A High-Bandwidth Memory Pipeline for Wide Issue Processors

Sangyeun Cho, *Student Member*, *IEEE*, Pen-Chung Yew, *Fellow*, *IEEE*, and Gyungho Lee, *Senior Member*, *IEEE*

Abstract—Providing adequate data bandwidth is extremely important for a future wide-issue processor to achieve its full performance potential. Adding a large number of ports to a data cache, however, becomes increasingly inefficient and can add to the hardware complexity significantly. This paper takes an alternative or complementary approach for providing more data bandwidth, called data decoupling. This paper especially studies an interesting, yet less explored, behavior of memory access instructions, called access region locality, which is concerned with each static memory instruction and its range of access locations at runtime. Our experimental study using a set of SPEC95 benchmark programs shows that most memory access instructions reference a single region at runtime. Also shown is that it is possible to accurately predict the access region of a memory instruction at runtime by scrutinizing the addressing mode of the instruction and the past access history of it. We describe and evaluate a wide-issue superscalar processor with two distinct sets of memory pipelines and caches, driven by the access region predictor. Experimental results indicate that the proposed mechanism is very effective in providing high memory bandwidth to the processor, resulting in comparable or better performance than a conventional memory design with a heavily multiported data cache that can lead to much higher hardware complexity.

Index Terms—Data bandwidth, data locality, instruction level parallelism, runtime stack, data stream partitioning, multiported data cache.

1 Introduction

Technological and architectural innovations have enabled development of powerful microprocessors that can execute several instructions concurrently at a very high clock rate [11], [36], [12]. These processors select and execute independent instructions at runtime, assisted by hardware mechanisms for control speculation, register renaming, and data-flow execution [15]. With ample on-chip hardware resources that will become available within a few years, researchers are actively proposing even more aggressive microarchitectures that can issue up to 16 or more instructions in a single cycle [17], [24], [27]. To increase the exploitable *instruction level parallelism* (ILP) by better utilizing the available hardware parallelism, various techniques to speculate on control [19], [37], data values [18], [29], and data dependences [20], [6] are being pursued.

In a future wide-issue processor with aggressive control and data speculation techniques, efficient handling of memory references will become a more critical factor that affects the overall performance. Cache memories have been used in virtually all recent microprocessors to shorten the average memory access latency. Temporal and spatial localities are two important operating principles for various cache memories. In addition to the memory latency problem caused by the processor-memory speed gap, cache's ability to provide enough memory bandwidth (or cache ports) is extremely important for a future wide-issue processor to achieve its full performance potential [31], [16], [25], [4]. For example, for a processor to sustain 10 instructions per cycle (IPC), the memory subsystem should provide a minimum bandwidth of four references per cycle, or more, to prevent excessive queuing delays, assuming that about 40 percent of all instructions are loads and stores [17].

1.1 Multiported Data Caches

A straightforward approach for increasing memory bandwidth is to implement a multiported data cache [31]. There are a number of techniques to provide multiple cache ports: ideal multiporting, time-division multiplexing, replicating the cache, and interleaving. Except for the very expensive ideal multiporting,[1] these techniques have been incorporated in recent superscalar processors to implement dual ports. For example, Alpha 21264 provides a two-ported data cache by doubling the cache access rate compared with the normal processor clock [12]. Alpha 21164, the predecessor of the 21264, uses a replicated data cache [8] and the MIPS R10000 implements a two-way interleaved data cache [36].

Each design, however, is either costly to implement, and/or can have significant drawbacks. The time-division multiplexing does not scale beyond a certain number of

- *S. Cho is with the Media IP Group, Samsung Electronics Co., Yong-In City, Kyoung-Ki, Korea. E-mail: sangyeun.cho@acm.org.*
- *P.-C. Yew is with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455.*
- *G. Lee is with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011.*

Manuscript received 19 Jan. 2000; revised 15 Sept. 2000; accepted 26 Feb. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111241.

1. Reportedly, dual-ported synchronous SRAMs are up to around 50 percent slower, nearly 150 percent larger, and/or consume over 120 percent more power, compared with the equivalent 8-KB or 16-KB single-ported synchronous SRAMs, all in a similar $0.18\mu m$ technology [14], [22], [28].

ports (seemingly two). The replication approach broadcasts a store to each copied cache for data coherence, effectively limiting the data bandwidth on stores. It also requires significantly more silicon area than other techniques. The interleaving technique suffers from bank conflicts [16], [25]. The cost and delay of the crossbar between reservation stations and load/store units can become prohibitively large, especially when a wider instruction window and many cache ports are to be employed. Moreover, it does not generally allow a scaling factor that is not a power of two, e.g., five or six; this can become a severe restriction to a balanced, cost-effective system design.

1.2 Data Decoupling

To tackle the memory bandwidth problem, this paper studies a processor pipeline and memory system design called *data-decoupled architecture* [4], [5]. The data-decoupled architecture divides the data memory stream into two or more independent streams before their actual addresses are known by using a prediction mechanism or static information from compilers. Partitioned memory accesses are then fed into multiple, independent pipelines. This allows the use of multiple independent caches with fewer ports, each of which is associated with a dedicated pool of reservation stations. Fig. 7 shows an example data-decoupled processor pipeline with two caches.

The data-decoupled approach to the memory system design can have two crucial advantages over a conventional design when used in a wide-issue processor. First, the cost and complexity of building a large cache with many ports is reduced. Implementing a reasonably sized data cache with more than two ports becomes increasingly inefficient. That is, such a cache may occupy significantly more chip area and/or can have longer access latency [25], [4]. More importantly, the network and the control logic for orchestrating memory accesses between a large number of reservation stations and cache ports become simpler. Such reduction in hardware complexity can lead to a shorter clock cycle time [23]. Second, partitioning memory references can facilitate more specialized handling of each partitioned group of memory references. Fast forwarding, described in Section 4, is one such technique.

This paper explores a very useful behavior of memory reference instructions for high-bandwidth processor memory system design, called *access region locality*, and how it is utilized to construct a data-decoupled memory system for a wide-issue processor. The access region locality states that a memory reference instruction typically accesses a single region[2] at runtime and (thus) the region it accesses is highly predictable. An important implication of the access region locality is that any two memory references known to access nonoverlapping regions are data independent. Fig. 1 presents a small code section to illustrate the access region of memory reference instructions.

Four (static) memory references of interest are highlighted: b[i] in line 8, c[i] in line 9, *parm1 in line 10, and a in line 13. When the function foo () is called, b[i] accesses the heap region and c[i] accesses the data region, determined by where b[] and c[] are allocated (see lines 6 and 1). The access region of *parm1 is unknown from the given code segment. It can access any region at runtime, depending on the address passed from the call site of the foo () function. Since a is a local variable whose address is taken (in line 12), the reference to a becomes a stack access.

2. An access region **R** is defined as (**L**, **U**), where **L** is the lower bound on the address of the accessed locations of a memory reference instruction at runtime and **U** is the upper bound. Program's data, heap, and stack segments are regions, for instance.

```
 1  int c[LIMIT];
 2
 3  void foo (int *parm1)
 4  {
 5    int i, a,* b;
 6    b = malloc (LIMIT * sizeof(int));
 7    for (i = 0; i < LIMIT; i++) {
 8      b[i] = ...;              // heap region
 9      ...   = c[i]             // data region
10              + *parm1;       // unknown
11    }
12    bar (&a);
13    printf ("%d", a);          // stack region
14  }
```

Fig. 1. An example code segment that shows memory references to different regions.

We show in Section 3 that memory reference instructions which can access more than one region, such as *parm1, are few and their access regions are accurately predictable at runtime using a simple predictor similar to the ones used for branch prediction. Even without the program's high-level information, i.e., given only the binary code, a simple one-bit predictor can classify the memory references into stack and nonstack references with an accuracy of well over 99 percent on average.

1.3 Paper Organization

The rest of this paper is organized as follows: Section 2 summarizes previous related work. Section 3 studies the access region locality using some experimental data and develops a prediction mechanism. Section 4 discusses the data-decoupled architecture and gives an implementation that exploits the access region locality. Evaluation results based on simulation then follow. Finally, conclusions are summarized in Section 5.

2 Related Work

Recent uncovering of useful behaviors of memory access instructions has facilitated development of very interesting and effective processor memory systems. First of all, Lipasti et al. showed that the values transferred from memory by load instructions present locality and are predictable [18]. They further devised and evaluated confidence, prediction, and verification mechanisms to utilize this *load value locality* in wide-issue pipelined processors. Under their scheme, predicted load values are provided to the consuming instructions after being filtered by a confidence mechanism, which then are executed *speculatively* on the predicted values. When it is known at a later time that the previous prediction was wrong (by the verification mechanism), those executed instructions that depend on the mispredicted value are squashed and reexecuted on the correct value. They reported measurable (3-6 percent on average, depending on the machine model) and, in some cases, dramatic (up to around 20 percent) speedups achievable by the load value prediction mechanism on realistic processor models. The potential of more advanced context-based predictors was investigated by Sazeides and Smith [29].

Second, it has been shown that the actual address of many memory instructions is highly predictable. Eickemeyer and Vassiliadis [9] proposed a stride-based predictor to speculate on the address of a load instruction, in order to hide the memory latency. Austin and Sohi [2] also proposed and studied address prediction schemes using operand-based predictors. These techniques have the potential to improve the dispatch-to-issue latency of load instructions and overall performance.

