VLIW Across Multiple Superscalar Processors On A Single Chip*

Soohong P. Kim, Raymond R. Hoare, and Henry G. Dietz
School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285
{ soohong , hoare , hankd } @ecn.purdue.edu

Abstract

Advances in IC technology increase the integration density for higher clock rates and provide more opportunities for microprocessor design. In this paper, we propose a new paradigm to exploit instruction-level parallelism (ILP) across multiple superscalar processors on a single chip by taking advantages of both VLIW-style static scheduling techniques and dynamic scheduling of superscalar architecture. In the proposed paradigm, ILP is exploited by a compiler from a sequential program and this VLIW-like-parallelized code is further parallelized by 2-way superscalar engines at run-time. Superscalar processors are connected by an aggregate function network, which can enforce the necessary static timing constraints and provide appropriate inter-processor data communication mechanisms that are needed for ILP. The aggregate function operations are statically scheduled and implement not only fine-grain communication and control, but also simple global computations resembling systolic array operations within the network.

1 Introduction

As the IC technology advances, an effective use of the enormous number of transistors that will soon be available for future microprocessors becomes an issue. The key question is what will be the dominant microarchitecture paradigm that can efficiently use the billions of transistors and can effectively exploit instruction-level parallelism. Possible competing paradigms include: superpipelined processors, wide-issue superscalar processors, wide VLIW processors, and chip multiprocessors (CMP).

1.1 Superpipelined Processors

Pipelining is a general technique used to increase the speed of any basic function by increasing the depth of the pipeline of the implementation. For superpipelining, the pipeline is divided further into smaller units so that instruction execution can be overlapped (e.g. instruction fetch can be overlapped with instruction decode of another instruction). Multiple function units can be added to the pipeline to increase arithmetic operations and by simplifying the operations of each of the stages the clock rate can be increased. This increase in the number of pipe stages adds more registers (for inter-stage storage) and logic, resulting in increased chip size and power consumption. Deeper pipelines also result in a higher penalty for mispredicted branches.

1.2 Wide-Issue Superscalar Processors

Rather than increasing the pipeline depth, superscalar architectures take advantage of the instruction-level parallelism to increase performance. Superscalar processors add more function units and dispatch more than one independent instruction in one cycle. They look at a window of sequential instructions and check for data and resource dependencies, to *dynamically* schedule as many independent instructions as possible to dispatch to function units.

Wide-issue superscalar architectures, or *Super-superscalars*, such as [13], can issue a maximum of 16 to 32 instructions per cycle without sacrificing code compatibility. History argues that such an engine could never run at peak performance. Limit studies have shown that even when all control and structural hazards are removed, single-thread performance is still severely limited by true data dependences [9]. Moreover, the implementation complexity of the dynamic issue mechanisms and size of the multi-port register files scales quadratically with increasing issue width and ultimately impacts the cycle time of the machine.

1.3 Wide VLIW Processors

VLIW (Very Long Instruction Word) architectures enable parallelism by specifying multiple independent operations as a single word or instruction. In this way, the parallelism is determined *statically* by the compiler and the task of the dynamic issue mechanism is removed from the hardware and given to the compiler.

The VLIW compiler can perform very aggressive, complex analysis to find more parallelism than is possible in hardware. However, the performance of static scheduling can be limited because the control flow of many programs is influenced by run-time data. Since VLIW architectures only contain a single program counter and branch mechanism, the parallelism is limited within a single thread of control. VLIW can lead both to low code density and inefficient memory usage because every word is long regardless of whether the instruction contains one or more computations. Wider instructions may worsen this problem.

1.4 Chip Multiprocessors (CMP)

Hammond et al. [5] proposed the chip multiprocessor (CMP) which consists of multiple (4 to 16) simple, faster processors on one chip. In this architecture, each processor is tightly coupled to a small, fast, level-one cache, and all processors share a large level-two cache. The processors may collaborate on a parallel job or run independent tasks. The CMP architecture lends itself to simpler design, faster validation, cleaner functional partitioning, and higher theoretical peak performance. However, the processor interconnection is not appropriate for exploiting ILP, and for this architecture to realize its performance potential, either programmers or compilers will have to make code explicitly parallel. Old ISAs will be incompatible with this architecture.

In this paper, we described a new paradigm to exploit instruction-level parallelism (ILP) on the chip aggregate multiprocessor (CAMP). The proposed paradigm can take advantages of VLIW's static scheduling and superscalar's dynamic scheduling techniques.

The remainder of this paper is organized as follows. Section 2 describes the chip architecture and the aggregate function network. The execution model is discussed in Section 3 and the compiler technology required is described in Section 4. Section 5 summarizes this paper and discusses the future research direction.

