

1-1-2003

# Data Forwarding Through In-Memory Precomputation Threads

Wessam Hassanein

José Fortes  
*University of Florida*

Rudolf Eigenmann

Follow this and additional works at: <http://docs.lib.purdue.edu/ecetr>

---

Hassanein, Wessam ; Fortes, José ; and Eigenmann, Rudolf, "Data Forwarding Through In-Memory Precomputation Threads" (2003).  
*ECE Technical Reports*. Paper 329.  
<http://docs.lib.purdue.edu/ecetr/329>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact [epubs@purdue.edu](mailto:epubs@purdue.edu) for additional information.

# Data Forwarding Through In-Memory Precomputation Threads<sup>◇</sup>

Wessam Hassanein<sup>§</sup>

José Fortes\*

Rudolf Eigenmann<sup>§</sup>

TR-ECE 03-16

<sup>§</sup>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

---

<sup>◇</sup> 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 .....	v
LIST OF FIGURES .....	vii
Abstract .....	ix
1. Introduction.....	1
2. Key Concepts of In-Memory Precomputation-based Forwarding Threads .....	3
2.1. Critical Load Identification.....	4
2.2. Slice Generation.....	4
2.3. Trigger Insertion .....	6
2.4. Slice Optimization .....	6
2.5. IMPT Execution, Slice Filtering and Prioritization .....	7
3. Hardware Support for IMPT .....	9
3.1. Memory Processor Architecture .....	9
3.2. Managing Memory Access in the Memory Processor .....	9
3.3. Value Verification and Trigger History Table in the Main Processor .....	10
4. Experimental Methodology .....	13
5. Experimental Results .....	15
5.1. Performance of IMPT .....	15
5.2. Effect of Memory-Processor Speed and Complexity on IMPT Performance .....	18
5.3. Forwarding to Cache vs. to IVT .....	19
5.4. Effects of Off-chip Data Accesses on IMPT Performance .....	19
6. Related Work .....	21
7. Conclusion .....	23
8. References.....	25



**LIST OF TABLES**

Table 1: Simulated microarchitecture parameters .....	14
Table 2: Benchmarks simulated.....	14
Table 3: Address slice, THT and IVT analysis.....	17



**LIST OF FIGURES**

Figure 1: IMPT system microarchitecture. ....	3
Figure 2: Precomputation slices and triggers generated by our compiler algorithm. ....	5
Figure 3: Comparison of different slice optimizations. ....	7
Figure 4: Buffered DRAM microarchitecture (BDRAM). ....	10
Figure 5: IMPT speedup using profile-based vs. static critical load selection. ....	15
Figure 6: Breakdown of critical load memory accesses with and without IMPT. ....	16
Figure 7: Normalized average load access latency. ....	16
Figure 8: Dynamic percentages of triggers with different leads from trigger to load. ....	17
Figure 9: 500MHz in-order vs. 1GHz out-of-order memory-processor IMPT performance. ....	18
Figure 10: IMPT Speedup when forwarding to IVT vs. to L1 cache. ....	19
Figure 11: Effect of Off-chip data latency on IMPT performance. ....	20



## Abstract

*In modern architectures, memory access latency is an increasingly performance-limiting factor. To reduce this latency, we propose concepts and implementation of a new technique that uses an in-memory processor to precompute future, critical load addresses and forward the computed values to the main processor. The acronym for this technique is IMPT for In-Memory Precomputation-based forwarding Threads. IMPT combines the advantages of precomputation-based techniques with the low memory access latency of processing-in-memory. To evaluate IMPT, we use a cycle-accurate simulation of an aggressive out-of-order processor with accurate simulation of bus and memory contention. The results show a performance gain of up to 1.47 (1.21 on average) over an aggressive superscalar processor. The average load access latency decreases by up to 55% (32% on average).*



## 1. Introduction

Memory latency is by far the largest overhead incurred by modern processors. Prefetching is a common technique to hide latency. It has traditionally been based upon prediction. However, memory-bound applications have large data working sets and complex data access patterns that defy address prediction. Therefore, to hide the memory latency: accurate address generation and early prefetches are needed.

Precomputation-based prefetching approaches this goal by pre-executing the code that generates complex irregular addresses. The program main thread initiates an address precomputation code slice on a precomputation thread when it expects a future load miss. Recently proposed precomputation-based prefetching techniques include Collins et al.'s Speculative Precomputation [4], Luk's Software Controlled Pre-Execution [16], Roth and Sohi's Speculative Data Driven Multithreading [20,21], Zilles and Sohi's Speculative Slices [24], Liao et al.'s Software-based Speculative Precomputation [15], and Kim and Yeung's Compiler Algorithms for Pre-execution [13]. While these approaches show significant gains, they also exhibit key shortcomings. Dynamic slice selection [4] incurs significant hardware cost, manual slice selection [16,24] is not feasible for real applications, the use of profiling feedback for slice construction [13,15] or instruction traces [20,21] depends on highly specialized profiling hardware, and source level analysis [13,16] does not allow fine tuning of the constructed slices. Our approach overcomes *all* of these shortcomings. We are the first that propose *in-memory* precomputation-based forwarding. In-memory pre-execution has the advantage of direct access to all data in memory at low latency. Moreover, by placing the precomputation thread in memory, it avoids the increase in fetch and execution resource contention typical of precomputation mechanisms in the main processor. Memory-side forwarding has also been proposed by Solihin et al.'s [22], however, they are prediction based.

We refer to our technique as in-memory precomputation threads (IMPT). The memory processor fetches a program slice from memory, executes it, and forwards the resulting load value to the main processor. Precomputation consists of four critical components: precomputation-slice generation (selection), trigger insertion, slice filtering, and slice prioritization. *Slice generation* selects the instructions for pre-execution. *Trigger insertion* adds to the program code a trigger instruction that invokes the precomputation. *Slice Filtering* decides

whether to execute a slice. *Slice Prioritization* ranks the slices. Instead of implementing the precomputation components in either software or hardware, IMPT proposes a hybrid technique. The first two components are implemented in software, whereas the last two are implemented in hardware. This design keeps the hardware complexity on the processor side low, while achieving high performance.

