1-1-2003

An Algorithm for Register-Synchronized Precomputation In Intelligent Memory Systems

Wessam Hassanein

José Fortes
University of Florida

Rudolf Eigenmann

An Algorithm for Register-Synchronized Precomputation In Intelligent Memory Systems[◊]

Wessam Hassanein[§]

José Fortes*

Rudolf Eigenmann[§]

TR-ECE 03-17

[§]School of Electrical & Computer Engineering
465 Northwestern Ave.
Purdue University
West Lafayette, IN 47907-2035
{hassanin,eigenman}@ecn.purdue.edu

*Department of Electrical & Computer Engineering
University of Florida
Gainesville, FL 32611-6200
fortes@ufl.edu

[◊] This material is based upon work supported by the National Science Foundation under Grant No. 0296005. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract

This paper presents a novel compiler algorithm for selecting program slices that prefetch load values concurrently with program execution. The algorithm is evaluated in the context of an intelligent memory system. The architecture consists of a main processor and a simple memory processor. The intelligent memory system pre-executes program slices and forwards values of critical loads to the main processor ahead of their use. The compiler algorithm selects program slices for memory processor execution, and inserts synchronization instructions that synchronize main and memory processors. Experimental results of the generated code on a cycle-accurate simulator show a speedup of up to 1.33 (1.13 on average) over an aggressively latency-optimized system running fully optimized code.

1. Introduction

Memory access latency is becoming a critical performance limiting factor, caused by the slower rate of memory improvements compared to processor speed and technology improvements. As memory access latency dominates instruction execution latency, processor pipelines stall, waiting for load instructions to fetch data from memory. Prefetching is one of the important techniques to hide memory access latency. Data prefetching can be prediction-based or precomputation-based. Prediction-based prefetching [16-18] is effective for applications with regular memory accesses. However, irregular applications are especially vulnerable to memory latency, as their load addresses cannot be predicted. Recent research has proposed several precomputation-based prefetching techniques [1,3,8,10,14,15,19], where the load address is precomputed through the pre-execution of a set of address-generating instructions. Examples include Collins et al.'s Speculative Precomputation [3], Luk's Software Controlled Pre-Execution [10], Roth and Sohi's Speculative Data Driven Multithreading [14], Zilles and Sohi's Speculative Slices [19], and Liao et al.'s Software-based Speculative Precomputation [8]. Through pre-execution, these techniques are able to adapt to the irregularity of load addresses. Precomputation uses a separate processing element or an unused thread in a multi-threaded processor. The above mentioned prefetching techniques are either performed by hardware support, by-hand, or by the compiler, using profile information.

Another venue of research has concentrated on developing intelligent memory systems [4-7,9,11-13,17] to solve the memory latency problem. These systems take advantage of the memory processor's low data access latency to execute memory-intensive calculations. Using intelligent memory systems, the work in [17] proposes memory-side prediction-based forwarding, where the memory forwards data to the processor, ahead of its use. Forwarding mechanisms push data from memory to the processor, which contrasts with the pull mechanism of prefetching techniques.

The main contribution of this paper is that it proposes a fully automated compiler algorithm for precomputation-based forwarding. In contrast to other compiler solutions, it does not need any profile information. We present the proposed algorithm in the context of in-memory pre-execution [6], however it is also directly applicable to processor-side pre-execution. The proposed algorithm selects program slices that execute in memory, inserts trigger instructions that initiate pre-execution, and synchronizes register values between the memory and main

processors. The algorithm selects program slices for memory execution, targeting a set of critical loads. When executed, these program slices will generate the load value ahead of its use by the main processor. The algorithm inserts register synchronization instructions that initiate precomputation. We refer to the proposed algorithm as Register-Synchronized Precomputation (RSP).

We implement the RSP algorithm and evaluate its performance using SPEC CPU2000 and Olden benchmarks. We use a cycle-accurate aggressive out-of-order processor simulator with accurate bus and memory contention. The experimental results show a speedup of up to 1.33 (1.13 on average) over an aggressively latency-optimized system running fully optimized code.

The rest of the paper is organized as follows: Section 2 describes the RSP execution model and compilation tool. Section 3 presents the RSP compiler algorithm. Section 4 describes dynamic slice scheduling and adaptation. Sections 5 and 6 discuss the experimental methodology and the results, respectively, and Section 7 discusses related work. Finally, we present in Section 8 the concluding remarks.

2. Register-Synchronized Precomputation

2.1. RSP Execution Model

In this section, we describe the RSP execution model and its components, shown in Figure 1. The RSP execution model consists of a main thread and a precomputation thread. The main thread runs the full program code. The precomputation thread runs only the marked program slices. The objective of the precomputation thread is to speedup the main thread by supplying values of critical load instructions ahead of their use. Figure 1 shows a single program slice that consists of three sections A, B and C. The slice is a sequence of instructions that, when executed, generate the addresses and values of the critical loads. The trigger from the main thread initiates pre-execution in the precomputation thread. While executing the program slice, the precomputation thread could forward several load values (data values 1,…,n in Figure1) to the main thread, where they wait until they are consumed by the corresponding critical loads. The precomputation thread accesses the same program as the main thread. However, it skips unmarked instructions.

Although RSP was developed to target an intelligent memory system the execution model is general and applicable to any architecture where an extra thread (in memory for forwarding or in the main processor for prefetching) is used for precomputation. The architecture's interpretation of the instructions added by the compiler will differ depending on the specific implementation, but the compiler view is the same. The compiler generates a single program that is loaded in memory. We describe the details of both threads in the following subsections.

Main Processor (Main Thread) Execution: The main thread executes all program instructions. Upon encountering a critical load, the main thread will check if a valid value has been received for this load and, if so, it will use it. Otherwise, the thread will issue a load instruction memory read as in conventional execution. The execution model does not require the precomputation to satisfy correctness constraints. Instead, correctness is guaranteed by the architecture, as in most speculative precomputation techniques (e.g. [3,8]).

