Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor

Sangyeun Cho, Pen-Chung Yew[†], and Gyungho Lee[‡]

MCU Team, System LSI Div.
Samsung Electronics Co.
Yongin-City, Korea

[†]Dept. of Comp. Sci. and Eng.
University of Minnesota
Minneapolis, MN 55455

[‡]Division of Engineering
Univ. of Texas at San Antonio
San Antonio, TX 78249

E-mail: `sangyeun.cho@acm.org`

Abstract

Providing adequate data bandwidth is extremely important for a wide-issue superscalar 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 the data-decoupled architecture. The approach, with support from the compiler and/or hardware, partitions the memory stream into two independent streams early in the processor pipeline, and feeds each stream to a separate memory access queue and cache. Under this model, the paper studies the potential of decoupling memory accesses to program's local variables that are allocated on the run-time stack. Using a set of integer and floating-point programs from the SPEC95 benchmark suite, it is shown that local variable accesses constitute a large portion of all the memory references, while their reference space is very small, averaging around 7 words per (static) procedure. To service local variable accesses quickly, two optimizations, fast data forwarding and access combining, are proposed and studied. Some of the important design parameters, such as the cache size, the number of cache ports, and the degree of access combining, are studied based on simulations. The potential performance of the proposed scheme is measured using various configurations, and it is concluded that the scheme can become a viable alternative to building a single multi-ported data cache.

1 Introduction

Efficient handling of memory references is one of the keys to achieving high performance in processors that exploit instruction-level parallelism (ILP). Although a significant body of work has been done to tolerate or hide the memory latency, the ever widening processor-memory speed gap calls for more aggressive and innovative techniques to tackle this difficult problem. Furthermore, the ability to provide the execution core with adequate (cache) memory bandwidth becomes extremely critical for the next generations of wide-issue processors [25, 31]. For example, for a processor to sustain ten 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% of all instructions are loads and stores [15].

A straightforward approach for increasing memory bandwidth is to implement a multi-ported data cache [25]. There are a number of techniques to provide multiple cache ports: ideal multi-porting, time-division multiplexing, replicating the cache, and interleaving [21]. Except for the very expensive ideal multi-porting, these techniques have been incorporated in recent superscalar processors. For example, DEC 21264 provides a two-ported data cache by running the cache twice as fast as the processor clock [13], DEC 21164, the predecessor of 21264, uses a replicated data cache [9], and MIPS R10000 implements a two-way interleaved data cache [33]. Each design, however, is either costly to implement, and/or has significant drawbacks. The time-division multiplexing does not scale beyond a certain number of ports (seemingly two). The replication approach broadcasts a store to each cache for data coherence, effectively limiting the data bandwidth on stores, and requires doubled silicon area. The interleaving technique suffers from bank conflicts. The cost and delay of the crossbar between reservation stations and load/store units can become significant. Moreover, it does not generally allow a scaling factor that is not a power of two, *e.g.*, five or six, which is a severe restriction to a balanced, cost-effective system design.

Instead of building a large set-associative data cache with multiple ports, we propose an alternative or complementary approach toward higher memory bandwidth, called the *data-decoupled architecture*. It divides the data memory stream into two independent streams before they enter the reservation stations, and feeds each stream to a separate memory unit. The separation of independent memory references, in an ideal situation, facilitates the use of dual data caches with a smaller number of ports, each of which is as-

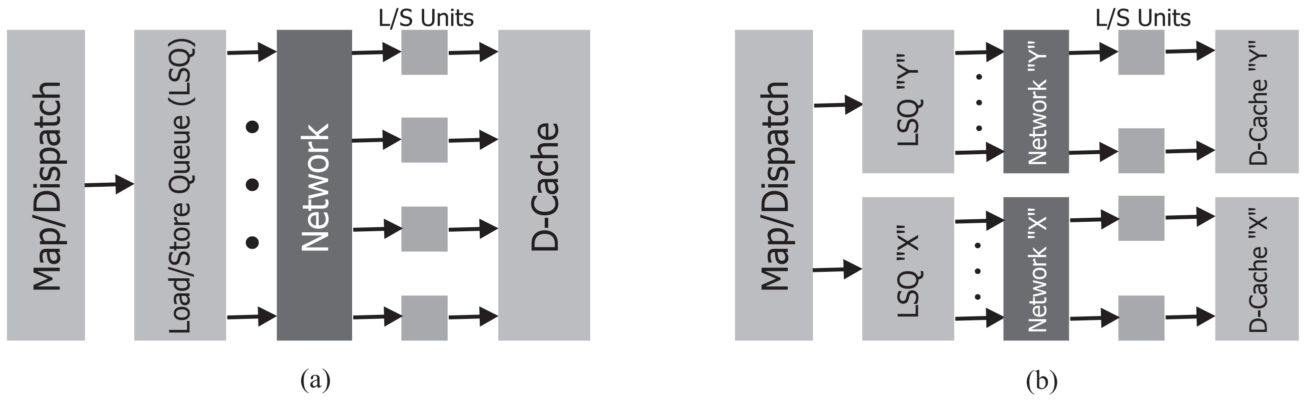

Figure 1. An example pipeline of (a) a conventional superscalar processor with a 4-port data cache and (b) a data-decoupled architecture with dual memory access queues and caches with 2 ports each.

sociated with a dedicated pool of reservation stations or an *access queue*.

The data-decoupled approach to the memory system design can have two crucial advantages over a conventional design in a wide-issue processor. First, the cost and complexity of building a large cache with many ports is avoided. Instead, existing cache designs with a smaller number of ports can be used. Further, 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 [18]. Second, dividing the data stream into smaller streams can open up new opportunities to optimize each with specialized techniques. For instance, various speculative techniques on data dependence and forwarding [16, 17, 30] can be tailored to each stream for higher efficiency.

This paper investigates the potential of decoupling local variable accesses from the memory stream and directing them to a small separate cache called the *local variable cache* (LVC) via an instruction queue called the *local variable access queue* (LVAQ). To service the local variable accesses efficiently, we study two optimizations – *fast data forwarding* and *access combining*. Although there have been efforts to optimize local variable accesses with hardware support in the past [8, 12, 26], little work has been done to study their performance impact in the context of a wide-issue superscalar processor with a multi-ported data cache. Experimental results based on an execution-driven simulator and a set of SPEC95 integer and floating-point programs suggest that the proposed approach can achieve comparable or better performance than a conventional approach. The additional hardware resources for data decoupling seem to be modest.

