Translation from Transactions of IPSJ

Efficient Execution of Fine-Grain Parallelism on a Tightly-Coupled Multiprocessor

TAKASHI MATSUMOTO*

In multiprocessor systems, the overheads caused by inter-processor synchronization and communication continue to be impediments to the efficient execution of parallel programs. Reduction of these types of overhead is necessary in systems that focus on large-scale and fine-grain parallelism. This paper proposes a Fine-Grain Multi-Processor (FGMP) based on a shared-memory/shared-bus architecture, which can efficiently handle fine-grain concurrency in parallel. New strategies for management of hardware resources in the system are discussed, and two innovative hardware mechanisms are proposed that work well for fine-grain parallelism with the above strategies: Elastic Barrier (a light synchronization mechanism), which is derived from a generalization of a barrier-type mechanism, and an Inter-Cache Snoop Control Mechanism that switches snoop-protocols dynamically to reduce the overhead associated with shared data handling. After introducing the FGMP system, which incorporates the above strategies and mechanisms, the paper closes with a discussion of the FGMP's characteristics and efficiency.

1. Introduction

Methods of extracting parallelism from programs according to the operations executed can be classified as follows. One approach is to leave the extraction of parallelism to the discretion of the programmers. Another is to automate this decision by using parallelizing compilers [1, 2] or parallelizers [3–5]. In the first approach, the programmer is also called upon to level and average the load—including synchronization and communication overheads—on each processor. Efficient execution in this approach hinges on the competence with which averaging is carried out. Additionally, in this method, parallelism can be extracted only by having programmers rewrite the massive software resources created in the past for uniprocessors. The second approach hold out the possibility of extracting parallelism by using software created in the past, as well as manually coded programs with explicit entries for parallelism. Furthermore, it is possible in this approach to consider interprocessor load averaging within a range that permits static analysis. All these factors indicate the advantages of the automatic approach, and the discussions in this paper will be based on these considerations. In other words, this paper will consider optimization of multiprocessor architecture by the use of parallelising compilers (or parallelizers).

Parallelism is most useful when a loop is encountered in a time-consuming control flow. Unless they involve inter-iteration data dependency or branching of control, iterations of loops can be uniformly assigned to processors and executed without processor synchronization. In general, however, loops involve inter-iteration dependency of data as well as branching. As a result, when iterations are distributed by assigning them to several processors, room has to be made within the flow for a number of synchronizations to maintain dependencies and communications so that data can be transmitted from one processor to another. The increase in overheads due to synchronizations and data transmissions tends to prolong the execution time when the load is distributed among processors. The greatest task confronting system architects, therefore, is to minimize such overheads. On the other hand, if the overheads are reasonably low, parallel operations also become possible in loops that do not normally permit parallel execution of iterations. In other words, fine-grain parallelism can be extracted in single iterations as in VLIW machines [6, 7], superscalars [8–11], or data flow machines [12], and these iterations may be handled in parallel. This will be explained with some examples in Section 6.

The multiprocessor system assumed in this paper consists of several connected uniprocessor chips, since a multiprocessor of this type is easy to build. The synchronization and data communication in VLIW machines or data flow machines could take place for each instruction or arithmetic operation. This, however, limits the efficiency of our approach. The

This is a translation of the paper that appeared originally in Japanese in Transactions of IPSJ, Vol. 31, No. 12 (1990), pp. 1840–1851.

*IBM Research, Tokyo Research Laboratory, 5-19, Sanban-cho, Chiyoda-ku, Tokyo 102, Japan.

Current address: Department of Information Science, University of Tokyo, Hongo, Bunkyo-ku, Tokyo 113, Japan.

multiprocessor system proposed here prevents overheads from becoming externally manifest even if processors are synchronized for data transactions for every couple of instructions.

Section 2 details basic strategies for the management of hardware resources in the system, and Section 3 discusses Elastic Barrier, a synchronization mechanism with negligible overheads. Section 4 introduces the inter-cache snoop control mechanism, which reduces bus contention in a shared-bus type multiprocessor through improved bus traffic. The fine-grain multiprocessor **(FGMP)** discussed in Section 5 is based on the above strategies and mechanisms. Section 6 evaluates the FGMP system on the basis of several examples, and Section 7 sums up the discussion.

2. Basic Strategies of the System

Most existing uniprocessor systems have been developed on the assumption that hardware resources should be operated indirectly by users, and so appear virtual. However, to execute fine-grain concurrency efficiently, consideration of only a virtual processor is not enough and may even be harmful, since the use of virtual interprocessor synchronization through an operating system (OS) imposes too heavy a burden on the system. The overhead of crossing the user-kernel boundary alone costs 50–5000 steps. Even if we used synchronization through shared memory, it would be accompanied by procedures that include costly accesses to shared memory. Therefore, there is a need for hardware mechanisms to synchronize processors, which should be controlled or managed directly by users. Moreover, the processor scheduling methods of existing OSs do not provide efficient fine-grain execution when synchronization among tasks takes places frequently. If tasks are not allocated to processors at the same time, synchronization is not completed until all of the tasks have been allocated, and this creates a substantial overhead. Such inefficiency cannot be avoided in current OSs, which do not have the capability to direct that relevant tasks be allocated to processors simultaneously. Such reasoning leads us to a reconsideration of systems on a virtual level and of OSs' facilities before we can implement efficient execution of fine-grain concurrency in multiprocessor systems.

We aim to optimize multiprocessor architecture by using parallelizing compilers. Our basic strategy is therefore that compilers should allocate and manage the critical hardware resources that can be decided statically. Stated another way, the idea here is to restrict the range of decisions required of the operating system and to widen the range of decisions required of users, including compilers. The highest efficiency is obviously achieved when codes can be fully optimized over the bare hardware without any OS. This, however, affects usability and does not allow for the development of multiuser and multijob functions.