Last, dependences between store and load instructions are shown to be predictable. In the most conservative form of static dependence prediction, load instructions queued after a store instruction whose address is not known are all considered *dependent* on the store. In an optimistic approach, a load instruction whose address does not match the address of any store before it (including "unknown" store addresses) in the queue is considered *independent* [13], [12]. Moshovos et al. [20] and Chrysos and Emer [6] showed that the actual dependences between store and load instructions are accurately identifiable at runtime. This memory dependence predictability can be utilized to reduce the misprediction penalty of the optimistic static prediction described above or to bypass the store data early to the dependent load(s) to shorten the store-to-load latency [21], [34]. The above three types of locality/predictability are, like the access region locality studied in this paper, based on per-instruction runtime information. We expect more studies on per-instruction memory access behaviors to come and cooperate with other types of locality, including traditional temporal and spatial localities, to build more effective and efficient memory systems.

Designing an effective multiported data cache has been a topic of active research as aggressive multiple-issue processors emerge. Sohi and Franklin [31] first predicted that the L1 cache bandwidth will eventually become a performance bottleneck for a wide-issue processor and proposed a nonblocking, multiported data cache design with interleaved banks as a solution. Wilson et al. [35] argued that adding more ports to the L1 cache can become costly and/or inefficient in terms of space and time. As an alternative to a dual-ported cache design found in some recent microprocessors, they proposed augmenting a small *line buffer* to a single-ported data cache to effectively increase the port efficiency. Rivers et al. [25] also studied the impact of using a line buffer per bank in a wide, interleaved cache. These studies have focused on increasing the efficiency of cache ports by adding a small buffer or understanding trade-offs of various multiporting strategies in terms of cost and performance under specific processor models. The data-decoupled architecture is largely orthogonal to multiported data cache design techniques. Rather, it tries to put together a number of caches to build a high-bandwidth memory system.

A more recent work worth mentioning is Yoaz et al. [38], which proposed cache bank prediction. The technique increases the cache port utilization through balanced scheduling of load instructions toward multiple cache banks. This technique also enables slicing the memory pipeline and eliminating the crossbar laid between cache banks and memory issue slots. We view the technique as a possible form of data decoupling more generally defined in this paper, where the partitioning criteria is the modulo cache line address of each memory access. This paper studies a type of memory access locality based on program semantics, which then leads to a simple, yet very accurate prediction mechanism for data decoupling.

3 Access Region Locality

A program's memory space is divided into a few regions or segments: data, heap, and stack regions under a typical runtime system [1].[3] We study in this section how each memory reference instruction accesses memory regions using a profiling tool and a set of benchmark programs. We first study how each static memory instruction accesses regions at runtime. Then, we develop and evaluate a runtime prediction mechanism. This section will serve as a basis for the discussions in Section 4.

3.1 Methodology

We use a memory reference profiler derived from the Simplescalar tool set [3] for the results reported in this section. In each simulated cycle, it fetches and executes one instruction as specified in the program. While doing so, it collects desired information, i.e., which region(s) a memory reference instruction accesses. We use eight integer and four floating-point (FP) programs from the SPEC95 benchmark suite [32], whose characteristics are summarized in Table 1. *101.tomcatv*, *102.swim*, *103.su2cor*, and *107.mgrid* are FP programs. All the programs were compiled using a version of gcc (EGCS V1.1b) at the -O3 optimization level with loop unrolling. Either *train* or *test* input is used in most cases, with some data set modification to control the simulation time. In all the experiments of this study, only user-level instructions are simulated. Operating system codes, such as system calls, are implemented within the simulator itself and they do not generate actual instructions to be simulated.

3.2 Per-Reference Memory Access Behavior

3.2.1 *Access Regions and Access Region Locality*

We analyze what region(s) each memory instruction accesses in a program execution. Depending on the accessed region(s), instructions are classified into seven different classes, as shown in Fig. 2. It is observed that majority of memory instructions, labeled "**D**" (accessing data region only), "**H**" (accessing heap region only), and "**S**" (accessing stack region only) classes, reference a single

3. Program's text region is yet another memory region. Accesses to the text region are directed to a separate instruction cache in many recent processors.

TABLE 1
Input, Dynamic Instruction Count, and Percentage of Dynamic Load and Store Instructions in Each Benchmark Program

Benchmark	Input	Inst. count	Loads	Stores
099.go	train	541M	22%	8%
124.m88ksim	ref	250M	14%	8%
126.gcc	stmt-protoize.i	220M	22%	14%
129.compress	train (100K)	293M	21%	13%
130.li	ctak.lsp	434M	28%	19%
132.ijpeg	penguin.ppm	621M	19%	9%
134.perl	scrabbl.pl	525M	26%	15%
147.vortex	train (1 iter.)	284M	29%	22%
101.tomcatv	test ($N = 253$, 1 iter.)	549M	21%	12%
102.swim	test (3 iter.)	473M	22%	8%
103.su2cor	test	676M	23%	10%
107.mgrid	train (1 iter.)	684M	32%	6%

Percentage of load or store instructions is relative to the total instruction count.

region at runtime. Only an average of 1.8 percent and 1.9 percent of all the *static* instructions access more than one region in the integer and floating-point programs studied, respectively. Although varied from one program to another, these instructions account for 0-9.6 percent of all the *dynamic* memory references. Programs such as *124.m88ksim*, *134.perl*, and *101.tomcatv* have more instructions that access multiple regions than other programs.

The strong correspondence between memory instructions and the memory regions they access is a natural consequence of how programs are written. Most memory instructions access either a fixed location (e.g., static variables and temporary local variables) or a set of locations that belong to (instances of) a single data structure, such as an array or C structure, that is allocated in a predetermined region. Therefore, even when it is difficult to predict the exact address of a memory reference, it is still feasible to predict its access region, as will be shown in this section.

Integer programs, except *099.go*, have a significant number of heap-accessing instructions as they allocate many data structures dynamically, while FP programs do not. It is interesting to note that, although each class varies much in its size, the sum of the "**D**" and "**H**" classes remains roughly comparable across programs.

Over 50 percent of all the static memory instructions only access the program's stack region at runtime on average. These instructions are for passing procedure parameters, spilling and reloading registers, and storing local variables. The static and dynamic distribution of the instructions from different classes will be determined by the writing style of the programmer, the programming language used, the underlying processor architecture, and how the compiler generates memory reference instructions, e.g., during register allocation, etc.

3.2.2 Interleaving of Accesses to Different Regions

We address two important questions for the data-decoupled architecture to be effective [4]: 1) How many dynamic references in application programs are directed to each access region and 2) how accesses to different regions are interleaved. Answers to these questions will provide a notable insight into how much memory bandwidth (or how many cache ports) is required by the memory accesses toward each region. Such information is also useful in

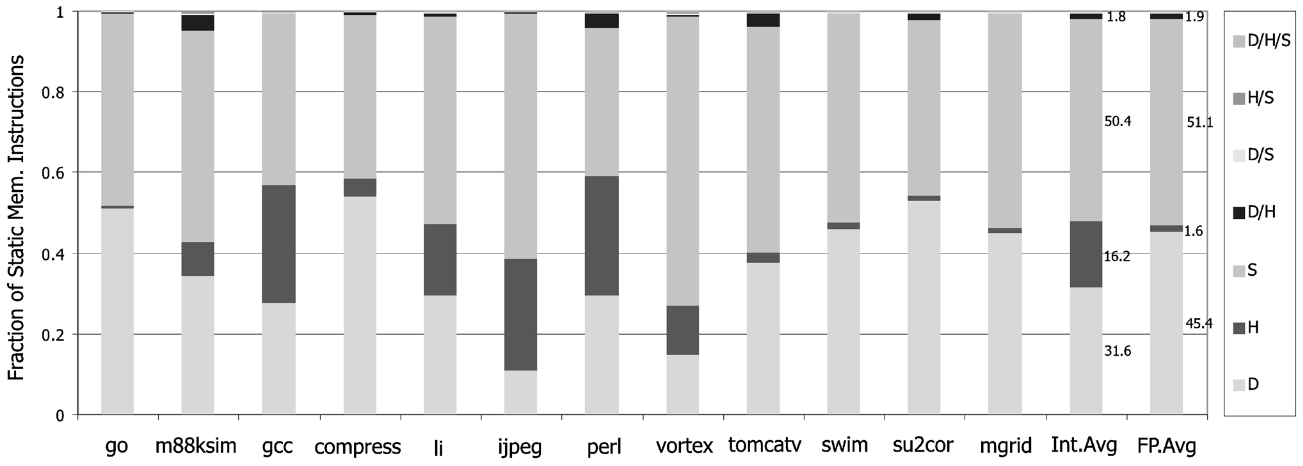

Fig. 2. Breakdown of static memory instructions based on the region(s) they access at runtime. "**D**" stands for data region, "**H**" heap region, and "**S**" stack region. "**D/S**" denotes the instructions that access both the data and the stack regions when the program is executed.