In the remainder of the paper, Section 2 presents the concept of the data-decoupled architecture, the motivation toward decoupling local variable accesses, and the related architectural/compiler issues. Section 3 then describes the experimental setup – the processor model, the execution-driven simulation environment, and the benchmark programs studied. Evaluation results are presented in Section 4. Related work is discussed in Section 5, and the conclusions are summarized in Section 6.

2 Data-Decoupled Architecture

2.1 Concept of data decoupling

To extract and exploit more parallelism, a future superscalar processor will establish a wide instruction window that consists of a large number of reservation stations, from which instructions are steered to a set of pipelined functional units [19, 15]. Building such a processor, unfortunately, poses many great challenges; Especially, the hardware complexity[1] of the logic that identifies and issues ready instructions from a pool of reservation stations becomes an increasingly severe impediment to a faster clock rate [18]. The situation is exacerbated when there are multiple functional units of the same type, such as identical integer ALUs, because extra time may be needed for arbitration.

An effective way to control the hardware complexity is to partition the instruction window among functional units so that only the instructions from a particular window can issue to the associated functional units. The MIPS R10000 processor, for example, partitions the window into an integer queue, a floating-point queue, and an address queue, based on the instruction type [33]. The *data-decoupled architecture* further partitions the instruction window for data memory accesses, and provides a separate cache for each partitioned window. An example of a pipelined, two-way data-decoupled architecture is depicted in Figure 1(b). Decoupling of data memory accesses at an early pipeline stage in a general-purpose processor has not been seriously considered in the past.[2]

The data-decoupled architecture has two fundamental operating issues: *memory stream partitioning* and *load balancing*. First, decoded memory access instructions should be partitioned into independent streams before they enter the instruction window, as shown in Figure 1(b). Either run-time or compile-time information on per-reference access type is needed. When run-time speculation [6, 15, 30, 17] is used for the classification, verification and recovery actions are required to handle mispredictions. For

[1]Hardware complexity in this paper refers to the critical path length that directly affects the clock cycle time.

[2]Instruction memory accesses have been handled via a separate I-cache.

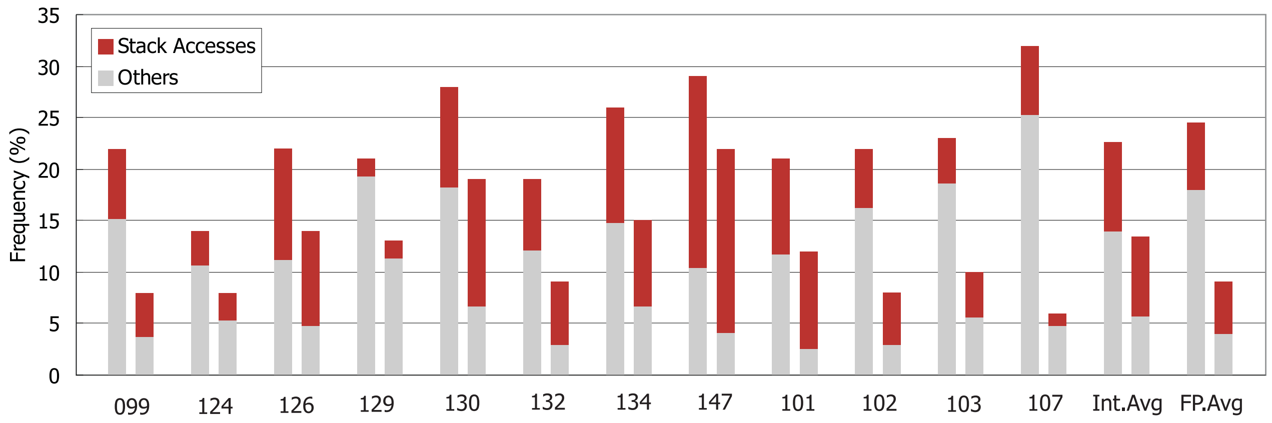

Figure 2. Frequencies of memory access instructions among all the instructions. Two bars for each program denote loads and stores from left. The corresponding program names of the labels (on X axis) are found in Table 2.

verification, a hardware mechanism is necessary to check whether an instruction was correctly classified or not, before (or while) it accesses the data. For instance, an annotation that indicates which stream an access belongs to, can be added to each *Translation Look-aside Buffer* (TLB) entry to this end. A run-time system should maintain the annotation bits properly when new pages are allocated, and the verification logic attached to each memory pipeline can then use this information to determine the validity of the memory access instruction in the pipeline. On a misprediction, *i.e.*, when it is detected that a memory instruction was inserted into a wrong queue, a recovery action similar to the one for a branch misprediction will be taken. Extracting classification information at compile time, on the other hand, can simplify the hardware design, while putting more burden on the compiler. Using a hybrid of both compile-time and run-time information can be more flexible and cost-effective than using either one alone [6].[3]

Second, each partitioned stream should contain an adequate amount of workload to justify the scheduling and possible communication overhead. Moreover, different types of memory references should be interleaved evenly for the approach to be effective. If only one type of references dominates, the resources for the other type are under-utilized or wasted.

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 – less 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 [18]. This is a very critical issue for the future wide-issue processor proposals [15, 19, 23]; They put more pressure on the data cache bandwidth as they aggressively speculate on control and register values, and use high-bandwidth instruction caches [34, 22]. Under such conditions, the proposed approach can have a performance advantage by providing more data bandwidth than a conventional technique at the same level of hardware complexity. The proposed approach can also expose opportunities for reduced memory access latency by properly partitioning memory references and optimizing each stream.

[3]For example, a compiler can mark as many memory references as it can, while leaving ambiguous references to the hardware for classification. At run time, the dispatch unit can insert such references into a particular memory access queue based on prediction. Alternatively, it can copy a reference into both the memory access queues to eliminate any communication between them; In this case, the wrongly inserted copy in LSQ or LVAQ will be killed at a later time.

2.2 A case for decoupling local variable accesses

2.2.1 Background

A variable whose live range spans only within a function is called a *local variable*.[4] Local variables, also known as *automatic* variables, are allocated on the run-time stack when the function that declares them is called, and are automatically deallocated when the function exits. Although not visible to a programmer, additional local variables are generated by a compiler for saving/restoring registers, passing arguments, and register spilling. Memory accesses to these variables, usually indexed by *stack pointer* ($sp), can constitute a large fraction of overall memory references [11]. For example, spill codes can produce a significant number of memory references at run time, as many as 20% of all the executed instructions [1]. Even though spill codes could be eliminated or reduced by increasing the number of (architected) registers and/or by using a more sophisticated register allocation scheme, such attempts are restricted by the current technological trends; Aggressive ILP optimizations often increase register pressure and could introduce extra spill codes. Ultra-fast processor clocks and considerations for instruction set architecture (ISA)

[4]Another definition of a local variable, from a programming language's point of view, is a variable *declared* within a function. A locally declared variable whose storage class is *static* is not "local" in our definition.