Before describing FGMP's strategies, we will explain our terminology to avoid confusion. A **task** expresses a fine-grain processing unit, which is a fundamental processing unit. Groups of tasks form **shreds** [13, 14], which in turn form **processes** and finally **jobs**. A process is a basic unit of processor scheduling by an OS. This paper assumes that a process can have multiple real processor allocations at the same time; a set of tasks in a process corresponding to an instruction flow of a real processor is called a shred, and thus each allocated shred always has a real processor. Stated another way, prior to execution, tasks are assigned statically to the shreds and scheduled accordingly. In a process, we are allowed to use and control hardware support mechanisms directly for synchronization or communication among tasks of different shreds. Among the processes in a job, we can synchronize or communicate through shared memory or the OS. The execution order of processes in a job can be directed by a user. A job expresses an independent processing unit. Among jobs, we can synchronize or communicate, but only through the OS. Figure 1 provides a view of job-process-shred interrelation.

We adopt the following strategies for FGMP systems:

Basic strategies

(1) The OS allocates the specified number (the number of shreds in a process) of real processors to a process at the processor scheduling time. A shred corresponds to a real processor. All shreds in a process are allocated at the same time.

(2) If a multiprocessor system consists of homogeneous processors, the OS has to make up the number of processors, but need not allocate specific ones to a particular process. In this sense the processors are virtual.

(3) The OS and bare hardware have a facility for detecting and removing illegal interferences among processes or jobs. That is to say, we provide protection mechanisms at process and job levels, but not among shreds in a process.

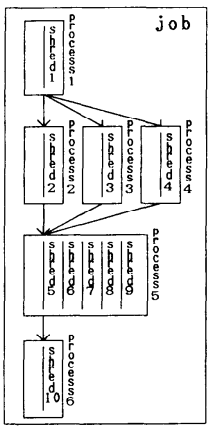

Fig. 1 Job-process-shred Interrelation.

(4) The OS has a facility for pre-empting all allocated processors in a process when a time-slice interrupt or external interrupt occurs, and for rescheduling the processors to other processes according to an appropriate scheduling algorithm.
(5) parallelizing compilers and/or programmers are responsible for dispersion and load-balancing of tasks in a process. That is, they group all of the tasks in a process into several shreds, taking account of the load-balances of processors (shreds).
(6) If a multiprocessor system has mechanisms to support inter-processor synchronization and/or communication, compilers and programmers are allowed to control them directly, provided they are allocated to the same process.

3. Elastic Barrier: A Generalized Barrier-Type Synchronization Mechanism

3.1 Barrier-Type Mechanism

We invented Elastic Barrier,[1] a generalized barrier-type synchronization mechanism [15-17], to allow the efficient execution of fine-grain concurrency. It is described briefly in this section. Under the strategies described in Section 2, it is easy to design a barrier-type light synchronization mechanism to rendezvous all the shreds in a process at a point. We first describe this barrier-type mechanism and then expand it to cover general light synchronization.

Figure 2 shows a diagram of the whole system, including a synchronization mechanism. Here we opt for a shared-bus method as an inter-processor connection for communication. To reduce contentions for data communication we provide another communication bus for synchronization information, called the **synchronization bus**. The number of lines in this synchronization bus is equal to the number of processors, with each line corresponding to a specific processor. Each processor has its own **synchronization controller**, which detects the completion of inter-processor synchronization by using the synchronization bus. When each processor reaches a barrier synchronization point, the processor activates the corresponding line of the synchronization bus through its own controller. Each controller has a **group register** to enroll processors that synchronize; all the allocated processors in a process are enrolled there. Each controller constantly monitors the synchronization bus by referring to the register, to check whether the lines corresponding to all of the processors in a process are active; that is, to check for the completion of barrier synchronization. Processors do not execute further instructions until the completion of the preceding synchronization has been detected.

Information on inter-processor (inter-shred) synchronization is put into instruction streams (shreds) in such a way that a new field or tag is established in a processor's instruction code or new prefix instructions for synchronization are provided. Barrier synchronization information requires only 1 bit in order to express a synchronization request for rendezvous just before (or after) execution of the instruction to which this bit is attached. To eliminate the overhead of the procedure that checks flags or variables for rendezvous, we stop processors temporarily with hardware mechanisms (just like the wait-state in slow memory accesses) until the synchronization is complete. To make ensure that the OS's pre-emption of processor resources is always possible, processors must have interrupt facilities that are valid during the wait-state.

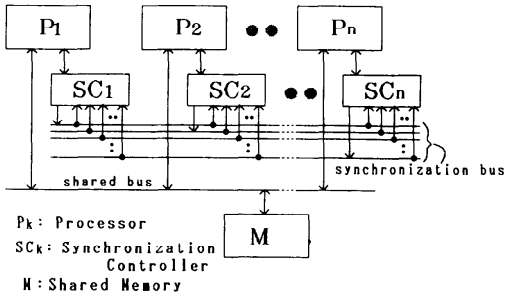

Fig. 2 System Incorporating Synchronization Mechanism.

3.2 Dummy Synchronization Requests

In general synchronization, the combination of processors that need to be synchronized varies according to the synchronization point, even if the processors (shreds) in the same process are considered. If a system holds information on the combination at each synchronization point, it synchronizes only the shreds that need synchronization at each rendezvous point. However, this method requires a rather large amount of hardware, because each processor must have data on all of the others' combinations or synchronization points. If it is adopted, the number of lines in the synchronization bus must be at least the square of the number of processors. Making a trade-off, we adopt a dummy synchronization request and insert it at a point in a shred where synchronization is not needed, but near which other shreds in the same process will be synchronized. In this approach, all shreds in a process rendezvous at each synchronization point, and no additional hardware is needed.

3.3 Extending Synchronization Information and Controller

Calculating the points at which to insert dummy requests is time-consuming and difficult. Moreover, if they are calculated as precisely as possible, an overhead may occur as a result of causes that are impossible to predict before execution, such as latencies of bus conten-

[1] A patent application for the basic mechanism of Elastic Barrier was filed in March 1989.

tion or cache miss. So far we have discussed only the type of synchronization in which processors wait until all of them have reached a certain point. However, in practice, most synchronization among fine-grain tasks is for producer-consumer relationships, and keeps the execution order of tasks intact. Since producer tasks do not need to wait for consumer tasks to begin, an unnecessary overhead is created if they wait. With only slight modifications of the mechanism, we avoid such overheads.

In the following pages, we use a new term, **'CompSCond'**. 'CompSCond' (Completion of Synchronization Condition) occurs when all lines on the synchronization bus according to processors enrolled in the group register become active.