Upon encountering a trigger (sync) instruction the main thread issues a start of execution to the precomputation thread. It sends to memory the program slice address and the initial value of registers at the beginning of the slice. The main thread dynamically decides whether to issue (execute) a trigger instruction or not, based on the history of execution. History-based prediction

is used to dynamically decide the likelihood of the need of the precomputation and thus the data forward/prefetch.

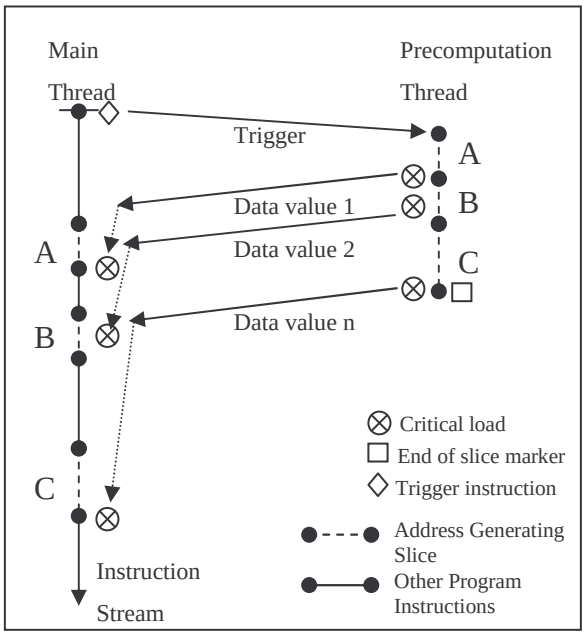


Figure 1: RSP Execution Model. Both main and precomputation threads access the same program. However, the main thread executes all instructions, while the precomputation thread executes only marked instructions. A single program slice execution is shown.

Memory Processor (Precomputation Thread) Execution: The memory processor executes annotated program instructions sequentially, while skipping unmarked instructions. The main thread initiates the precomputation thread execution by executing a sync instruction. Execution continues until a special instruction, marking the end of the program slice. No store instructions are executed by the precomputation thread. The precomputation thread does not alter the architecture state of the main thread.

2.2. RSP Compilation Tool

The compiler is responsible for identifying precomputation program slices, trigger points and initial slice values. No run time or hardware cost is incurred achieving these tasks. A compilation tool (Figure 2) is used to generate the binary code for RSP. The RSP pass is a separate pass, following optimization and assembly code generation.

The RSP pass starts by selecting critical load instructions. The algorithm then identifies the program slices that generate the addresses of these loads. It identifies the trigger points for each corresponding program slice and the initial register values of the address precomputation.

Finally, it generates a new assembly code, containing trigger instructions, initialization and marked program slices. The details of the RSP algorithm are presented next.

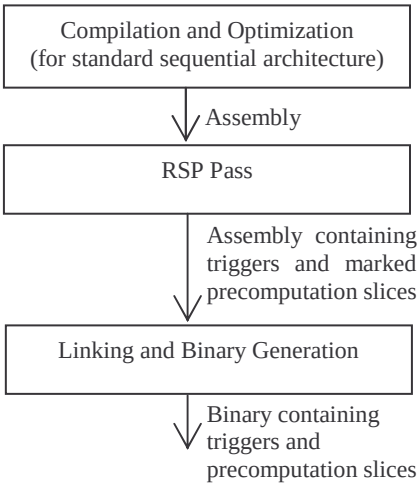


Figure 2: Structure of the RSP Compiler.

3. RSP Algorithm

3.1. Critical Load Selection

The first step of RSP is to select and mark the set of load instructions whose execution is likely to cause the main processor to stall. RSP targets these loads, which we call “critical loads”. Loads that miss in the cache incur large memory access latencies and therefore are most likely to stall the processor.

RSP classifies all load instructions in the program into three categories; 1- loads whose address is a register (*register*); 2- loads whose address is based on the global pointer register (*global pointer*); 3- loads whose address is based on the stack pointer and frame pointer registers (*stack and frame pointers*). Figure 3 shows the percentage of each load type in a set of fully optimized SPEC CPU2000 and Olden benchmarks. On average ~47% of the loads are *register* loads, 29% are *global pointer* and 24% are *stack and frame pointers*. Figure 4 shows that, on average, the majority of cache misses (~99%) are due to *register* loads. This is because *register* loads are dependent on other instructions and therefore, could be part of irregular address generating chains (eg. in pointer chasing). RSP therefore considers these loads only. In doing so, it deals with merely 47% of the total loads and still achieves most of the potential performance gain. In contrast with recently proposed prefetching techniques [8,14], RSP selects the critical loads without needing a profile run.

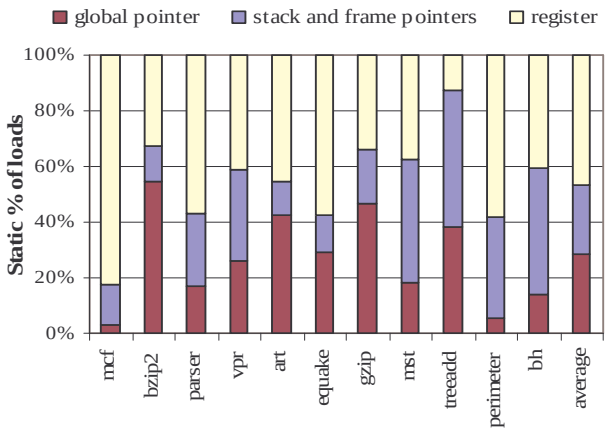


Figure 3: Static percentage of the different load types in each benchmark.

