denote the time when the targeted load is executed in the main thread and when the prefetch is executed in the speculative thread, respectively. The above mathematical definition is *ideal* and indeed is likely to vary each time the trigger and the load are encountered. In practice, our SSP tool employs a heuristic in the following sections to estimate this slack value. Positive slack values are desirable, because they indicate the speculative thread executes the instructions ahead of the main thread. The goal of our algorithm is to find slices that contain large enough slack, but not too large. Having too much slack may cause adverse cache interference, or the prefetched data may be replaced by the time the main thread encounters the delinquent load.

We first present the slicing tool and scheduling algorithm in Sections 3.1, and 3.2. Section 3.3 describes the identification of all the trigger points in the main thread's binary for launching the slice computation. Finally, Section 3.4 describes how the tool generates the new binary including p-slices and code for copying their live-in values from the main thread.

3.1 Slicing for Speculative Precomputation

To generate p-slices, the post-pass tool first reads in the program representation and identifies the delinquent loads. The tool employs cache profile data from the simulator to determine the set of loads that contributes to the majority of cache misses. Typically, only a small number of static loads are selected. Next, the tool applies program slicing to each load address to reduce the code to only the instructions relevant to the load's address computation.

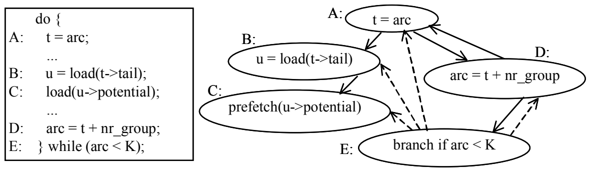
The slice selection process determines the content for speculative precomputation. A common technique for computing program slices is to transitively follow all of the control and data dependence edges originating from the reference being sliced. But this notion of slicing, originally proposed by Weiser [32], may result in large slices. Specifically, this original context-insensitive algorithm may suffer from inaccuracy due to unrealizable paths [18] because they compute the slice as the union of all the statements in all paths that reach the reference without *matching* in- and out-parameter bindings. To solve this imprecision problem, the slicing algorithm proposed in [20] defines a context-sensitive slice of a reference *r* with respect to a calling context *C* as:

$$slice(r, [c_1,...,c_n]) = slice(r,\phi) \cup \bigcup_{f \in F} slice(contextmap(f,c_n),[c_1,..., c_{n-1}])$$

In the equation, let $C = [c_1, ..., c_n]$ be the call sites currently on the call stack maintained by the post-pass tool, with c_n being the one on the top of the call stack, and *F* be the subset of the formal parameters of the procedure upon which *r* depends on. The function *contextmap(f,c)* returns the actual parameter passed to a formal variable *f* at call site *c*. *Slice(r,ϕ)* represents the set of instructions found by transitively traversing the dependence edges backwards within the procedure in which *r* is located and its callees. In summary, the computation of a context-specific slice only builds the slice up the chain of calls on the call stack, which reduces the size of the slice. Our slicing tool also ignores loop-carried anti dependences and output dependences in order to produce smaller slices.

3.1.1 Region-Based Slicing

A slice with large slack ensures the corresponding speculative thread runs early enough to prefetch timely. On the other hand, unnecessary expansion of slack may cause prefetched data to be evicted from cache before its use. In this paper, we propose a region-based slicing method that allows us to increase the slack value incrementally from one code region to its outer ones, to find slices with large enough slack to avoid untimely prefetches, but small enough slack to avoid early eviction. A region represents a loop, a loop body, or a procedure in the program. Derived using CFG information, a region graph is a hierarchical program representation that uses edges to connect a parent region to its child regions, that is, from callers to callees, and from an outer scope to an inner scope. In addition, we build a dependence graph that contains both control and data dependence edges. For each delinquent load identified, we locate the backward slice of the load address in the dependence graph, starting from the innermost region in which this load occurs. The demand-driven slice computation accepts a request on a delinquent load's address and uses a primarily recursive descent algorithm region by region until the slack of precomputing the address is large enough. We will use the simplified code excerpt from the function `primal_bea_map()` in the *mcf* benchmark from SPEC CPU2000 [14] as an example to illustrate the algorithms in this paper. Figure 3 shows the code excerpt and a corresponding slice in the loop region. A solid arrow denotes a data dependence, and a dashed arrow denotes a control dependence. Note that there are no false loop-carried dependences in this figure. By finding the slice of the address of the delinquent load "`t->tail->potential`", SSP executes fewer instructions than the main thread does. Because many instructions in the loop body are not present in the slice, the region-based slicing achieves enough slack and selects this region for precomputation. The region selection is detailed in Section 3.4.1.



```
    do {
A:    t = arc;
      ...
B:    u = load(t->tail);
C:    load(u->potential);
      ...
D:    arc = t + nr_group;
E:  } while (arc < K);
```

(a) Simplified excerpt. load(u->potential) is a delinquent load.

(b) Slice of the address of the delinquent load in the loop region. Arrows are dependence edges.

Figure 3. Simplified code example from *mcf* and a slice.

When the current region being examined is a loop or a procedure with recursive procedure calls, this region may introduce recurrences in the equation to compute the slice of a reference *r*. To allow the reuse of previous slices, a slice summary is recorded to exploit redundancy in slice computation [20]. We resolve recurrences encountered when computing a slice summary by using an iterative algorithm that is guaranteed to reach a fixed point. The algorithm locates the recurrences by using a stack to track the slice summaries of the references being constructed. If a new slice summary to be computed is already on the stack, a recurrence is detected and the algorithm simply uses the approximate slice summary already built. If the approximate summary is used in the computation of another summary, we deem that the latter depends on the former. All such dependence relationships are recorded. When the approximate slice summary is finalized, its dependent summaries are placed on a worklist. The algorithm finds the fixed point solution by iteratively removing a slice summary from the worklist, recomputing it, and adding new dependents on the worklist if the result changes. The fixed point computation terminates when the worklist is empty. The fixed point solution is guaranteed to terminate because the number of static instructions in a program is finite.

3.1.2 *Speculative Slicing*

Another source of enhancing the likelihood of timely prefetch is from removing unprofitable instructions from a given p-slice. To achieve that, we use speculation techniques to reduce the size of a slice. Since the SSP threads do not alter the architecture state, unlike traditional speculation mechanisms, we do *not* have to generate recovery code when a mis-speculation is encountered. Although reducing the size using speculation techniques is prone to lower the accuracy of address computation, we exploit profile information to maintain a reasonable tradeoff between the size and accuracy of slices.