Since our mechanism is based on barrier synchronization, one CompSCond is required at each synchronization point. However, we have circuits for synchronization that are separate from the processors; namely, synchronization controllers. We can eliminate unnecessary processor waits by running the controllers as independently as possible. For this purpose, we extend the synchronization information from 1 bit to 2 bits (thereby giving four directions for synchronization), and provide the synchronization controller with three new counters. Each processor outputs the information to the corresponding synchronization controller just before execution of the instruction to which it is attached.

Figure 3 shows a block diagram of the controller that is connected to the corresponding processor through data lines and three control lines: **SSIG0** (Synchronization SIGnal 0) and **SSIG1** are inputs for synchronization information, and **SACK** (Synchronization ACKnowledge) gives the processor permission to leave the wait-state.

The rough meanings of the four types of synchronization information are as follows:

NONE The processor does nothing for synchronization.

RREQ After outputting this to the controller, the processor stays in the wait-state until it receives a SACK signal from the controller (Real REQuest).

APRV After outputting this, the processor continues execution of instructions without waiting for any signals from the controller. This indicates that the processor need not wait for the end of execution of a task on another processor. APRV relates to dummy synchronization requests or the producer in the producer-consumer pair. This enables the controller to approve the completion of later synchronization (APpRoVal).

PREQ After outputting this, the processor continues execution without waiting for any signals. PREQ is placed before RREQ to give advance notice of its arrival. It is used to check for completion of synchronization in advance of the actual synchronization point (PreREQuest).

Note: RREQ and NONE correspond to 1 bit of synchronization information before expansion.

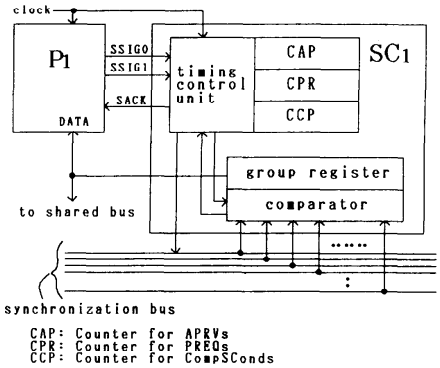

Fig. 3 Configuration of Synchronization Controller.

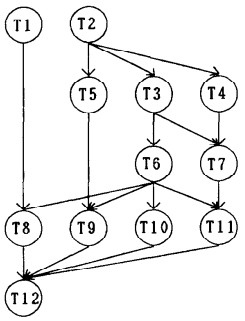

Fig. 4 A Sample Task Graph.

Of the three counters, one is for APRVs before their CompSConds. We call it **CAP**. The second (**CPR**) is for PREQs before their CompSConds. The third (**CCP**) is for CompSConds before their RREQs.

3.4 Illustration of Elastic Barrier Functions

Figure 4 is a sample task graph to clarify the use and operations of the synchronization mechanism. The nodes in the graph correspond to fine-grain tasks and the directed edges between tasks their interdependencies. Figure 5 illustrates a task allocation in which the sample procedure is divided into four shreds (instruction streams). The length of a task indicates the number of instructions, and also the task process time with no processor waiting. In the figure, the combination of a cross and a circle that have the same index number corresponds to an edge in Fig. 4, and the instruction at the cross must not be processed earlier than the instruction at the circle, because of interdependency. In more concrete terms, interdependency involves the use of an instruction immediately preceding a circled instruction to assign the results of calculation to variables and a

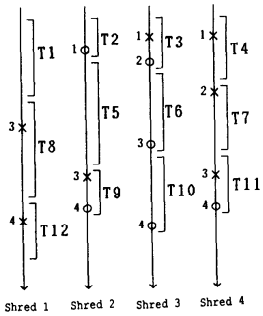

Fig. 5 Shreds (Instruction Streams) of the Sample.

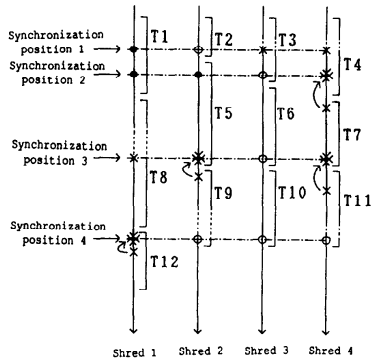

Fig. 6 Estimated Execution Time of the Sample.

crossed instruction to refer to the results.

To clarify the idea further, let us first take an example of synchronization where a simple two-word (RREQ, NONE) message is used. In this case, all the shreds are involved in barrier synchronization, and four synchronization requests (RREQ) must be inserted into each shred for four occurrences of synchronization. Figure 6 shows the estimated execution time, inclusive of synchronization, used to determine the positions of insertion. The dotted line shows the suspension of operations in a processor waiting for synchronization. The solid dots indicate the positions at which "dummy" synchronization requests must be inserted. Double crosses are used instead of ordinary crosses to indicate the positions that seem best suited for insertion of synchronization requests. Synchronization requests are also inserted where circles and crosses occur without any corresponding double crosses.

Next, let us examine synchronization for extended operations (Elastic Barrier operations). When several shreds approving synchronization occur, as at the dummy synchronization request positions (solid dots) or synchronization position 4 in Fig. 6, APRV and CAP (the counter for APRVs) are used in order to simplify the determination of the position at which to insert requests and to minimize the synchronization overhead. Instructions can now be executed without interruption, since shreds issuing dummy synchronization requests or those approving synchronization (circles) need not wait for CompSConds. Thus, for each dummy request (solid dot), an "APRV" is inserted into the shred as far ahead as possible without disturbing the sequence of CompSConds. For each approving synchronization, an APRV instead of a RREQ is inserted at the position of the circle. CAP counts the number of times that APRV has been issued without reaching its CompSCond. The synchronization controller uses this value to control the synchronization signal bus independently of the processor. In Fig. 6, the crossed positions are those that have received approval for continuation of execution from another shred. For this reason, as long as the sequence of CompSConds remains unchanged, theoretically a CompSCond can be satisfied at any position preceding a cross. Accordingly, a PREQ is inserted before a crossed-RREQ, and the stretch between PREQ and RREQ is made a potential range for the CompSCond to be detected. Moreover, to increase the range, the position for insertion of PREQ is made transferable past other crosses to a position as far ahead as possible in the shred. CPR (the counter for PREQs) counts the number of occurrences of PREQs corresponding to undetected CompSConds. When a CompSCond is detected by the controller, this counter counts down by intervals of 1, while CCP (the counter for CompSConds) counts up by intervals of 1. As soon as a processor reaches RREQ, CCP is reduced by 1 if it is 1 or above, and the processor continues its execution. If the counter reads 0, the processor is suspended until another CompSCond is detected. These extensions widen the range of insertion positions to which solid dots and/or double crosses in Fig. 6 are attached without increasing synchronization overheads and increasing the number of instances of covered-up delays, such as are caused by contentions for data communication. In this example, the insertion positions of the four types of synchronization information turn out to be as shown in Fig. 7, where circles indicate insertion positions for APRV, small triangles insertion positions for PREQ, and crosses insertion positions for RREQ. Positions with no marks indicate where NONE is inserted.