2 Microarchitecture

In this section, we present *chip aggregate multiprocessors* (CAMPs), the microarchitecture of our proposed paradigm. CAMP, shown in figure 1, consists

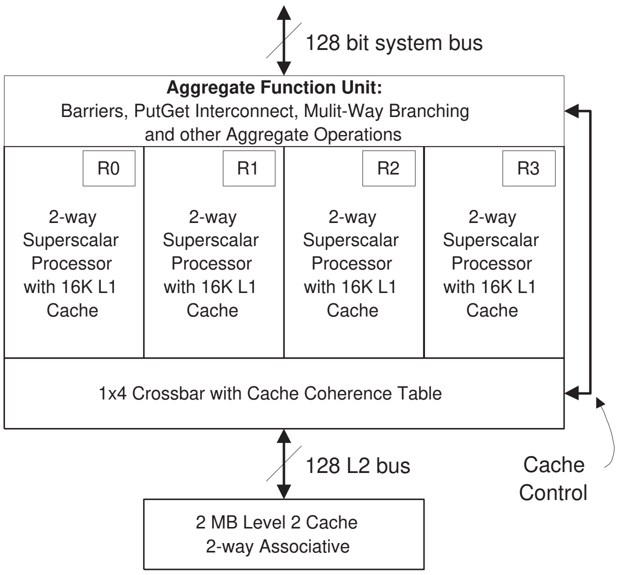


Figure 1: Our microarchitecture, CAMP (Chip Aggregate Multiprocessor)

of four 2-way superscalar processors, on-chip level-one cache, and the aggregate function unit.

2.1 Memory Hierarchy: Shared Cache

Nayfeh et al. [11] compared the performance of three realistic architectures at different levels of the memory hierarchy that could be used in CMPs: shared level-one (L1), shared level-two (L2), and share-memory multiprocessors. Both share-L1 and share-L2 architectures performed similarly. Table 1 shows the contention-free access latencies of both shared cache architectures in CPU clock cycles, used in [11].

Unfortunately, cache contention cannot be ignored with CMP architectures because all inter-processor communication takes place through the shared cache. This is an inherent problem with shared memory architectures because both the writer(s) and the reader(s) need access to the same memory location.

In shared-L1 CMP, a 4x4 crossbar is placed between the processors and the four banks of L1 cache. Each processor of CMP is a 2-way superscalar executing up to two instructions per clock; thus, each processor needs up to four operands per clock for a total of 16 operands in the worst case. Taking a more conservative estimate of each processor making only one memory reference per clock the probability of L1 cache contention can be calculated. Upon each memory reference there is a one in four chance of accessing any particular bank of memory. If all processors access

Data Access Type	Shared-L1 CMP	Shared-L2 CMP
Level 1 Cache	3 cycles	1 cycle
Level 2 Cache	10 cycles	14 cycles
Main Memory	50 cycles	50 cycles

Table 1: Contention-free latencies for shared-L1 and shared-L2 multiprocessors [11]

memory at the same time, the probability of having at least one cache contention is one minus the probability of having no contention ($4!/4^4$). Thus, the probability of having at least one cache contention is greater than 90% (i.e. $1-(4!/4^4)$). This results in pipeline stalls and a drastic reduction in performance.

In our microarchitecture, to reduce cache contention, an aggregate function unit (AFU) is added for the interconnection between processors' register files. Data communications between register files are statically scheduled by the compiler. The aggregate function unit provides a mechanism for fast data communications between processors and will be discussed in section 2.2.

Because inter-processor communication does not use the L1 or L2 caches, the L1 cache can be localized, which reduces latency of L1 cache access. The L2 cache can be shared but this requires that there be some mechanism for providing cache coherency if shared data segments exist. This coherency is solved through the register files or through a combination of the AFU and a cache coherency table located within the 1x4 crossbar.

2.2 Aggregate Function Unit

In our microarchitecture, an aggregate function unit [6] is connected between processors' register files and provides mechanisms for barrier synchronizations, multi-broadcast data communication, and other aggregate function operations.

2.2.1 Synchronization

In CMPs, synchronizations are done through the shared cache and every processor must write to and read from the same cache line at least once. These memory accesses would have to be serialized because they all access the same cache line and therefore, the same cache bank. Thus, for four processors and a shared L1 cache, four writes and at least four reads must be executed serially. Based on the latencies in table 1, this would take upwards of 24 cycles because every access to L1 goes through the crossbar and takes 3 cycles. This would cause all of the superscalar pipelines to stall and performance would drop drastically.

Barrier synchronization is implemented in hardware and is directly attached to all of the processors. Each processor can execute a barrier synchronization, or `wait`, which will stall the processor until all of the other processors also execute their *wait* instruction. This can be trivially implemented as an *AND* or *NAND* gate. AFU collects one-bit from each processor and sends the *ready* signal to all processors so that each of the processors can detect that the `wait` has completed. If all of the processors execute a `wait` at the same time, the ready signal will be received on the following clock. During the clock cycle that follows the completion of the `wait` the inputs of the NAND gate are reset so that complex acknowledgments are not needed.