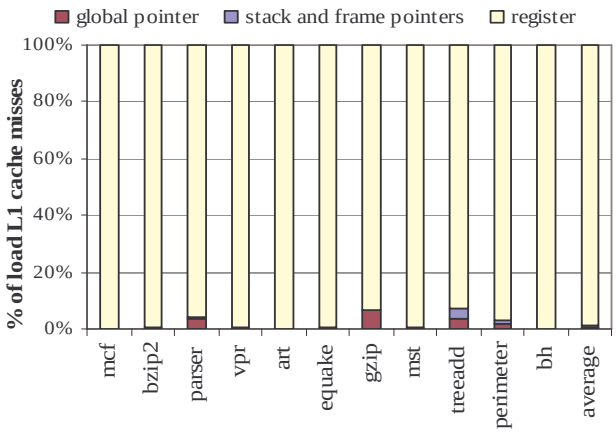


Figure 4: Percentage of L1 cache misses due to different load types.

3.2. RSP Program Slicing

The second step of RSP is to select and mark program slices for execution in the memory processor. These program slices generate the address of the load value that will then be forwarded (or prefetched). The timing of the forward operation is critical. The forwarded value needs to arrive at the main thread before the target load instruction reaches the pipeline issue stage to take full benefit of the forwarding process. To achieve this, equation *(1)* below should be satisfied. Partial benefit is achieved if the forwarded value arrives after the targeted load was issued but before the load's value arrives from memory. To achieve this, equation *(2)* needs to be satisfied.

$$Load\_issue\_time - Trigger\_time \geq Slice\_precomp\_interval + PreData\_forward\_time + Trigger\_init\_time \quad (1)$$

$$Data\_arrival\_time - Trigger\_time \geq Slice\_precomp\_interval + PreData\_forward\_time + Trigger\_init\_time \quad (2)$$

where;

Load_issue_time: Instant at which the targeted load is issued.

Trigger_time: Instant at which the precomputation is triggered to start pre-execution.

Slice_precomp_interval: Amount of time needed for slice precomputation (generation of targeted load's address).

PreData_forward_time: Amount of time needed to forward (prefetch) the data value from memory. In prefetch case, this variable includes the prefetch transfer time to memory.

Trigger_init_time: In forwarding, the amount of time needed for trigger to reach precomputation thread in memory. In prefetching, the amount of time to initialize (copy live-in values) the precomputation thread.

Data_arrival_time: Instant at which the load instruction's data arrives from memory.

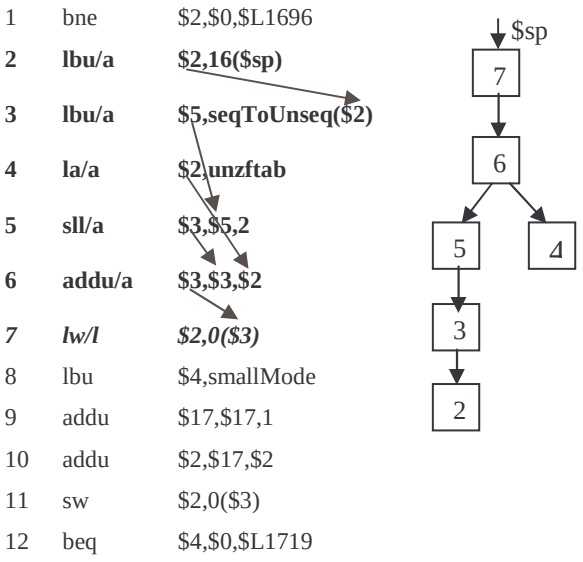
The performance objective of the RSP algorithm is to minimize the *Slice_precomp_interval* (and therefore the precomputation slice size) and the *Trigger_time*.

To select the precomputation slice, data dependence analysis (register dependence analysis; no memory disambiguation is needed by RSP) is done, starting from the critical load. The analysis follows all dependence chains backward and selects all instructions that contribute to the address generation of the load instruction. This is shown in the RSP algorithm Figure 5, lines 3-6. Figure 6(a) shows an example. Starting from the critical load at line 6, the source of the data dependence indicates the instruction generating the address of the load. Since the address of the load is based upon register $3, which is produced by instruction 6, number 6 is next in the dependence tree in Figure 6(b). This backward analysis is repeated, selecting instructions 5, 4, 3 and 2. To minimize the size of the slice, the algorithm stops the backward analysis at the beginning of the load's basic block. To minimize the number of slices, the algorithm combines several loads into one slice. The following two sub-steps optimize the program slice further.

```
1 Main
2     Build CFG of full program, each node representing a basic block
3     For each node N do this:
4         For each critical load L do this:
5             Build data dependence Tree T for L
6             Mark all instructions in T for memory execution
7             For each register R in T generated outside N do this:
8                 Mark sync instruction addition point of register R and node N
9     Generate new code containing sync instructions
10 End

11 Mark sync instruction addition point of register R and node N
12    For each parent node P of N do this:
13        If node P is not marked as visited do this:
14            Mark P as visited
15            If R is defined in P do this:
16                Mark the position right after the definition of R for sync insertion
17            Else
18                Mark sync instruction addition point of register R and node P
19 End
```

Figure 5: RSP algorithm for (1) identifying program slices for precomputation (2) inserting triggering and value initialization (register synchronization) instructions for the precomputation thread.



(a) Code (b) Data Dependence Tree

Figure 6: RSP program slicing using backward data dependence analysis, starting from the critical load. Instruction marked "/l" is the critical load, instructions marked "/a" are the address generation instructions selected for precomputation. The program slice consists of the "/a" and "/l" instructions. This example is generated from the SPEC CPU2000 bzip2 benchmark.

First, to increase the size of the basic block, the first pass of the compilation tool uses loop unrolling and subroutine inlining. In addition, loop unrolling allows the algorithm to consider loads from several loop iterations in the slice. As a result, slices can forward load values several iterations in advance.