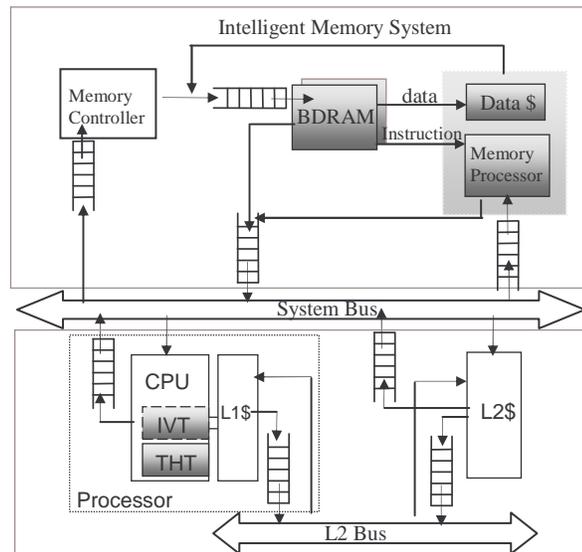
In this paper we make three main contributions: 1- We introduce precomputation techniques that execute on the memory side. 2- We propose a compiler algorithm for automatic slice selection and trigger insertion. 3- We present a new dynamic technique for slice filtering and prioritization based on Earliest Deadline First (EDF) scheduling.

IMPT uses a fully automated compiler algorithm that marks instructions for slice selection and trigger insertion. Since IMPT constructs slices and inserts trigger instructions statically at the assembly level, no dynamic trace information is needed. We study both static and profile based critical load selection. We simulate a cycle-accurate aggressive out-of-order processor with accurate bus and memory contention to evaluate IMPT. The simulation results show that IMPT achieves a performance gain of up to 1.47 (1.21 on average) on Olden [3] and SPEC CPU2000 benchmarks over a superscalar processor running fully optimized code. Our results also show a reduction of the load latency by up to 55% (32% on average).

The rest of the paper is organized as follows: Section 2 describes the key concepts of IMPT. In Section 3, we present details of IMPT hardware. Sections 4 and 5 discuss the experimental methodology and the results, respectively, and Section 6 discusses related work. Finally, conclusions are provided in Section 7.

## 2. Key Concepts of In-Memory Precomputation-based Forwarding Threads

This paper proposes a hybrid architecture-compiler approach to hide the memory access latency. The IMPT architecture precomputes program sections in memory and forwards data to the main processor prior to its use. Figure 1 shows the proposed IMPT microarchitecture. It consists of a general-purpose memory-processor (precomputation thread), an interface to the DRAM (Buffered-DRAM), a trigger history table (THT, described in Section 3.3) and an optional load Instruction Validation Table (IVT, described in Section 3.3) used to store the forwarded load values. The memory processor can reside on a DRAM chip or in the memory controller. Both cases are considered in our results. The memory processor forwards load-values to the main processor, which the IVT validates. If valid, a value is directly moved into the destination register of the load instruction. Otherwise, the instruction performs a normal load operation. Therefore, IMPT hides L2 miss latencies, as well as L1 miss latencies of loads whose values are in memory (non-dirty in the L2 data cache).



**Figure 1:** IMPT system microarchitecture (described in detail in Section 3). Highlighted components correspond to changed parts of the uni-processor microarchitecture. The system uses an IVT table only if forwarding to the main processor IVT and not to the cache (we study both cases).

IMPT differs from main-processor side pre-execution in the following aspects. 1- Memory-side execution is decoupled from main-processor side execution. Fetch units, caches, etc. are not

shared between the two threads and, therefore, do not cause resource contention. 2- Instead of prefetch requests, trigger requests are sent to memory. 3- A trigger request is sent per code slice instead of per load address. The number of requests sent to memory is reduced, as the slices tend to contain multiple loads. 4- Due to the memory-side low data access latency, IMPT leads to faster generation of load addresses that depend on other loads missing in the cache.

The following subsections describe the main tasks of the IMPT architecture. They include the identification of performance-critical loads (Section 2.1), *Slice generation* (Section 2.2), *Trigger insertion* (Section 2.3), and *Slice Filtering* and *Slice Prioritization* (Section 2.5). The hardware support is described in Section 3.

## 2.1. Critical Load Identification

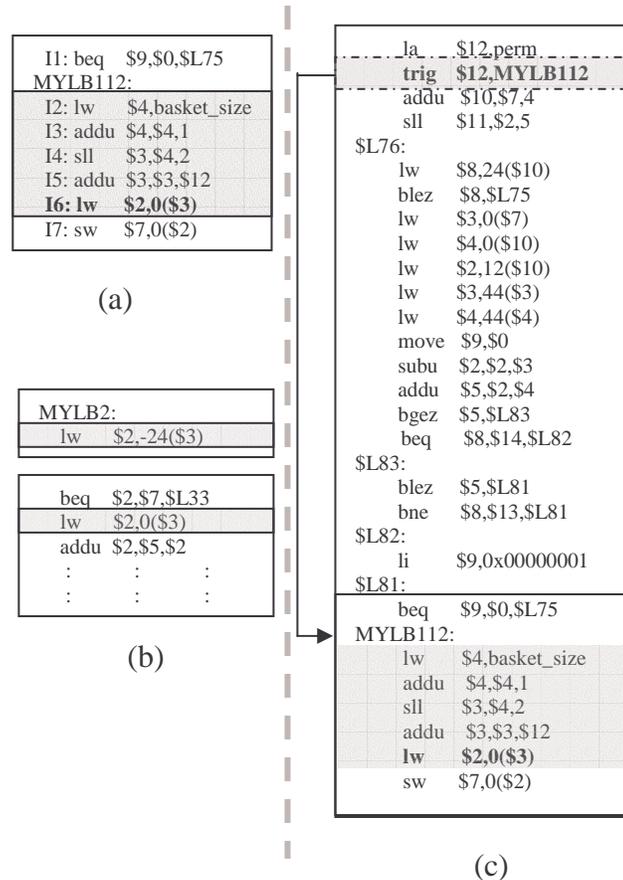
The first step in precomputation-based prefetching or forwarding is the selection of performance-critical load instructions. The common method in recent prefetching techniques [13,15] is the use of profiling runs to mark the loads causing the majority of cache misses. Profiling has the advantage of using accurate run-time information to correctly mark critical loads. However, its disadvantage is that it is only guaranteed to be precise for a single input data set. This paper investigates both the profile approach and a purely static approach.