2.2.2 "Shared" Register File

VLIW processors have a single shared register file and allow each of the function units to have access to data in a fixed amount of time so that the VLIW compiler can properly schedule the instructions. Given that we are using a 2-way superscalar processor, the register files for each of the processors must be able to perform at least four reads and two write simultaneously in a single clock. To enable all four 2-way superscalar processors to share a single register file requires 16 reads and 8 writes to take place simultaneously every clock cycle. Although this may be feasible at some time in the future, it is not currently a reasonable assumption to make. Thus, there are four local register files that each of the superscalar processors have immediate access to. To implement the "shared" register file there must be a mechanism by which data can be transferred between all four register files in a minimal amount of time. This is done through the aggregate function unit using a PutGet interconnect which has some of the characteristics of a crossbar except that it

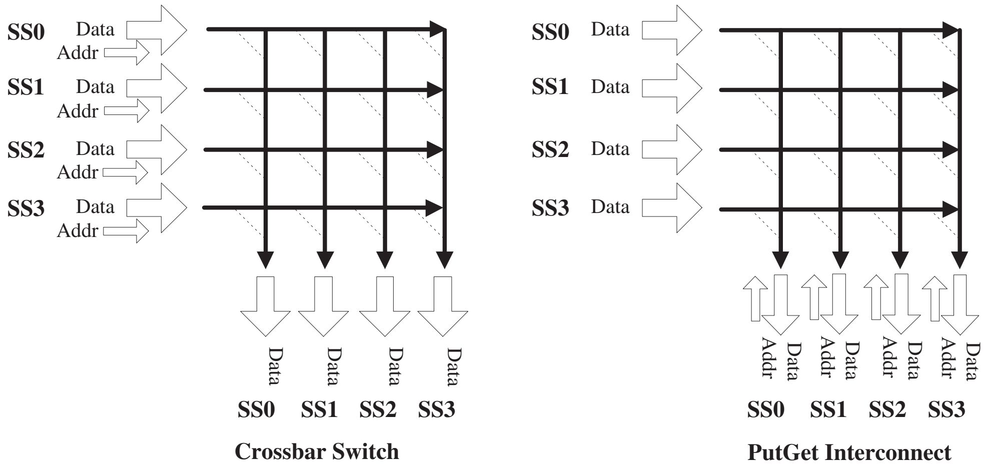


Figure 2: Crossbar Switch vs. PutGet Interconnect

is contention-free.

As shown in Figure 2, a 4x4 crossbar is capable of sending data from four sources to four destinations as long as the destinations are all different. In essence there are four writers that can write to any of four destinations as long as no two writers want to write to the same location. In a Crossbar, superscalar processor #0 (SS0) places both Data0 and Addr0 into the crossbar and then receives that data that has been sent to it. Likewise for SS1, SS2 and SS3.

For a PutGet, SS0 executes the *multi-broadcast* operation [1] `putGet Data0, Addr0`. This *put*s Data0 into the network and *get*s data from Addr0. Likewise for SS1, SS2 and SS3. In essence, there are four readers that specify the location of the data that they wish to read from. Unlike a crossbar, the PutGet is contention-free because multiple readers can read the same data. In other words, each SS places a datum into the PutGet and selects which SS it wishes to receive data from. For a crossbar both the Data and the Address are specified by the source and there can be contention at the destination.

This subtle difference drastically changes the capability and characteristics of the interconnect. With a crossbar, data can be transferred asynchronously because both the data and the address are given at the same time. The PutGet is synchronous and thus, the Data must arrive at the same time or prior to the Address. A second major difference is that a PutGet can have four processors reading the same data within a single cycle. With a crossbar, this requires four cycles because there are four destination addresses. In essence, the PutGet supports single writer, multiple reader whereas the Crossbar supports single reader, single writer.

One of the implied requirements of the PutGet interconnect is knowledge of which datum a particular processor is placing into the PutGet. This is not a problem if the communication operations are statically scheduled and are synchronous. As will be discussed in the next section, there are two phases to each meta-instruction, a synchronous phase and an asynchronous phase. All interprocessor communication and coordination that is needed for the asynchronous (and independent) phase, takes place during the synchronous phase. Thus, all register file reading is performed synchronously across all of the processors. This is scheduled by the compiler because it knows all of the registers that are needed for the given meta-instruction. Thus, the processors *put* and *get* the correct registers. This will take at most k cycles where k is the maximum of: the number of *put*s that any processor needs to make from its local register file for the other processors to read; or the number of *get*s that any single processor needs from a remote register file. This can also be thought of in terms of reads and writes. If a $N \times N$ Crossbar is used, k can be $(N-1)$ times greater because all broadcasts have to be serialized.