Second, selected program slices are combined to create larger slices, using a full program Control Flow Graph (CFG). After slices are selected within the basic block, all register dependencies to instructions outside the basic block are marked. Let the register dependence marked for slice S_1 be register R. The marked dependence R is tracked outside of the basic block through the control flow paths leading to that slice S_1. If R is also the register dependence belonging to another slice selected for precomputation, S_2, and slice S_2 is sequentially located before S_1 in the code, the two slices are combined. Figure 7(a) shows slice combining. If R is not a register dependence in S_2 then the two slices are not combined.

Two slices S_1 and S_2 are also combined if S_1 follows S_2 sequentially in the code and S_1 has no marked register dependencies (no register dependences outside of slice). Figure 7(b)

illustrates this case. Both cases in Figure 7 require that the two slices being combined are in sequence in the program code. Therefore, the sequential precomputation can directly execute the slice without the need to add a special jump instruction to direct precomputation from one slice to the other.

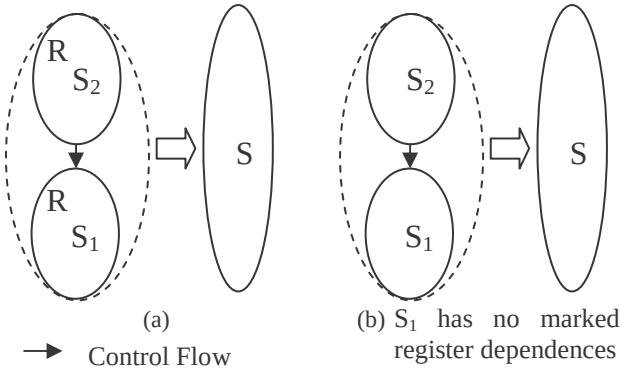


Figure 7: Combining slices S_1 and S_2 into a single slice S. S_1 and S_2 are sequentially located in the code. (a) Marked register dependence R from S_1 is also in S_2. If not, slices are not combined. (b) S_1 has no marked register dependences.

Any marked register dependencies not satisfied by the slice combination process are marked as initial values of their slice. Section 3.3 describes how RSP initializes these slices. All instructions in the slices are marked for precomputation. The instruction at the end of each slice is marked as the final instruction.

3.3. RSP Trigger Insertion and Slice Initialization

The third step of RSP is to select the insertion point of the trigger instruction. The objective of this step is to maximize the dynamic distance between the trigger instruction and the targeted critical load instruction as explained in the previous section.

RSP combines trigger insertion and slice initialization analysis. This is because the trigger instruction (register synchronization instruction or *sync*) also supplies the slice's initial value to the memory processor. This instruction, executed by the main thread to start a precomputation, will synchronize the initial register by sending its value to the precomputation thread.

To select the sync insertion point, a full program Control Flow Graph (CFG) is used. RSP builds a CFG across procedure calls and files, through interprocedural analysis, taking advantage of the compiler knowledge of the full program. Figure 5, lines 1-10 illustrates the steps of the

RSP algorithm including sync insertion. Each node in the CFG represents a basic block. Nodes that contain critical loads are marked with the registers that are needed in the critical load address generation and whose values are generated outside of the node. These registers are obtained by the analysis described in Section 3.2. For each marked register, a backward analysis is performed, starting from its node through all control flow paths that lead to the node (Figure 5, lines 11-19). The algorithm checks for the last definition that generated the value of the marked register and marks the corresponding instruction. Visited nodes are marked to avoid repeated analysis. The analysis in a path stops either when reaching a last definition for the register or a node that has already been visited. The analysis is repeated for all marked nodes.

At the sync insertion point, the algorithm adds a sync instruction that contains the marked register number whose value will be used for initialization and an offset to the corresponding program slice. The precomputation thread combines sync instructions targeting the same program slice with different register numbers dynamically by initializing both registers.

3.4. Illustrative Example

Figure 8 illustrates an example of the interprocedural analysis used in the RSP algorithm. First the algorithm builds a CFG as illustrated in Figure 8(b). From the load selection criteria in Section 3.1, the load marked “/l” in Figure 8(a) is a register load and therefore a critical load for our algorithm. Applying the data dependence analysis described in Section 3.2 on the “/l” load instruction, five instructions marked “/a” are recognized to be the address generating instructions of this load within its basic block (labeled “$L46”). The program slice for this load consists of the load itself and the instructions marked “/a” in Figure 8(a).

The registers whose values are generated outside of the basic block for the selected load are registers $4 and $8. A backward traversal of all the paths leading to the node containing the candidate load, as described in Section 3.3, yields two instructions generating values for register $8, and one for register $4. The sync instructions are added directly after the selected instructions, as illustrated in Figure 8(b). Both the register number and the offset between the synchronization instruction and the start of the program slice (represented as “MYLB3” in Figure 8) are added to the sync instruction.

3.5. RSP Code Generation

In the RSP code generation stage, the algorithm marks all precomputation slices by annotating their instructions (setting one bit in the instruction that is unused by the ISA). Since

slices do not overlap, there is no need to differentiate between slices. Therefore, only one bit is needed to mark instructions for precomputation. Similarly, the end of each precomputation slice is annotated using another bit. The sync instruction is similar in format to a jump instruction and contains an address offset and a register number.