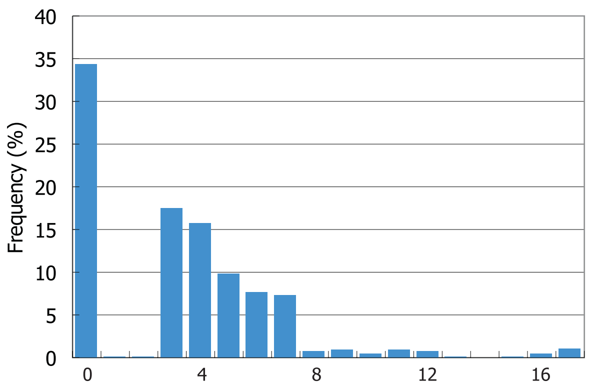

Figure 3. Dynamic frame size distribution of the integer programs studied. The results under 99% percentile are shown to ease reading. Ditzel and McLellan [8] report similar results.

compatibility, as exemplified by the Intel's x86 architecture, may disallow increasing the size of a register file.

Figure 2 shows the frequency of local memory access instructions in a set of SPEC95 programs [27].[5] A large fraction of memory references are to local variables, with an average of 30% of loads and 48% of stores in the programs studied. Over 60% of loads and 80% of stores are local variable accesses in *147.vortex*. They correspond to 10% (*129.compress*) to 71% (*147.vortex*) of all the memory references, with an average of 36%.

Despite the large amount of local variable accesses, their reference space tends to be small. Figure 3 shows the dynamic distribution of the function frame size in the programs studied. The average frame size was only about 3 words. Our static analysis also shows that most frames are very small, typically less than 25 words. The average frame size of 4746 functions in the studied integer programs was only 7 words, while the largest frame was 282 words. Floating-point programs produced similar numbers also. The results suggest that if a separate cache is used to hold the local variables, it need not be large to obtain a high hit rate. In fact, this has been the motivation for some previous work [8, 12, 26, 29].

The high frequency of local variable accesses and their strong locality motivate us to consider decoupling and servicing the local variable accesses separately. Moreover, identifying local variables in the stack frames is relatively easy for hardware or compiler.

2.2.2 Architectural support

To service local variable accesses efficiently, a specialized hardware organization to simulate the run-time stack may appear attractive [8, 12, 26]. 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.** In recent superscalar processors [33, 13], data is forwarded from a store to a later load of the same address in the LSQ. This data forwarding enables faster loads without accessing the data cache. There is another opportunity to perform an even faster forwarding in the LVAQ. 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 the instructions to identify a matching store-load pair within a function frame, even before their effective addresses are calculated, thereby allowing a faster bypassing of the 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** [31]**.** When a program or a program region contains many local variable accesses, the number of LVC ports can become a performance bottleneck. In fact, a procedure call/return generates bursty 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, releaving the bandwidth requirement on it.

2.2.3 Compiler issues

Most local variable accesses are accurately identifiable by a compiler when the compiler comes across their declarations, handles a procedure call, or performs register allocation. There are, however, cases when it is ambiguous to determine if an instruction accesses a local variable or not. Passing a local variable as a parameter, for instance, may hinder correct identification. Figure 4 depicts the situation; bar() reads a value from the frame of foo () via a pointer. If bar () is called only by foo (), the compiler could simply mark the load in bar () as "local". If that is not the case, Z can actually point to a variable allocated in the heap or global region. The compiler cannot classify the load as either local or non-local in this case. To get around this problem, the compiler could either re-allocate X as a "non-local" static variable, or duplicate or expand bar () to provide a local variable accessing version. The studied programs had less than 1% of (static) memory access instructions on average that access both the local and non-local variables at run time. Using a simple 1-bit hardware predictor storing the previous access region of these small number of instructions results in

[5]Measurements in Figure 2 and 3 are based on the machine architecture, the input data, and the compiler options outlined in Section 3.

```
foo()                          bar(struct test *Z)
{                              {
  struct test X,Y; // auto       . . .
  . . .                          . . .
  bar(&X);                       . . . = Z->this; // local?
  . . .                          . . .
}                              }
```

Figure 4. An example of accessing a local variable via a pointer.

about 99.9% of all the dynamic memory references correctly classified into local and non-local accesses [6]. Therefore, this paper assumes that a processor can accurately separate the local accesses from others with such hardware and software techniques.

To convey the classification information to the processor, each memory access instruction may be associated with a bit, indicating to which memory access queue (LSQ or LVAQ) the instruction need be steered. Alternatively, the processor can assume those accesses indexed by $sp (or *frame pointer*, $fp) as local variable accesses [8]. Not all local accesses, however, may be indexed by $sp or $fp; When the address of a local variable is taken to index through the data structure, $sp is not used. Also, having too many local variables in a procedure can overflow the offset used in conjunction with $sp, forcing the compiler to use another register for indexing.[6] The studied benchmark programs had less than 5% of stack references that are not indexed by $sp or $fp.

Optimizations to allocate more variables on the run-time stack are possible. For example, inter-procedural data-flow analysis on statically allocated variables can discover variables whose live range actually is within a function. Programs written in Fortran will benefit from this technique. A recursive function can be restructured to a non-recursive version, either by a programmer or a compiler, if it degrades the LVC hit rate.

As the cost of register spilling becomes relatively small and predictable when the proposed technique is used, the compiler could find more opportunities for aggressive optimizations. Loops which previously couldn't benefit from the unrolling optimization or software pipelining due to high register pressure, may use such program optimizations based on a new cost model.

3 Experimental Setup

3.1 Simulator and machine model

We develop and use a cycle-accurate execution-driven simulator derived from the *sim-outorder* simulator in the SimpleScalar tool set [3]. The machine model used in the experiments is a superscalar processor that supports out-of-order issue and execution, based on the *Register Update Unit* (RUU) [24]. 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, writeback, and commit. Depending on the instruction type, more than one cycle can be taken in the execution stage.

[6]Two such functions have been found in the programs studied – loadcore() and dumpcore() in *124.m88ksim*. These functions use over 11K words of stack space, by allocating a huge C structure and a buffer. A compiler can easily recognize these memory references as local.