3.5 Detailed Description of the Controller

Figure 8 shows the algorithm for synchronization controller operations. For simplicity, it is assumed that the synchronization controllers are provided with a common clock. They follow the clock for reception of synchronization infomation and checks on CompSConds. Also, each synchronization controller executes a series of operations, as shown in the figure. To simplify matters, reception of synchronization information and checks of CompSConds are treated separately. In actuality, however, these two operations are treated as occurring together at the same clock signals. Activate (SL) is a procedure for activating the synchronization signal

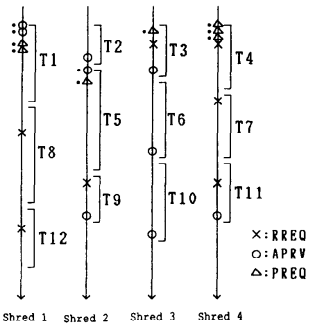

Fig. 7 Insertion Positions for Synchronization Information.

line. It executes no operation if the synchronization signal line is already active. Similarly, the Negate(SL) procedure makes the synchronization signal line inactive. Output (SACK) is for output of SACK to a processor. Waiting (SACK) is a logical function that is true if the processor is waiting for a SACK.

The synchronization controller also follows another condition to ensure correct synchronization in keeping data dependencies. Thus, it does not activate its synchronization signal bus line from the inactive state until the end of a data communication occurring in response to instructions executed earlier. In other words, this restriction prevents crossing over of the signal for synchronization of a previous data communication.

3.6 Summary of the Elastic Barrier

Before using this Elastic barrier mechanism, compilers arrange the order of synchronization in each basic block of a process. The mechanism then performs the lightest synchronization while keeping the predetermined order intact. The overhead of each synchronization point is at most an electric signal delay (1−2 clocks) from one processor's SSIG to the others' SACK. For details on the Elastic Barrier (such as comparison with other synchronization mechanisms, capacity extensions based on FIFO rather the use of counters, and the pattern characteristics of insertion of synchronization information), refer to a paper by the author [17].

Let us now briefly compare the Elastic Barrier with the 'Fuzzy Barrier' [18]. While our Elastic Barrier has aspects that are similar to the 'Fuzzy Barrier', it was designed quite independently in 1988. Owing to the adoption of APRV information and counters, ours is more general for producer-consumer type synchronization, more convenient for the insertion of dummy requests, and superior in terms of the elimination of overheads. Our mechanism can also realize 'fuzzy barriers' easily [16].

4. Inter-Cache Snoop Control Mechanism

When frequent inter-processor communication is needed among conventional processors, communication through shared memory is most appropriate. There is a simple and economical shared bus method for connecting shared memory and processors. If the contention of data communication on the bus can be avoided, the cost of access to the shared memory is rather low. The snoop cache method [19] was therefore invented and adopted in many systems to reduce the frequency of contention to a considerably lower level. Because data communication takes place very frequently in the execution of fine-grain concurrency, it is necessary to minimize the traffic on a shared bus and to reduce bus contention and data communication overheads even further. We therefore propose an inter-cache snoop control mechanism [15].[1]

```
0. Initialization of the synchronization controller
CCP = 0; CAP = 0; CPR = 0; negate(SL);

1. When synchronization information is received
switch (SSIG) {
case RREQ:
    if (CCP > 0) {
        CCP--;
        output(SACK);
    } else activate(SL);
    break;
case APRV:
    CAP++;
    activate(SL);
    break;
case PREQ:
    CPR++;
    activate(SL);
    break;
}

2. When a CompSCond is detected
if (CAP > 0) {
    CAP--;
    if ((CAP == 0) && (CPR == 0))
        negate(SL);
} else {
    if (CPR > 0) {
        CPR--;
        if (waiting(SACK)) output(SACK);
            else CCP++;
        if (CPR == 0) negate(SL);
    } else {
        output(SACK);
        negate(SL);
    }
}

CCP:  counter for CompSConds
CAP:  counter for APRVs
CPR:  counter for PREQs
SL :  controller synchronization signal line
```

Fig. 8 Algorithm for Operation of the Synchronization Controller.

In shared-bus/shared-memory type multiprocessor systems with snoop caches, the protocol of shared data among processors—that is, the way of handling the shared data—keeps the data consistent, determines the frequency of traffic on shared bus, and significantly affects the performance of the system. Here, we briefly compare two commonly used protocols (update-type and invalidate-type). For fine-grain parallel execution, the update-type protocol is more convenient, because frequent inter-processor communication is attained by having caches put data into other caches automatically. However, when processor-local variables become shared variables, owing to the arrangements of page

[1] A patent application for this mechanism was filed in March 1989.

management, the update-type protocol causes extra traffic on the shared bus. Moreover, in medium- and/or coarse-grain parallelism there are many reported applications [20] for which the invalidate-type protocol is convenient because of the locality of the programs. Thus we cannot state which of the protocols is better for general purposes, since their usefulness depends on the variables or work areas to which they are applied. Switching protocols dynamically according to memory areas is an important means of reducing traffic on the shared bus and of improving performance.

An example of protocol switching in conventional multiprocessors with shared bus architecture is provided by TOP-1 [21]. Here, however, the processor has help from the software and sets only one fixed protocol at the same time in each cache. The system does not take into account the possibility of dynamic and fine protocol switching within the same application. The mechanism considered in this paper differs from that of TOP-1 in that it dynamically specifies protocols for all snooping caches at the respective bus accesses.