2.2.3 Cache Coherence

One of the problems with having a shared L1 cache is contention. This can occur even when each of the processors are using different data. For example, if

[1] Multi-broadcasts allow multiple processors to send data.

Operations	Number of Cycles	Size of NAND Tree
Barrier Synchronization	1 cycle	1 x 4-input NAND
Any or All Operations	1 cycle	1 x 4-input NAND
64-bit OR, AND, NAND, NOR	1 cycle	64 x 4-input NANDs

Table 2: Aggregate operations

four pieces of data are placed randomly into the L1 cache there is a 90% ($1 - 4!/4^4$) chance that at least two pieces of the data will reside on the same cache. The primary reason for having a shared L1 is to reduce the latency required for interprocessor communications for shared-memory (i.e. cache) based communications. Since this restriction has been removed, a shared L1 is no longer required. Thus, the access time for L1 is reduced from more than three cycles to one cycle which increases memory bandwidth by a factor of at least 3. Another advantage is that a more complex cache replacement scheme can be used.

However, a cache coherency mechanism must be used to insure that two copies of a cache line are not modified in separate L1 caches. This can be done in a number of ways. The simplest way is to use the caches only for the register files and for instructions which are read-only. This solves the problem because of the "shared" register file mechanism discussed earlier.

Another mechanism is to take advantage of the bottleneck at the L2 cache. Only a single cache line can pass through the 1x4 crossbar at a time even though the cache lines can be 128 or 256 bits wide. A snooping mechanism can be place at that point and from there a list of all of the L1 caches line request can be kept. If these requests includes the location in the L1 cache that it will be placed in when it is retrieved, then a directory can be created so that the contents of every L1 cache is known. A number of cache coherency protocols can use this table to either invalidate duplicate cache lines when a potential conflict occurs or actively update the duplicate L1 cache line. In either case, there is the possibility that the cache coherence mechanism might fall behind the L1 cache accesses. Therefore, a *CacheWait* instruction can be implemented within the aggregate function unit that, when executed, will wait until the L1 cache is coherent. This can be done every meta-instruction or at other points in the schedule depending on the protocol used. By allowing this operation to be used, unnecessary performance hits do not have to be taken just to avoid potential problems that may never occur.

2.2.4 Other Aggregate Operations

Due to having a four-bit wide *NAND* tree for multiway branching and barrier synchronization, other operations can be implemented using the *NAND* tree and barrier unit. These aggregate, or global, operations take data from each of the processors and return the result of a function to all of the processors. Table 2 describes some of these operations and the size of the NAND tree that they need [6].

3 Execution Model

In our chip aggregate multiprocessor, ILP is exploited with *a single meta-instruction stream* (figure 3b) across four superscalar processors that can stall independently but which are synchronized every few cycles for communication.

3.1 The Program and Meta-Instructions

In the proposed paradigm, a program consists of a sequence of *meta-instructions*. A meta-instruction, shown in Figure 3, is a conceptual counterpart to a "very long instruction word" in VLIW architectures which has a set of op-code fields for function units. Similarly, each *field* of a meta-instruction contains a sequence of instructions for its corresponding superscalar engine.

Within a meta-instruction, fields are *data-independent* of others. That is, any pair of instructions contained in different fields within a meta-instruction are pair-wise data-independent. Consequently, four data-independent instruction streams of a meta-instruction can be executed in the corresponding superscalar processors simultaneously. At run time, each superscalar processor independently fetches, executes instructions in its field of a meta-instruction, and performs the out-of-order execution and dynamic scheduling for further parallelization across its two function units.

3.2 Two Phases Of A Meta-Instruction

A meta-instruction contains a group of synchronous aggregate operations (shaded areas in figure 3a), fol-