TABLE 2
Average Number of Data, Heap, and Stack Accesses in the Last 32 and 64 Instructions

	Window Size = 32			Window Size = 64		
Benchmark	Data	Heap	Stack	Data	Heap	Stack
go	6.11 (2.71)	0.00 (0.00)	3.61 (4.62)	12.23 (4.37)	0.00 (0.00)	7.23 (7.83)
m88ksim	2.91 (2.45)	2.14 (3.69)	1.90 (2.20)	5.82 (2.18)	4.29 (7.21)	3.81 (3.35)
gcc	3.48 (4.23)	1.69 (2.36)	6.45 (5.13)	6.96 (7.97)	3.38 (4.09)	12.91 (8.54)
compress	9.94 (3.70)	0.00 (0.02)	1.08 (1.50)	19.86 (6.42)	0.00 (0.01)	2.15 (2.05)
li	2.70 (1.94)	5.24 (3.77)	7.09 (4.64)	5.40 (3.21)	10.48 (6.25)	14.17 (7.44)
ijpeg	1.41 (2.22)	3.45 (3.72)	4.10 (4.94)	2.82 (4.33)	6.90 (6.95)	8.20 (8.80)
perl	2.06 (2.01)	4.79 (2.91)	6.29 (5.42)	4.11 (3.01)	9.59 (4.34)	12.58 (8.92)
vortex	1.92 (1.42)	2.80 (3.74)	11.81 (5.06)	3.84 (2.10)	5.60 (6.63)	23.63 (7.88)
tomcatv	3.96 (3.33)	0.63 (1.38)	5.97 (5.83)	7.93 (5.72)	1.26 (2.47)	11.92 (10.05)
swim	6.06 (5.09)	0.00 (0.00)	3.35 (4.45)	12.11 (8.18)	0.00 (0.00)	6.69 (6.58)
su2cor	7.38 (4.81)	0.44 (1.19)	2.98 (4.53)	14.76 (8.72)	0.88 (2.12)	5.98 (8.29)
mgrid	9.57 (2.98)	0.00 (0.02)	2.58 (1.75)	19.15 (4.41)	0.00 (0.04)	5.17 (3.00)
Average	4.79 (3.27)	1.77 (2.48)	4.77 (4.41)	9.58 (5.52)	3.54 (4.37)	9.54 (7.34)

Standard deviation of the distribution is shown in parentheses.

estimating the performance impact when separate cache ports are provided for a certain memory region.

To answer the questions, we counted the number of memory references in the last 32 or 64 instructions executed (in a 32 or 64-wide "sliding instruction window") every cycle. After constructing the distribution of the collected numbers (per region), we draw from it two major metrics used in this section: 1) the average number of memory accesses in the window and 2) the standard deviation of them. The standard deviation shows the "bursty-ness" a group of memory references exhibit; the higher it is, the more bursty the memory references are. The standard deviation becomes large when the occurrences of memory accesses are clustered. Table 2 reports the results.

Three observations are made. First, either data or stack accesses consume more memory bandwidth than the other two types of accesses in all the programs studied. There are six programs (*099.go*, *124.m88ksim*, *129.compress*, *102.swim*, *103.su2cor*, and *107.mgrid*) that have more data accesses than heap or stack accesses. The other six programs have more stack references than data or heap references. All the programs except one (*124.m88ksim*) have fewer heap references than stack references.

Second, comparing the average number of accesses and the standard deviation, accesses to the data region are less bursty than accesses to the heap or stack region. Data accesses are *strictly bursty*[4] in only two programs (*126.gcc* and *132.ijpeg*), while heap accesses are strictly bursty in eight programs and stack accesses in six programs when the window size is 32. When the window size is 64, however, only three programs (*099.go*, *132.ijpeg*, and *103.su2cor*) have bursty stack accesses.

Third, as pointed out earlier, there are few heap accesses in floating-point programs. In programs that have many heap accesses, such as *130.li*, *132.ijpeg*, and *134.perl*, there are relatively fewer data accesses, suggesting that these programs distribute their data structures and the related accesses among the data and the heap regions as necessary. Furthermore, it is shown that heap accesses are quite bursty, even when the window size is 64. This implies that processing heap accesses separately will not generally bring much benefit, especially for the floating-point programs.

From the above observations, it is concluded that many programs have relatively constant memory bandwidth demand for data and stack accesses, especially when the processor buffers many instructions to search for independent instructions to achieve higher performance.

3.3 A Case for Decoupling Stack Accesses

The previous subsection suggests that if an extra cache for stack references is provided, the data cache bandwidth can be saved significantly for many programs. Heap access distribution showed irregular shapes among different programs and in different phases in a single program. Therefore, if one wishes to partition the set of memory access instructions into two groups that access distinct memory regions, the stack-accessing instructions are a good candidate to decouple from others. The remainder of this paper focuses on servicing stack and nonstack references separately with two exclusive caches. The details of the architectural change and the effects will be discussed in Section 4.

In addition to the above observations, there are also other practical considerations that favor separate handling of stack accesses. We will show in Section 4.5 that stack accesses show a very high degree of data locality and a separate cache for stack need not be as large as a conventional data cache to attain a high hit rate (also in [7]). A 4-KB stack cache achieved over 99.5 percent hit rate for the SPEC95 benchmark programs, with an average of about 99.9 percent.

3.4 Predicting Access Regions

3.4.1 *Dynamic Access Region Prediction Strategies*

Memory instructions can be categorized into three classes: ones that always access the stack ("S" in Fig. 2), ones that

4. In this paper, accesses to a region are considered *strictly bursty* if the average number of accesses in the given instruction window is smaller than the standard deviation. *132.ijpeg*'s data, heap, and stack accesses are all strictly bursty, for instance.

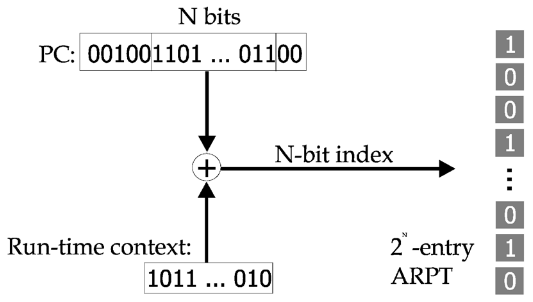

Fig. 3. Operation of the ARPT with 1-bit entries. *N*-bits from PC (XOR'ed with context bits) is used to index an 2^N-entry ARPT.

always access nonstack regions ("D," "H," or "D/H"), and ones that can access both the stack and nonstack regions. The goal of the discussions in this section is to develop an efficient mechanism to accurately predict which region (stack or nonstack) an instruction will access at runtime, given the program counter (PC) of it. We use a hardware table called *access region prediction table* (ARPT), similar to *branch prediction table*, and a set of heuristics for higher prediction accuracy. The prediction result, available as early as at the fetch stage, is used to guide the instruction dispatcher for data decoupling, as discussed in Section 4. A schematic figure to illustrate how the ARPT operates is given in Fig. 3.

Note that it is already shown that most instructions access only a single region during program execution in Fig. 2. Therefore, if a single bit is allocated for each static memory instruction to store the previously accessed region, e.g., using "1" to indicate stack and "0" to indicate nonstack, the access region of most instructions can be accurately predicted from next time by just looking up their history bit. This is called the *simple one-bit scheme* in this paper. Alternatively, we can use two bits per instruction to add hysteresis or inertia to prediction (*simple two-bit scheme*).

There are two issues in obtaining good prediction accuracy given the above base predictors. First, when a memory instruction is first executed, i.e., the history of the instruction is not available, how do we predict the access region of it?[5] The second issue is how we can effectively handle a memory instruction that accesses the stack ("S") and nonstack ("N") regions irregularly, e.g., "SNSNSNNNSN."

To cover the above first issue, we note that the addressing mode of many memory instructions immediately reveals the regions that will be accessed. We use the following baseline heuristics in our study:[6]

(Static Prediction):

1. If the addressing mode is constant, the instruction will access a nonstack region.
2. If \$sp (stack pointer) or \$fp (frame pointer) is used for indexing, the instruction will access the stack region.
3. If \$gp (global pointer) is used for indexing, the instruction will access a nonstack region.
4. For other cases, i.e., instructions with an index register other than \$sp, \$fp, or \$gp, *predict* that a nonstack region will be accessed.

The addressing mode information can be obtained early from the predecoding logic or the decoding stage at the latest. In any case, the information from the ARPT at the fetch stage will be discarded by the instruction dispatcher if the addressing mode information is available from the (pre)decoder. Fig. 4 shows that many instructions, over 50 percent on average, manifest their access regions in their addressing mode (lower dark bars). In order to save space in the ARPT, these instructions are not recorded. It is noted that a compiler has more knowledge of the access region of memory instructions than is disclosed in the addressing mode of them. The impact of transporting compiler information to the hardware will be studied in Section 3.5.2.

Next, we use an ARPT index function with additional runtime information to tackle the second issue. Fig. 3 shows how the ARPT is indexed with the PC of the memory reference instruction XOR'ed with a string of bits, labeled "runtime context." We consider two types of runtime context in our work: *global branch history* (GBH) and *caller's identification* (CID). Modern branch predictors already use the GBH to index the branch prediction table [19]. The path through which the control reaches a memory instruction, like in a branch predictor, may help predict the behavior of the memory instruction. Considering which procedure called the current procedure gives a valuable insight into which region a pointered reference will access (such as *parm1 in Fig. 1) because it is likely that the calling procedure passes to the called procedure the same set of arguments or at least the arguments of the same types repeatedly. The *link register* usually keeps the next PC of the call instruction (that resides in the calling procedure) and thus can be used as a unique CID. It is also possible to combine these two types of information by concatenating a certain number of bits from the GBH and also from the CID to form a value to be used as context bits.