For ease of implementation, we pay attention to the page management mechanism (memory management system), which has been implemented in existing processors and/or memory management units. To each page, we attach additional information describing the type of protocol. The example of the page entry (and TLB) of this mechanism is illustrated in Fig. 9. P1 and P0 express the right to access each page of the memory. A, D, and E manage virtual memory. T2, T1, and T0 are original bits for this mechanism and express the type of protocol. To access a page of the shared memory, processors output these bits to show what type of protocol should be chosen in a page. Caches also output these bits on the shared bus when they need to use it (see Fig. 10). Snooping caches select a proper protocol dynamically, using the protocol type signal on the bus as determined by the bits. We can also use the most significant bits of the physical address to express the types of protocols for conventional processors, which might not have output pins corresponding to a non-used field in the page entry.

As a result of compilers' analysis of access patterns and/or programmers' management of protocol types, each variable or work area is allocated to a page with the appropriate protocol type.

Since we are able to switch protocols dynamically, protocols that are good for limited types of data are applicable to the system. As an example of such protocols, we introduce the **'all-read'** type protocol.[1] With this,

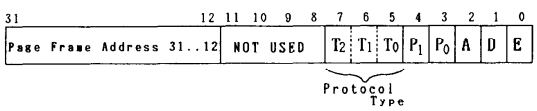

Fig. 9 Extended Page Entry (Identical for TLB).

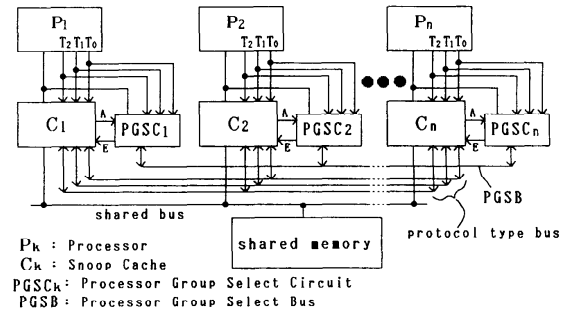

Fig. 10 System with Inter-Cache Snoop Control Mechanism.

snooping caches take in via the bus the data that some cache (processor) reads from the shared memory. However, they take in the data only when they do not need to make extra bus accesses by write-backs to the shared memory. When all processors need to reference the same data at approximately the same time, and this protocol is used, only one access is needed on the bus for each data item. Without this protocol, accesses would cause traffic in proportion to the number of processors, and thus increasingly demonstrate the efficiency of the protocol in such a situation.

In protocols such as all-read type, it is desirable that the group of caches that take in a specific data item simultaneously should be selectable to make it possible for the OS (and users) to group processors. To realize this facility, the lines on the bus that inform caches of the group number (that is, the identifier, such as, the process number) of the data, and a register for each cache that holds the identifier to be output at bus accesses are required. In addition, each snooping cache monitors these lines, but only takes in the data on the bus when the access is all-read type and when the cache is included in the group. The Processor Group Select Circuit **(PGSC)** and Processor Group Select Bus **(PGSB)** in Fig. 10 are the corresponding parts of this facility. In the figure, signal A between a cache and a processor-group-specifying circuit indicates the timing for the output of IDs on the bus specifying the processor group. Signal E indicates the result of monitoring IDs; it is activated if its processor has been selected.

5. The FGMP System

In this section, we describe the FGMP system, which is based on the strategies explained in Section 2 and includes the Elastic Barrier mechanism described in Section 3 and the inter-cache snoop control mechanism described in Section 4. We characterize it in relation to VLIW machines, which are noted for their architecture based on parallelizing compilers.

[1]This protocol was suggested by N. Suzuki, Director of IBM Tokyo Research Laboratory, through private communications in August 1988.

Figure 11 shows a cluster of the FGMP system. Since we adopt the shared-bus method as a basis, we cannot avoid the problem of contentions on the bus when we expand FGMP to a large-scale parallel system. We overcome this impediment by clustering [22]. In this system, the facility of the distributed shared memory [23] is implemented with a memory directory at each cluster, so as to realize a single memory space, which simplifies the OS's management and programming models. Multiple shreds in a process for fine-grain concurrency would be allocated to the same cluster, while processes in a job of medium- or coarse-grain parallelism might be scheduled in more than one cluster. The cluster in Fig. 11 has a Cluster Synchronization Controller (**CSC**) for inter-cluster synchronization, which utilizes the data communication network for exchanges of synchronization information. It allows Synchronization Controllers (**SCs**) to perform inter-cluster light synchronization. This CSC is only for inter-process synchronization [24], which is not described in this paper. Some new protocols for the cache control mechanism are convenient for efficient management of the distributed shared memory. For example, a protocol in which a write access through the network invalidates all clusters' caches, but not the writer's cluster, is useful when inter-cluster shared data have locality in the cluster.

In comparison with VLIW machines that have repetitive function units, the first advantage of the FGMP system is the ease with which processors are implemented. This results from a trade-off in which several instructions are lumped together as the finest-grain task, sacrificing the fineness of separate instructions. In return, we are able to use existing processors as elements of the FGMP system with little or no modification. In VLIW machines, multiple function units and a multi-port register file have to be integrated in a chip to achieve high performance. This is no easy task with present technology, and even if it were, it would still be difficult to make multiple memory interfaces in a chip, owing to the limited number of pins. Moreover, to attain pipeline processing in a unit, additional hardware items, such as by-passes to interconnect the stages of the pipelines, are required. Even if future technology makes it feasible to incorporate a number of function units in a VLIW chip, there will redundant units in jobs that do not have enough inherent parallelism. In the FGMP system, surplus processors are freely allocated to other jobs, and when a job has a lot of coarse-grain parallelism, the FGMP system can handle it as a conventional multiprocessor would. Moreover, in a VLIW machine, because of its clock-synchronize-executions of all units, unpredictable irregularities of instruction executions (such as memory contentions or cache misses) cause considerable overheads. However, in the FGMP system, the Elastic Barrier mechanism keeps processors as independent as possible, thus reducing the overheads if such irregularities take place, though the amount of