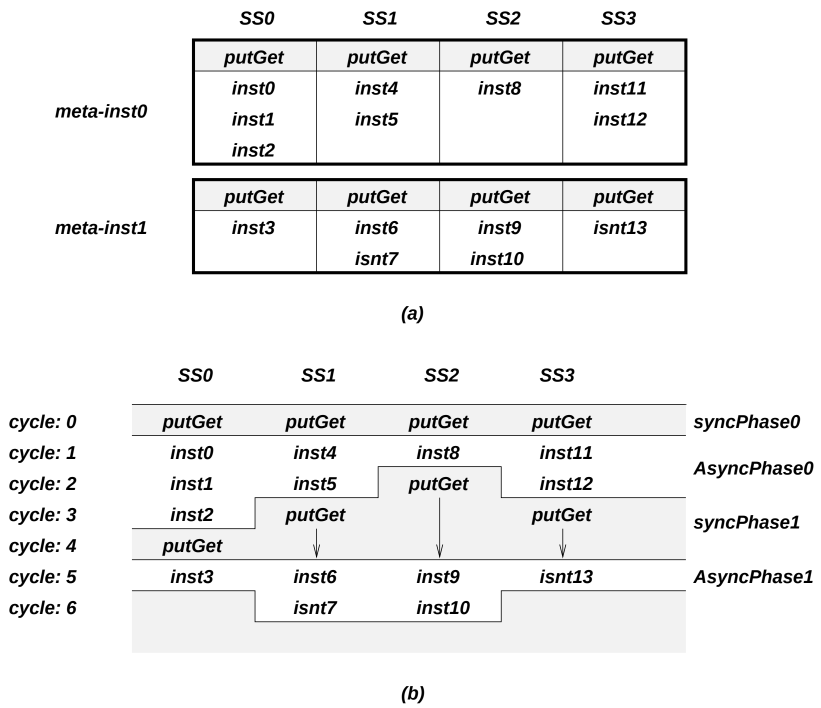


Figure 3: Meta-instructions (a) at compile time and (b) at run time (SS = SuperScalar processor)

lowed by ordinary instructions. During the *synchronous phase*, processors wait and execute synchronous aggregate operations as soon as all processors complete their sub-instruction stream of the previous meta-instruction. synchPhase0 in figure 3b is a synchronous phase of the first meta-instruction (meta-inst0).

While executing aggregate operations, such as `putGet`, processors localize data needed by instructions in the following asynchronous phase because there is no share register file between processors. After the synchronous phase, processors simultaneously begin to execute the rest of the current meta-instruction.

Figure 3b shows the timing diagram of two consecutive meta-instructions. To simplify the example, we assume that the latency of every instruction is one clock cycle and no parallelization is done at run time. SS1 stalls itself after the completion of its instruction stream (`pugGet, inst4,` and `inst5`) and waits at the barrier for other processors. As soon as all processors reach a barrier, they exchange data and simultaneously begin to execute instruction streams independently until the next meta-instruction.

Processors synchronize *only* when they need data from other processors. These synchronous data communications are inevitable data transfers which can be found in architectures determined by a number of instructions which can be executed across the processors in parallel without data communication.

Meta-instructions eliminate redundant synchronizations which occur in VLIW architectures at the end of every very long instruction word although they are implicit synchronizations. This provides latency tolerance between processors so that each processor can execute its instruction stream with its own timing between synchronous phases. For example, in figure 3, `inst1` of SS0 can be fetched and executed even though `inst4` of SS1 causes cache miss. This is because `inst1` is data-independent of `inst5` and there is no synchronization needed between SS0 and SS1 until the next meta-instruction.

Since each processor stalls independently while executing synchronous aggregate operations, *No-op* is not used (no-op is not fetched, nor stored in a memory), resulting in high code density and efficient memory usage.

3.3 Control Flows In A Meta-Instruction

A VLIW machine only contains a single program counter and branch mechanism. That is, only one control operation can be executed each cycle and, therefore, all function units follow a single thread of control. In our paradigm, fields of a meta-instruction can contain independent *components* of a control flow graph

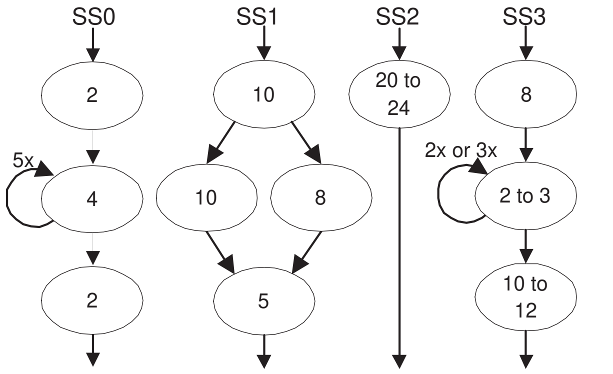


Figure 4: Four data-independent components of control flow graph (CFG) within a single meta-instruction

(CFG) of the program. For example, a single meta-instruction, shown in figure 4, contains a loop, if-then-else, a straight code, and another loop for SS0, SS1, SS2, and SS3, respectively. The ellipses represent basic blocks and their associated numbers indicate latencies. In this example, the execution time ranges from 20 to 29 cycles. For a VLIW architecture, this example meta-instruction can not be executed in parallel even though the fields are data-independent of each other.

3.4 Program Layout and Compatibility

Object-code compatibility between processor generations is an open issues for VLIW architectures and is directly related to the program layout in the main storage or memory. Figure 5 shows two possible schemes of program layout for our proposed paradigm.

Each processor's program can be stored contiguously in a main storage shown in figure 5a. The one superscalar processor's program is stored in a memory, followed by other processors' programs without sharing code address space between processors. This scheme is simple and no special hardware is needed for instruction fetching. Code can be executed in wider chip aggregate multiprocessors, but not in a single superscalar processor.