Empirical results have shown that pure static slicing may introduce a large number of unnecessary instructions in the slice due to the lack of run-time information [19]. However, pure dynamic slicing [2] can often be prohibitively expensive. We use a hybrid slicing [13] approach to improve precision while maintaining efficiency. This approach, called control-flow speculative slicing, alleviates the imprecision problem of static slicing by exploiting block profiling and dynamic call graphs. This control flow information is used to filter out unexecuted paths and unrealized calls. In our experience, the quality and efficiency of static analysis drops rapidly if the code being analyzed contains indirect calls that we cannot resolve. To tackle this issue, we instrument all the indirect procedural calls to capture the call graph during profiling, and provide the result back to the slicing algorithm. The slicing tool currently does not use data speculation information to trim the slices, since the static disambiguator in the compiler has proven to be effective [11].

In summary, both region-based slicing and speculative slicing perform *slice-pruning* operations [20]. That is, after the slack value becomes large enough, region-based slicing prunes the traversals of the dependence edges to avoid prefetching the data too early. Similarly, speculative slicing prunes the slice computation at those nodes that are unlikely to yield effective speculative precomputation. Slice-pruning is key for SSP, because a precise slicing tool may not produce useful slices if the precomputation does not result in timely prefetches.

3.2 Scheduling the Slice

As defined in Section 3.1, a slice was simply considered a set of instructions. In this section, we describe the generation of an *execution* slice. Using the slice of the code in Figure 3 as an example, we show a corresponding execution slice in Figure 4. Scheduling and synchronization determine when and which instructions in the slices are assigned to the available thread contexts. Our algorithm aims at scheduling enough slack for the prefetches. For each targeted code region, the scheduling algorithm considers two precomputation models: *chaining speculative precomputation* (chaining SP) and *basic speculative precomputation*. Chaining SP in this context means using one precomputation thread to spawn another precomputation thread.

```
    do {
A:      t = arc;
B:      u = load(t->tail);
C:      prefetch(u->potential);
D:      arc = t + nr_group;
E:  } while (arc < K);
```

Figure 4. An execution slice for the code in Figure 3.

In this work, both threads execute identical code and are collectively referred to as *chaining threads*. Chaining SP allows parallelism among the chained thread invocations. Scheduling for chaining SP is often the key to create enough slack, because the spawning inside the speculative threads enables *long-range* prefetching without incurring the spawning overhead on the main thread. In contrast, basic SP uses only one speculative thread. The scheduling algorithms for both SP models are presented in Section 3.2.1 and 3.2.2, respectively.

3.2.1 *Scheduling for Chaining SP*

Although parallelism is not a necessary condition for chaining SP to produce enough slacks, increased thread-level parallelism in the chaining threads is important as its existence can be exploited to produce much larger slack between the prefetches and the main thread. Hence, given a p-slice generated by the slicing tool in Section 3.1, our algorithm constructs a do-across prefetching loop whose iterations are executed in parallel by the chaining threads. This parallel execution is constrained by the fine-grained synchronization between the threads, because a do-across loop may contain loop-carried dependences. To increase thread-level parallelism, the algorithm first reduces the number of dependences (described in Section 3.2.1.1) and then minimizes the delays among the threads for the given set of dependences (described in Section 3.2.1.2). A chaining thread can be setup to iterate over multiple iterations or targets for multiple regions. For simplicity, the tool currently uses one chaining thread to target one iteration in a loop region of the main thread. Our scheduling algorithm is also applicable to *speculative multithreading*, since that paradigm needs to address a very similar delay minimization problem [9].

Note that the scheduling algorithm requires latency information in combination with the dependence graph. The latency of a memory operation is determined by cache profiling, and the machine model provides latency estimates for other instructions. The latency information is annotated on a dependence graph edge. A slice and the annotated dependence edges between the nodes in the slice form the dependence graph of the slice, as shown in Figure 3(b). Using the region graph information, the instructions in a region are scheduled together. Currently we do not move instructions across region boundaries.

3.2.1.1 *Dependence Reduction*

In addition to removing loop-carried false dependences, we develop two optimizations for dependence reduction, *loop rotation* and *condition prediction*. The parallelism of chaining SP is highly sensitive to actual code sequence in the slice for a loop iteration. For example, if the first instruction of a chaining thread depends on the last instruction of the previous chaining thread, no amount of thread scheduling efforts can improve further as the thread executions are serialized. However, it is possible to prevent SSP thread serialization by reordering the SSP code. We are able to reorder this code without affecting the main thread since, unlike traditional parallelization, our main thread and speculative threads execute different code. *Loop rotation* reduces loop-carried dependence from the bottom of the slice in one iteration to the top of the slice in the next iteration. The algorithm greedily finds the new loop boundary that converts many backward loop-carried dependences into true intra-iteration dependences. The algorithm enforces the property that new boundary does not introduce new loop-carried dependences. Loop-carried anti dependences and output dependences are ignored when scheduling the chaining SP code. By shifting the loop boundary, we may expose more parallelism for chaining SP.

The second optimization is to use the prediction techniques on some conditional expressions in the slice. For instance, spawning chaining threads often occurs when the loop contains a delinquent load, because the load misses many times during the program execution. The spawn condition becomes highly predictable. Thus, we use the prediction when the parallelism is little, or the delinquent load occurs before the spawning. The computation of the chaining spawn condition usually incurs loop-carried dependences; hence by using prediction, the delay among the threads is oftentimes reduced. The prediction breaks the dependences leading to the spawn condition after predicting the spawn condition. After such removal of dependences, more instructions can be executed after the spawning point instead of before the point. This results in higher thread-level parallelism.