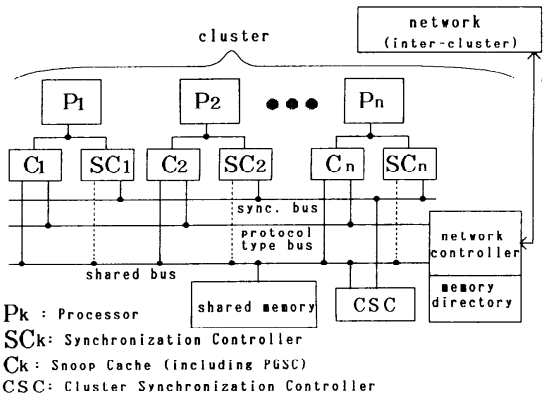

Fig. 11 A Cluster of the FGMP System.

reduction varies according to the situation. Undoubtedly, with regard to the fineness of the target grain of concurrency, VLIW machines are superior to the FGMP system. Accordingly, when a VLIW-type processor with a couple of function units in a chip has finally been developed, the FGMP sysem will adopt it as an element processor. VLIW is an architecture to be implemented in a chip; in contrast, FGMP is an architecture that is appropriate for the construction of multiprocessor systems beyond physical packaging limitations.

6. Early Evaluations of the FGMP System

6.1 Example Programs used for FGMP Evaluations

As we mentiond before, our research on multiprocessor architectures took account of their parallelizing compilers. Methods for extracting parallelism from programs or jobs are very important, and research on ways of developing such methods has been continuing for a considerable time. Many methods have been developed, but most are not fine-grain-oriented, since there were no systems for fine-grain parallelism at the time of their development. In this section, we use some existing extraction methods related to fine-grain parallelism and evaluate the performance of the FGMP system by using simple loop examples. Note that the FGMP system indisputably executes coarse-grain parallel programs, which are compiled and/or written for present multiprocessors, at least as fast as the latter.

First, we consider three examples that do not have parallel execution, because of the overhead of fine-grain concurrent executions. Example 1 in Fig. 12(a) is taken from a paper by Polychronopoulos [25], and is transformed into the form shown in Fig. 12(b) by using his 'cycle shrinking' method to enable it to be executed concurrently. Since only the inner loop can be executed in parallel, barrier synchronization is required at each outer iteration. The example is executed in parallel among four shreds (processors). Example 2 in Fig. 13(a)

```
DO I = 1, N
   X(I) = Y(I)+Z(I)
   Y(I+4) = X(I-5)*W(I)
ENDO
        (a)
DO J = 1, N, 4
   DOALL I = J, J+3
      X(I) = Y(I)+Z(I)
      Y(I+4) = X(I-5)*W(I)
   ENDOALL
ENDO
        (b)
```

Fig. 12 Cycle Shrinking (Example 1).

```
DO i=1, N
 S1: A(i) = B(i-1)*C(i)+37
 S2: B(i) = A(i)*C(i-1)
 S3: C(i) = B(i)*D(i-1)
 S4: D(i) = C(i)*E(i-1)
 S5: E(i) = D(i)+77
ENDDO
        (a)

0  S1(1)
1  S2(1)
2  S3(1)  S1(2)
3  S4(1)  S2(2)
4  S5(1)  S3(2)  S1(3)
5         S4(2)  S2(3)
6  S1(4)  S5(2)  S3(3)
        (b)
```

Fig. 13 Pipeline Execution (Example 2).

is taken from papers by Cytron [26] and Midkiff [27], and like Fig. 13(b), has potential parallel execution among three shreds. A lot of synchronization is required to ensure that parallel execution is carried out and to preserve the dependency of variables. Example 3 in Fig. 14(a) is an image processing procedure whose loop is almost perfectly sequential on the iteration level; there is no parallelism that is syntactically extractable on that level. We divided the procedure of one iteration of the loop into many fine-grain tasks, and restructure them into three shreds (see Fig. 14(b)) by using a task scheduling method based on one given by Kruatrachue [28]. Further parallelism can be extracted by a loop unwinding (unrolling) technique [29] in which the loop is unrolled several times and then acted upon by the above procedures.

Next, we deal with Example 4 in Fig. 15(a). If we divide the loop into independent loops that have only one statement each, we can extract parallelism on the iteration level or even vectorize each loop. However, if we attach importance to reducing the traffic on the bus by means of snoop caches, then dividing the loop is not the best way when the number of iterations is significantly large. Here, the iterations of the loop are executed in parallel without it being divided, but the dependency of the variables is preserved by synchronization (see Fig. 15(b)). It is assumed here that the program is executed in four shreds.

For reference, each row in Table 1 indicates the grain size (the number of instructions and the cost from one synchronization to the next).

```
for (i=0;i<=N;i++){
   j = (i & 0x3ff)+1;
   w = (a0*e[j-1]+a1*e[j]+a2*e[j+1]+a3*tmp+a4*d[i]) >> 5;
   p = t[w];
   if (j != 1)
      e[j-1] = tmp;
   else
      e[1024] = tmp;
   tmp = w-p;
   g[i] = p;
}
        (a)
```

```
for (i=0;i<=N;i++){          | for (i=0;i<=N;i++){      | for (i=0;i<=N;i++){
   j = (i & 0x3ff)+1;        |    w4 = a3*tmp;          |    w5 = a4*d[i];
   w3 = a2*e[j+1];           |    w2 = a1*e[j];         |    w1 = a0*e[j-1];
   s2 = w3+w4+w5;            |                          |
   w = (s2+w1+w2) >> 5;      |                          |    if (j != 1)
   p = t[w];                 |                          |       e[j-1] = tmp;
                             |                          |    else
   tmp = w-p;                |    g[i] = p;             |       e[1024] = tmp;
}                            | }                        | }
     shred 1                 |      shred 2             |      shred 3
                        (b)
```

Fig. 14 Parallelism in a Sequential Loop (Example 3).

```
DO i=1,N
 S1: A(i) = W(i)*A(i)
 S2: B(i) = W(i)*B(i)+A(i-1)
 S3: C(i) = W(i)*C(i)+B(i-1)
 S4: D(i) = W(i)*D(i)+C(i-1)
 S5: E(i) = W(i)*W(i-1)
ENDDO
        (a)

0  S1(1)\ S1(2)\ S1(3)\ S1(4)
1  S2(1)\S2(2)\S2(3)\S2(4)
2  S3(1)\S3(2)\S3(3)\S3(4)
3  S4(1) \S4(2) \S4(3) \S4(4)
4  S5(1)   S5(2)   S5(3)   S5(4)
5  S1(5)\ S1(6)\ S1(7)\ S1(8)
6  S2(5) \S2(6) \S2(7) \S2(8)
        (b)
```

Fig. 15 Example 4.