The profile approach marks loads that cause 99% of the cache misses. By contrast, the static compiler approach classifies load instructions based on their addresses. Our experiments have shown that loads whose address is a register (not a stack, frame or global address) cause the majority of L1 cache misses (99% on average). This is the case because complex and irregular addresses (e.g. pointer-chasing, etc.) are *register loads*. Therefore, our static algorithm marks all *register loads* as critical.

## 2.2. Slice Generation

The second step of the IMPT compiler algorithm selects the instructions that generate the address of each critical load. These instructions are referred to as the precomputation slice (they have also been called p-slices, p-threads and data-driven threads). IMPT selects the precomputation slice through backward data-dependence (register-dependence) analysis at the assembly level, starting from the critical load instruction. This method yields precise code slices containing only the necessary instructions.

The compiler proceeds in two steps. First, it selects slices within basic block boundaries. Second, it combines such slices into global slices, spanning multiple blocks. Figure 2(a) shows an example of selected basic block slices. The analysis stops at the critical load's basic block boundary. Since the slice is limited to a basic block, it includes no control flow instructions. At the end of the slice selection analysis, a set of registers are identified whose values are needed by the slice but are not generated within the critical load's basic block. These registers represent the initialization (init\_reg) needed by each slice. Using this information, the algorithm generates global slices as follows. (1) Slices not requiring initialization are grouped with the preceding slice. (2) Subsequent slices that require the same register initialization are combined into a single slice. Figure 2(b) shows an example of slice combination.



**Figure 2:** Precomputation slices and triggers generated by our compiler algorithm. (a) Slice selection. I6 is the critical load. Highlighted instructions are the address generating slice. (b) Slices combined if sequential and have the same initialization register (\$3). Therefore, both load instruction slices are combined into a single slice. (c) Trigger instruction insertion (trig instruction highlighted) at the assignment of \$12. Code is from the mcf benchmark (SPEC CPU2000).

### 2.3. Trigger Insertion

Trigger instructions have two main functions. 1- They spawn (i.e. initiate) pre-execution slices in memory. 2- They supply the register initialization values needed by the slice; thus they synchronize registers between main and memory threads. For each program slice and each `init_reg` value (from Section 2.2), the algorithm adds a trigger to the program code. The trigger is placed after the register's most recent definition, using the following steps. First, the algorithm builds a full, inter-procedural program control flow graph. Second, the algorithm traverses this graph backwards from the critical load's basic block, locating the `init_reg`'s last definition point. As several paths may be traversed, several insertion points may be found. At each point, the algorithm adds a trigger instruction identifying the needed register (as shown in Figure 2(c)) and the point's offset from the targeted program slice. The distance of the trigger instruction from the corresponding load is not limited by loop or subroutine boundaries.

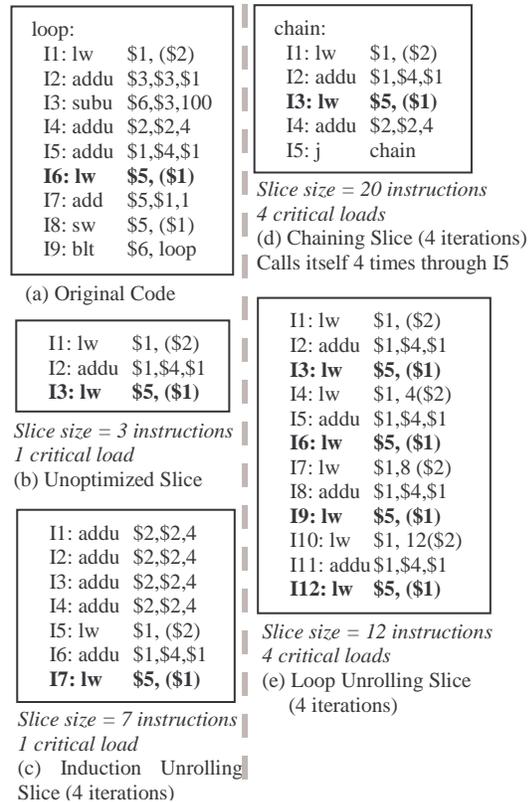
### 2.4. Slice Optimization

**Loop Unrolling:** To include several loop iterations within the slice (and therefore critical loads from several iterations of the loop), IMPT uses loop-unrolling as a pre-step prior to slice selection (Figure 3(e)). Loop-unrolling generates several instances of each critical load within a loop and combines these iteration instances into one slice. This allows a single trigger instruction to spawn several iterations of the loop including all inter-iteration dependencies.

The final code generated by the IMPT compiler is the same as the original program code with slices marked through instruction annotation and trigger instructions added. Four annotations are used; *start*, *end*, and *part of slice*, as well as *critical load*.