The other possible scheme is to store a meta-instruction by a meta-instruction in a main memory (figure 5b). Since fields are data-independent of one another, fields of a meta-instruction are stored sequentially and can be executed serially in any order. The code can be executed in a single superscalar processor and wider CAMP architectures. The dotted arrows point to the next instruction for processors. Since address spaces for processors are interwoven in a single

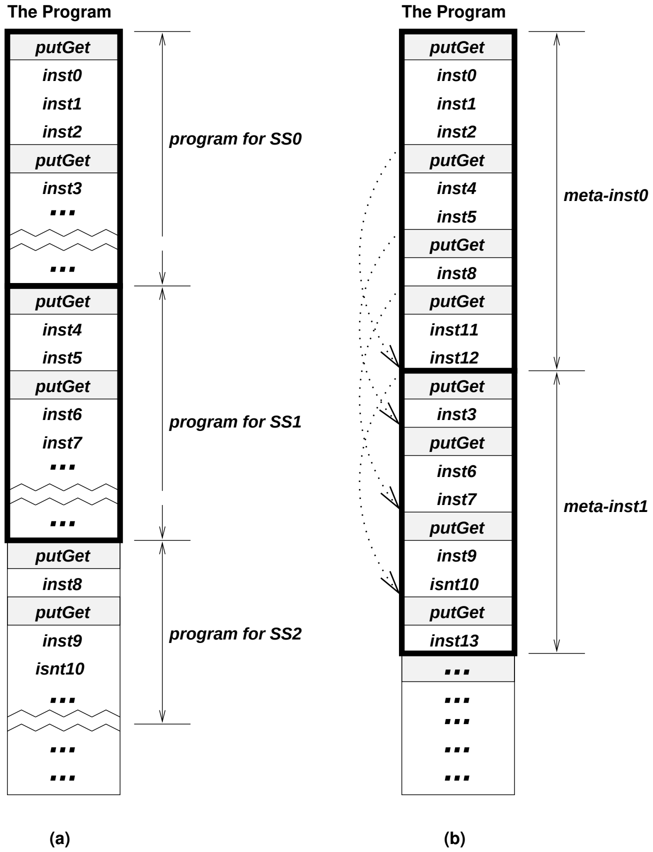


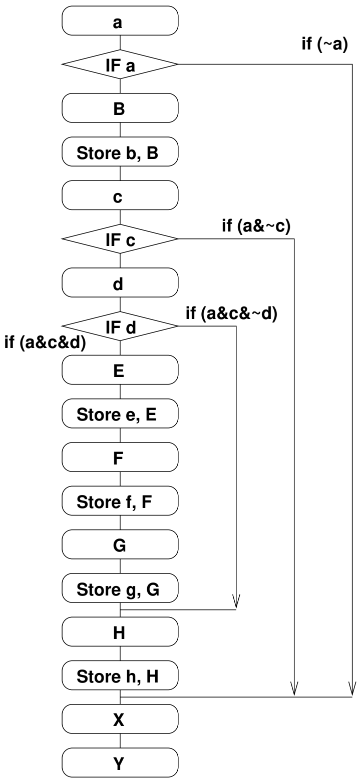
Figure 5: Two program memory layout schemes (for meta-instructions in figure 3) (a) processor by processor and (b) meta-instruction by meta-instruction

memory, there are extra labels needed for the last instructions of each meta-instruction. To improve performance on a single processor, aggregate operations can be ignored or bypassed with a special hardware.

3.5 Compute-While-Transfer

A *reduction* is a class of aggregate function operations which takes operands from participating superscalar processors, performs a simple computation, and broadcasts the result back to processors. The reduction operations are statically scheduled to implement simple global computations resembling systolic array [8] operations *within* the network. Scheduling reduction operations is profitable only if the operands for the reduction aggregate operation do not cause additional data transmission overhead.

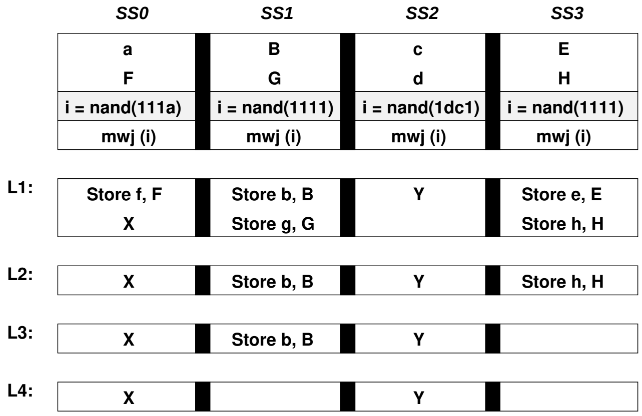
With *compute-while-transfer*, the operands and operation need not be in the same superscalar processor – eliminating unnecessary scheduling constraints – and the reduction result is broadcast to participat-