Table 1 Program Grain Sizes in the Examples.

	Smallest	Average	Largest
Example 1	11 (32)	11 (32)	11 (32)
Example 2	2 (3)	3.7 (12.8)	6 (20)
Example 3	3 (5)	6.4 (9.2)	9 (15)
Example 4	4 (15)	7.7 (25.0)	15 (45)

Note: The upper part of the cell shows the number of instructions, and the lower part the cost.

6.2 Assumptions on the Simulation

Conditions and assumptions of the simulation are as follows: Each example is written in C language and the compiled assembler source is used as the basic data. The cost of an instruction with only register operations is 1 (unit cost) but for multiplication or division the cost is 4. An instruction with a memory-read operation costs 2 when the cache hits the data, and 6 (pre-process 2+shared-bus access 3+post-process 1) when it misses. An instruction with a memory-write operation always costs 2, because of buffered-write for the shared memory, but in the case of a write for a shared variable

Table 2 Speed-Up Ratio.

	FGMP (sync)	FGMP (all)	usual	ideal	FGMP (unwind)
Example 1	2.67 [1.60]	3.20 [2.00]	0.96 [0.76]	4.00	
Example 2	1.65 [1.38]	2.13 [1.83]	1.23 [1.05]	2.20	
Example 3	1.67 [1.67]		1.08 [0.96]		2.81 [2.41]
Example 4	2.73 [1.71]	3.05 [2.33]	2.15 [1.35]	4.00	

Note: The upper part of the cell relates to double shared-bus use and the lower part of the cell, in parentheses, to single shared-bus use.

or in that of a cache-miss, the cache uses the bus for a time corresponding to a cost of 3 after the execution of the instruction. The processor is free to execute succeeding instructions during the bus write-access by the cache. The buffer of this facility has only one slot for each cache, and the order of bus access by a processor is maintained. The system has two shared buses, which are implemented by using memory interleaving or some other technique (for comparison, the lower part of each cell in the Table 2 provides data on one shared bus). No traffic on the bus is caused by other jobs, but contentions are taken into account in the example. Owing to the evaluation of the loops, all the instructions are in the cache and fetching them costs nothing. Additionally, the cache contains the variables that are used in the continuous iterations within the same processor. In the cache snoop protocol, write operations always correspond to "update," and "all-read" to read operations of array variables in the case where the "all-read" protocol is used. In the FGMP system, the synchronization cost involving synchronization signal transmission is assumed to be 1. In other words, APRV and PREQ cost 0, whereas if the synchronization controller has detected synchronization conditions in advance, RREQ is assumed to cost 0, and if not, it will include the cost, 1, of transmission referred to above and will also raise the cost if execution is suspended until synchronization is complete. In the non-FGMP case, barrier synchronization (using a shared counter, a shared flag, and a local flag in each shred) takes place in the program in Example 1. In the other examples where there are producer-consumer type synchronizations, producers signal that they have finished assigning values to shared variables by writing the loop count value in the shared variables provided at synchronization positions in the respective iterations. At the same time, consumers synchronize by comparing the value with their own loop counts.

As mentioned earlier, the cost in this simulation of the execution of a basic register operation instruction and that resulting from execution of an instruction involving a read operation using the shared bus are assumed to be in the ratio 1:6. The ratio varies between 1:2 and 1:3 in CISC-based conventional multiprocessors, but the above value is set on the assumption that speed-up resulting from the improvement of technology is more difficult to achieve in a shared bus than in a uniprocessor. This setting is not advantageous to fine-grain parallel processing. Although it has not been discussed in detail in this paper, the latency of the memory accesses can be nullified by prefetching data and advanced (and/or speculative) execution of instructions.

6.3 Simulation Results

Table 2 lists the speed-up ratios achieved by simulating the four examples above. Here, the speed of execution achieved by a processor in the program before parallelization is taken to be 1 (unit speed). The first column, "sync," relates to the use of FGMP's synchronization mechanism alone. In the second column, "all," the all-read protocol is used for arrays referencing the same array element in each shred by additionally using the inter-cache snoop control mechanism. The third column, "usual," refers to a contrasting case, that of parallel execution without the above mechanisms. Here, synchronization is done by using the shared memory. The fourth column relates to an "ideal" case free from bus contention and free of synchronization and shared variable updating costs. Example 3 is omitted from this column because the parallelization technique in it is based on cost considerations. As for "unwind," this column shows the effects of parallelizing after four iterations by applying the aforementioned loop-unwinding technique to Example 3. The operation is executed by five shreds.

Table 2 leads us to the following considerations. Examples 1, 2, and 4 are of programs using many arrays as data. They involve frequent use of the shared bus and the overhead resulting from bus contention determines the extent of their speed-up. In such examples, use of the all-read protocol and duplication of the shared bus by, say, interleaving, make a considerable speed-up possible under the above conditions. More complex calculations will increase the grain size, making room for a further acceleration. The compilation technique is rather complex in Example 3. This, however, is an example in which parallelism may be extracted from the sequential loop without loss of speed. As in VLIW machines, it shows the possibility of high-speed execution by sequential loops making use of fine-grain parallelism in an FGMP system with conventional uniprocessors.

7. Conclusions

We have described the FGMP architecture, which can efficiently execute fine-grain concurrency in

parallel. It is based on a shared-memory/shared-bus multiprocessor architecture, and even when existing uniprocessors are used as element processors, can efficiently support a fine-grain quantum, that has relatively few machine instructions. To illustrate the FGMP architecture, we discussed new strategies for managing hardware resources, taking account of its OS and parallelizing compilers, and proposed two hardware mechanisms that work well on the strategies: Elastic Barrier, a light synchronization mechanism; and an inter-cache snoop control mechanism that reduces the overhead associated with shared data handling. We also described the FGMP's potential for improving performance in fine-grain parallel executions, with some simulated examples. Future tasks in this context will include an investigation of the efficiency of program execution amid disturbances such as these resulting from other jobs using the bus or from preemption by the operating system. Other areas that should be considered include additional mechanisms for further increasing efficiency and for extracting/achieving parallelism among grains involving several instructions.