BASE MACHINE MODEL	
Issue width	16
No. of regs.	32 GPRs/32 FPRs
ROB/LSQ size	128/64
Func. units	16 integer + 16 FP ALUs, 4 integer + 4 FP MULT/DIV units.
L1 D-cache	2-way set-assoc. 32 KB. 2-cycle hit time.
L2 D-cache	4-way. 512 KB. 12-cycle access time.
Memory	50-cycle access time. Fully interleaved.
I-cache	Perfect I-cache with 1 cycle latency.
Br. prediction	Perfect.
Inst. latencies	Same as those of MIPS R10000 [33].

Table 1. The base machine model. Decode and commit widths are the same as the issue width.

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.

For the experiments in this paper, a processor model that can issue up to 16 instruction per cycle is used, which represents a future wide-issue processor with aggressive issue bandwidth from a large instruction window. The ROB has 128 entries and the LSQ has 64 entries, which are derived from the MIPS R10000 implementation [33]. The ROB and the LSQ effectively form the instruction window of the processor. The primary on-chip data cache is 32 KB in size and 2-way set-associative, and has 2-cycle hit time as in some recent machines [33, 13]. The 512 KB L2 cache, either on-chip or off-chip, has a 12-cycle hit latency. Both caches are lock-up free. When data decoupling is used, a direct-mapped 2 KB LVC is employed. The line size of the caches is 32 Bytes.

Because the goal of the experiments is to study the potential of the proposed scheme, we employ an ideal front-end for the processor model in our study – a perfect instruction cache (100% hit ratio) with a perfect (or oracle) branch predictor, in order to assert a maximum memory bandwidth demand on our memory system as well as to isolate the impact of the proposed scheme from other factors. Important parameters of the base machine model are summarized in Table 1.

3.2 Benchmark programs

We use eight integer and four floating-point programs from the SPEC95 benchmark suite [27], whose inputs and instruction

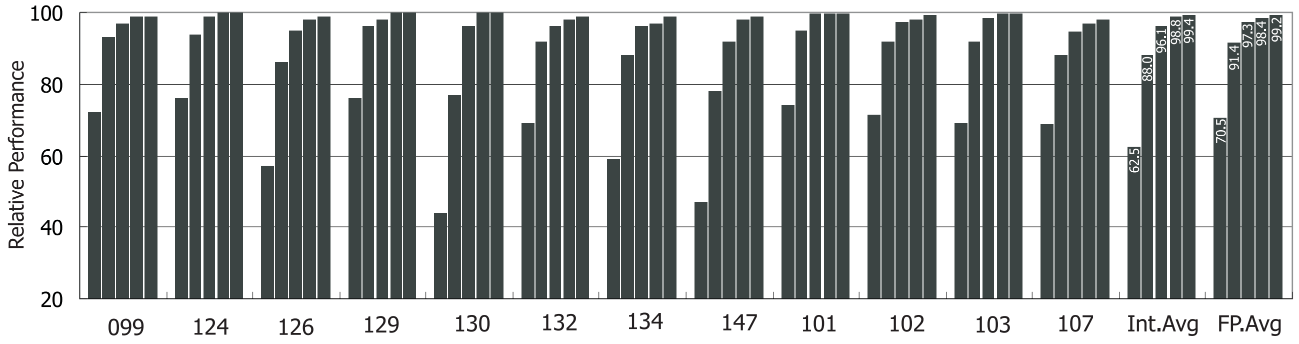

Figure 5. Relative performance of (N+0) configurations to the (16+0) configuration where N is varied from 1 to 5.

counts are described in Table 2. Instruction mixes of these programs, in terms of memory and non-memory instructions, are shown in Figure 2. All the programs were compiled using EGCS[7] version 1.1b [10] 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.

Benchmark	Input	Inst. count
099.go	train	541M
124.m88ksim	ref	250M
126.gcc	stmt-protoize.i	220M
129.compress	train (100K)	293M
130.li	ctak.lsp	434M
132.ijpeg	penguin.ppm	621M
134.perl	scrabbl.pl	525M
147.vortex	train (1 iter.)	284M
101.tomcatv	test ($N = 253$, 1 iter.)	549M
102.swim	test (3 iter.)	473M
103.su2cor	test	676M
107.mgrid	train (1 iter.)	684M

Table 2. Input and dynamic instruction count of each benchmark program.

4 Evaluation Results

This section begins by addressing the question: "How much data cache bandwidth does each program need ?" in the first subsection, to provide a basis for the following discussions. A cache or an LVC port used in the experiments is assumed to be "ideal"[8] not to limit the results and discussions to a specific multi-ported data cache implementation. The notation "(N+M)" is used to denote a configuration with an N-port data cache and an M-port LVC. If M is zero, no LVC or LVAQ is used.

[7]EGCS is based on widely used GCC. It has a global CSE pass and a global instruction scheduling pass additionally, assisted by an improved alias analysis algorithm.

[8]Under this assumption, an N-port cache can service N data requests in any combination per cycle.

4.1 Program bandwidth requirements

Figure 5 shows the performance of (N+0) configurations relative to the performance of the (16+0) configuration (*i.e.*, the limit case with a maximum bandwidth) when N is varied from 1 to 5. Results suggest that a three- or four-ported data cache provides the processor with enough bandwidth to achieve the maximum performance. A cache with two ports obtains almost 90% of the maximum performance on average. Programs with frequent memory accesses, such as *130.li* and *147.vortex*, are more sensitive to the data cache bandwidth. In the following experiments, we focus on the (2+M), (3+M), and (4+M) configurations, which more or less correspond to the situations where the cache bandwidth is rather constrained, adequate, and amply available, respectively. When comparing the performance of different configurations, we use the relative performance over the (2+0) configuration.

4.2 LVC and LVAQ parameters

It is difficult to determine the most cost-effective combination of the parameters for the data cache, the LVC, and the LVAQ. Instead of exhaustively searching for a "best" combination, we study the individual performance impact and characteristics of some of the important parameters. For the LVC, the LVC size and the number of ports to the LVC are studied. For the LVAQ, the impact of the degree of access combining and the fast data forwarding is analyzed. We use an LVAQ of 64 entries in our experiments.

4.2.1 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.