**Comparison with Induction Unrolling and Chaining:** Other slice optimization techniques include induction unrolling [4] and chaining [4,15]. Induction unrolling updates the loop induction variable multiple times to target a critical load multiple iterations ahead, Figure 3(c). However, the slice only targets a single iteration. By contrast, chaining targets multiple loop iterations. A chaining slice spawns multiple instances of itself, Figure 3(d). Loop-unrolling, Figure 3(e), is similar to chaining in targeting several loop iterations and therefore, critical loads. Chaining allows the number of loop iterations to be altered dynamically; however, it requires special hardware to limit the number of chain-calls, as well as compiler or hardware generation of the slice. Software-based chaining [15] requires the use of profiling information. By contrast,

loop-unrolling executes a smaller slice, has the advantage of being readily available in most compilers, and does not require any extra hardware or profiling.



**Figure 3:** Comparison of different slice optimizations.

## 2.5. IMPT Execution, Slice Filtering and Prioritization

**Main-Processor Execution of Trigger Instructions:** The execution of a trigger instruction by the main processor sends a precomputation request to the memory processor. The request contains the value of the initialization register (specified in the trigger instruction) and the address of the program slice (trigger instruction offset+PC). The request also contains a *lead time*, which is the difference (in cycles) from the trigger to the start of execution of the corresponding program slice in the main processor. The lead time is computed using the trigger history table, described next. It is used for two purposes. (1) The memory processor decides which precomputation request to take (or drop) from its queue based on the request's lead time (*Slice Prioritization* and *Slice Filtering*). (2) The main processor suppresses triggers that have a lead time of less than a threshold (*Slice Filtering*).

The trigger history table (THT) determines the trigger lead time. An entry in the trigger history table consists of the trigger's PC, the corresponding program-slice entry PC (generated from the trigger instruction, as described above), a trigger flag, the last trigger-time and a trigger deadline (lead time). When a trigger instruction is executed, the THT clears the trigger flag and updates the last trigger-time. As the main processor executes a start of program slice, the THT sets the corresponding trigger flag and updates the deadline (which equals the difference between the program-slice start time and the last trigger-time). As the trigger instruction is re-executed, if the trigger flag is set, its corresponding program-slice had been executed the previous instance and therefore this trigger instruction should be executed. In order to target the most promising triggers, THT uses a deadline threshold of 50 cycles for executing triggers. Triggers with deadlines less than the threshold are not executed. *Slice Filtering* is based on a cycle trigger-to-load lead time (deadline) rather than an instruction count. This allows IMPT more precision to include small slices that contain several dependent critical loads, exclude large slices with small execution time and abort queued slices whose deadlines have expired.

**Memory-Processor Handling of Trigger Requests:** Trigger requests arriving at the memory processor contain the program-slice address, a register number and initialization value, and a deadline. The memory processor queues and executes the trigger requests according to an Earliest-Deadline-First (EDF) scheduling scheme. Alternatively to spawning slices in order, EDF optimizes slice ordering by allowing newer and more urgent triggers to initiate their slices first. The memory processor discards requests whose deadline has expired. It combines requests for the same program-slice but with different initial registers by initializing both registers. The memory processor executes a trigger request by loading the initial register value and starting the execution at the program-slice address. Execution proceeds until it reaches the end-of-slice mark. All loads within the slice are executed and forwarded.

### 3. Hardware Support for IMPT

#### 3.1. Memory Processor Architecture

The memory processor is a general-purpose processor that executes the instructions marked by the compiler. These instructions never contain branches, stores, and floating point operations (except floating point critical loads). The processor does not need to support these operations nor speculative execution. We study both in-order and out-of-order memory-processor organizations.

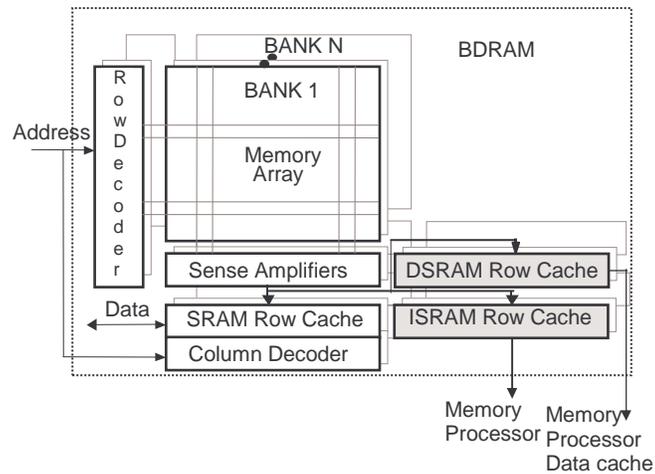
The memory-processor includes a data cache, which it reads but never writes. This cache is updated when a memory write occurs from the main processor to an address that this cache contains. No other cache coherence operations are needed. The memory processor cache contains values that are in the memory only and therefore is not linked to the main processor caches. Memory could potentially contain stale values that are dirty in the main-processor caches. Using such a value in an address precomputation leads the IVT to discard the corresponding load value. No correctness issues arise.

Program slices executed by the memory processor are the same code executed by the main processor and therefore generate a virtual load-address. We use a mechanism similar to [7] for virtual to physical address translation.

#### 3.2. Managing Memory Access in the Memory Processor

In the proposed microarchitecture, the memory processor interfaces to the DRAM through two extra SRAM buffers (ISRAM and DSRAM) as shown in Figure 4 (ESDRAM [5,8] and low latency DDR2 [6] use a similar approach to save an SRAM row for further accesses). These buffers decouple memory-processor requests from main-processor requests. Each buffer is a direct-mapped, single SRAM cache line, the size of the row buffer. ISRAM is used as an instruction cache for the memory processor. DSRAM is used as an interface between the memory-processor data cache and the DRAM. All requests made by the memory processor to the DRAM cause a memory page to be read in either the ISRAM or the DSRAM, but do not affect the value in the SRAM row cache. Therefore, this configuration preserves the value in the original SRAM buffer from reads generated by the memory processor. Main-processor requests are given a higher priority in both the DRAM and the memory bus than memory-processor

requests. We refer to this microarchitecture as Buffered DRAM (BDRAM). It is shown in Figure 4.