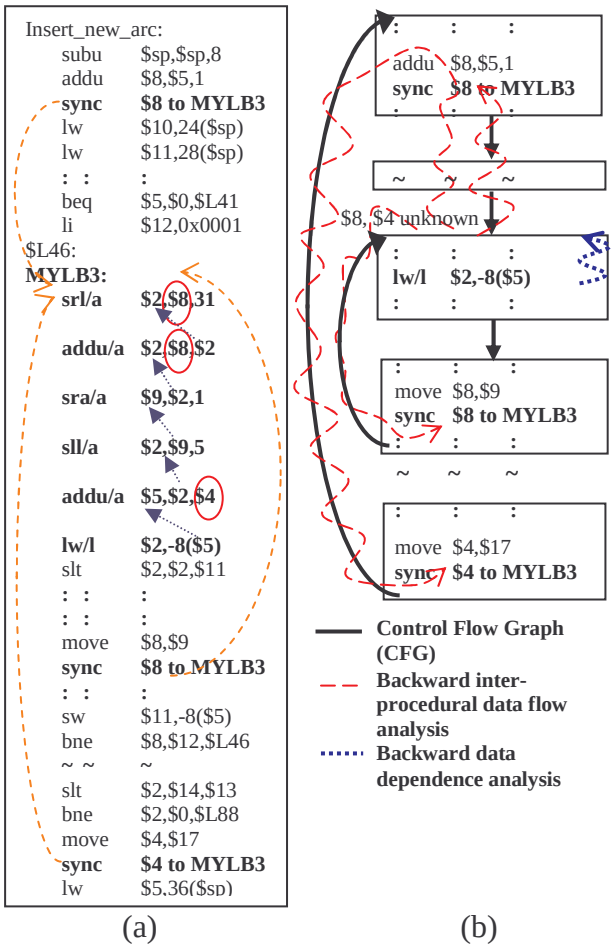


Figure 8: (a) Code including generated trigger instructions (sync). (b) CFG and backward interprocedural analysis to locate where the sync instruction should be added. Backward data dependence analysis (represented as small upward dotted arrows) within the critical load's basic block generates the program slice. Starting from the basic block of the critical load, backward global inter-procedural data flow analysis (represented as large dashed arrows) is done using last definition analysis of each basic block visited to locate where the sync should be added. (This example is generated from the SPEC CPU2000 mcf benchmark).

4. Dynamic Slice Scheduling and Adaptation

This section presents important aspects of the execution model in our architecture.

4.1. RSP Dynamic Slice Scheduling

Slice scheduling determines when a slice will be assigned to the precomputation thread. Scheduling is needed because multiple triggers could occur in the program code, during main thread execution, while the precomputation thread is busy pre-executing a program slice. As each trigger is executed by the main thread, it generates a request for the pre-execution of the corresponding program slice. This request is queued if the precomputation thread is busy. If not, the request is serviced by the precomputation thread by executing the requested program slice. After the precomputation-thread pre-executes the program slice, the queued precomputation requests are checked and one of the requests is chosen to be the next request assigned to the precomputation thread. We refer to the process of choosing a request from the queued requests for precomputation as “slice scheduling”.

Slice scheduling is important because the order in which slices are pre-executed could determine the effectiveness of each slice’s pre-execution. Effective slices should be pre-executed early with respect to the main thread’s execution of the same slice. Therefore, the critical load values generated by the precomputation thread should arrive at the main thread (after being fetched from memory) before the main thread reaches the load instruction. A straight-forward scheduling mechanism is FIFO (First In First Out). This mechanism schedules requests for precomputation in their arrival order. However, the order in which requests are received is not always the best order in which they should be scheduled (e.g. a later request could have a closer deadline than an earlier request). Therefore, RSP uses an Earliest Deadline First (EDF) slice scheduling mechanism. EDF prioritizes requests based on the deadline by which they should be executed. If a schedulable solution for all requests exists, EDF will find it, while FIFO is not guaranteed to do so. The deadline of each request is generated by the main thread from the history of the previous execution of the request’s program slice as shown by equation (3).

Trigger_deadline_current = Slice_timestamp_previous - Trigger_timestamp_previous *(3)*

A new request replaces an older queued request if both target the same slice and specify the same initialization register. The requests are combined if they specify the same slice with

different initialization registers. Requests are dropped (not queued if new, or deleted if old) if their deadline is below a threshold value as there might not be enough time to pre-execute that request.

4.2. RSP Dynamic Adaptation

Dynamic adaptation uses run time information to decide which slices should be precomputed. This is important to filter out slices on paths that are not executed or are rarely executed. The decision whether to pre-execute a slice or not occurs at the trigger (sync) point. The RSP uses a “sync history table” to adaptively select the trigger instructions to execute based on the sync’s previous history. To reduce the complexity and size of the history table, RSP only considers the most recent history of the sync instructions. If the last instance this sync was executed the corresponding slice was executed by the main thread then the sync will be executed. Otherwise, the sync will not be executed.

5. Experimental Methodology

We have implemented the described RSP compiler algorithm in C++. The SPEC CPU2000 and Olden benchmarks shown in Table 2 are used. Benchmarks are initially run through the SimpleScalar simulator [2] gcc 2.6.3 with –O3 –funroll-loops compiler optimizations and assembly output is produced. The output is then used as input to the proposed RSP algorithm as shown in Figure 2. The algorithm performs the described assembly-level analysis in Section 3 and produces new assembly code containing the new generated synchronization instructions. This code is then compiled through SimpleScalar gcc to produce the final binary output. We compare against a base code which is a fully-optimized including latency hiding techniques such as loop-unrolling and scheduling (-O3 –funroll-loops compiler optimizations).

To study the dynamic performance of RSP, we developed a cycle-accurate simulator based on the SimpleScalar 3.0a simulator [2]. We added major enhancements to the simulator to implement accurate bus and memory contention and memory-side precomputation and forwarding. The system consists of an aggressive out-of-order superscalar main processor executing the main thread and a memory-processor executing the precomputation thread. Forwarded values are saved in a buffer that takes 1 cycle to access. The parameters chosen in the simulations are shown in Table 1.