Figure 6 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% for all the programs except *126.gcc.* A 4 KB LVC obtains a hit rate of 99.5% or more for all the programs, with an average of about 99.9%. The major reason why a small LVC can achieve a high hit rate is that function frames tend to be very small

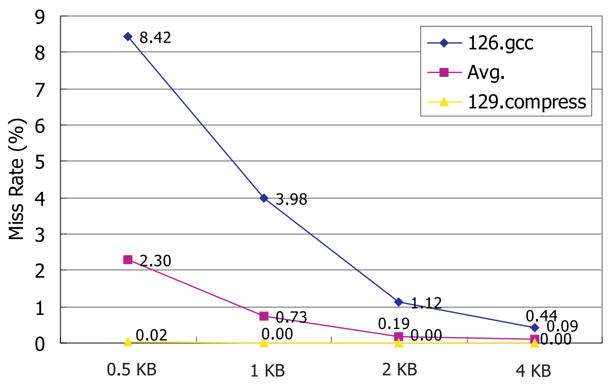

Figure 6. Miss rates of the LVC of different sizes. 126.gcc and 129.compress show the highest and lowest miss rate, respectively. A direct-mapped LVC with four ports is used for measurement.

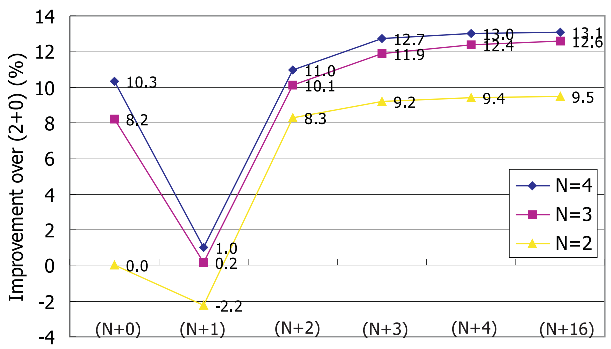

Figure 7. Performance of various (N+M) configurations. The (N+16) configuration provides an unlimited LVC bandwidth for the machine model.

as discussed previously. Furthermore, most of the programs have a call depth of four or five routines [28]. The line size of the LVC, being it 32 or 64 Bytes, has a negligible effect on the hit rate when the LVC size is larger than or equal to 2 KB. The hit rate of an LVC is also relatively insensitive to the input data, because the function frames are generally determined at compile time.

For the rest of the experiments, a 2 KB, direct-mapped LVC with one-cycle hit latency is used.[9] We prefer this design to a 4 KB LVC or a set-associative LVC, because a small direct-mapped cache is likely to have an access time advantage when a fast clock is used [14]. Furthermore, adding additional ports to this small LVC is much cheaper than to a large data cache like the one used in our study (32 KB).

The additional caching space provided by a 2 KB LVC resulted in slight decrease in traffic to the L2 cache for all the programs except *126.gcc*, which experienced a slight increase. In *130.li* and *147.vortex*, there was a considerable reduction in the L2 cache accesses (and accordingly the traffic on the memory bus), of 24% and 7%, respectively, implying that these programs have many conflicts between local variables and other data, attempting to occupy space in the data cache. This decrease in the memory bus traffic will improve the overall performance of a processor in the presence of heavy traffic on the bus.

4.2.2 Number of LVC ports

Figure 7 shows the impact of having an LVC with a varying number of ports. In all cases, the addition of an one-port LVC (the (N+1) configurations) degrades the performance due to poor load balance, since the LVC becomes the performance bottleneck. Adding another port to the LVC (the (N+2) configurations) quickly restores the lost IPC and attains a speedup of about 1 – 10% over the (N+0) configurations. Providing more than three LVC ports produces only a slight performance gain, suggesting that a three-port LVC offers nearly unlimited bandwidth for local variable accesses in the studied model.

[9] Results with a two-cycle LVC are discussed in Section 4.3.

4.2.3 Impact of fast data forwarding in LVAQ

Table 3 shows the performance improvement provided by fast data forwarding. Speedups of up to 3.9% were observed. *124.m88ksim* does not benefit from the technique at all, because only about 1% of the loads actually find their values in the LVAQ. On the other hand, *129.compress* gains a speedup of 1.2% even though it has fewer local variable accesses, because almost 80% 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. Figure 11 shows that when the memory bandwidth is the performance bottleneck (N = 2), adding a two-port LVC achieved a spectacular speedup of over 25%, whereas a speedup of less than 2% was observed when there is sufficient bandwidth already (N = 4).

Program	099	124	126	129	130	132
Speedup	2.1%	0%	1.2%	1.2%	0.3%	1.9%
Program	134	147	101	102	103	107
Speedup	3.1%	3.9%	3.9%	0.2%	0.5%	0%

Table 3. Performance improvement with fast data forwarding under the (3+2) configuration.

4.2.4 Impact of access combining in LVAQ

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

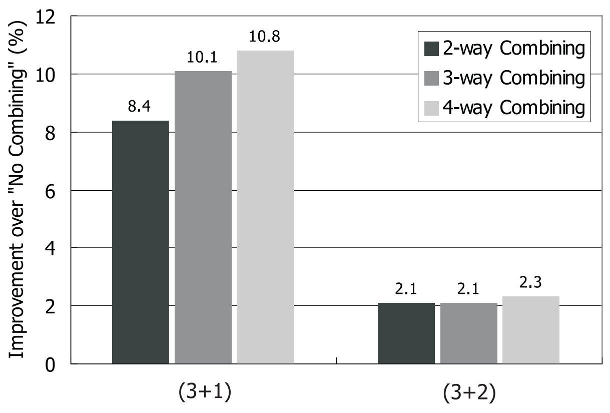

Figure 8. Performance of access combining. N-way combining looks at up to N consecutive entries in the LVAQ for access combining.

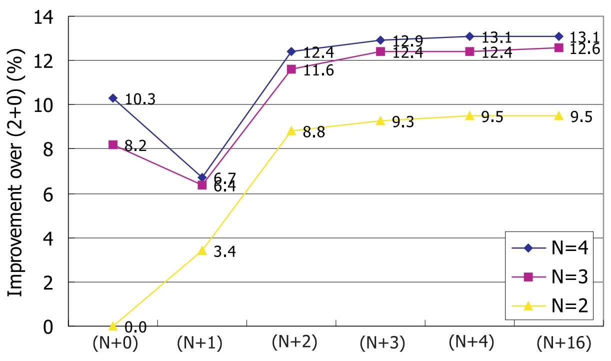

Figure 9. Performance of various (N+M) configurations with the proposed optimizations.

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 of the access combining, the two-way combining seems to be a reasonable choice for implementation. Figure 9 presents the performance of various (N+M) configurations with the fast data forwarding and access combining. Compared with Figure 7, the performance of the (N+1) configuration is noticeably improved. The performance of various (N+M) configurations for 4 selected programs is plotted in Figure 11.