We evaluated the following five different schemes with an unlimited ARPT: the baseline static classification (STATIC), simple 1-bit scheme (1-BIT), 1-bit with GBH (1-BIT-GBH), 1-bit with CID (1-BIT-CID), and 1-bit with both the GBH and CID (1-BIT-HYBRID). When GBH and CID are used together, we use the lower 8 bits from the GBH and lower 24 bits from the CID concatenated together to form a 32-bit context.[7] Fig. 4 shows the results; Overall, 1-BIT-HYBRID performed the best in terms of prediction rate (99.89 percent and 100 percent for the integer and floating-point programs respectively).[8]

The 1-BIT scheme outperformed 1-BIT-GBH and 1-BIT-CID. This is because adding some runtime context in the indexing function increases the number of entries in ARPT (as shown in Table 3) and causes unnecessary cold misses for certain memory instructions that don't need any

5. Because we don't use a tag or valid bit in the ARPT, this is similar to "How do we initialize the content of the ARPT entries?"

6. This is based on Simplescalar's *Portable ISA* (PISA) [3]. However, the discussions can be easily extended to other ISAs.

7. Experiments revealed that this combination provides reasonable performance across programs.

8. We omit presenting the performance of 2-bit schemes because their performance is consistently lower than that of 1-bit schemes.

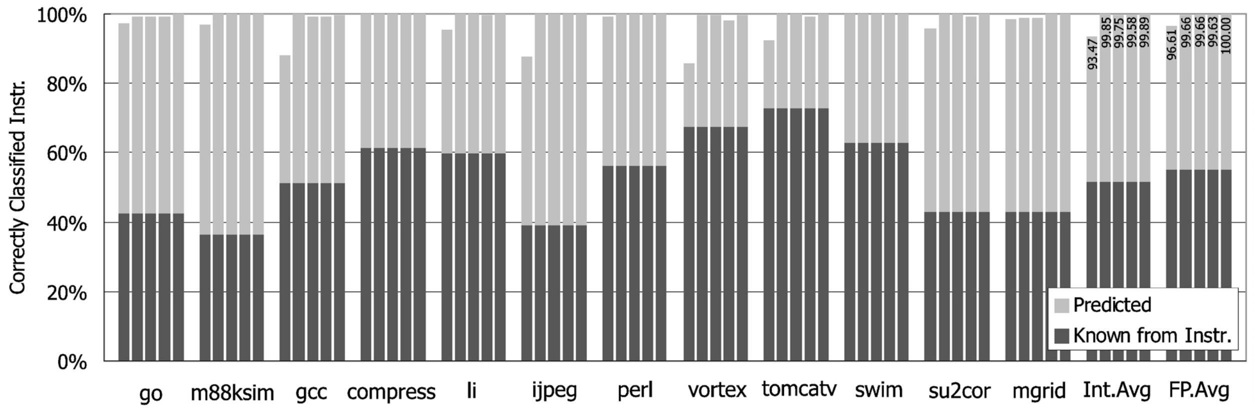

Fig. 4. Percentage of dynamic memory instructions that are correctly classified into stack and nonstack regions using various schemes. Bars for each benchmark program denote, from left, the static classification, the simple 1-bit scheme, the 1-bit scheme w/ GBH context, the 1-bit scheme w/ CID, and the 1-bit scheme w/ hybrid context (8-bit GBH and 24-bit CID).

runtime context, but still can be reached via a number of different procedures or control flows. These misses offset the benefit of using the context bits and, in many cases, lead to lower overall prediction rate unless the obtained benefit is large enough. *147.vortex* and *103.su2cor* are the examples where 1-BIT is the clear winner over 1-BIT-CID. Notice, however, that indexing with either runtime context can lead to better performance than 1-BIT, as is observed in *107.mgrid*.

3.5 Performance of the Access Region Prediction Table (ARPT)

3.5.1 ARPT of Limited Size

We study the impact of the ARPT size on the prediction rate. Fig. 5 (lower light bars) shows the performance of 1-bIT-HYBRID when the ARPT size is varied from 64K to 8K entries. In general, the ARPT loses the prediction accuracy when its size becomes smaller. Certain programs, such as *099.go*, *126.gcc*, *147.vortex*, and *101.tomcatv*, are more sensitive to the ARPT size than others. Table 3 indicates that these programs require significantly more entries than other programs when a type of runtime context is used (except *101.tomcatv*), putting high pressure on the ARPT size. In the case of *101.tomcatv*, it seems that there are negative interferences in the ARPT. This is indirectly supported by the fact that the availability of compiler information removes any deficiencies caused by the limited ARPT size. It is interesting to observe that a limited ARPT performs better than an unlimited ARPT in the case of *126.gcc*. This means that there are positive interferences between different memory instructions that map to a single ARPT entry.

To summarize, a 32K-entry ARPT achieved over 99.9 percent prediction accuracy for both the integer and floating-point programs studied. The necessary hardware resources for implementing a 32K-entry ARPT is small—only 4 KB of space.

TABLE 3
Number of Entries Occupied in an Unlimited ARPT, when Used with Different Context Bits for Indexing

Bench.	STATIC	w/ GBH	w/ CID	w/ HYBRID
go	7896	9733 (23%)	13465 (71%)	34421 (336%)
m88ksim	1227	1319 (7%)	1326 (8%)	2093 (71%)
gcc	10521	11484 (9%)	15949 (52%)	30517 (190%)
compress	556	590 (6%)	573 (3%)	765 (38%)
li	822	862 (5%)	973 (18%)	1467 (78%)
ijpeg	3137	3467 (11%)	2681 (-15%)	5002 (59%)
perl	2140	2255 (5%)	2874 (34%)	4229 (98%)
vortex	6836	7094 (4%)	5833 (-15%)	12200 (78%)
tomcatv	877	954 (9%)	1033 (18%)	1430 (63%)
swim	958	1062 (11%)	1072 (12%)	1520 (59%)
su2cor	1760	2220 (26%)	3214 (83%)	5887 (234%)
mgrid	1095	1322 (21%)	1519 (39%)	2620 (139%)

Increase in number compared to "Static Prediction" is shown in parentheses.

3.5.2 Effects of Compiler Hints

A compiler often has knowledge of which region a memory instruction will access at runtime. For example, b[i] and c[i] in Fig. 1 are easily identifiable as heap and data references. Most stack accesses are accurately identifiable when the compiler comes across variable declarations, handles a procedure call, or performs register allocation and reloading. For nonstack accesses, the type and storage information of a variable symbol provides necessary information. A simple compiler algorithm to determine the access region of a memory instruction is shown in Fig. 6.

This section studies the effects of augmenting each static memory instruction with a tag that indicates if it is a stack access, a nonstack access, or that the compiler cannot distinguish (such as *parm1 in Fig. 1). For the experiments, we used profiled region information gathered from program runs instead of implementing the compiler algorithm and performing static analysis. The accessed region(s) of each instruction is saved in the profile data and we assumed that an instruction can be classified by a compiler if it is shown to access only a single region during program execution. Therefore, the quality of the compiler information used in this section should be interpreted as one that is from a very accurate compiler analysis (i.e., upper bound).

The compiler information helps to improve the performance of the ARPT in two ways. First, the resultant prediction rate becomes higher because many instructions have been taken care of by the compiler and bypass the

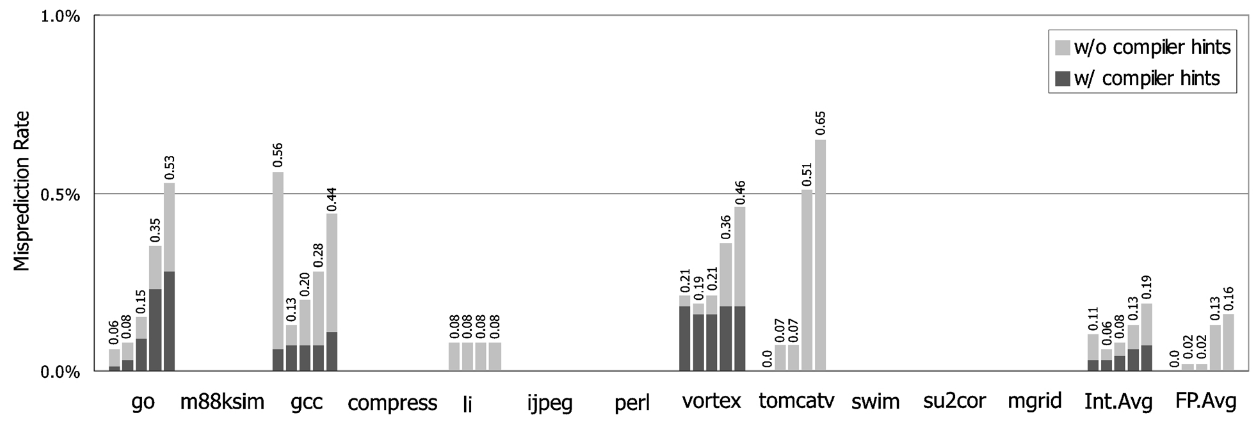

Fig. 5. Misprediction rate of the 1-BIT-HYBRID when ARPT size is varied. Results for the ARPT of unlimited size, 64K, 32K, 16K, and 8K entries are shown from left. The dark, lower portion of each bar denotes the ARPT misprediction rate when compiler information is available. For *130.li* and *101.tomcatv*, the misprediction rate was 0 if compiler information is provided.

prediction mechanism. Second, the space of the ARPT is saved since many correctly tagged instructions do not need to occupy the entries in the ARPT. This will decrease the effects of negative interferences and allow us to use a smaller ARPT to achieve a given prediction rate. In other words, compiler hints will reduce the pressure on the ARPT size, especially when a lot of entries are needed (i.e., when a type of context bits is used).