Acknowledgements

The author wishes to thank to Mr. Toshiaki KUROKAWA for his constant support during this study. His thanks are also due to Mr. Takao MORIYAMA and Mr. Shigeru UZUHARA for their valuable criticisms, and Prof. Hidehiko TANAKA of the Faculty of Engineering, the University of Tokyo, under whose guidance the author has been a postgraduate research student.

(An earlier version of this paper was presented as: Matsumoto, T., A Study of FGSM: Fine-Grain Support Multiprocessor, 'IBUSUKI' Summer Workshop on Parallel Processing (Kagoshima, Japan, August 3-5 1989) [13])

References

1. ELLIS, J. R. Bulldog: A Compiler for VLIW Architectures, The MIT Press (1986).
2. HONDA, H. et al Implementation and Performance Evaluation of the Fortran Parallel Processing System on OSCAR (in Japanese), *IEICE Japan SIG Reports*, **89**, 168, CPSY 89-57 (Aug. 1989), 75-80.
3. ALLEN, F. et al. An Overview of the PTRAN Analysis System for Multiprocessing, *Journal of Parallel and Distributed Computing 5*, Academic Press (1988), 617-640.
4. WOLFE, M. Optimizing Supercompilers for Supercomputers, The MIT Press (1989).
5. POLYCHRONOPOULOS, C. D. Parallel Programming and Compilers, Kluwer Academic Publishers (1988).
6. FISHER, J. A. Very Long Instruction Word Architectures and the ELI-512, *Proc. IEEE 10th Int. Symp. on Computer Architecture* (1983), 140-150.
7. COLWELL, R. P. et al. A VLIW Architecture for a Trace Scheduling Compiler, *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (Oct. 1987), 180-192.
8. MYERS, G. J. and BUDDE, D. L. The 80960 Microprocessor Architecture, John Wiley & Sons, Inc., New York (1988).
9. MURAKAMI, K. et al. SIMP (Single Instruction Stream/Multiple Instruction Pipelining): A Novel High-Speed Single Processor Architecture, *Proc. 16th Annual Int. Symp. on Computer Architecture* (May 1989), 78-85.
10. IBM Corp., IBM RISC System/6000 Technology, IBM Corp. Austin, TX (1990).
11. SMITH, M. D., JOHNSON, M. and HOROWITZ, M. A. Boosting Beyond Static Scheduling in a Superscalar Processor, *Proc. 17th Annual Int. Symp. on Computer Architecture* (May 1990), 344-354.
12. SHIMADA, T. et al. An Architecture of a Data Flow Machine and Its Evaluation, *Proc. COMPCON 84 Spring, IEEE* (1984), 486-490.
13. MATSUMOTO, T. A Study of FGSM: a Fine-Grain Support Multiprocessor (in Japanese), *IEICE Japan SIG Reports*, **89**, 167, CPSY 89-37 (Aug. 1989), 37-42.
14. MATSUMOTO, T. et al. Classification of Parallel Programs Based on Grain Size Information and Its Application to a Multiprocessor Resource Management Scheme (in Japanese), *Proc. 7th Conf. of Japan Society for Software Science and Technology* (Oct. 1990), 133-136.
15. MATSUMOTO, T. Fine-Grain Support Mechanisms (in Japanese), *IPS Japan SIG Reports*, **89**, 60, ARC-77-12 (July 1989), 91-98.
16. MATSUMOTO, T. A Generalized Barrier-Type Synchronization Mechanism (in Japanese), *Proc. of Joint Symp. on Parallel Processing '90, IPSJ/IEICE/JSSST* (May 1990), 49-56.
17. MATSUMOTO, T. Elastic Barrier: A Generalized Barrier-Type Synchronization Mechanism (in Japanese), *Trans. of IPS Japan*, **32**, 7 (July 1991), 886-896.
18. GUPTA, R. The Fuzzy Barrier: A Mechanism for High-Speed Synchronization of Processors, *Proc. Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (Apr. 1989), 54-63.
19. PAPAMARCOS, M. and PATEL, J. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories, *Proc. 11th Annual Int. Symp. on Computer Architecture* (1984), 348-354.
20. EGGERS, S. J. and KATZ, R. H. A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation, *Proc. 15th Annual Int. Symp. on Computer Architecture* (May 1988), 373-382.
21. MORIWAKI, A. and SHIMIZU, S. A High-Performance Multiprocessor Workstation (TOP-1) (in Japanese), *Proc. 38th Annual Convention IPS Japan* (Mar. 1989), 1456-1457.
22. JONES, A. K. and GEHRINGER, E. F. The Cm* Multiprocessor Project: Research Review. Dept. of Computer Science, Carnegie-Mellon Univ., CMU-CS-80-131 (1980).
23. AGARWAL, A. et al. An Evaluation of Directory Schemes for Cache Coherence, *Proc. 15th Int. Symp. on Computer Architecture* (May 1988), 280-289.
24. MATSUMOTO, T. Synchronization and Processor Scheduling Mechanisms for Multiprocessors (in Japanese), *IPS Japan SIG Reports*, **89**, 99, ARC-79-1 (Nov. 1989), 1-8.
25. POLYCHRONOPOULOS, C. D. Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design, *IEEE Trans. Comput.*, **37**, 8 (Aug. 1988), 991-1004.
26. CYTRON, R. G. Doacross: Beyond Vectorization for Multiprocessors, *Proc. 1986 Int. Conf. on Parallel Processing*, St. Charles, IL (Aug. 1986), 836-844.
27. MIDKIFF, S. P. and PADUA, D. A. Compiler Generated Synchronization for Do Loops, *Proc. 1986 Int. Conf. on Parallel Processing*, St. Charles, IL (Aug. 1986), 544-551.
28. KRUATRACHUE, B. Static Task Scheduling and Grain Packing in Parallel Processing Systems, *PhD dissertation*, Electrical and Computer Eng. Dept., Oregon State Univ., Corvallis (1987).
29. NICOLAU, A. Loop Quantization: A Generalized Loop Unwinding Technique, *Journal of Parallel and Distributed Computing 5*, Academic Press (1988), 568-586.