4.3 Sensitivity to cache access latency

In this subsection, we study the performance impact of adding an extra clock cycle to the cache hit latency. The situation may occur if the hardware complexity related to the memory system becomes excessive and the machine designer has to increase the memory access time, not to prolong the clock cycle time. Figure 10 presents the results.

The (4+0) configuration with a three-cycle hit time (fourth bar) degrades the performance by as much as 13.4% (*099.go*) compared with the (4+0) configuration with a two-cycle hit time. In certain cases (*099.go*, *128.m88ksim*, and *129.compress*), it was outperformed by the (2+0) configuration (bars below “zero”). The figure also shows that the (2+2) configuration whose performance is comparable to that of the (4+0) configuration with two-cycle access latency, performs consistently better than the (4+0) configuration with three-cycle cache access time for the integer programs.

For floating-point programs, however, the (4+0) configuration performed better than the (2+2) configuration. In these programs, the occurrences of local and non-local accesses are not interleaved well to benefit from the partitioned caches. Hence, the performance of the (2+2) configuration is close to that of the (2+0) configuration. The (2+2) configuration shows a slight performance degradation in *103.su2cor* due to undesirable interactions between cache accesses and the data forwarding in the LSQ, *i.e.*, more instructions access the 2-cycle data cache instead of being serviced by the 1-cycle forwarding in the LSQ, since accesses are divided into the LSQ and the LVAQ now, making the queue lengths shorter than before.

We also studied the impact of increasing the LVC access time to two (not shown in the figure); It was observed that the overall performance is almost insensitive to the LVC latency. This is in part because many LVC accesses (50 – 90%) find their values in the LVAQ before they reach the LVC, and in part due to the dynamic scheduling capability of the processor. The (3+3) configuration provides sufficient bandwidth for both local and non-local accesses and performs about 5% better than the (4+0) configuration for integer programs.

4.4 Discussions

Our experiments show that data decoupling with an LVC can achieve a comparable performance to a single heavily multi-ported data cache. Therefore, when the hardware complexity of such a multi-ported data cache puts a severe pressure on the clock cycle time or the cache hit latency, the proposed scheme can become a viable and attractive alternative. In *099.go*, *132.ijpeg*, and *147.vortex*, the use of the fast access path for local variable references yields a performance improvement that is not achievable by adding more ports to the data cache. Hence, data decoupling is also complementary to the conventional cache design.

Finally, one may envision that a small (*e.g.*, 2 KB) fast L1 cache (with one or more levels of larger caches on chip) is a more effective solution to the bandwidth and latency problem in a wide-issue processor. The cost-effectiveness of this approach should be studied in depth; However, our preliminary simulation results (not shown) suggest that the inevitably higher miss rates negate the performance gain due to a short access latency unless the L2 cache latency is less than four cycles. It is worth noting that the recent 21264 processor [13] has incorporated a 64 KB L1 data cache, eight times larger than that of its predecessor [9].

5 Related Work

The idea of optimizing accesses to local variables on run-time stack is not new. Ditzel and McLellan [8] studied a transpar-

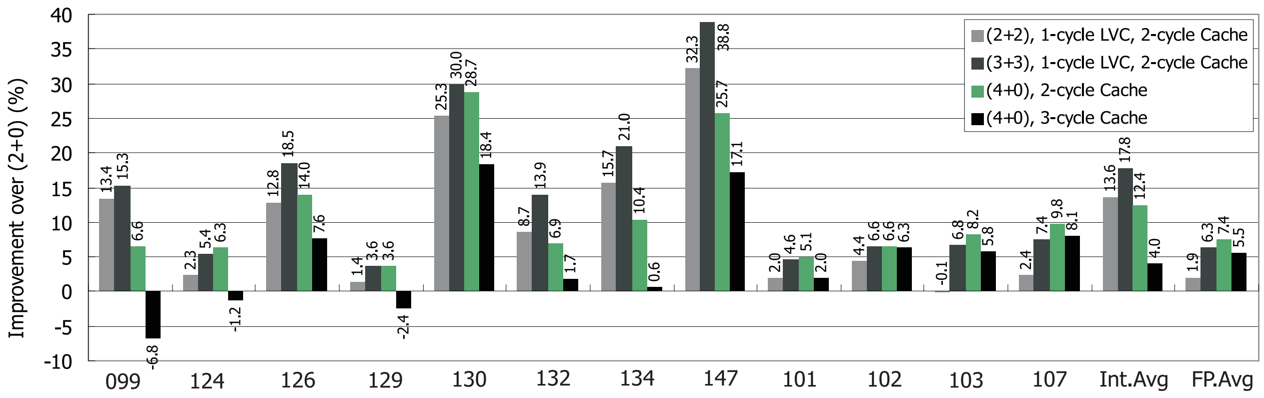

Figure 10. Performance of various configurations.

ent data buffer as a close mapping of the run-time stack, called the stack cache. The stack cache is effectively a large register file to simulate the run-time stack that replaces the general register file. The contour buffer proposed by Flynn and Hoevel [12] in their *Directly Executed Languages* model, is a programmer-addressable buffer that is used in conjunction with the run-time stack in memory. Stanley and Wedig [26] proposed three buffer management algorithms for a Top of Stack (TOS) buffer, which is a register file designed to cache the top elements of the stack. These studies aimed primarily at reducing the impact of a procedure call/return on the processor performance, motivated by an observation that programs written in a high-level language tend to have many procedure calls and returns [11], and that a function call is the most costly source language statement [20]. Unlike the previous approaches, the technique proposed in this paper does not require processor intervention or complex algorithms to manage the buffer, which were mandated in the previous techniques to deal with buffer overflow/underflow and context switches. The proposed LVC under the data-decoupled architecture framework fits in the memory hierarchy, and the data movement between the LVC and lower-level memory is initiated automatically. It is also noted that no previous work has considered separating local variable accesses to relieve the data cache bandwidth in the context of a wide-issue superscalar processor. Among current microprocessors, Sun UltraSparc employs a special register file structure called *register window* to reduce the cost of a procedure call/return [29].

Chow and Hennessy [7] categorize memory traffic into five types of references after register allocation – unallocated references, global scalars, save/restore memory references, a required stack reference, and a computed reference. Register allocation techniques with various heuristics [4, 2, 7, 1] try to efficiently assign a set of hard registers to the live ranges. Increasing the number of registers or using a sophisticated register allocation scheme will cut down the number of memory references in the first category above. Using a large number of registers, however, may increase the overhead of the third type of memory traffic. The proposed approach is complementary to such software efforts, and offers the processor a fast access mechanism for the first, third, and fourth types of memory traffic.