Fig. 5 (dark portions) shows that, in the presence of compiler information, 1) the prediction accuracy is higher and 2) the performance of the ARPT becomes less sensitive to its size. *101.tomcatv* benefits greatly from the compiler information when the ARPT has less than 32K entries. It is shown also, however, that the misprediction rate without compiler information is already very low when the ARPT has 32K entries or more; therefore, we conclude that, although compiler information is useful, the hardware mechanism proposed in this paper is accurate enough for effective data decoupling, as will be shown in the next section. Use of the dynamic technique allows running existing binaries on a data-decoupled processor without noticeable loss of performance.

```
mem_type classify_mem (mem_instr* instr)
{
  if (is_local_var (instr)) return MT_STACK;
  if (is_static_var (instr)) return MT_NONSTACK;

  // Pointer deref.; assume ptr is the pointer for instr
  int flag = -1;
  for all def in UD-chain for ptr {
    if (is_function_param (def)) return MT_UNKNOWN;
    if (point_to_unknown (def)) return MT_UNKNOWN;
    if (point_to_stack (def))
      if (flag == 0) return MT_UNKNOWN;
      else flag = 1;
    if (point_to_nonstack (def))
      if (flag == 1) return MT_UNKNOWN;
      else flag = 0;
  }
  if (flag) return MT_STACK;
  return MT_NONSTACK;
}
```

Fig. 6. A simple compiler algorithm to classify the access region of a memory instruction.

4 Data-Decoupling for a Wide-Issue Superscalar Processor

4.1 Definition

The idea of *data decoupling* is to partition the set of memory access instructions into two (or more) separate instruction streams before the instructions enter the instruction window, where they wait until they become ready to issue. Partitioning of instructions will be performed in such a way that the data locations accessed by the instructions in one stream will not be overlapped by the ones from another stream. Instructions in each partitioned stream are then sent to a set of dedicated reservation stations (that compose a part of the instruction window) after they are decoded; eventually, each set of reservation stations feeds a separate data cache (see Fig. 7).

The proposed approach can ease the hardware complexity of a wide-issue superscalar processor in two ways. First, the bandwidth requirements on the data cache can be mitigated since multiple caches serve the memory references. Considering that existing multiported data cache schemes become very expensive and inefficient as more ports are required [16], [25], the reduction in port requirements can lead to a much cheaper and more efficient cache design. Second, the window logic to select and arbitrate the ready memory instructions and the network to route them to the actual cache ports can be simpler since each memory window can now be made smaller and each cache can have fewer ports. The window logic has been pointed out to be one of the most critical hardware components as the processor becomes wide and a more advanced process technology with smaller feature sizes is used [23]. Because the delay of the window logic is proportional to the square of the instruction window size and the square of the issue bandwidth, this partitioning can become a crucial advantage.

It should be noted that, given the same number of available cache ports, the net effect of data decoupling is *not* an increased IPC, but decreased hardware complexity—

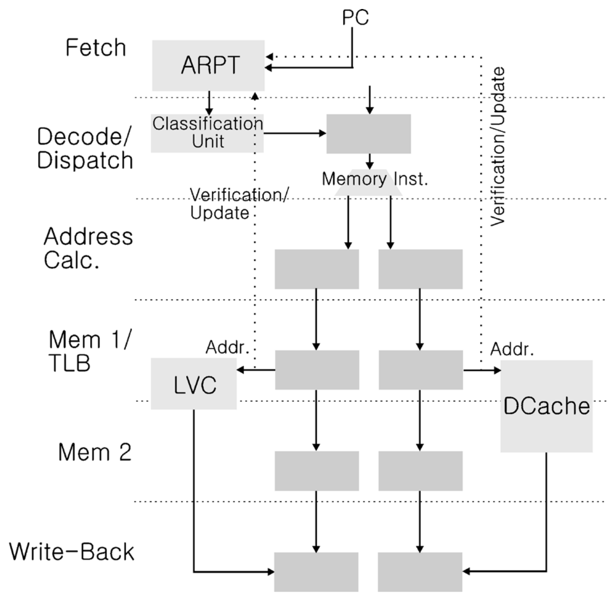

Fig. 7. an example processor pipeline augmented with the ARPT and a separate memory pipeline for stack and nonstack references each.

fewer cache port requirements and simpler instruction issue logic. In fact, the resulting IPC could be impaired due to unbalanced resource utilization. However, since it is increasingly difficult to add more than two ports to a cache and to build a wide instruction window without penalizing clock cycle time, achieving a comparable performance with simpler hardware is a valid goal [23]. This is a very critical issue for the future wide-issue processor proposals [17], [24], [27]; they put more pressure on the data cache bandwidth as they aggressively speculate on control and register values and use high-bandwidth instruction caches [37], [26]. Under such conditions, the proposed approach can in turn have a performance advantage by providing more data bandwidth than a conventional technique at the same level of hardware complexity. Furthermore, the proposed approach can expose opportunities for reduced memory access latency by properly partitioning memory references and optimizing each stream.

4.2 Architectural Support

To service local variable accesses efficiently, a specialized hardware organization to simulate the runtime stack may appear attractive [7], [10]. However, we use a more general cache design, called the *local variable cache* (LVC), in the framework of our data-decoupled architecture. This approach has two advantages; First, the LVC is a conventional cache and can leverage the most efficient current design. Second, certain events, such as a buffer overflow due to bursty stack growth (that can happen when a recursive function is called, for example) and a context switch, are easily handled without CPU intervention.

The LVC is associated with a group of reservation stations, called the *local variable access queue* (LVAQ). It has the same organization as the conventional *load store queue* (LSQ). Since the LVC is placed at the same level as the L1 cache, it will be attached to the memory bus connecting to the L2 cache and will make the bus arbitration logic slightly more complex.

Further optimizations are possible for the LVAQ and the LVC. Two such techniques to improve the local variable accesses are introduced:

- **Fast data forwarding.** To decrease the memory load latency, data from a store can be forwarded to a later load. The MIPS R10000 processor forwards data to address-matching loads in the LSQ on a refill [36]. By tracking previously manifested dependences and keeping store data in a separate hardware table, data forwarding from a store to a load can be performed speculatively [21], [34]. There is another opportunity to perform fast forwarding in the LVAQ without keeping dependence tables (i.e., no speculation). Accesses to the stack region in a procedure are usually based on the same value of \$sp, i.e., \$sp is not updated within a procedure. The dependence checking hardware can use the offset field in instructions to identify a matching store-load pair within a function frame, even before their effective addresses are calculated, allowing faster bypassing of data. This technique will be beneficial when there are many local store-reload pairs within a small section of code, such as spill codes generated by a compiler.
- **Access combining** [35]**.** When a program or a program region contain many local variable accesses, the number of LVC ports can become a performance bottleneck. In fact, a procedure call/return generates a sequence of stack accesses for saving/restoring registers and passing parameters. These stack accesses show strong spatial locality, i.e., accessing adjacent memory locations in a row. *Access combining* tries to combine two or more contiguous references that fall onto the same LVC line at the expense of wider LVC ports, buffers, and associated logic. The technique will decrease the traffic to the LVC, relieving the bandwidth requirement on it.

4.3 Pipeline Design

We describe in this subsection an example memory pipeline of a data-decoupled processor equipped with an ARPT.

- **Fetch stage.** ARPT is accessed with current PC XOR'ed with runtime context bits. A single bit value returned from the ARPT is passed to the decode stage. Note that this way of accessing the ARPT is a baseline design and that other ways of implementing the ARPT access mechanism are possible. For example, memory instructions in the trace cache [26] lines can be tagged with prediction bits to relieve the high bandwidth requirements on the ARPT. A part of ARPT accesses can be deferred to the decode stage also.
- **Decode (and Dispatch) stage.** Fetched instructions are decoded and dispatched to issue slots of various execution units. If a decoded instruction is a memory

TABLE 4
The Base Machine Model

Base Machine Model	
Issue width	16
Number of regs.	32 GPRs/32 FPRs
ROB/LSQ size	256/128
Functional units	16 integer + 16 FP ALUs, 4 integer + 4 FP MULT/DIV units.
Value predictor	Stride-based predictor for register values. 16K-entry table.
L1 data cache	2-way set-assoc. 64 KB. 2-cycle hit time. Varying number of ports.
L2 data cache	4-way set-assoc. 512 KB. 12-cycle access time.
Memory	50-cycle access time. Fully interleaved.
Local Var. Cache	Direct-mapped. 4 KB. 1-cycle access time.
ARPT	32K 1-bit entries.
Instruction cache	Perfect I-cache with a 1-cycle latency.
Branch predictor	Perfect.
Inst. latencies	Same as those of MIPS R10000 [36].

Decode and commit widths are the same as the issue width.

instruction, checking is made if the addressing mode shows the access region of the instruction. If so, the ARPT bit from the fetch stage is simply discarded; however, if the addressing mode does not give any hint, the ARPT bit guides which of the two instruction queues, LSQ or LVAQ, the instruction is steered into. Since this *dynamic memory stream partitioning* is speculative, it is required that each memory access be verified against the correct access region information available later when the actual address is calculated.