(a)

nand() arguments				nand() Result	Jump Table	
SS0	*SS1*	*SS2*	*SS3*		index	label
1111	1111	1111	1111	*0000*	0	L1
1110	1111	1111	1111	*0001*	1	L4
1111	1111	1101	1111	*0010*	2	L3
1110	1111	1101	1111	*0011*	3	L4
1111	1111	1011	1111	*0100*	4	L2
1110	1111	1011	1111	*0101*	5	L4
1111	1111	1001	1111	*0110*	6	L3
1110	1111	1001	1111	*0111*	7	L4

(b)



(c)

Figure 6: (a) Sequential code (b) multi-way jump table for SS0, and (c) scheduled code based on multi-way jump

ing processors – possibly reducing future data transmissions. In aggregate function units which support dynamic barrier mechanism [12], different aggregate network operations can be scheduled for different processing elements, even within a single partition. For example, SS0 and SS3 execute `reduceOr` while SS1 and SS2 execute `reduceNor` at the same time.

3.6 Multi-way Jumps

To facilitate more aggressive code motions, VLIW machines allow multiple branch tests to be executed in parallel [1, 3, 4, 10]. As a result, multi-way jumps reduce the number of branches and significantly increase the inter-basic block parallelism. It is possible to implement a multi-way jump using a bitwise NAND aggregate operation [7].

The basic strategy uses a NAND reduction to collect a multi-bit value from each processor. Each bit of this value could be controlled by any processor and is the branch-condition result (1≡true, 0≡false) computed on the controlling processor. The NAND result is used as a key value for every processor to index its local jump table for this multi-way jump. This *generalized* multi-way jump eliminates an unnecessary scheduling constraint by allowing more than one branch condition to be evaluated by one processor within a particular multi-way jump.

Figure 6 shows how a multi-way jump is implemented with an aggregate function unit using `nand()`. The example given in Figure 6a contains three nested 2-way branches, the jump table for SS0 is shown in Figure 6b, and the corresponding code structure is shown in Figure 6c. The resulting code provides not only "branch parallelism" by performing three branch tests on SS0 and SS2 simultaneously, but also aggressive code motions by allowing execution of code blocks `B`, `E`, `F`, `G`, and `H` before the four-way branching of Figure 6a is resolved.

It is useful to recognize that the precise coding of the multi-way branch may need to be appropriately optimized to match the branch handling mechanisms of the individual processors. For example, some superscalar processor designs can, dynamically, speculatively execute past a binary conditional branch, later quashing instructions on paths that were not supposed to be taken. Using this type of processor design, the indirect jump implied by table lookup could block dynamic speculative execution past the jump, resulting in insertion of unnecessary pipeline bubbles at each multi-way branch operation. To optimize for such a processor, the lookup table could be coded as a tree of ordinary binary conditional branches so that the most speculatively profitable paths occur earlier in the tree (which does not necessarily correspond to the original lexical ordering of the binary branches in the source program). In this paper, we simply note that this type of transformation is possible; details of code transformations that are specific to a precise processor implementation are beyond the scope of the current paper.

In our paradigm, speedup is achieved by:

- executing independent instructions within a meta-instruction in parallel across superscalar processors;
- allowing asynchronous execution (latency tolerance) within a meta-instruction;
- taking advantages of out-of-order execution and dynamic scheduling by the superscalar engines;
- performing simple computations within the aggregate function network;
- providing branch parallelism across processors with aggressive code motions (multi-way jump).

4 Compiler Support

Given this compilation target, two questions arise: how much of the existing ILP compiler technology can be used and what new compiler technology is needed to optimize for the aggregate multiprocessor (CAMP)? The obvious differences between CAMP and traditional VLIW architectures do not seriously alter the basic technology for finding instruction-level parallelism in a serial code, except for finding data-independent *components* of program's control flow graph (CFG) (figure 4). Slight additional complexity is required to incorporate detailed static timing analysis [2] into the meta-instruction scheduling. Scheduling aggregate function operations require new compiler technology to improve the performance with the effective usage of aggregate function networks. In this section, we briefly overview the techniques for scheduling multi-broadcast operations.

The multi-broadcast communication of an aggregate function network serves the same purpose as the shared registers and memory of a conventional VLIW. As the scheduler creates each meta-instruction by packing a number of independent instructions, communication operations are inserted as needed. The compiler's goal is to yield the maximum parallel speedup with minimal communication overhead. Because any datum generated on one processor and used on another will require a communication operation, minimizing execution time involves striking a balance between using parallelism and incurring communication latency. The carefully scheduled aggregate operations can reduce the cache contention between processors by taking advantages of the additional interconnection between register files. We use three basic types of scheduling optimizations: operation-operand alignment, value cloning, and pre-transmission.