**Figure 4:** Buffered DRAM microarchitecture (BDRAM). The components added to a conventional DRAM are highlighted.

### 3.3. Value Verification and Trigger History Table in the Main Processor

In most main-processor prefetching architectures, cache lines are directly prefetched into the L1 or L2 caches [1,4,16,22,23,24]. Forwarding into the cache can also be used with IMPT. The performance of this option is discussed in the results section. In this case, forwarded cache lines are discarded if already available in the cache. The alternative is to forward load values directly to the main processor, to an Instruction Validation Table (IVT), by-passing the cache. This approach has no effects on the caches and the value is available directly in the main processor. While this approach requires extra hardware, it yields better performance and uses less bandwidth, as shown in Section 5.

The memory processor forwards load-values together with their addresses. The main processor stores the load value in the IVT, if it is not stale. The value is stale if its address exists dirty in the L1/L2 caches or in the Load-Store Queue. Store instructions also update values kept in the IVT.

Each entry in the IVT consists of: a valid bit, the load-value address, the load value and the load instruction address. The IVT is indexed using the load value address, as this access is in the critical path and requires 1 cycle access time (similar to a 2-way L1 cache). Updating the IVT is not in the critical path. An IVT update indexed by the load's PC is done through a table giving

the IVT location. As the main processor generates the address of a marked load instruction, it is checked against the address in the IVT. If a valid match occurs, the load value in the IVT is directly transferred to the destination register of the load. The THT is implemented similar to the IVT where it is indexed using the trigger's PC (critical path access). Updating the THT is not in the critical path and is indexed by the program slice's entry PC through a table giving the location in the THT.

As instruction cache lines are removed from the instruction cache, the IVT entries that refer to PCs in these cache lines are considered no longer needed and are thus removed. Owing to this mechanism, the size of the IVT can be kept small even with a large number of critical load instructions.



## 4. Experimental Methodology

To study the performance of IMPT, we developed a cycle-accurate simulator based on SimpleScalar 3.0a [2]. We added major enhancements to the simulator to implement accurate bus and memory contention. To this end, we have changed the buses and memory models of SimpleScalar into event driven models. The system consists of an aggressive out-of-order superscalar main processor and the microarchitecture shown in Figure 1. Table 1 shows the simulation parameters. Recent advances in DRAM technology motivate the choice of memory processor speed, where the integration of a processor running at the same speed as a logic only chip seems possible [10,11]. For comparison, we also study a slower memory processor. The latency of a memory access by the memory-processor is the same as for ESDRAM [8]. Instructions are annotated through extra bits in the SimpleScalar ISA.

To evaluate the performance of IMPT, we use the full Olden [3] suite and five SPEC CPU2000 benchmarks shown in Table 2. We concentrate on pointer-intensive C benchmarks that can run on our compiler and simulator. All benchmarks are run either to completion or for 1 Billion committed instructions. SPEC CPU2000 benchmarks are fast-forwarded for 2 Billion instructions. The extra instructions inserted by our compiler algorithm are not included in the count.

We compiled all benchmarks using the SimpleScalar version of gcc 2.6.3 with `-O3 -funroll-loops` compiler optimizations. This allows us to compare the results of IMPT against a fully optimized code including loop-unrolling.

**Table 1:** Simulated microarchitecture parameters

Module	Parameter	Value
Main Processor	Frequency	1GHz
	Issue width	Out-of-order, 4 issue
	Functional Units	4Int+4FP+4Ld/St
	Branch Prediction	2level
	Round-Trip memory latency	79 cycles (row miss) , 67 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
	Write Strategy	Writeback
Memory Processor	Frequency	1GHz & 500MHz
	Issue Width	Out-of-order & in-order, 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
Memory	Banks	4
	Page Size	4KBytes

**Table 2:** Benchmarks simulated

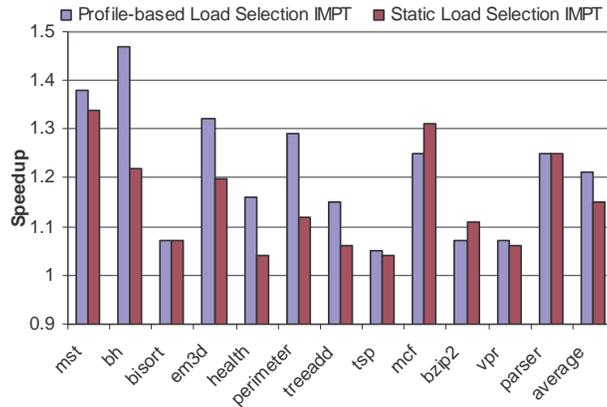
Suite	Benchmark	Input
Olden	Mst	1024
	Bh	8192
	Bisort	300000
	Em3d	25K
	Health	5 500 1
	Perimeter	12
	Power	1
	Treeadd	20
	Tsp	150000
	Voronoi	80000
SPEC CPU 2000	Mcf	train/input/inp.in
	Parser	2.1.dict -batch < train.in
	Bzip2	input.source 58 input.graphic 58 input.program 58
	Vpr	train/input/net.in arch.in place.in -nodisp -route_only -route_chan_width 15 -pres_fac_mult 2 -acc_fac 1 -first_iter_pres_fac 4 -initial_pres_fac 8
	Gzip	input.combined 32

## 5. Experimental Results

### 5.1. Performance of IMPT

In this section we present simulation-based performance evaluation of the IMPT system. The baseline in our comparison uses the same main-processor as IMPT, but no memory processor. This system, referred to as *original*, runs fully optimized, unmodified code. All performance results in this paper are normalized with respect to the *original* system.