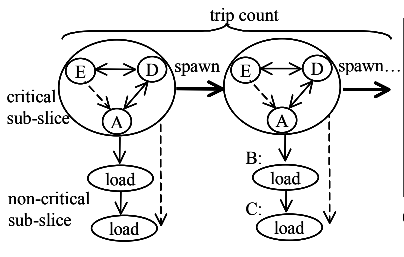
3.2.1.2 Delay Minimization

The key for an efficient schedule is to minimize the delay of the consecutive chaining threads when there is any inter-slice dependence. Minimizing the delays results in larger slack values. Cytron showed that the delay-minimizing scheduling problem is NP-complete, even for simpler loop parallelization problems that assume no loop-independent dependences exist [9]. Hence Cytron proposed a two-phase scheduling heuristics, which first partitions the dependence graph and then reorders the nodes in each partition according to their priority value. Specifically, the first phase in [9] partitions the graph by level-sorting the instructions considering only the forward dependences. The second phase computes the node priority by subtracting the maximal edge latency among the incoming backward edges from the maximal edge latency among the outgoing backward edges from the node. Our algorithm is two-phase as well: graph partitioning followed by list scheduling on the resulting acyclic graph. While Cytron used adjacent edges to compute the node's priority, we use the maximum node height in the dependence graph of the slice as the priority value of a node. The two phases are presented in Section 3.2.1.2.1 and 3.2.1.2.2, respectively.

3.2.1.2.1 Graph Partitioning

We use the strongly connected components (SCC) algorithm in [24] to partition a dependence graph. In a strongly connected subgraph, there exists a path from a node to any other nodes in the same subgraph. The SCC is defined as the maximal strongly connected subgraph. A degenerate SCC contains only one instruction node. Furthermore, we form SCC's without considering any false loop-carried dependences. In comparison, Cytron's partitioning phase is relatively restrictive since all loop-carried forward dependences are enforced to produce code schedule. Any occurrence of non-degenerate SCC in the dependence graph consists of one or more *dependence cycles*, which implies the existence of loop-carried dependences. A speculative thread executing a loop iteration must resolve the loop carried dependence in a dependence cycle in order for the next chaining thread to start executing the same dependence cycle in the next iteration. Hence the span of the dependence cycle should be minimized. To achieve this, our heuristics schedules all instructions in an SCC first before scheduling instructions in another SCC.

Figure 5(a) shows the SCC's in the slice's dependence graph in Figure 3(c). The three instructions, A, D, and E, in Figure 3 are merged into one SCC node in Figure 5. Each of the load instructions, B and C, forms a degenerate SCC node, because it is not part of any dependence cycles. Node `B` and `C` do not compute live-in values for the next chaining thread. Since the partitioning uses SCC's to tighten the dependence cycles into one SCC, the algorithm can then schedule the chaining thread to first execute the non-degenerate SCC in Figure 5, spawn the next chaining thread, and then execute the two degenerate SCC's in Figure 5. This partitioning is important for the Itanium, because when the load instruction B in Figure 3 misses in cache, executing instruction C, which depends on the outcome of instruction B, will stall an in-order processor while waiting for the miss being serviced. Chaining SP triggers several speculative threads to overlap multiple prefetches. To overlap the miss cycles in the code in Figure 3, chaining SP forms several instances of a pointer dereference and executes the chaining threads in parallel, as shown in Figure 5(a).



(a) Chaining SP (Thin arrows are dependence edges)

```
L1:
A:  t = arc;
D:  arc = t + nr_group;
    /* live-in: arc */
E:  if (arc < K) spawn(L1);
B:  u = load(t->tail);
C:  prefetch(u->potential);
    thread_kill_self();
```

(b) The automatically generated do-across prefetching loop for SSP

Figure 5. Chaining SP for the code in Figure 3

3.2.1.2.2 Scheduling an Acyclic Graph

The resulting partitioned graph is a directed acyclic graph (DAG) and thus can be scheduled by list scheduling algorithm. We select the forward cycle scheduling with maximum cumulative cost heuristics. As the heuristics accumulates the cost, or latency, for each path, the node with longer latency to the leaf nodes of the slice has a higher priority. If two nodes have the same cost, the node with the lower instruction address in the original binary has a higher priority. Finally, the instructions within each non-degenerate SCC are list scheduled by ignoring all the loop-carried dependence edges. The resulting code is shown in Figure 5(b). Node `B` and `C` have smaller node heights than the non-degenerate SCC does and are assigned lower priority.

As shown in Figure 5, we partition the slice into a *critical sub-slice* and a *non-critical sub-slice*. The term critical sub-slice denotes the set of SCC nodes before the spawn point. The remaining nodes on the DAG become the non-critical sub-slice. The instructions in the former are often in a long dependence chain due to the chaining effect. The non-degenerate SCC in Figure 5(a) is identified as the critical instructions. On the other hand, no future instructions depend on the non-critical sub-slice, which does not compute the live-in values for the next chaining thread.

Although heuristics are needed for this NP-complete problem [9], it is important to investigate whether the output of the algorithm is comparable to the optimal schedule. Cooper et al. showed that if the available instruction-level parallelism (ILP) is very small, the forward scheduling with maximum dependence height heuristics performs very well on real codes and there is little opportunity for improving its performance further [8]. The available ILP is computed as the length of the worst possible schedule divided by the length of the best possible schedule. This is equivalent to the ratio of the sum of the latencies of all operations in the dependence graph to the critical path length. Our tool automatically computes the available ILP from the dependence graph. We observe that the dependence chains leading to the delinquent loads do not exhibit much instruction-level parallelism. Thus, simply using the height of a node in the dependence graph is a reasonable metric for assigning priority values.

For chaining SP, we compute the slack of a slice at the *i*-th iteration of the generated do-across prefetching loop for the region as:

$$slack_{csp}(i)=(height(region)-height(critical\ sub\text{-}slice) \\ -latency(copy\ live\text{-}in\text{'}s\ and\ spawn)) * i$$

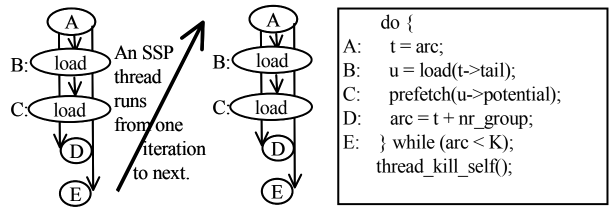
The function *height* computes the height of the dependence graph for a region or for a slice by finding the maximum of the node heights in a region or a slice. The above equation assumes that there are enough hardware threads. We model the communication overhead (copying live-in's) as a factor in decreasing the slack.