- **Address calculation stage.** The address of a ready memory instruction in LSQ or LVAQ is calculated in this stage. While waiting for dependences to be resolved and for available cache ports, a load instruction could receive bypassed data from previous stores.
- **Memory access stages.** Using the address calculated in the previous stage, TLB and cache are accessed in the first memory access stage. In the same cycle, access region verification is done through a verification unit. Alternatively, the TLB can be used for verification, where each TLB entry is extended with a single bit, indicating whether the translated page belongs to the designated region (in this paper, the stack) or not. Storing such information can be done accurately and efficiently when a page is allocated by the runtime system. Note that both accessing the cache and the verification action are done simultaneously.

When it is known that an instruction has been placed in a wrong pipeline, the instruction will be reissued to the right queue and the ARPT will be updated. As the depending instructions are still waiting in the instruction queue, the recovery action on detecting a mispredicted memory instruction will be either squashing those instructions, or just reestablishing the dependence chain from the previously mispredicted instruction [17]. Because the verification process will eventually prevent any store instruction from writing data to a wrong cache, as well as any instruction from executing on a wrong load value, the coherence in the current dual cache scheme will be guaranteed.

4.4 Methodology and Architecture Model

We use a detailed timing simulator based on the Simplescalar tool set [3]. The machine model used in the timing simulator supports out-of-order issue and execution, based on the *Register Update Unit* (RUU) [30]. The RUU scheme uses a reorder buffer (ROB) to automatically perform register renaming and hold the results of pending instructions. In each cycle, the ROB retires completed instructions in program order to the architected register file. The processor pipeline consists of six stages: fetch, dispatch (decode and register renaming), issue, execution, write-back, and commit. Depending on the instruction type, more than a cycle can be taken in the execution stage.

The processor's memory system employs a *Load Store Queue* (LSQ). Store values are placed in the queue if the store is speculative. Loads are dispatched to the memory system when the addresses of all previous stores are known. Loads may be satisfied either by the memory system or by an earlier store value residing in the queue. In the latter case, the store-to-load forwarding delay is one cycle.

We use an aggressive processor model that can issue up to 16 instructions per cycle. The model represents a future wide-issue processor with aggressive issue bandwidth from a large instruction window. The ROB has 256 entries and the LSQ has 128 entries, which are derived from the MIPS R10000 implementation [36]. The ROB and the LSQ effectively form the instruction window of the processor. The primary on-chip data cache that is 64 KB and 2-way set-associative has a 2-cycle hit time (unless otherwise stated), as in some of the recent machines [36], [12]. The 512-KB L2 cache has a 12-cycle hit latency. Both the caches are lock-up free.

We implemented a stride-based value predictor for the register values as the performance improvement achieved by value prediction techniques is mainly from the register data flow [17]. The value prediction table has 16K entries in our processor model.

We use an aggressive front-end with a perfect instruction cache and a perfect branch predictor to assert the maximum

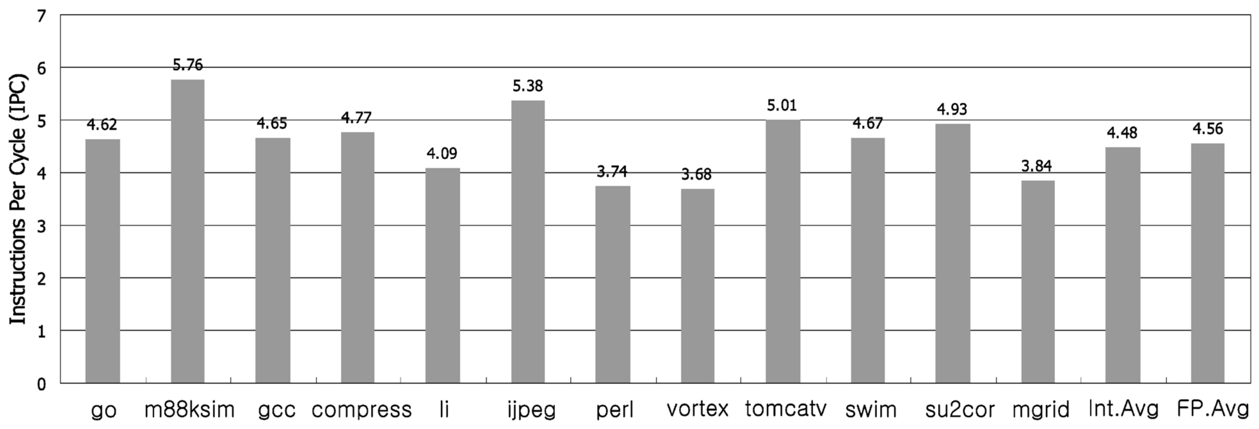

Fig. 8. Instruction Per Cycle (IPC) of the baseline (2+0) configuration.

pressure on the data memory bandwidth. This setting is necessary to accurately study the impact of the proposed techniques by eliminating other factors that affect the measured performance. Due to this, the primary performance metric used in the result section is relative performance, after presenting the base IPC numbers at the beginning. Important parameters of the base machine model are summarized in Table 4.

For data decoupling, the direct-mapped 1-cycle LVC is sized to 4 KB. The ARPT has 32K 1-bit entries with no tags or valid bits (4 KB table size). For ARPT indexing, the 15 bits of PC (above least-significant zeros due to a large instruction size) XOR'ed with context bits composed of 8 bits from global history and 7 bits from link register. The LSQ/LVAQ size was set to 96/96 entries each. On detecting an ARPT misprediction, the mispredicted instruction will be reissued to the right instruction queue in a single cycle after accessing the verification unit (e.g., TLB). As a result, the dependent instructions will further reside in their respective reservation stations, delayed at least three cycles in our pipeline model.

In the following discussions, the notation "(N+M)" is used to denote a processor configuration with an N-port data cache and an M-port LVC. When M equals 0, the configuration is a conventional memory design with an N-port data cache.

4.5 Experimental Results

4.5.1 Baseline Performance

The aggressive processor model with the perfect front end resulted in very high baseline IPC numbers, as shown in Fig. 8. The perfect branch prediction assumption especially made the performance of integer programs comparable to that of numerical applications. The performance of this (2+0) configuration is, however, severely restricted by the cache bandwidth. Fig. 11 shows that if cache has unconstrained bandwidth, represented by the (16+0) configuration, the performance is boosted by up to 80 percent (*147.vortex*), with an average of over 20 percent. Experiments further revealed that adopting a cache with unlimited size will not create such performance gain, implying that the cache hit rate becomes a secondary performance factor to the cache bandwidth, as long as a reasonable cache size is used.

4.5.2 LVC Size

The size of the LVC should be carefully chosen to keep the miss rate low. At the same time, the LVC should be sufficiently small and simple to keep the access time short.

Fig. 9 shows the measured miss rates when the LVC size is varied from 0.5 KB to 4 KB. A 2-KB LVC achieves a hit rate of over 99 percent for all the programs except *126.gcc*. A 4-KB LVC obtains a hit rate of 99.5 percent or more for all the programs, with an average of about 99.9 percent. The major reason why a small LVC can achieve a high hit rate is that function frames tend to be very small [7], [4]. Furthermore, most of the programs have a call depth of four to five routines [33]. The line size of the LVC, be it 32 or 64 Bytes, had negligible effects on the hit rate with an LVC of 2 KB or a larger size. The hit rate of an LVC was also relatively insensitive to the input data because the function frames are generally determined at compile time. For the rest of the experiments, a 4-KB, direct-mapped LVC with an one-cycle hit latency is used.

The additional caching space provided by a 4-KB LVC resulted in slight decrease in traffic to L2 cache for all the programs as some conflicts between stack and nonstack accesses in the data cache are removed. Therefore, when it is difficult to have a large L1 data cache in a wide-issue processor (in order to keep the clock cycle time short), data decoupling can provide a way to effectively increase the cache size. For example, *130.li* and *147.vortex* showed a

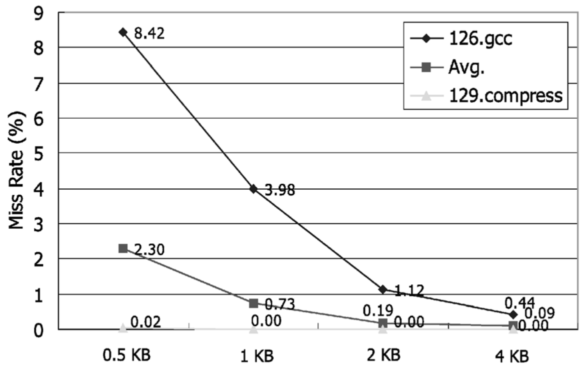

Fig. 9. Miss rates of the LVC of different sizes. 126.gcc and 129.compress show the highest and lowest miss rate, respectively. A direct-mapped LVC with four ports is used for measurement.

TABLE 5
Performance Improvement with Fast Data Forwarding under the (3+2) Configuration

Program	*099.go*	*124.m88ksim*	*126.gcc*	*129.compress*	*130.li*	*132.ijpeg*
Speedup	2.1%	0%	1.2%	1.2%	0.3%	1.9%
Program	*134.perl*	*147.vortex*	*101.tomcatv*	*102.swim*	*103.su2cor*	*107.mgrid*
Speedup	3.1%	3.9%	3.9%	0.2%	0.5%	0%

considerable reduction in the L2 cache accesses and, accordingly, the traffic on the memory bus, of 24 percent and 7 percent, respectively, when a 32-KB data cache was assisted by a 2-KB LVC [4]. This decrease in the memory bus traffic will improve the overall performance of a processor in the presence of heavy traffic on the bus, as well as the power consumption caused by accessing larger memories and the (high capacitance) bus.