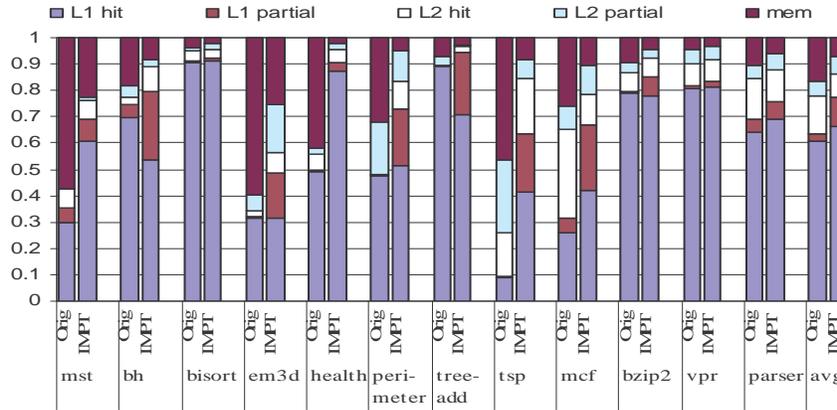
**Execution Time:** Figure 5 shows the performance results of IMPT using profile-based vs. static critical load selection for memory-bound benchmarks. All programs show speedups. Profile-based IMPT produces better performance in all benchmarks except mcf and bzip2. However, our results show that IMPT also produces good performance improvements (up to 1.34, 1.15 on average) with the conservative static load selection criteria. The rest of this paper will concentrate on the profile-based approach. IMPT improves performance by up to 1.47 (1.21 on average). IMPT does not cause any performance degradation for non memory-bound applications such as power, gzip and voronoi. Instead, only a modest performance improvement is noticed, up to 2%.



**Figure 5:** IMPT speedup using profile-based vs. static critical load selection.

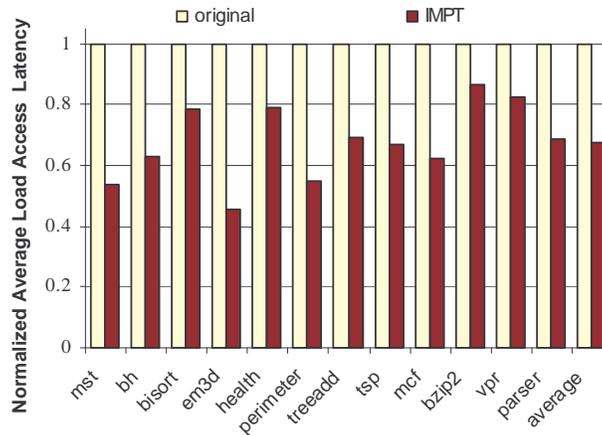
**Breakdown of Critical Load Memory Accesses:** Figure 6 shows the breakdown of critical load memory accesses with and without IMPT. Accesses are satisfied by both the memory hierarchy and the IVT and are divided into five categories based on their latencies: L1 hit (1 cycle), L1 partial (2-15 cycles), L2 hit (16-21 cycles), L2 partial (22-66 cycles) and memory (67

cycles and larger). When compared to the *original* system, the use of IMPT reduces the percentage of memory accesses in all simulated benchmarks. This reduction is considerable in highly memory-bound applications.



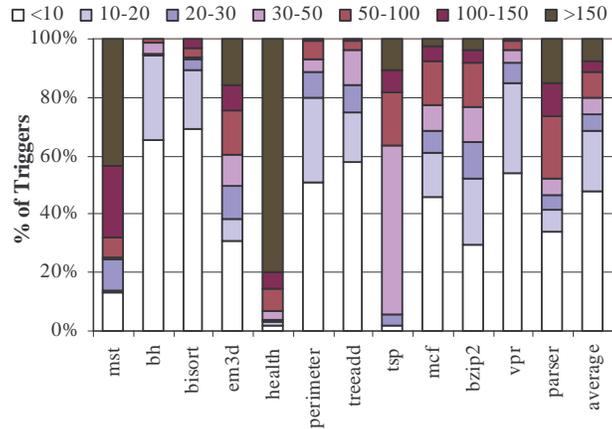
**Figure 6:** Breakdown of critical load memory accesses with and without IMPT.

**Load Latency:** Figure 7 compares the normalized average load latency, measured from the load issue time to the load write-back time. Eight benchmarks show a latency reduction of more than 30%. IMPT reduces the average latency by up to 55% (32% on average).



**Figure 7:** Normalized average load access latency.

**Trigger Lead Time:** The dynamic distance (number of 1GHz cycles) between the trigger instructions and the corresponding critical load issue on the main thread is a measure of the system's ability to initiate the precomputation thread early. This *lead time* and its distribution are characterized in Figure 8. On average, 57% of possibly sent triggers (as limited by THT deadline threshold at 50 cycles described in Section 2.5) have a lead time of 100 cycles or more.



**Figure 8:** Dynamic percentages of triggers with different leads (number of cycles) from trigger to load instruction. 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.

**Address-slice Analysis:** Table 3 shows the number of slices selected in each benchmark. On average, 47% of the intermediate loads of a slice are also critical. Critical loads that depend on such other critical loads have a compounding impact on performance, when both loads miss in the cache. This is because the prefetch or forward of the dependent load cannot be issued until all earlier load values have been received from memory. Executing these chains of load instructions directly in memory leads to significantly shorter latencies, compared to precomputation on the main processor side. This effect is key to the superior performance of IMPT.