3.2.1.2.3 Synchronizations across the Threads

After a slice is scheduled by the two phases above, we need to insert synchronizations in the resulting do-across prefetching loop due to the remaining loop-carried dependences. The partitioning algorithm via SCC facilitates efficient synchronization, because the nodes in the same dependence cycle are tightened into one SCC. Synchronization placements determine the degree of the thread-level parallelism and the synchronization overhead. For instance, if we have more than one non-degenerate SCC nodes, assuming no synchronization cost, synchronizing across threads after the execution of each non-degenerate SCC may result in shorter delays across the threads than synchronizing once after all the non-degenerate SCC's have been executed. However, the former scheme requires more instances of handshaking and the implementation of faster synchronization hardware primitives than the latter. To minimize both the synchronization overhead and the hardware support, the tool currently allows one point-to-point communication between the threads. Thus, we synchronize only once after all the non-degenerate SCC's have been executed. As a result, we pass all the live-in values at the spawn point.

3.2.2 Scheduling for Basic SP

Basic SP does not allow a speculative thread to spawn another thread. The advantage of using basic SP is that it saves the thread spawning and communication overhead. However, basic SP uses only one sequential speculative thread and thus may stall if the thread encounters a data dependence after the delinquent load on an in-order execution machine. Figure 6(a) shows the sequential execution of the loop iterations for the slice in Figure 3. Because the slice contains pointer dereferences, the speculative thread stalls at the second load when the first load misses in the cache. This reduces the memory parallelism across different loop iterations using basic SP.



```
     do {
A:     t = arc;
B:     u = load(t->tail);
C:     prefetch(u->potential);
D:     arc = t + nr_group;
E:   } while (arc < K);
     thread_kill_self();
```

(a) Basic SP. Thin arrows are loop-independent dependence edges.

(b) The automatically generated prefetching loop for Basic SP

Figure 6. Basic SP for the code in Figure 3

The instructions in Figure 3 are list scheduled by ignoring all the loop-carried dependence. The resulting code for basic SP is shown in Figure 6(b). For basic SP, the slack value at the *i*-th iteration of the generated prefetching loop for the region is computed as:

$$slack_{bsp}(i)=(height(region)-height(slice)) * i$$

Contrasting the two equations that compute the slack show that basic SP saves more thread spawning overhead, while chaining SP can provide a higher slack value when a non-critical sub-slice incurs high latency. The basic SP for a loop region is illustrated in Figure 6. If the region is a loop body, basic SP uses a speculative thread to execute one iteration (loop body) and in each iteration of the loop, the main thread triggers new speculative thread for the next iteration, as done in [7].

3.3 Slice Triggering for SSP

After we find the slice inside a region, locating good trigger points in the main thread would ensure enough slack while minimizing the communication between the main thread and the speculative thread. The set of triggers should form a *cut set* on the control flow graph to ensure that each execution path leading to the delinquent load has only one trigger point. As infrequent edges are filtered out in a pre-pass, the optimal solution is to find the minimum total cost of the cut weighted by the frequency, $\Sigma_i (f_i * c_i)$, where f_i and c_i are the frequency and the triggering cost of the edge *i*, described below. Given the set of costs associated with the edges, if we map the problem to the max-flow min-cut problem by representing cost as capacity [12], the complexity for finding the optimal cut is polynomial to the product of the number of the edges and the number of the nodes. However, computing the precise cost is difficult in practice. As a result, our traversal on the dependence graph for finding a trigger point is conservative. During the traversal, we compute the slack of the slice by subtracting the dependence height of the slice from the length of program schedule in the main thread.

With basic SP, the communication between the main thread and the speculative thread would slow down the main thread. The main thread needs to allocate the live-in buffer (LIB) and copy the live-in values to the buffer. Thus, we model the copying overhead as decreasing the slack with a higher cost. Minimizing the live-in copying takes precedence over increasing the slack value. In addition, we maintain control dominance information intra-procedurally and terminate the traversal at the procedural boundaries. With such information, we only consider the nodes that control-dominate the delinquent loads as potential trigger points and prune the slicing when the slack value is too high. If the slack value increase remains low after some traversal, we would prune the slicing as well. This is because continuing the slicing would include too many nodes while the slack value remains the same. Finally, the tool would first place the trigger after the instruction that produces the last live-in to the slice, and then move the trigger points to the immediate control dominant nodes if the slack value of the immediate dominant node remains the same. Note that the trigger points do not need to be in the dependence graph of the slice. By moving the triggers to a control dominance point, several triggers may be combined and thus reduce the number of trigger placements.

3.4 SSP-Enabled Code Generation

Finally, the post-pass tool selects the region and the precomputation model for a delinquent load and then generates the new binary with the p-slices attached.

3.4.1 Selecting Regions and Precomputation Models

If there is no overlap among the execution of different threads, conventional do-across parallelization may yield no speedups. However, parallelizing an application for SSP may still produce enough slack between the main thread and the speculative thread when there is no parallelism. Thus, instead of the degree of parallelism, we use the number of reduced miss cycles and the slack as our optimization targets. The region-based traversal starts with the innermost region where the delinquent load occurs and finds the first region in which the reduced miss cycles for basic or chaining SP is greater than a threshold value. The value is calculated as the product of the cutoff percentage and the miss cycles from cache profiling. We experiment with different percentage values and discover that the resulting performance is not highly sensitive to the percentage as long as it is reasonably selected. To avoid a slice becoming too big that often leads to wrong address calculations, we also stop the traversal of the region graph when it is nested several levels deep. The traversal also stops when the outermost region is reached. If none of the regions reduce the miss cycles beyond the threshold percentage, we pick the region with the largest percentage of miss cycles. However, when the reduced miss cycles are about the same for two regions, we prefer the inner one to the outer.

For a given region, we assign one to be its trip count (or the number of iterations) if the region is not a loop. For a loop region, the trip counts are derived from block profiling if available; otherwise, they are estimated. If the trip count is small or the slack value for basic SP is larger than that for chaining SP, we use $slack_{bsp}$ in Section 3.2.2 to compute the $slack_{sp}$ function below. Otherwise, chaining SP is used. Reduced miss cycle count for a region is computed by summing over all the iterations the number of miss cycles reduced in each iteration.

$$reduced\_misscycle=\Sigma_i\ min(miss\_cycle\_per\_iteration,\ slack_{sp}(i))$$

We record the decision to use chaining or basic SP for each region traversed and the slack values. Thus, the final code may have nested chaining SP. Finally, different slices are combined if they share nodes in the dependence graph.