4.5.3 Impact of Fast Data Forwarding in LVAQ

Table 5 shows the performance improvement provided by fast data forwarding. Speedups of up to 3.9 percent were observed. Looking at individual programs, *124.m88ksim* does not benefit from the technique at all because only about 1 percent of the loads actually find their values in the LVAQ. On the other hand, *129.compress* gains a speedup of 1.2 percent even though it has fewer local variable accesses because almost 80 percent of all the local variable loads find their values in the LVAQ. This suggests that the reuse distance of local variable accesses in *129.compress* is relatively short.

In spite of many local variable accesses, *130.li* didn't get a noticeable speedup because most of the local variable accesses are not on the critical path of the program. Bandwidth, therefore, is more important than latency in this case. Fig. 11 shows that, when the memory bandwidth is the performance bottleneck (baseline (2+0) configuration), adding a two-port LVC achieved a spectacular speedup of over 50 percent (represented by the (2+2) configuration), whereas much smaller speedup is achievable when there is sufficient bandwidth already (the (3+3) configurations compared with the (3+0) configuration).

The net performance gain produced by fast data forwarding was not impressive. However, when cache access latencies should be prolonged to meet the desired (fast) clock rate, the impact of this technique can become more significant.

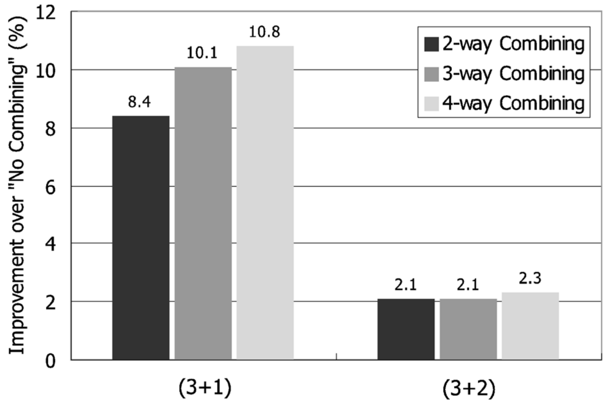

Fig. 10. Performance of access combining. N-way combining looks at up to N consecutive entries in the LVAQ for access combining.

4.5.4 Impact of Access Combining in LVAQ

Fig. 10 shows the effect of access combining under the (3+1) and (3+2) configurations. Two-way combining achieves a speedup of around 8 percent and 2 percent over "No Combining" in each configuration. Two programs, *130.li* and *147.vortex* (not shown), exhibited a speedup of 16 percent and 26 percent, respectively, in the (3+1) configuration. *147.vortex* still crops over 12 percent speedup in the (3+2) configuration.

The results suggest that access combining can considerably reduce the bandwidth requirements on the LVC, especially when the memory pressure is high (or when the provided bandwidth is insufficient). Taking into account the hardware complexity to implement access combining,[9] the two-way combining seems to be a reasonable choice for implementation.

4.5.5 Overall Performance

Fig. 11 shows the performance of various configurations relative to that of the (2+0) configuration. The results clearly show that the baseline configuration with a 2-ported data cache does not provide enough bandwidth: The (16+0) configuration (i.e., unlimited bandwidth) improves the performance by 33 percent (integer) and 25 percent (floating-point) on average, indicating that there is large room for performance improvement when more data bandwidth is provided beyond 2 ports. For *147.vortex*, the improvement was as high as 80 percent. It is also worthwhile to note that our experiments showed that the (2+0) configuration with a 128 KB data cache produces little performance improvement over the same configuration with a 64 KB data cache (by less than 1 percent).

Two (3+0) configurations with a 2- and 3-cycle latency achieved 21 percent and 18 percent improvement for the integer programs, respectively, and 14 percent for the floating-point programs. This shows that, when the processor performance is limited by the data bandwidth, the impact of access latency becomes smaller, especially in a dynamically scheduled processor like our baseline model. Only *099.go* shows a sharp performance drop as a 3-cycle latency is used for cache access. Floating-point programs were hardly affected by the latency change, implying that the latency was effectively hidden by out-of-order execution. Having a 4-ported data cache, represented by the (4+0) configuration, improves the performance by 25 percent (integer) and 20 percent (floating-point). Considering that

9. To implement two-way combining, for example, will require wider (doubled) bus per port and two-input multiplexors for each subblock corresponding to access unit, with slightly more complex control.

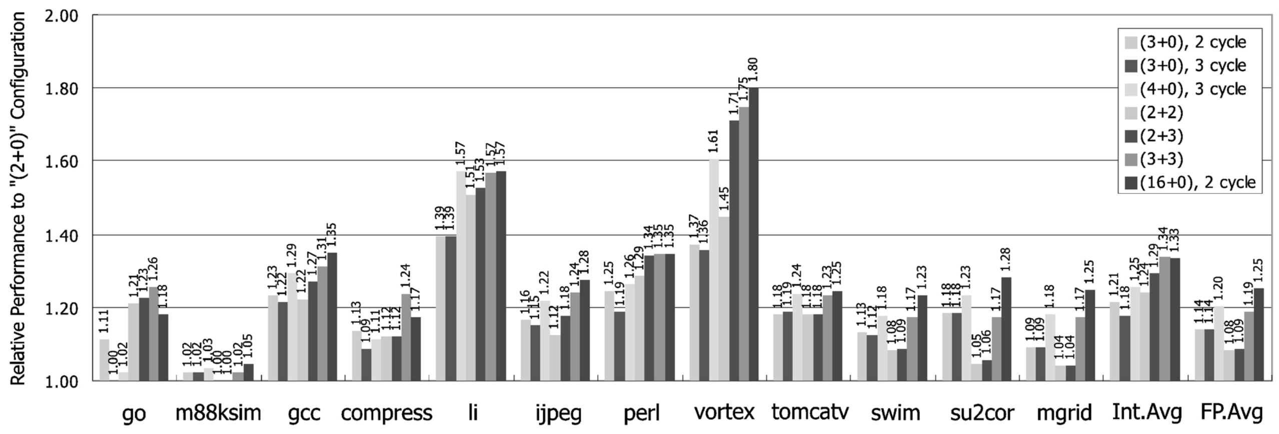

Fig. 11. Performance of various configurations compared with that of the baseline (2+0) model. The (16+0) configuration on the rightmost side is shown as an upper bound (unlimited bandwidth).

adding more ports beyond four gives diminishing returns, this configuration might become the choice for a conventional design. However, the hardware complexity caused by the large 128-entry LSQ and the cache can become excessive. We have accordingly set the cache access time to be three cycles for the configuration so as not to increase the clock cycle time.

Comparing the (2+2) and (4+0) configurations, they achieved similar performance for integer programs, but the (4+0) configuration performed better for the floating-point programs. Under the (2+2) configuration, floating-point programs showed limited performance, mainly because they have more bandwidth demand for data region than stack. Therefore, attaching one more port to the LVC, the (2+3) configuration, doesn't help improve the performance of these programs at all. On the other hand, certain integer programs, like *126.gcc* and *147.vortex*, obtain additional speedups.

The (3+3) configuration performs well for both the integer and floating-point programs. For the integer programs, this configuration was as good as the (16+0) configuration on average, which is the limit case for our study. For the floating-point programs, the (3+3) configuration was close to the (4+0) configuration. In summary, when it is not feasible to build a single many-ported data cache attached to a large instruction window, a data-decoupled configuration, such as (3+3), can become a viable alternative that achieves a similar performance level. The configuration choice should be made, however, after thoroughly investigating the cost of implementing a particular multiported data cache and related hardware complexity, as the studied models in this paper assume perfect multiporting.

Finally, thanks to the high prediction accuracy of the ARPT, increasing the misprediction penalty hardly affects the overall performance. Adding five more cycles to the misprediction penalty incurred only 0.6 percent performance degradation on average.

5 Concluding Remarks

This paper studied an important behavior of memory access instructions, called the access region locality, and how it is used to predict the region of memory accesses. Utilizing the access region locality, we showed the effectiveness of the data decoupling as a way of increasing on-chip data memory bandwidth. The following contributions are made in the paper:

- A novel memory pipeline organization for a wide-issue processor, called data-decoupled architecture, is proposed. To achieve high data bandwidth without placing much pressure on the data cache port requirements and the window logic complexity, the data-decoupled architecture partitions the memory stream early before instructions are directed into the instruction window.
- The notion of access region locality is introduced. Also given is a set of detailed profile data, showing that most memory reference instructions access only a single region at runtime. The definition of access region can be further refined so that the number of access regions can be made larger to match increasing need of bandwidth.
- Techniques to predict the access region of a memory instruction before the actual effective address is calculated are developed and evaluated. Results show that the proposed prediction mechanism with reasonable hardware resources can precisely determine the access region of an instruction (to be either stack or nonstack reference) with an accuracy of over 99.9 percent on average.
- Using a detailed cycle-by-cycle simulator, we studied the potential performance of a wide-issue superscalar processor with the proposed data decoupled memory pipeline that exploits the access region locality. Results show that the proposed decoupled approach secures comparable performance compared to a conventional unified data cache design with the same number of ports. In certain cases, the proposed technique offers an opportunity for performance improvement that is not achieved by merely adding more ports to the data cache.