There are dynamic techniques to decouple a portion of data references and service them using separate, specialized functional units. Lipasti introduced a notion called *load stream partitioning* in his Superflow processor model [15], which partitions loads into multiple streams based on their run-time behavior, and sends them to disjoint functional units for processing. The functional units used include a constant verification unit, a queue for load/store folding, a stream buffer/prefetch engine, and a conventional data cache. Techniques to detect dependent memory access instructions and explicitly synchronize and forward data between them have been proposed [30, 17]. They provide a dynamic technique to detect a producer operation and a consumer operation within the instruction window, and try to forward the data in a special buffer before the effective addresses are calculated, without accessing the cache memory. Compared to these approaches, data decoupling requires much simpler mechanism for dynamic classification – a small hardware table and simple instruction decoding logic [6]. The technique proposed in this paper can be implemented with the above techniques together. In fact, it may expose more opportunities for dynamic dependence speculation methods. For example, more tailored prediction techniques can be used for the streams of memory reference in the LSQ and the LVAQ, and the large combined window provided by the LSQ and the LVAQ will allow more dependent instructions far away from each other to be linked together.

Designing an effective multi-ported cache has been a continuing topic of active research [25, 31, 32, 21]. These studies have focused on increasing the efficiency of cache ports by adding a small buffer, or understanding tradeoffs of various strategies in terms of cost and performance under specific processor models. The data-decoupled architecture is orthogonal to the data cache design techniques. Decoupling local variable accesses, however, may dramatically change the memory access patterns seen by the data cache. The change in the memory access behavior may favor a particular data cache design over another.

6 Concluding Remarks

This paper studied the potential of decoupling local variable accesses using a processor model that represents a future wide-issue superscalar processor. Following contributions are made in the paper:

- We introduce the notion of the data-decoupled architecture, which splits the instruction window for memory access into two independent windows, each of which is connected to a dedicated cache. The approach is expected to have implementation and performance advantages over a conventional multi-ported memory system, especially when it is difficult to control the hardware complexity because a processor adopts a large number of reservation stations and requires a data cache with many ports.
- It is shown that the local variable accesses in a program constitute a large fraction of the total memory references using a set of SPEC95 programs. It is also shown that the space required by local variables for each static function is very small, averaging around 7 words. Some of the compiler issues are discussed, which include identifying local variable accesses and possible optimizations.
- We give a preliminary evaluation of the *local variable cache* (LVC) and the *local variable access queue* (LVAQ). They are shown to effectively provide more data bandwidth and facilitate faster local variable accesses. A small 2 KB LVC could achieve over 99% hit rate for most of the programs. Two optimization techniques, fast data forwarding and access combining, are proposed and evaluated.
- We study the impact of decoupling local variable accesses using execution-driven simulations. 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 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.

Acknowledgment

This work was supported in part by NSF grants MIP-9610379 and CDA-9502979, a gift from Intel Corp., and by the U.S. Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order no. D346. 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 U.S. Army Intelligence Center and Fort Huachuca, or the U.S. Government. 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 DoD/Air Force grant F49620-96-1-0472. This work was done while the first author was with the University of Minnesota. The authors are grateful to David Lilja, Dongsoo Kim, Jinseok Kong, Qing Zhao, and the referees for their comments and encouragements on drafts of this paper. Special thanks go to those who have developed and maintained the SimpleScalar tool set used in this work, especially Todd Austin and Doug Burger. Prof. Soo-Mook Moon at Seoul National University helped with the final version of the paper by providing computing resources.

References

[1] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe. "Spill Code Minimization via Interference Region Spilling," *Proc. of the 1997 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 287 – 295. June 1997.

[2] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. "Coloring Heuristics for Register Allocation," *Proc. of the 1989 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 275 – 284. July 1989.

[3] D. Burger and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0," *Computer Sciences Department Technical Report*, No. 1342, Univ. of Wisconsin, June 1997.

[4] G. J. Chaitin. "Register Allocation and Spilling via Graph Coloring," *Proc. of the 1982 ACM SIGPLAN Symp. on Compiler Construction*, pp. 98 – 105, June 1982.

[5] S. Cho, P.-C. Yew, and G. Lee. "Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor," *Technical Report #98-020*, Dept. of Computer Sci. and Eng., Univ. of Minnesota, May 1998.

[6] S. Cho, P.-C. Yew, and G. Lee. "Access Region Locality for High-Bandwidth Processor Memory System Design," *Technical Report #99-004*, Dept. of Computer Sci. and Eng., Univ. of Minnesota, Feb. 1999.

[7] F. C. Chow and J. L. Hennessy. "The Priority-Based Coloring Approach to Register Allocation," *ACM Trans. on Programming Languages and Systems*, 12:4, Oct. 1990.

[8] D. Ditzel and R. McLellan. "Register Allocation for Free: The C Machine Stack Cache," *Proc. of the Symp. on Architectural Support for Programming Languages and Operating Systems*, pp. 48 – 56, March 1982.

[9] J. Edmondson *et al.* "Internal Organization of the Alpha 21164, a 300-MHz, 64-Bit, Quad-Issue, CMOS RISC Microprocessor," *Digital Technical Journal*, Volume 7, Number 1, 1995.

[10] EGCS Project. `http://egcs.cygnus.com`.

[11] J. Emer and D. Clark. "A Characterization of Processor Performance in the VAX-11/780," *Proc. of the 11th Int'l Symp. on Computer Architecture*, June 1984.

[12] M. J. Flynn and L. W. Hoevel. "Execution Architecture: The DELtran Experiment," *IEEE Trans. on Computers*, C-32(2): 156 – 175, Feb. 1983.

[13] L. Gwennap. "Digital 21264 Sets New Standard," *Microprocessor Report*, Volume 10, Issue 14, Oct. 1996.

[14] M. D. Hill. "A Case for Direct-Mapped Caches," *IEEE Computer*, pp. 25 – 40, Dec. 1988.

[15] M. H. Lipasti and J. P. Shen. "Superspeculative Microarchitecture for Beyond AD 2000," *IEEE Computer*, pp. 59 – 66, Sept. 1997.

[16] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. "Dynamic Speculation and Synchronization of Data Dependences," *Proc. of the 24th Int'l Symp. on Computer Architecture*, pp. 181 – 193, June 1997.