**Table 3:** Address slice, THT and IVT analysis

(x indicates no intermediate loads)

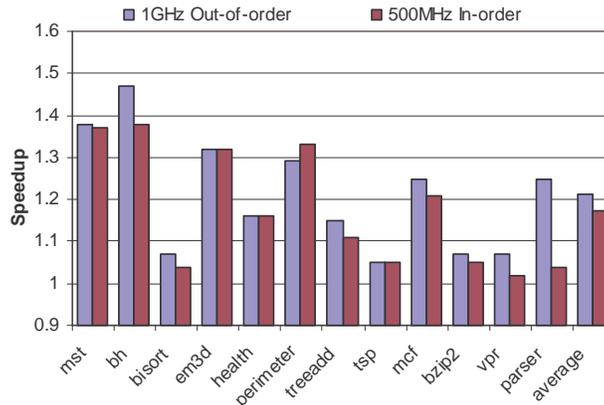
Benchmark	Number of slices	Percentage of intermediate loads that are critical	Max. No. of THT entries	Max. No. of IVT entries
mst	5	x	9	9
bh	8	x	15	12
bisort	12	x	27	34
em3d	18	55%	29	37
health	10	100%	18	24
perimeter	11	x	38	20
treeadd	4	x	4	5
tsp	8	x	15	11
mcf	41	50%	51	61
bzip2	42	x	86	40
vpr	158	6%	48	46
parser	266	25%	906	161
<b>average</b>	<b>48.58</b>	<b>47%</b>	<b>103.83</b>	<b>38.33</b>

**IVT and THT Table Sizes:** Table 3 shows the maximum size (number of entries) of the Instruction Validation Table (IVT) and the Trigger History Table (THT) needed and used by our simulations. On average, the IVT needs 39 entries, and the THT needs 104 entries. The size of the THT table indicates the dynamic number of unique trigger instructions in the program. The size of both tables is dependent upon the number of unique loads identified as critical in each benchmark.

**Bus Bandwidth and DRAM Utilization:** As in all prefetching techniques, IMPT hides memory-access latency at the expense of using more bandwidth. IMPT gives main-processor bus and DRAM requests a higher priority and therefore uses idle bus and DRAM cycles. Our simulation results show that, on average, extra requests caused by IMPT consume 20.2% of the total available bandwidth and 8.3% of the total DRAM time. The *original* system uses on average 10.2% of the total available bandwidth and 13.2% of the total DRAM time.

## 5.2. Effect of Memory-Processor Speed and Complexity on IMPT Performance

The sensitivity of IMPT performance on memory processor parameters is an important consideration. The experiments presented so far used a 1GHz out-of-order memory processor. This section investigates the effect of reducing the memory-processor speed to 500MHz and using a simpler in-order memory processor. As shown in Figure 9, using an in-order and lower-speed memory-processor has only a small effect on IMPT performance (4% on average). This is because the precomputation occurs early enough to allow the system to mask the extra latency.



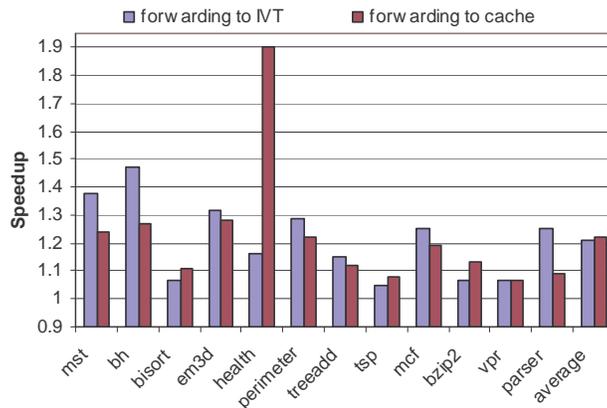
**Figure 9:** 500MHz in-order vs. 1GHz out-of-order memory-processor IMPT performance.

### 5.3. Forwarding to Cache vs. to IVT

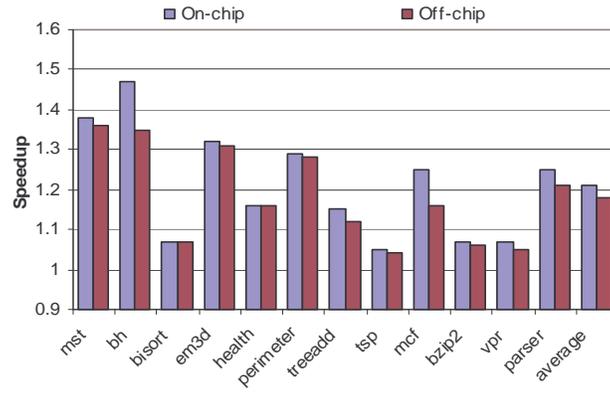
The presented IMPT mechanism forwards load values to the IVT on the main processor. Alternatively, it could forward directly to the L1 cache. The first approach requires the IVT hardware but uses less bandwidth, as only the load value is forwarded from memory. The second approach requires no IVT addition to the main processor. It forwards the whole cache line and, therefore, requires more bandwidth and could potentially pollute the cache. Figure 10 studies the effects of both alternatives on performance. Forwarding to the cache gives noticeably better performance only in health, bisort, tsp and bzip2. In all other cases IVT performs better.

### 5.4. Effects of Off-chip Data Accesses on IMPT Performance

IMPT results presented so far are using a memory-processor integrated on a single DRAM chip. In this section, we investigate the effect of the location of the memory processor on IMPT performance. In a memory organization with several DRAM chips, with one containing the memory processor, data could be located off-chip. The same would apply if the memory processor were located in the memory controller. This configuration would increase the latency from the memory processor to memory (an increase by 18 cycles, approximately doubling the latency) and reduce the bandwidth (a factor of 16 reduction), using a similar memory bus as the system bus in Table 1. The results of a system with all data accesses being off-chip (worst-case) are shown in Figure 11. There is an average performance difference of 3% between these two extreme cases. We attribute this to the tolerance of the extra latency by most precomputations.



**Figure 10:** IMPT Speedup when forwarding to IVT vs. to L1 cache.



**Figure 11:** Effect of Off-chip data latency on IMPT performance.

## 6. Related Work

**Memory-Side Forwarding:** Memory-side *prediction-based* forwarding has been presented by Solihin et al. [22]. A user-level helper thread executes correlation prediction code in memory and forwards L2 cache lines. By contrast, IMPT is precomputation-based and runs actual program slices. Therefore, it is more capable of handling irregular data accesses. In addition, IMPT forwards data-values directly to the main processor, addressing both L1 and L2 data-cache misses that are available in memory. Yang and Lebeck [23] 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. By contrast, IMPT is general and targets all applications, including those that use linked data structures. IMPT uses memory-side precomputation of program instructions that are directly read from memory. No processing is performed in the caches.