To evaluate the performance of the RSP algorithm we used four Olden benchmarks and five integer, two floating point SPEC CPU2000. We concentrate on pointer-intensive C benchmarks that can run on our compiler and simulator. All SPEC benchmarks are fast-forwarded for 2 Billion instructions and then run for 1 Billion committed instructions. The art benchmark is not fast-forwarded since its total execution is less than 2 Billion instructions. The train data set is used in all SPEC benchmarks except bzip2 where the reference data set is used because it gives a larger number of L2 misses, and art where we use the test input due to its large simulation time. Olden benchmarks, mst (1024 nodes), treeadd (20 levels) and perimeter (12 levels) are small and therefore run to completion, while bh (8192 bodies) is run for 1 Billion committed instructions. The additional instructions generated by our compiler algorithm are not included in the count.

Table 1: Simulated microarchitecture parameters.

Numbers do not include bus or memory contention affects that are simulated.

Module	Parameter	Value
Main Processor (Main Thread)	Frequency	1GHz
	Issue width	Out-of-order, 4 issue
	Functional Units	4Int+4FP+4Ld/St.
	Branch Prediction	2level
	Round-Trip memory latency	73 cycles (row miss), 61 cycles (row hit)
	I/DTLB miss latency	60 cycles
L1 Instruction/Data caches	Size	Split 16KB/16KB
	Latency	1 cycle
	Associativity	2-way set associative
	Line size	32 Byte
	Write Strategy	Writeback
	MSHRs	16
L2 cache	Size	Unified 512KB
	Latency	16 cycles
	Associativity	4-way set associative
	Line size	32 Byte
	Write Strategy	Writeback
Memory Processor (Precomputation Thread)	Frequency	1GHz & 500MHz
	Issue Width	Out-of-order & Inorder, 4 issue
	Functional Units	4Int+no FP+4Ld/St.
	Branch Prediction	no branch prediction
	Round-Trip memory latency	23 cycles (row miss), 11 cycles (row hit)
Memory Data cache	Size	32KB
	Latency	1 cycle
	Associativity	2-way set associative
	Line size	64 Byte
System Bus	Speed	500MHz
	Width	64bits
Memory Controller	Latency	30ns

6. Results

In this section we present the experimental results of the proposed RSP compiler algorithm on the architecture simulation described in Section 5 and Table 1. The base system ("original") uses the same main-processor as RSP but with a regular memory system that does not incorporate any memory processing. This original system runs fully optimized, unchanged code. All performance results in this paper are normalized with respect to the original system.

6.1. Performance Analysis

Execution Time: Figure 9 illustrates the performance results of the proposed RSP algorithm. The results show that RSP gives a speedup of up to 1.33 (1.13 on average) over the *original*. Figure 9 also shows the performance overhead due to the addition of instructions to the code (*instruction overhead*). This overhead is measured by running the new code, including inserted sync instructions, on the base system. The overhead is very small (much less than 1% in most cases, maximum 2% in bzip2) and therefore has little effect on the system. The reduction in the normalized execution time of the *instruction overhead* in equake is due to a change in the instruction L1 (IL1) cache access pattern. In this case, by adding instructions to the code, fewer IL1 misses and replacements were observed. This caused a dip in execution time as shown in Figure 9, as well as a change in the L2 and data L1 cache access patterns.

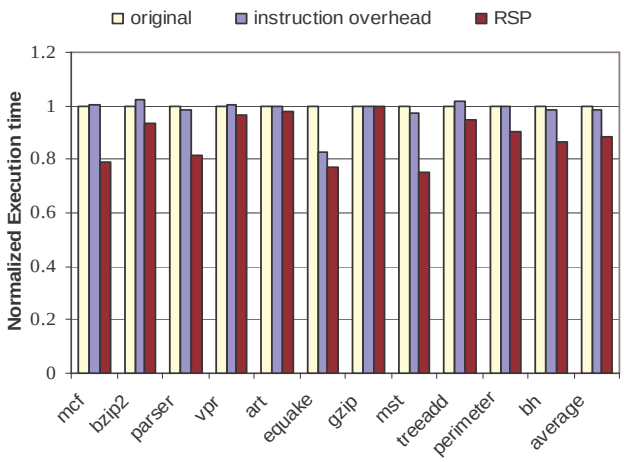


Figure 9: Normalized execution time

Load Access Latency: Figure 10 compares the normalized average load access latencies. This latency is measured from the load issue time to the load writeback time. RSP gives a reduction of average load access latency of up to 38% (18% on average). The increase in average load access latency in equake is a result of the change in cache behavior of the new code (1.06 normalized average load latency), which is then reduced to the shown value by RSP.

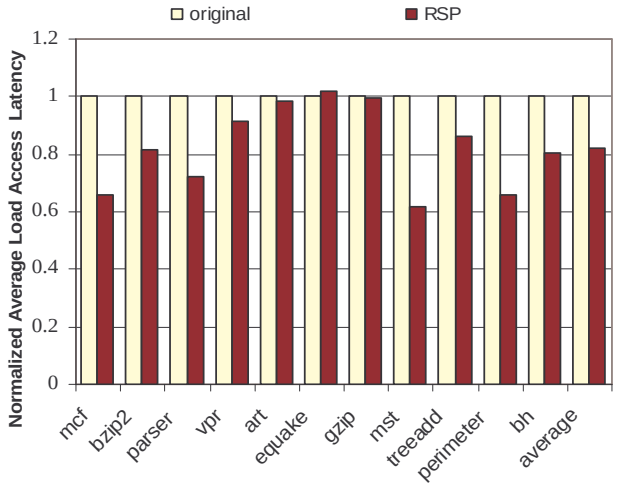


Figure 10: Normalized average load access latency

Effect of Memory-Processor Speed/Complexity on Performance: Experiments presented so far have been using a 1GHz out-of-order memory processor. This section investigates the effect of halving the memory-processor speed to 500MHz and using a simpler inorder memory processor. As shown in Figure 11, using an inorder and lower-speed memory-processor has only a small effect on RSP performance. This is because the precomputation occurs early enough to allow the system to mask the extra latency.