[17] A. Moshovos and G. S. Sohi. "Streamlining Inter-operation Memory Communication via Data Dependence Prediction," *Proc. of the 30th Annual Int'l Symp. on Microarchitecture*, pp. 235 – 245, Dec. 1997.

[18] S. Palacharla, N. P. Jouppi, and J. E. Smith. "Complexity-Effective Superscalar Processors," *Proc. of the 24th Int'l Symp. on Computer Architecture*, pp. 206 – 218, June 1997.

[19] Y. N. Patt, S. J. Patel, D. H. Friendly, and J. Stark. "One Billion Transistors, One Uniprocessor, One Chip," *IEEE Computer*, pp. 51 – 57, Sept. 1997.

[20] D. A. Patterson and C. H. Sequin. "A VLSI RISC," *IEEE Computer*, pp. 8 – 21, Sept. 1982.

[21] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin. "On High-Bandwidth Data Cache Design for Multi-Issue Processors," *Proc. of the 30th Annual Int'l Symp. on Microarchitecture*, pp. 46 – 56, Dec. 1997.

[22] E. Rotenberg, S. Bennet, and J. E. Smith. "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. of the 29th Annual Int'l Symp. on Microarchitecture*, pp. 24 – 34, Dec. 1996.

[23] E. Rotenberg, Q. Jacobson, and J. E. Smith. "Trace Processors," *Proc. of the 30th Annual Int'l Symp. on Microarchitecture*, pp. 138 – 148, Dec. 1997.

[24] G. S. Sohi. "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Trans. on Computers*, 39(3):349 – 359, March 1990.

[25] G. S. Sohi and M. Franklin. "High-Bandwidth Data Memory Systems for Superscalar Processors," *Proc. of the Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 53 – 62, April 1991.

[26] T. J. Stanley and R. G. Wedig. "A Performance Analysis of Automatically Managed Top of Stack Buffers," *Proc. of the 14th Int'l Symp. on Computer Architecture*, pp. 272 – 281, June 1987.

[27] The Standard Performance Evaluation Corporation, `http://www.specbench.org`.

[28] Y. Tamir and C. H. Sequin. "Strategies for Managing the Register File in RISC," *IEEE Trans. on Computers*, C-32(11): 977 – 989, Nov. 1983.

[29] M. Tremblay, B. Joy, and K. Shin. "A Three Dimensional Register File for Superscalar Processors," *Proc. of the 28th Annual Hawaii Int'l Conf. on Systems Sciences*, IEEE CS Press, 1995.

[30] G. Tyson and T. M. Austin. "Improving the Accuracy and Performance of Memory Communication Through Renaming," *Proc. of the 30th Annual Int'l Symp. on Microarchitecture*, pp. 218 – 227, Dec. 1997.

[31] K. M. Wilson, K. Olukotun, and M. Rosenblum. "Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors," *Proc. of the 23th Int'l Symp. on Computer Architecture*, pp. 147 – 157, May 1996.

[32] K. M. Wilson and K. Olukotun. "Designing High Bandwidth On-Chip Caches," *Proc. of the 24th Int'l Symp. on Computer Architecture*, pp. 121 – 132, June 1997.

[33] K. C. Yeager. "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, Volume 16, Number 2, pp. 28 – 40, April 1996.

[34] 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. of the 7th Int'l Conf. on Supercomputing*, pp. 67 – 76, July 1993.

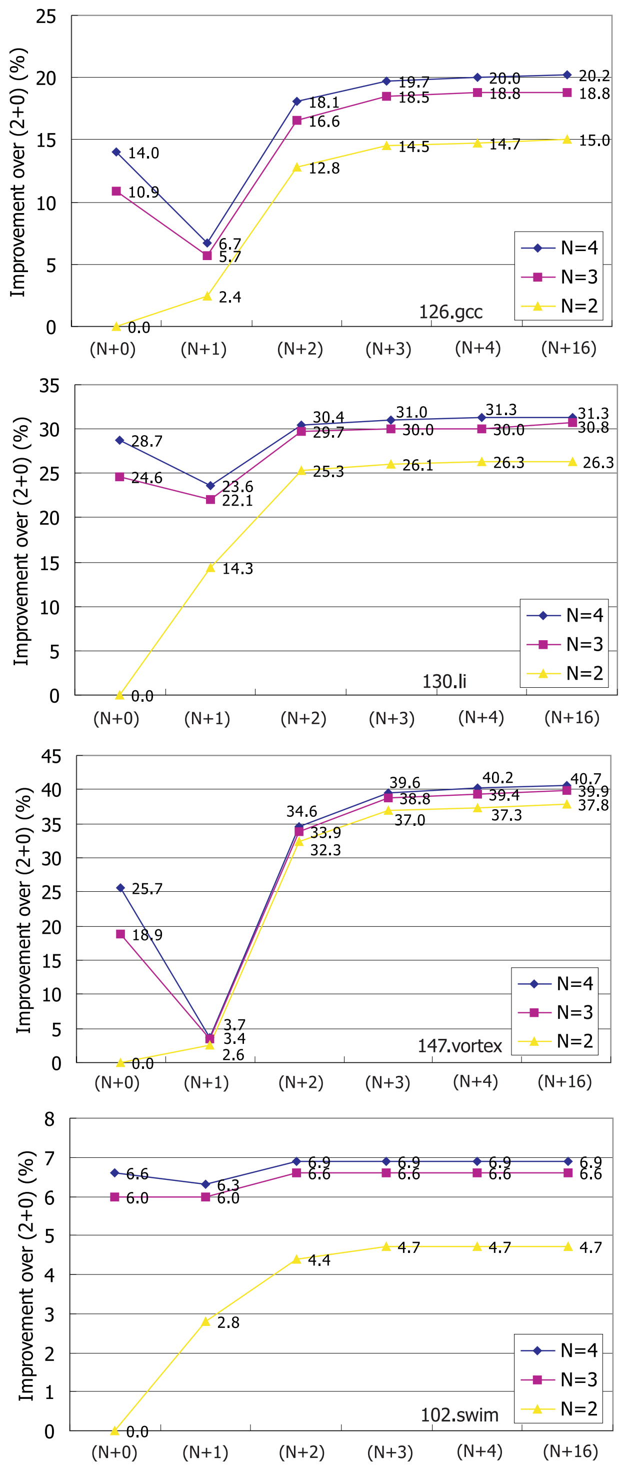

Figure 11. Performance of *126.gcc*, *130.li*, *147.vortex* **and** *102.swim* **under various (N+M) configurations. Results of other programs can be found in [5].**