3.4.2 Generating the Binary

After the selection above, we insert the trigger instructions and generate slices in the binary. The trigger instruction is to bind a spawned thread to a free hardware context. The thread spawning is done via the existing lightweight exception-recovery mechanism in the Itanium architecture. This mechanism uses the speculation check instructions to determine if an exception should be raised for mis-speculation recovery. We take advantage of this feature by introducing a new check instruction, `chk.c`, for available context check. The trigger instruction, `chk.c`, raises an exception if free hardware context is available for spawning. Otherwise, `chk.c` behaves like a `nop`. Figure 7 shows that the tool adapts the binary by replacing a single `nop` instruction with a `chk.c` instruction and by appending to the function the slice code as the exception recovery code.

Function Body	**Attachment 1**	**Attachment 2**
…		
instruction *n*	stub_block_1:	stub_block_2:
instruction *n+1*	copy live-in to buf	copy live-in to buf
…	*spawn* slice_block_1	*spawn* slice_block_2
chk.c stub_block_1		
…	slice_block_1:	slice_block_2:
chk.c stub_block_2	copy live-in from buf	copy live-in from buf
…	slice body	slice body

Figure 7. Code layout in the enhanced binary for SSP

As shown in Figure 7, each spawn instance consists of two blocks, the stub block and the slice block. The stub block is executed by the main thread as the recovery code for a `chk.c` instruction. The stub code copies the live-in values to the live-in buffer and issues an instruction to spawn the speculative thread at its end. The main thread resumes its normal execution after executing the stub block as its recovery code. At the same time, the spawned thread begins executing the slice block, which contains code to copy live in values from the live-in buffer to the thread's register file. The code layout for two thread spawns and their individual trigger instructions are illustrated in the Figure 7. The tool can form a slice block by extracting instructions from various procedures. Finally, to implement a live-in buffer that the main thread and the speculative thread can both access, we use the on-chip memory buffer, which is the spill area for the backing store of the Register Stack Engine on the Itanium processor family, as the live-in buffer [26].

4. EXPERIMENTS AND ANALYSIS

In this section, we begin with a description of the simulation environment and the benchmark suites. We then present the performance data.

4.1 Experimental Framework

The experiments are carried out on SMTSIM/IPFsim, a version of SMTSIM simulator [27] adapted to work with Intel's Itanium simulation environment [29]. This infrastructure is execution-driven and cycle-accurate. It supports a variety of single-threaded and multi-threaded Itanium research processor models, including in-order pipeline, out-of-order pipeline, and SMT. It enables comprehensive performance study of several speculative precomputation machine models.

All benchmarks studied in this paper are compiled with the Intel compiler [4][17] using the advanced instruction-level parallelism techniques such as aggressive use of speculation and prefetching. To ensure that the baseline binaries achieve the best that our compiler can deliver, we use the peak options including profile-guided optimizations and prefetching. While the automated tool leverages production compiler infrastructure [4], it is currently for research investigation only and is not yet intended as a new feature in the production compiler. The tool targets the pointer-intensive applications in the Olden suite [5] and the SPEC CPU2000 suite [14]. The Olden benchmarks consist of the pointer-intensive codes that are known to suffer frequent L2 and L3 cache misses. *Em3d* solves electromagnetic propagation in three dimensions. *Health* models Colombian health care system. *Mst* computes the minimum-spanning tree of a graph. Finally, *treeadd* performs depth-first search on a balanced B-tree. To study prefetching for both depth-first and breadth-first traversals, we enhance the program to perform both. Thus, two versions are used: *treeadd.df* for the depth-first traversal of a balanced B-tree and *treeadd.bf* for the breadth-first counterpart. From the SPEC CPU2000 suite, we use memory-intensive applications such as *mcf* and *vpr* that miss the L2 and L3 cache often. These two benchmarks exhibit an average of about four-times speedup when we assume zero-miss-penalty memory on an in-order model, as shown in Figure 2. *Mcf* is a combinatorial optimization program. *Vpr* performs FPGA circuit placement and routing. For 12 SPEC CPU2000 programs, Aamodt et al. [1] have shown that the average number of stores per slice is only 0.87 and the average slice includes only 22.5 instructions even when following load-store dependences using perfect disambiguation and extracting slices over a window of 256 committed instructions from the main thread. Hence they demonstrate that these programs do not update the dynamic data structures very heavily while executing delinquent loads, and that in any case, exploitable parallelism exists between the prefetch threads and the main thread.

4.2 Slice Characteristics

As shown in Table 2, the post-pass tool successfully locates several static slices to target a small number of selected delinquent loads. The column labeled "Interproc slices" indicates the number of p-slices that are interprocedural. We find that interprocedural slices contribute to larger slack value and hence higher performance. Furthermore, the slice-pruning methods such as speculative slicing and region-based slicing, effectively extracts *short* sequence of instructions to produce the address for a delinquent load. Finally, the last column in Table 2 shows that the average number of live-in values for the slices identified above is relatively small.

Benchmark	Slices (#)	Interproc slices (#)	Average size	Average # live-in
em3d	8	0	10.3	2.8
health	2	1	9.0	3.5
mst	4	1	28.3	4.8
treeadd.df	3	0	11.3	3.0
treeadd.bf	2	0	12.5	4.5
mcf	5	0	14.0	4.4
vpr	6	0	13.5	4.0

Table 2. Slice characteristics

The tool automatically selects basic or chaining SP for a given region. The benchmark *treeadd.df* uses basic SP. Most loops in the benchmark suite use chaining SP. We find that chaining SP produces bigger slacks and achieves long-range prefetching. The performance impact of the scheduling is presented in the next section.

4.3 Performance of SSP-Enhanced Binaries

This section presents the speedups gained by SSP on both in-order and OOO processors, over the baseline in-order processor. In Figure 8, the three bars associated with each application denote the speedup of SSP on the in-order machine, that of the OOO machine, and that of SSP on the OOO machine, respectively. The baseline is the in-order processor without the precomputation threads. SSP achieves an average speedup of 87% over the baseline in-order processor on the seven pointer-intensive benchmarks. Although our machine can only issue one bundle from each of the two threads per clock cycle, SSP achieves at least 2x speedups for *em3d*, *health,* and *treeadd.bf* on the in-order processor. This demonstrates an advantage of the SSP model over traditional parallelization that relies on partitioning data or computation: In the SSP model, one instruction executed by the speculative thread may save more than one clock cycle for the main thread if the instruction is executed early. A single prefetch instruction executed 230 cycles ahead of the main thread may save 230 cycles if the data had only been in the main memory. SSP is effective for in-order processors especially because the in-order pipeline stalls when an instruction attempts to use the destination register of an outstanding load miss. The first bar for each benchmark in Figure 8 shows that the post-pass tool effectively enhances the binary to use multithreading for preventing such pipeline stalls in the main thread's execution.