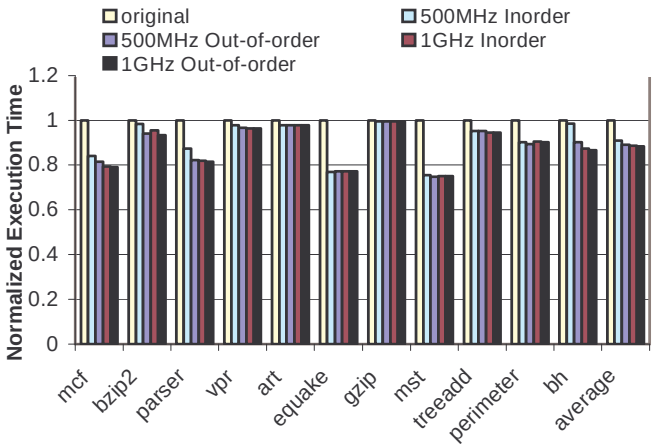


Figure 11: 500MHz vs. 1GHz (In-order and Out-of-order) memory-processor RSP performance.

6.2. Synchronization and Slice Analysis

Sync to Load Distance: The dynamic distance (number of 1GHz cycles) between the register synchronization instructions (trigger) and the corresponding load instructions is characterized as shown in Figure 12. The categories are as follows: less than 10 cycles (<10), 10 to less than 20 cycles (10-20), and so on, and finally 150 cycles and above. The importance of this measure is that it illustrates the distribution of the time between the register-synchronization value availability and the load-value use. The register synchronization value is available when the instruction producing this register value is executed in the main processor and the register value is written. At this time, the main processor can send the sync request to the memory processor (trigger slice precomputation). The load-value is used at the issue time of the load instruction in the main processor. As shown in Figure 12, on average over 40% of all sync instructions have a latency of 30 cycles or more.

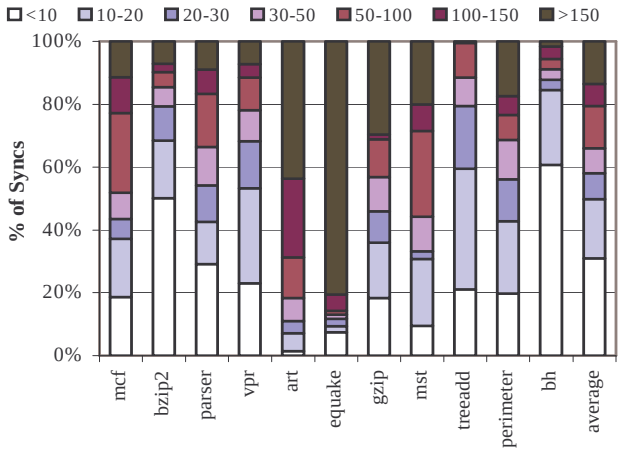


Figure 12: Dynamic percentages of synchronizations with different distances (number of cycles) from sync to load instruction.

Slice Characteristics: Table 2 shows the dynamic number of unique sync instructions executed. On average, RSP executes 334 unique sync instructions. The numbers also represent the maximum size of the sync history table. The average static number of slices is 282. The average selected program-slice size is 4.18 instructions per slice. The majority of the instructions in the slice are loads and on average 2.84 critical loads are in each slice. Therefore, on average, slices generate multiple load values. “Intermediate critical loads” are critical loads that are part of an address generation slice of another critical load. On average 49% of intermediate loads are critical loads.

Table 2: Number of unique sync instructions and slice sizes

(x indicates no intermediate loads)

Benchmark	Number of Dynamic Syncs	Number of Static slices	Average slice size	Number of critical loads per slice	Number of total loads per slice	Percentage of intermediate loads that are critical
mcf	119	147	4.52	4.19	4.22	96%
bzip2	400	1072	5.13	3.25	3.32	82%
parser	2542	314	4.6	2.38	2.93	9%
vpr	103	810	6.36	3.17	3.89	23%
art	40	95	3.13	2.20	2.56	3%
equake	35	91	7.11	3.99	4.66	57%
gzip	260	495	2.86	1.82	1.91	21%
mst	46	11	4	3.09	3.09	100%
treeadd	3	3	1.67	1.67	1.67	x
perimeter	54	27	3.74	3.00	3.00	x
bh	74	35	2.86	2.54	2.54	x
Average	**334**	**282**	**4.18**	**2.84**	**3.07**	**49%**

7. Related Work

Main-processor Precomputation-based Prefetching: For simultaneous multithreading (SMT) processors, Luk [10] proposes software-controlled precomputation-based prefetching in idle threads of an SMT processor. Based on a C-source level analysis of the programs, prexecution instructions are manually inserted in the code to identify where to start and end execution for several threads. The analysis targets the pre-execution of a pointer chain or a procedure call, etc., and the scheme is dependent on the application under study. Collins et al. [3] uses hardware to analyze, extract and optimize instructions for precomputation in an SMT processor. When a trigger load instruction reaches some point in the pipeline, the corresponding slice is spawned into an available thread. Later Liao et al. [8] propose a software-based speculative precomputation technique. Zilles and Sohi [19] target loads and branches by manually selecting and optimizing the precomputed instruction slices in an SMT processor. Roth and Sohi [14] propose Speculative Data Driven Multithreading and later [15] propose a framework for automated pre-execution thread selection. In contrast to SMT approaches, the proposed RSP algorithm executes on a single thread processor in memory. Program analysis is performed automatically at the assembly level by the compiler. No profiling is used and no extra instructions are added to start and end pre-execution; instead, the precomputation slice is annotated (marked).