Compiler and architectural considerations for efficient memory handling remain as a very important part of designing a balanced, cost-effective processor. We expect that the *decouple-and-conquer* approach to the memory bandwidth and/or latency problem, as proposed in this paper, will be of greater significance as more aggressive wide-issue processors emerge.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation under grant numbers MIP-9610379 and CDA-9502979; by the US Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order number D346, and a gift from the Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US Army Intelligence Center and Fort Huachuca or the US government. The authors thank Todd Austin and Doug Burger for developing the Simplescalar tool set that has been indispensable for our study. The anonymous referees provided many constructive comments that greatly helped improve the quality of this paper. Sangyeun Cho was supported in part by a doctoral fellowship from the Korea Foundation for Advanced Studies (KFAS). Gyungho Lee was supported in part by the US National Science Foundation under grant number CCR-0073259 and by Alpha Processor, Inc.

REFERENCES

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Principles, Techniques, and Tools.* Addison-Wesley, 1986.
[2] T.M. Austin and G.S. Sohi, "Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency," *Proc. 28th Int'l Symp. Microarchitecture,* pp. 82-92, Nov. 1995.
[3] D. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report No. 1342, Computer Sciences Dept., Univ. of Wisconsin, June 1997.
[4] S. Cho, P.-C. Yew, and G. Lee, "Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor," *Proc. 26th Int'l Symp. Computer Architecture,* pp. 100-110, May 1999.
[5] S. Cho, P.-C. Yew, and G. Lee, "Access Region Locality for High-Bandwidth Processor Memory System Design," *Proc. 32nd Int'l Symp. Microarchitecture,* pp. 136-146, Nov. 1999.
[6] G. Chrysos and J. Emer, "Memory Dependence Prediction Using Store Sets," *Proc. 25th Int'l Symp. Computer Architecture,* pp. 142-153, July 1998.
[7] D. Ditzel and R. McLellan, "Register Allocation for Free: The C Machine Stack Cache," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems,* pp. 48-56, Mar. 1982.
[8] J. Edmondson et al., "Internal Organization of the Alpha 21164, a 300-MHz, 64-Bit, Quad-Issue, CMOS RISC Microprocessor," *Digital Technical J.,* vol. 7, no. 1, 1995.
[9] R.J. Eickemeyer and S. Vassiliadis, "A Load-Instruction Unit for Pipelined Processors," *IBM J. Research and Development,* vol. 9, no. 2, 1993.
[10] M.J. Flynn and L.W. Hoevel, "Execution Architecture: The DELtran Experiment," *IEEE Trans. Computers,* vol. 32, no. 2, pp. 156-175, Feb. 1983.
[11] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report,* vol. 9, no. 2, Feb. 1995.
[12] L. Gwennap, "Digital 21264 Sets New Standard," *Microprocessor Report,* vol. 10, no. 14, Oct. 1996.
[13] D. Hunt, "Advanced Performance Features of the 64-bit PA-8000," *Proc. COMPCON,* pp. 123-128, 1995.
[14] IBM, *ASIC SA-27E Databook,* 2000.
[15] M. Johnson, *Superscalar Microprocessor Design.* Prentice Hall, 1991.
[16] T. Juan, J.J. Navarro, and O. Temam, "Data Caches for Superscalar Processors," *Proc. Int'l Conf. Supercomputing,* pp. 60-67, July 1997.
[17] M.H. Lipasti and J.P. Shen, "Superspeculative Microarchitecture for beyond AD 2000," *Computer,* pp. 59-66, Sept. 1997.
[18] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen, "Value Locality and Load Value Prediction," *Proc. Seventh Int'l Symp. Architectural Support for Programming Languages and Operating Systems,* pp. 138-147, Oct. 1996.
[19] S. McFarling, "Combining Branch Predictors," WRL Technical Note TN-36, Digital Equipment Corp., June 1993.
[20] A. Moshovos, S.E. Breach, T.N. Vijaykumar, and G.S. Sohi, "Dynamic Speculation and Synchronization of Data Dependences," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 181-193, June 1997.
[21] A. Moshovos and G.S. Sohi, "Streamlining Inter-Operation Memory Communication via Data Dependence Prediction," *Proc. 30th Int'l Symp. Microarchitecture,* pp. 235-245, Dec. 1997.
[22] NEC, *Block Library CB-11 Family Databook,* 2000.
[23] S. Parlacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 206-218, June 1997.
[24] Y.N. Patt, S.J. Patel, D.H. Friendly, and J. Stark, "One Billion Transistors, One Uniprocessor, One Chip," *Computer,* pp. 51-57, Sept. 1997.
[25] J.A. Rivers, G.S. Tyson, E.S. Davidson, and T.M. Austin, "On High-Bandwidth Data Cache Design for Multi-Issue Processors," *Proc. 30th Int'l Symp. Microarchitecture,* pp. 46-56, Dec. 1997.
[26] E. Rotenberg, S. Bennet, and J.E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. 29th Int'l Symp. Microarchitecture,* pp. 24-34, Dec. 1996.
[27] E. Rotenberg, Q. Jacobson, and J.E. Smith, "Trace Processors," *Proc. 30th Int'l Symp. Microarchitecture,* pp. 138-148, Dec. 1997.
[28] Samsung Electronics Co., *STD130 Databook,* 2000.
[29] Y. Sazeides and J.E. Smith, "The Predictability of Data Values," *Proc. 30th Int'l Symp. Microarchitecture,* pp. 248-258, Dec. 1997.
[30] G.S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Trans. Computers,* vol. 39, no. 3, pp. 349-359, Mar. 1990.
[31] G.S. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 53-62, Apr. 1991.
[32] The Standard Performance Evaluation Corporation, http://www.specbench.org, 1995.
[33] Y. Tamir and C.H. Sequin, "Strategies for Managing the Register File in RISC," *IEEE Trans. Computers,* vol. 32, no. 11, pp. 977-989, Nov. 1983.
[34] G. Tyson and T.M. Austin, "Improving the Accuracy and Performance of Memory Communication Through Renaming," *Proc. 30th Int'l Symp. Microarchitecture,* pp. 218-227, Dec. 1997.
[35] K.M. Wilson, K. Olukotun, and M. Rosenblum, "Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors," *Proc. 23rd Int'l Symp. Computer Architecture,* pp. 147-157, May 1996.
[36] K.C. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro,* vol. 16, no. 2, pp. 28-40, Apr. 1996.
[37] T.-Y. Yeh, D.T. Marr, and Y.N. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," *Proc. Seventh Int'l Conf. Supercomputing,* pp. 67-76, July 1993.
[38] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation Techniques for Improving Load Related Instruction Scheduling," *Proc. 26th Int'l Symp. Computer Architecture,* pp. 42-53, May 1999.

Sangyeun Cho received the BS degree in computer engineering from Seoul National University, Seoul, Korea, in 1994. He earned the MS degree in computer science from the University of Minnesota in Minneapolis in 1996, where he has been a PhD candidate since 1998. Since 1999, he has been with Samsung Electronics Company, where he is now a senior engineer. He has designed several generations of the CalmRISC™ processor core and their caches. Prior to joining Samsung, he was a summer intern at Intel Microcomputer Research Laboratory in 1998. His current research interests are in low-power embedded processors and their memory hierarchy design, as well as high-performance computer architecture. He is a student member of the IEEE.

Pen-Chung Yew received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1981. He is currently a full professor and the head of the Department of Computer Science and Engineering, University of Minnesota. Previously, he was an associate director of the Center for Supercomputing Research and Development at the University of Illinois. From 1991 to 1992, he served as the program director of the Microelectronic Systems Architecture Program in the Division of Microelectronic Information Processing Systems at the National Science Foundation, Washington, D.C. Pen-Chung Yew is a fellow of the IEEE and has served on the program committee of various conferences. He also served as a cochair of the 1990 International Conference on Parallel Processing, a general cochair of the 1994 International Symposium on Computer Architecture, and the program chair of the 1996 International Conference on Supercomputing. He served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems* from 1992 to 1996 and the *Journal of Parallel and Distributed Computing* from 1989 to 1995. His research interests include high-performance multiprocessor system design, parallelizing compilers, computer architecture, and performance evaluation.

Gyungho Lee received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1986. He is currently a professor of computer engineering at Iowa State University. His research and teaching interests are in computer design and architecture, high-performance computing, and computer security. His industrial experiences include leading an effort to develop a next generation microprocessor based on Compaq's Alpha 21264 at Samsung Austin Design Center in 1997 and working as the principal architect of a shared-bus symmetric multiprocessor SSM 7000 at Samsung Electronics from 1990 to 1992. His academic research experiences include participating as a research assistant in the Parafrase, a parallelizing compiler project, and the Cedar, a shared-memory multiprocessor project at the University of Illinois at Urbana-Champaign from 1982 to 1986. From 1992 to 1996, he led the DICE project at the University of Minnesota, which invented the bus-based cache-only memory multiprocessor (US patent no. 5,692,149) and noninclusive memory access mechanism (US patent no. 5,937,431). He was a recipient of the 1986 Outstanding Paper Award from the 15th International Conference on Parallel Processing, St. Charles, Illinois, for his work on "combining switch." He is a subject area editor for the *Journal of Parallel and Distributed Computing* and is on the editorial board for the *Journal of Parallel Computing*. He currently serves as a distinguished visitor of the IEEE Computer Society and is a senior member of the IEEE.