Even though SSP only targets the delinquent loads, its performance for *health* approaches that of the OOO execution, which attempts to cover all misses. However, we observe that the OOO processor can on average achieve 175% speedup over the baseline in-order processor. The reason is that OOO can execute beyond dependent instructions and tolerate many L1 cache misses on memory-intensive applications. The SSP tool achieves an average of 5% speedup on the OOO processor.

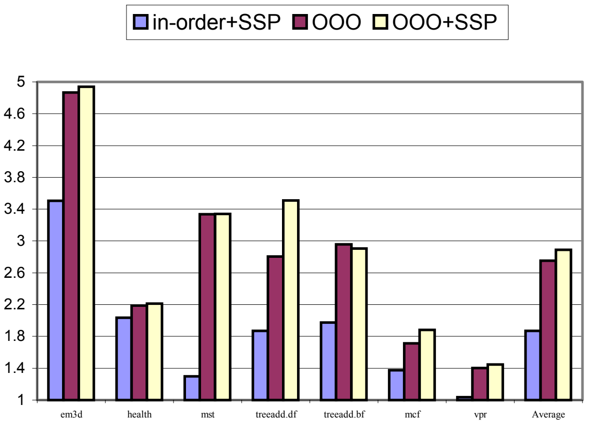


Figure 8. Speedups of SSP, OOO model, SSP+OOO model over the baseline in-order model.

4.4 Dynamic Statistics for SSP-Enhanced Binaries

To evaluate the effectiveness of our tool in detail, we measure the cache miss reduction for the benchmark programs. Figure 9 shows the percentage breakdown of which level of the memory hierarchy is accessed. The height of any bar in the figure is the L1 cache miss rate. In the figure, the four configurations for each benchmark are presented in the following order: the baseline in-order model, the in-order model with SSP, the OOO model, and the OOO mode with SSP. All the partial misses in the figure denote the percentage of accesses to cache lines which were already in transit to L1 cache due to accesses by prior loads from the main thread or from a prefetch.

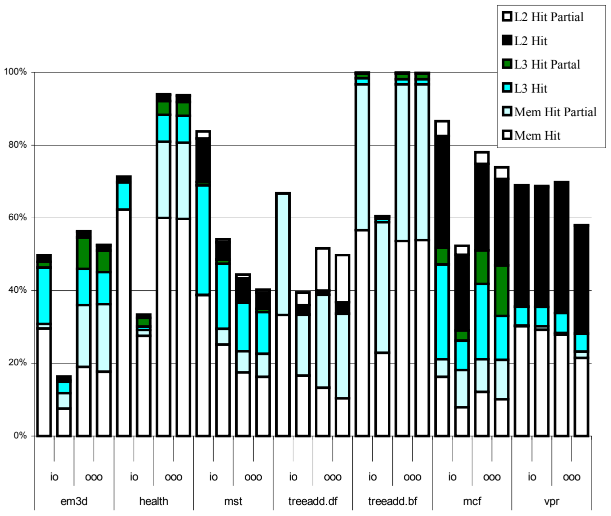


Figure 9. Percentage of where delinquent loads are satisfied when missing in L1. Height of a bar is those loads' miss rate.

Figure 9 shows that on the in-order model, most of the reduction of cache misses happens in the lower cache levels, which are categorized in the bottom portions of the bars in the figure. Thus, the chaining SP schedules many long-range prefetches and hence reduces many L3 and L2 cache misses. Chaining threads can achieve these long-range prefetches because even if one thread stalls due to the in-order execution, many chaining threads can still run ahead and prefetch data. Traditional intra-thread prefetching techniques cannot overcome this stall problem on the in-order model. Although some benchmarks cause more L1 cache misses on the OOO model than on the in-order model, the former model executes the benchmarks faster than the latter. This is because the OOO model overlaps many cache misses with program execution. Furthermore, because OOO hides cache misses better, and relies less on thread-based prefetching, SSP reduces fewer misses on the OOO model than on the in-order model.

Even if a slice computes the load address correctly, the prefetch is useless if it is untimely. The reduction in cache misses in Figure 9 shows not only that the slices compute many addresses correctly but also that the scheduling algorithm generates many timely prefetches. The number of wrong addresses generated by speculative slicing is small for these benchmarks.

4.4.1 Cache Latency Reduction by SSP

To further understand the speedups, we show in Figure 10 the detailed cycle breakdown for SSP on both an in-order and OOO model. All data are normalized to the execution cycle count of the baseline in-order processor. This reveals how much miss penalty SSP manages to reduce at different levels of cache hierarchy. The height of each bar in the figure denotes the cycle counts, normalized to the cycle count of the baseline in-order processor. As shown in Figure 10, the total cycles are *partitioned* into six categories: *L3*, *L2*, *L1*, *Cache+Exec*, *Exec*, and *Other*. The first three, shown as the bottom three partitions of a bar in the figure, denote the miss cycles for L3, L2, and L1 cache respectively, while no instruction is issued for execution. If the cache hierarchy and instruction issue are both active in the same cycle, we account the cycle as *Cache+Exec*. If only the latter is active in a cycle, the cycle belongs to *Exec* category. The *Other* category accounts for all other cycles such as bubble cycles due to branch misprediction. Figure 10 shows that the benchmarks suffer performance loss from cache misses at nearly all levels of the memory hierarchy.

Figure 10 shows that SSP effectively reduces the *L3* cycles, which is the main reason for the 87% speedup on the in-order processor. Over the seven programs, SSP achieves a speedup of 135% on average for the *L3* category. The reason is that the tool enhances the binary to issue long-range prefetches. Because in-order processors are not as latency-tolerant as OOO models, the benefit of using SSP is more dramatic on the former. However, we still observe the reduction in the *L3* cycles for *all* programs on the OOO processor. The reduction is due to the long-range prefetches resulting from SSP, attacking load misses beyond the reach of the OOO instruction window. On top of the OOO processor which can hide both cache misses in most cache levels (especially L1) and functional unit latency, SSP can further reduce the latency on the lower levels of cache hierarchy by performing long-range prefetching for few loads.