Annavaram et al. [1] proposed a data prefetching by dependence-graph precomputation in a separate engine located in the main processor. At run-time, a hardware generator is used to generate the dependence graph from the instruction fetch queue (IFQ), based on profiling information that selected the targeted loads. By contrast, RSP proposes a compiler algorithm for program slicing and slice triggering. Therefore, no run-time overhead is incurred for selecting the instructions. RSP is not limited to the execution of instructions that are in the IFQ but is decoupled from main-processor execution. Therefore, RSP can execute any instructions in memory, including those that do not exist in the caches and can execute far ahead of where the processor is executing.

Memory-Side Forwarding: Memory-side prediction-based forwarding has been recently presented by Solihin et al. [17], where a user-level helper thread executes software that implements correlation prediction in memory. No program slice precomputation is done and therefore, no slicing technique is used. Instead, the proposed technique is prediction based. Yang

and Lebeck [18] propose a push model that adds a prefetch controller to each level of the memory hierarchy (L1 and L2 caches, and memory) to target linked data structures. The prefetch engines execute linked list traversal kernels that are downloaded into these engines initially. In contrast, RSP is general and targets all applications including those that include linked data structures. RSP has no processing in the caches.

Processing In Memory: Several processing-in-memory (PIM) architectures have been proposed, Active Pages [12], FlexRAM [7], IRAM [13], DIVA [4,5], Smart Memories [11] as well as others [9,17]. Other than [17], described in the previous section, these architectures use distributed processing by partitioning the code between all the processors. This is a different objective from that of RSP forwarding.

8. Conclusion

This paper presents a new automated compiler algorithm for Register-Synchronized Precomputation. The algorithm selects program slices for precomputation in a memory processor or dedicated thread, and inserts instructions that synchronize main and memory processor values, and trigger precomputation. At run time, the slice precomputation thread generates values of critical load instructions, which are then forwarded to the main thread. By forwarding the load values ahead of their use by the main thread, the algorithm hides the memory access latency of these loads. The main thread dynamically selects which trigger instructions to execute. This method allows the system to adaptively direct the execution of the precomputation thread and update it with only the necessary values for its calculations. We have implemented and evaluated the proposed RSP algorithm on a simulated intelligent memory system using seven SPEC CPU2000 and four Olden benchmarks. The results show performance improvements of up to 1.33 (1.13 on average) over a fully optimized code running on an aggressively latency-optimized out-of-order processor with lockup-free caches.

9. References

[1] M.Annavaram, J.M.Patel and E.S.Davidson, "Data Prefetching by Dependence Graph Precomputation", In *Proc. International Symposium on Computer Architecture*, May 2001.

[2] D.Burger and T.Austin, "The SimpleScalar Tool Set, version 2.0", *Tech.Rep.CS-1342, University of Wisconsin-Madison,* June 1997.

[3] J.D.Collins, D.M.Tullsen, H.Wang, J.P.Shen, "Dynamic Speculative Precomputation", In *Proc. International Symposium on Microarchitecture*, December 2001.

[4] J.Draper, "The Architecture of the DIVA Processing-In-Memory Chip", In *Proc. International Conference on Supercomputing,* 2002.

[5] M.Hall, et. al., "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture", In *Proc. International Conference on Supercomputing,* November 1999.

[6] W. Hassanein, J. Fortes and R. Eigenmann, "Towards Guided Data Forwarding using Intelligent Memory", *Proceedings of the Workshop on Memory Performance Issues (WMPI), held in conjunction with the 29th Annual International Symposium on Computer Architecture (ISCA),* Anchorage, Alaska, May 2002.

[7] Y.Kang, M.Huang, S.Yoo, D.Keen, Z.Ge, V.Lam, P. Pattnaik, and J.Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System", In *Proc. International Conference on Computer Design*, October 1999.

[8] S.S. Liao, P.H. Wang, H. Wang, G. Hoflehner, D. Lavery and J. Shen. "Post-Pass Binary Adaptation for Software-Based Speculative Precomputation". In *Proc. PLDI*, June 2002.

[9] J.Lee, Y.Solihin, and J.Torrellas. "Automatically Mapping Code on an Intelligent Memory Architecture". In *Proc. International Symposium on High-Performance Computer Architecture*, January 2001.

[10] C.Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors", In *Proc. International Symposium on Computer Architecture*, May 2001.

[11] K.Mai, T.Paaske, N.Jayasena, R.Ho, W.J.Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture", In *Proc. International Symposium on Computer Architecture*, June 2000.

[12] M.Oskin, F.Chong, and T.Sherwood, "Active Pages: A Computation Model for Intelligent Memory", In *Proc. International Symposium on Computer Architecture,* June 1998.

[13] D.Patterson, T.Anderson, N.Cardwell, R.Fromm, K.Keeton, C.Kozyrakis, T.Tomas, and K.Yelick, "A Case for Intelligent DRAM", *IEEE Micro*, March/April 1997.

[14] A. Roth and G.S. Sohi. "Speculative Data-Driven Multithreading". In Proc. *International Symposium on High-Performance Computer Architecture*, January 2001.

[15] A. Roth and G.S. Sohi. "A Quantitative Framework for Automated Pre-Execution Thread Selection". In *Proc. MICRO-35*, Nov. 2002.

[16] C.Selvidge, "Compilation-Based Prefetching for Memory Latency Tolerance", *PhD.Thesis, MIT,* May 1992.

[17] Y.Solihin, J.Lee, and J.Torrellas "Using a User-Level Memory Thread for Correlation Prefetching", In *Proc. International Symposium on Computer Architecture*, May 2002.

[18] C.Yang and A.Lebeck, "Push vs. Pull: Data Movement for Linked Data Structures", In *Proc. International Conference on Supercomputing*, 2000.

[19] C.Zilles and G.Sohi, "Execution-based Prediction Using Speculative Slices", In *Proc. International Symposium on Computer Architecture,* May 2001.