**Main-processor Precomputation-based Prefetching:** Annaram et al. [1] proposed data prefetching by dependence-graph precomputation that is generated by a separate engine located in the main processor from the instruction fetch queue. By contrast, IMPT forwards data from memory, employing memory-side precomputation. IMPT uses the compiler for *Slice Generation* eliminating run-time and hardware overheads of this step.

In simultaneous multithreading (SMT) processors, Luk [16] proposes software-controlled precomputation-based prefetching in idle threads of an SMT processor. Based on a C-source analysis, pre-execution instructions are manually inserted in the code to identify slices. 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.'s [4] Speculative Precomputation uses hardware to analyze, extract and optimize instructions for precomputation in an SMT processor. Automated software Speculative Precomputation based on profiling feedback is proposed by Liao et al. [15]. Zilles and Sohi [24] target loads and branches by manually selecting and optimizing the precomputed instruction slices in an SMT processor. Roth and Sohi [20,21] propose Data Driven Speculative Precomputation as well as a framework for pre-execution based on instruction traces. Kim and Yeung [13] propose an automated C source level compiler algorithm for pre-execution. In contrast to SMT approaches, IMPT executes on a single thread

processor in memory. Program analysis is performed automatically at the assembly level by the compiler.

**Processing In Memory:** Several processing-in-memory (PIM) architectures have been proposed, Active Pages [18], FlexRAM [12], IRAM [19], DIVA [7,9], Smart Memories [17] as well as others [14,22]. Other than the approach [22] described in the previous subsection, these architectures use distributed processing by dividing the code between all the processors. This is a very different approach from IMPT and can be viewed as complementary.

## 7. Conclusion

In this paper we present a hybrid architecture-compiler approach to hide memory-access latency. Our approach makes use of In-Memory Precomputation-based Threads (IMPT), which forward data values from memory to the main processor, prior to their use. Memory-side precomputation decouples memory pre-execution of load-address code slices from main-processor fetch and execution, and takes advantage of the low memory access latency and full access to memory data. We propose a fully automated compiler algorithm for *Slice Generation* and *Trigger insertion*, as well as hardware techniques for *Slice Filtering* and *Slice Prioritization*. To the best of our knowledge, this paper is the first that studies memory-side precomputation threads with the goal of data forwarding. Our results show that IMPT improves performance over a fully optimized superscalar processor by up to 1.47 (1.21 on average) and reduces the load latency by up to 55% (32% on average). Our work shows good results for in-memory precomputation-based forwarding, improving in *all* benchmarks over superscalar performance.



## 8. 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] M. Carlisle. "Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines", *PhD. Thesis*, Princeton University Department of Computer Science, June 1996.
- [4] J.D.Collins, D.M.Tullsen, H.Wang, and J.P.Shen, "Dynamic Speculative Precomputation", In *Proc. International Symposium on Microarchitecture*, December 2001.
- [5] V.Cuppu, B.Jacob, B.Davis and T.Mudge, "A Performance Comparison of Contemporary DRAM Architectures", In *Proc. International Symposium on Computer Architecture*, May 1999.
- [6] B.Davis, T.Mudge, V.Cuppu and B.Jacob, "DDR2 and Low Latency Variants", In *Proc. Memory Wall Workshop, held in conjunction with the International Symposium on Computer Architecture*, June 2000.
- [7] J.Draper, "The Architecture of the DIVA Processing-In-Memory Chip", In *Proc. International Conference on Supercomputing*, 2002.
- [8] Enhanced Memory Systems Inc., "64Mbit-Enhanced SDRAM preliminary data sheet", [http://www.edram.com/library/datasheets/sm2603ds\\_r1.0.pdf](http://www.edram.com/library/datasheets/sm2603ds_r1.0.pdf), 2000.
- [9] M.Hall, et. al., "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture", In *Proc. International Conference on Supercomputing*, November 1999.
- [10] IBM Microelectronics. "Blue Logic SA-27E ASIC", *News and Ideas of IBM Microelectronics*, <http://www.chips.ibm.com/news/1999/sa27e>, February 1999.
- [11] S.S.Iyer and H.L.Kalter, "Embedded DRAM Technology: Opportunities and Challenges", *IEEE Spectrum*, April 1999.
- [12] 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.

- [13] D. Kim and D. Yeung, "Design and Evaluation of Compiler Algorithms for Pre-Execution". In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
- [14] 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.
- [15] 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. Conference on Programming Language Design and Implementation*, June 2002.
- [16] C.Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors", In *Proc. International Symposium on Computer Architecture*, May 2001.
- [17] 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.
- [18] M.Oskin, F.Chong, and T.Sherwood, "Active Pages: A Computation Model for Intelligent Memory", In *Proc. International Symposium on Computer Architecture*, June 1998.
- [19] 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.
- [20] A. Roth and G.S. Sohi. "Speculative Data-Driven Multithreading". In *Proc. International Symposium on High-Performance Computer Architecture*, January 2001.
- [21] A. Roth and G.S. Sohi. "A Quantitative Framework for Automated Pre-Execution Thread Selection". In *Proc. International Symposium on Microarchitecture*, November 2002.
- [22] 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.
- [23] C.Yang and A.Lebeck, "Push vs. Pull: Data Movement for Linked Data Structures", In *Proc. International Conference on Supercomputing*, 2000.
- [24] C.Zilles and G.Sohi, "Execution-based Prediction Using Speculative Slices", In *Proc. International Symposium on Computer Architecture*, May 2001.