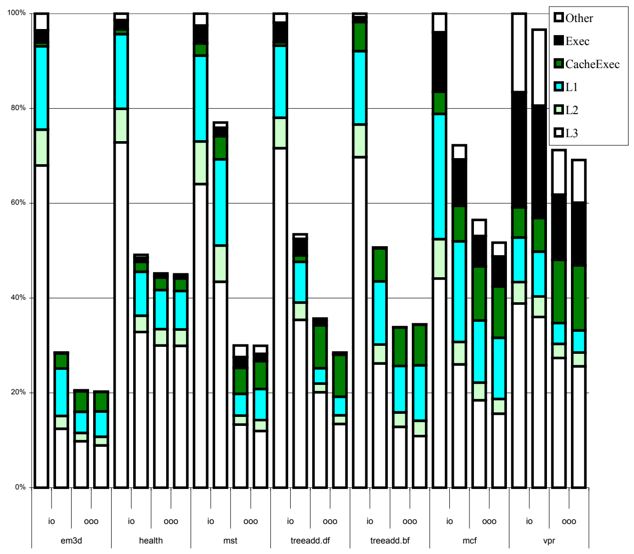


Figure 10. Cycle breakdown of in-order, in-order+SP, OOO, and OOO+SP, normalized to in-order model. It shows the impact of SP on cache hierarchy on both in-order and OOO.

While SSP reduces the *L3* cycles effectively, it sometimes increases the *L1* cycles. As a result, SSP achieves only 5% speedups on the OOO processor. Because we assume SSP without special hardware support, speculative threads can only be spawned at the retirement stage of the pipeline. And thread spawning is assessed with similar penalty to exception handling that incurs pipeline flushes. This makes the SSP scheme less effective in producing timely prefetches for L1 cache misses. Our results show that there is potential in further improving the tool for an OOO processor. The opportunity lies in judiciously applying SSP to even more selective loads, targeting long-range prefetching for reducing L3 cache misses without interfering with L1 misses that are covered by OOO. Future dynamic optimizers can monitor the coverage and timeliness data associated with a prefetching thread and if the thread does not help reduce latency, future `chk.c` instructions for that thread will return no available context. Alternatively, on OOO we need to create a slice that achieves even longer-range prefetching. For instance, precomputation via hand-adaptation in [31] achieves 3-times speedup on *health* on an OOO processor by creating a bigger interprocedural slice. This is due to the inlining of a few levels of recursive function calls by the programmer's hand adaptation to create large enough slack. The tool could not perform such aggressive optimization. We are investigating SSP techniques to complement, instead of interfering, the prefetching by OOO execution.

4.5 Automatic vs. Hand Adaptation

Wang et al. performed hand adaptation on three memory-intensive benchmarks for speculative precomputation [31]. In contrast, we use the automated binary adaptation tool to enhance the binary for SSP. We compare the performance of both approaches on the same simulator. The common programs from both works are *mcf* and *health*. On an in-order processor, hand-adaptation achieves a speedup of 73% on *mcf,* while the post-pass tool achieves 37% speedup. Our tool loses 20% of the overall performance of the manual version. On an OOO processor, both approaches achieve little improvements.

For the *health* benchmark*,* the enhanced binary from SSP achieves 103% speedup on the in-order processor, while hand adaptation achieves a speedup of 130%. We lose about 12% of the overall performance of the manual version. On the OOO processor, the hand-adaptation achieves 200% speedup, while our tool reports a speedup of only 120%. We lose 27% of the overall performance. As explained in Section 4.4, the loss is due to the fact that our tool could not perform the aggressive inlining of recursive function calls done by hand.

5. RELATED WORK

Longer memory latencies (relative to processing time) have motivated the research on building more complex pattern-based predictors of program behaviors in hardware. In comparison, several research groups have recognized recently that the program itself could be used as a predictor. This paper presents a software tool for such program-as-predictor prefetching. Luk proposed *software controlled pre-execution* that uses available hardware thread contexts to execute inserted code for prefetching [21]. The hand-inserted code provides prefetches for a non-speculative thread and yields an average speedup of 24% in seven irregular applications on an out-of-order SMT processor based on Alpha. The inserted code was not trimmed using the concept of slicing. Roth and Sohi proposed *data driven multithreading* that uses hardware contexts to prefetch for future memory accesses and predict future branches [25]. There was no automated compiler for identifying the triggers or extracting a minimal sequence of instructions to produce the address of a future memory access. Zilles and Sohi performed analysis of dynamic backward slices for execution-based prediction [33][34]. They target more delinquent events such as *problem branching*. Our work focuses on the automated tool that generates p-slices and triggers for load operations.

Collins et al. used the simulator to capture the dependence graph that forms p-slices for basic triggers [7]. This graph-capture code is equivalent to performing dynamic slicing, which is shown to be potentially prohibitively expensive [2]. P-slices using chaining triggers were constructed manually. The post-pass binary tool aims at automating it in the compiler. *Dependence graph precomputation* [3], *dynamic speculative precomputation* [6], and *slice processors* [23] use all-hardware approaches. The hardware complexity may increase if future processors try to target a very long-range prefetch. Furthermore, to identify a program subset with minimal size and maximal accuracy may require a sophisticated program analysis, and the complexity of attainable analysis in hardware is typically constrained. However, hardware has the advantage of being able to track program behaviors and dynamically adjust accordingly. In comparison to the all-hardware approaches, SSP uses the existing SMT hardware and relies on the compiler to perform binary adaptation. Aamodt et al. approached pre-execution as a generalized form of computation prediction [1]. They introduced and measured *slice-locality*, a necessary property for history-based methods such as those used for value and branch outcome patterns to be extended for dynamically predicting repeating patterns of computations. They provided the insight that program execution exhibits *slice-locality* and that by recording the few most recently seen unique slice traces per problem load or branch, the majority of problem branches and load instances are covered. Our SSP tool exploits *slice-locality* by statically capturing the dominant slices, instead of building hardware to track and adjust slice-traces dynamically.

6. CONCLUSIONS AND FUTURE WORK

It is difficult to parallelize pointer-intensive applications for multithreading architectures. However, we demonstrate an SMT approach that leverages otherwise-idle threads to perform precomputation and prefetches for the main thread. By exploiting the increasing memory bandwidth available on modern processors, we can reduce the memory latency for the main thread. This new form of multithreading, unlike traditional parallelization which focuses on partitioning the data or computation, minimizes the changes to the existing binaries and to the execution of the main thread. The main thread does not integrate computation results from the speculative threads. This makes it possible to speed up existing optimized binaries via multithreading.