The scheduler can minimize the number of aggregate operations through *operation-operand alignment*, by assigning both producer and consumer instructions to the same processor. In effect, this optimization technique uses the same basic analysis that determines data layouts for larger-grain constructs in languages like HPF, but the result is treated as an instruction scheduling constraint rather than a data layout issue.

Although it is conceptually sufficient to have a single copy of each datum anywhere within the cluster, the fact that memory is not uniformly accessible to all processors makes it possible to improve performance by cloning certain data objects and computations. In effect, *value cloning* is a combination of forward substitution, common subexpression induction, and detailed tracking of which processors own copies of each datum.

Normally one thinks of scheduling aggregate operations to transmit data only when an operation demands the data, but *pre-transmission* can significantly improve performance. If an earlier multi-broadcast communication operation had data transmission slots unused, a smart compiler can eliminate an additional communication by utilizing unused slots of a prior communication operation to transmit data.

5 Summary and Future Work

In this paper, we described a new paradigm to exploit instruction-level parallelism (ILP) on the chip aggregate multiprocessor (CAMP). A CAMP can be implemented by replicating simpler, 2-way superscalar processors and by augmenting the aggregate function interconnect between processors' register files. ILP is exploited with a single meta-instruction stream across four superscalar processors that can stall independently but which are synchronized every few cycles for communication. An aggregate function network provides low-latency multi-broadcast communications, reductions, and a mechanism for multi-way branching.

The execution model allows more latency tolerance because each sub-instruction stream of a meta-instruction can stall independently and can be executed with processor's own timing, possibly out-of-order. With the addition of a smart interconnection network between register files, the cache contention can be reduced by statically scheduled communications.

This research work is in progress, which makes it difficult to quote real performance numbers. A simulator for our microarchitecture will be implemented and used for measuring performance of the proposed paradigm. A number of benchmark programs will be simulated on the simulator and the benchmark result will provide the insight into open issues, such as the cache structure. Support for multiprocessing will be studied for parallel systems using multiple CAMPs. Scalability becomes an issue for the systems built with multiple CAMPs. One of the issues would be scaling the AFU if a large number of parallel-chips are to be interconnected.

Acknowledgments

The authors would like to thank Kenneth Butler, Russell Rivin, and David R. Levine at Analog Devices Inc. for their support, comments, and suggestions. Special thanks to the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly. This work was supported by the Office of Naval Research (ONR) under Grant N00014-91-J-4013 and the National Science Foundation (NSF) under Grant CDA-9015696.

References

[1] C. J. Brownhill and A. Nicolau. Percolation Scheduling for Non-VLIW Machines. Technical Report TR 90-2, University of California, Irvine, 1990.

[2] H. G. Dietz, M. O'Keefe, and A. Zaafrani. Static Scheduling for Barrier MIMD Architecture. *Journal of Supercomputing*, 5:263–289, 1992.

[3] K. Ebcioglu. Some design ideas for a VLIW architectures for sequential natured software. In *Proceedings of IFIP 10.3 Working Conference Parallel Processing*, pages 3–21, April 1988.

[4] J. A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proceedings of 10th Annual International Symposium on Computer Architecture*, pages 140–150, 1983.

[5] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, September 1997.

[6] R. Hoare, H. Dietz, T. Mattox, and S. Kim. Bitwise Aggregate Networks. In *Proceedings of Symposium on Parallel and Distributed Processing*, October 1996.

[7] Soohong P. Kim and Henry G. Dietz. VLIW-Style Parallelism On Aggregate Function Clusters. *Workshop on Interaction between Compilers and Computer Architectures in conjunction with HPCA-3, San Antonio, TX*, February 1997.

[8] H. T. Kung and C. E. Leiserson. Systolic Arrays (for VLSI). In Duff and Stewart, editors, *Sparse Matrix Proceedings*, pages 25–74. SIAM, Philadelphia, 1978.

[9] Mikko H. Lipasti and John Paul Shen. Superspeculative Microarchitecture for Beyond AD 2000. *IEEE Computer*, 30(9):59–66, September 1997.

[10] S.-M. Moon and S. D. Carson. Generalized Multiway Branch Unit for VLIW Microprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6:850–862, August 1995.

[11] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. In *Proceedings of 23rd Annual International Symposium on Computer Architecture*, pages 66–77, 1996.

[12] Matthew T. O'Keefe. *Barrier MIMD Archicture: Design and Compilation.* PhD thesis, Purdue University, West Lafayette, Indiana, Aug 1990.

[13] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, and Jared Stark. One Billion Transistors, One Uniprocessor, One Chip. *IEEE Computer*, 30(9):51–57, September 1997.