Unlike some previous work that either assumes complex hardware mechanisms or relies on manual adaptation, this paper presents an effective post-pass compilation tool for the software-based speculative precomputation. The tool achieves an average of 87% speedup on an in-order processor for seven applications. Motivated by our results on the research Itanium model and by our assumption of no expensive hardware support, we recently performed SSP on the out-of-order Pentium 4 processor with the Hyper-threading technology [22] and achieved 7% to 45% speedups on real silicon [30]. In the future, we plan to extend this tool for other delinquent events in the program and for using SSP more judiciously on an OOO processor. Furthermore, we plan to use this automated tool to enable broader and more productive application of SSP to programs such as database applications.

7. ACKNOWLEDGEMENTS

We thank Tor Aamodt, Murali Annavaram, Jesse Fang, Monica Lam, Yong-fong Lee, Ken Lueh, Justin Rattner, and Xinmin Tian for their valuable comments on this paper. We appreciate the helpful suggestions from the referees for this conference.

8. REFERENCES

[1] T. Aamodt, A. Moshovos, and P. Chow. The Predictability of Computations that Produce Unpredictable Outcomes. In *5th Workshop on Multithreaded Execution, Architecture and Compilation*, pp. 23-34, Austin, Texas, December 2001.

[2] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 246-256, June 1990.

[3] M. Annavaram, J. Patel, E. Davidson. Data Prefetching by Dependence Graph Precomputation. In *28th International Symposium on Computer Architecture*, Goteborg, Sweden, July 2001.

[4] J. Bharadwaj, W. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, J, Pierce. The Intel IA-64 Compiler Code Generator. In *IEEE Micro*, Sept-Oct 2000, pp. 44-53.

[5] M. Carlisle. Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines, *Ph.D. Thesis*, Princeton University Department of Computer Science, June 1996.

[6] J. Collins, D. Tullsen, H. Wang, J. Shen, Dynamic Speculative Precomputation. In *Micro conference*, December 2001.

[7] J. Collins, H. Wang, D. Tullsen, C, Hughes, Y. Lee, D. Lavery, J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *28th International Symposium on Computer Architecture*, Goteborg, Sweden, July 2001.

[8] K. Cooper, P. Schielke, D. Subramanian. An Experimental Evaluation of List Scheduling. Rice University Technical Report 98-326, September 1998.

[9] R. Cytron. Compiler-time Scheduling and Optimization for Asynchronous Machines. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1984.

[10] J. Emer. Simultaneous Multithreading: Multiplying Alpha's Performance. In *Microprocessor Forum*, October 1999.

[11] R. Ghiya, D. Lavery, and D. Sehr. On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 47-58, June 2001.

[12] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum-Flow Problem. In *Journal of the Association for Computing Machinery*, 35(4):921-940, October 1988.

[13] R. Gupta and M. L. Soffa. Hybrid Slicing: an Approach for Refining Static Slicing Using Dynamic Information. In *The Foundations of Software Engineering*, pp. 29-40, September 1995.

[14] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. In *IEEE Computer*, July 2000.

[15] G. Hinton and J. Shen. Intel's Multi-Threading Technology. In *Microprocessor Forum*, October 2001.

[16] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, R. Zahir, Introducing the IA-64 Architecture. In *IEEE Micro*, Sept-Oct 2000.

[17] R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, C. Lim, J. Ng, D. Sehr. An Advanced Optimizer for the IA-64 Architecture, In *IEEE Micro*, Nov-Dec 2000.

[18] W. Landi and B. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 235-248, June 1992.

[19] S. Liao. SUIF Explorer. Ph.D. thesis, Stanford University, August 2000, Stanford Technical Report CSL-TR-00-807.

[20] S. Liao, A. Diwan, R. Bosch, A. Ghuloum, M. S. Lam. SUIF Explorer: an Interactive and Interprocedural Parallelizer. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 37-48, Atlanta, Georgia, May 1999.

[21] C. K. Luk, Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors, In *28th International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001.

[22] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. In *Intel Technology Journal,* Volume 6, Issue on Hyper-threading, February 2002.

[23] A. Moshovos, D. Pnevmatikatos, A. Baniasadi. Slice Procesors: an Implementation of Operation-Based Prediction. In *International Conference on Supercomputing*, June 2001.

[24] E. Reingold, J. Nievergelt, N. Deo. Combinatorial Algorithms: Theory and Practice. Prentice-Hall Publishers, 1977.

[25] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *7th IEEE High-Performance Computer Architecture*, January 2001.

[26] H. Sharangpani and K. Aurora, Itanium Processor Microarchitecture. In *IEEE Micro*, Sept-Oct 2000.

[27] D. M. Tullsen. Simulation and Modeling of a Simultaneous Multithreaded Processor. In *22nd Annual Computer Measurement Group Conference*, December 1996.

[28] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *22nd International Symposium on Computer Architecture*, June 1995.

[29] R. Uhlig, R. Rishtein, O. Gershon, I. Hirsh, and H. Wang. SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture. In *Intel Technology Journal*, Q4 1999.

[30] H. Wang, P. Wang, R. D. Weldon, S. Ettinger, H. Saito, M. Girkar, S. Liao, J. Shen. Speculative Precomputation: Exploring Use of Multithreading Technology for Latency. In *Intel Technology Journal,* Volume 6, Issue on Hyper-threading, February 2002.

[31] P. Wang, H. Wang, J. Collins, E. Grochowski, R. Kling, J. Shen. Memory Latency-tolerance Approaches for Itanium Processors: out-of-order execution vs. speculative precomputation. In *Proceedings of the 8th IEEE High-Performance Computer Architecture,* Cambridge, Massachusetts, February 2002.

[32] M. Weiser. Program Slicing. In *IEEE Transactions on Software Engineering*, 10(4), pp. 352-357, 1984.

[33] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *27th International Symposium on Computer Architecture*, Vancouver, BC, Canada, May 2000.

[34] C. Zilles and G. Sohi. Execution-Based Prediction Using Speculative Slices. In *28th International Symposium on Computer Architecture*, Goteborg, Sweden, July 2001.